Models, Algorithms and Architectures for Reconfigurable Computing

DARPA Contract No.: DABT63-96-C-0049

Final Technical Report

Submitted by:

Principal Investigator: Viktor K. Prasanna
Department of Electrical Engineering–Systems
University of Southern California
Los Angeles, CA 90089-2562.
Ph: (213) 740-4483, Fax: (213) 740-4418
prasanna@usc.edu

Subcontractor: Krishna Palem
ReaCT-ILP Laboratory
New York University
palem@cs.nyu.edu

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave Blank)*

2. REPORT DATE
03/06/2000

3. REPORT TYPE AND DATES COVERED
Technical Report - Final

4. TITLE AND SUBTITLE
Architecutres, Models, Algorithms, and Software Tools for Configurable Computing

5. FUNDING NUMBERS
DARPA
Contract Number:
DABT63-96-C-0049

6. AUTHORS
Viktor K. Prasanna

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Southern California
Departments of Contracts and Grants
University Park
Los Angeles, CA 90089-1147

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Office of Naval Research
4520 Executive Drive, Suite 300
San Diego, CA

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
None

12a. DISTRIBUTION / AVAILABILITY STATEMENT
DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Models, Algorithms, and Architectures for Reconfigurable Computing (MAARC) project developed a sound framework for algorithmic configurable computing and for exploiting this technology for embedded signal and image processing applications. Fundamental configurable computing models and performance metrics were developed to evaluate the scalability of configurable hardware. The developed models and the performance metrics were utilized to analyze dynamic reconfiguration and design model based algorithm mapping techniques for signal processing applications. Mapping techniques were developed to identify the core computational kernels of signal processing applications and map them onto configurable hardware. The mapping techniques are efficient and yield significant performance speed-ups and logic utilization. An interpretive simulation framework was proposed to analyze and visualize dynamic reconfiguration and the proposed mapping techniques. A prototype of the framework, Dynamically Reconfigurable Systems Interpretive Simulation and Visualization Environment (DRIVE) was developed and demonstrated. A model based compiler framework and compiler optimization technologies were designed targeting reconfigurable platforms.

14. SUBJECT TERMS
Reconfigurable Computing, Modeling, FPGA, Hybrid Architectures, HySAM Model, Mapping Techniques, Algorithm Synthesis, String Matching, FFT, DCT, Model-based ATR.

15. NUMBER OF PAGES
451

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
UNCLASSIFIED

18. SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

19. SECURITY CLASSIFICATION OF ABSTRACT
UNCLASSIFIED

20. LIMITATION OF ABSTRACT
UNCLASSIFIED

Contents

I USC Efforts

1 **Summary of Accomplishments**

1.1 Models for Configurable Computing

1.2 Performance Metrics for Evaluating Configurable Systems

1.3 Algorithmic Techniques

1.3.1 Algorithm Specialization

1.3.2 Mapping onto Coarse Grained Configurable Architectures

1.3.3 Mapping Computations onto Hybrid Reconfigurable Architectures

1.3.4 Dynamic Precision Computations

1.3.5 String Matching and Genetic Programming

1.3.6 Instance-dependent Mapping Techniques

1.3.7 Model-based ATR on Configurable Hardware

1.3.8 Mapping Irregular Applications onto Configurable Hardware

1.4 DRIVE Software

2 **References**

3 **Copies of Reports/Publications**

4 **DRIVE Software Source Listing**

5 **Final Review Presentation Slides**

II NYU Efforts

1 **Summary of Accomplishments**

1.1 Model for Compilation onto Adaptive Systems

1.2 Compilation Framework

1.3 Compilation Techniques

2 **Final Review Presentation Slides**

Abstract

The MAARC project developed a sound framework for algorithmic configurable computing and for exploiting this technology for embedded signal and image processing applications. The project was executed by the MAARC research group at USC in collaboration with the ReaCT-ILP lab at New York University. The USC efforts developed fundamental configurable computing models and performance metrics to evaluate the scalability of configurable hardware. The developed models and the performance metrics were utilized to analyze dynamic reconfiguration and design model based algorithm mapping techniques for signal processing applications. Mapping techniques were developed to identify the core computational kernels of signal processing applications and map them onto configurable hardware. The mapping techniques are efficient and yield significant performance speedups and logic utilization. An interpretive simulation framework was proposed to analyze and visualize dynamic reconfiguration and the proposed mapping techniques. A prototype of the framework, *Dynamically Reconfigurable Systems Interpretive Simulation and Visualization Environment* (DRIVE) was developed and demonstrated. As part of the collaborative effort the NYU group designed a model based compiler framework and compiler optimization technologies targeting reconfigurable platforms. This report summarizes the accomplishments of these efforts and the details are provided in the manuscripts submitted as part of the report. The first part of the report describes the accomplishments by the USC MAARC group and the second part describes the accomplishments of the NYU ReaCT-ILP group.

Part I
USC Efforts

1 Summary of Accomplishments

1.1 Models for Configurable Computing

One major problem in using FPGAs to speed-up a computation is the design process. The "standard CAD approach" used for digital design is typically employed. The required functionality is specified at a high level of abstraction via an HDL or a schematic. FPGA libraries specific to a given device (e.g. Xilinx, Altera, etc.) and time consuming placement and routing steps are required to perform the logic mapping. This approach of *logic synthesis* as opposed to *algorithm synthesis* allows the user to specify the design using a behavioral model. But this abstraction is achieved at the expense of performance. The semantics and nature of the algorithm are lost in the mapping phases.

The model based mapping environment takes into account the capabilities and limitations of current as well as projected hardware technologies. In this effort parameterized models for algorithm design and analysis have been developed which possess the following characteristics:

- Cost models for analysis of reconfigurable architectures.
- Techniques for partitioning and placement of designs exploiting algorithm and input structure.
- Cost analysis incorporating the cost of reconfiguration and *partial* and *dynamic* reconfigurability.
- Impact of off-chip communication in designing reconfigurable computing solutions.
- Tradeoffs between reconfigurability and redundancy of hardware.

A Configurable Linear Array model of coarse grained architectures has been developed. The model consists of identical powerful PEs, where the datapaths as well as the functionality of the PEs can be dynamically configured. I/O is performed only at the boundaries which limits the required memory bandwidth. The model has been utilized to map homogeneous computations onto coarse grained architectures [9].

Hybrid System Architecture Model (HySAM), a parameterized model of reconfigurable architectures has been developed. The model encompasses systems with configurable logic attached to a traditional microprocessor. HySAM is a compilation model and facilitates development of architecture independent mapping algorithms. The model has been utilized to develop mapping algorithms for various problems [1, 2, 3].

1.2 Performance Metrics for Evaluating Configurable Systems

A configurable computing solution can be based on generic implementation of the problem in a HDL. But for efficient designs, the nature of the algorithm and the specific input have to be exploited. In designing such configurations, the mapping from a specific instance to actual configuration plays an important role. A configurable computing solution has three components:

1. Design compilation to generate the configurations.
2. Configuring the logic on the device.
3. Execution of the computation tasks on the configured hardware.

The performance of the configurable computing solution can be measured by the total time:

$$T = T_d + T_c + T_e$$

T_d, T_c and T_e correspond to the three steps mentioned above. T_d is the design time, T_c is the configuration time (including the reconfiguration time), and T_e is the actual execution time on configurable logic.

The design time, T_d, is the time needed to map a description of the design in a HDL to low level netlist format by using various high level synthesis tools and technology mapping tools. In current configurable computing designs the design time, T_d, varies anywhere from hours to weeks of computation time on a traditional workstation. The configuration time, T_c, varies from milli-seconds to seconds. The execution time, T_e, varies from nano-seconds to milli-seconds for typical tasks. The execution time in hardware (once the hardware is configured) is usually much lower than that in software because of hardware efficiency. To obtain high performance, techniques are being developed to exploit the structure in the input instance. In such cases, the configurations are generated for each input instance.

The total time to compute a solution has to include the time elapsed from the time the input data is submitted to the time all the the outputs are obtained. This total latency is the metric used in traditional performance measures. But, existing framework takes into account only the actual execution time, T_e, of the developed design in evaluating the performance of the design. It is incorrect to compare only the execution time, especially when the design compilation time is many orders of magnitude greater than the execution time (typical designs take hours to weeks on workstations to compile). In this effort the performance comparison was based on the total time elapsed rather than just the execution time.

1.3 Algorithmic Techniques

1.3.1 Algorithm Specialization

Configurable architectures have architectural characteristics different from traditional computing architectures. It is necessary to explore the space of algorithms for a given problem to map onto configurable architectures. This effort proposed a fast parallel implementation of Discrete Fourier Transform (DFT) using FPGAs. The design is based on the Arithmetic Fourier Transform (AFT) using zero-order interpolation. For a given problem of size N, AFT requires only $O(N^2)$ additions and $O(N)$ real multiplications with constant factors. The design employs $2p+1$ PEs ($1 \leq p \leq N$), $O(N)$ memory and fixed I/O with the host. It is scalable over p ($1 \leq p \leq N$) and can solve larger problems with the same hardware by increasing the memory. All the PEs have fixed architecture. The proposed implementation is faster than most standard DSP designs for FFT. It also outperforms other FPGA-based implementations for FFT, in terms of speed and adaptability to larger problems [8].

1.3.2 Mapping onto Coarse Grained Configurable Architectures

Some configurable architectures address the problem of reconfiguration cost by using coarse grain reconfigurable logic blocks. This reduces the flexibility but also significantly decreases the reconfiguration cost. This effort developed an efficient design for 2D-DCT on dynamically configurable coarse grained architectures. A novel technique for deriving computation structures for *two dimensional* homogeneous computations was developed. In this technique, the speed of the data channels is dynamically controlled to perform the desired computation as the data flows along the array. This results in a space efficient design for 2D-DCT that fully utilizes the available computational resources. Compared with the state-of-the-art designs, the amount of local memory required is reduced by 33% while achieving the same high throughput [9].

1.3.3 Mapping Computations onto Hybrid Reconfigurable Architectures

Loop statements in traditional programs consist of regular, repetitive computations which are the most likely candidates for performance enhancement using configurable hardware.

This effort developed a formal methodology for mapping loops onto reconfigurable architectures. The HySAM parameterized abstract model of reconfigurable architectures developed in this effort (see Section 1.1) is used to define and solve the problem of mapping loop statements onto reconfigurable architectures. A polynomial time algorithm was developed to compute the optimal sequence of configurations for one important variant of the problem [2]. These techniques were also utilized to develop algorithms for mapping loop computations onto multi-context devices [4].

1.3.4 Dynamic Precision Computations

Reconfigurable architectures promise significant performance benefits by customizing the configurations to suit the computations. Variable precision for computations is one important method of customization for which reconfigurable architectures are well suited. This effort developed a formal methodology to manage the variable precision computations. For managing dynamic precision in loop computations, intelligent choices on the use of appropriate modules from the available set of modules with different precision need to be made. These configurations then have to be scheduled to achieve optimal execution schedule. An optimal schedule is based on the metrics defined in Section 1.2. Exploiting dynamic precision using the proposed Dynamic Precision Management Algorithm (DPMA) resulted in a 33% reduction in the computation of a multiplication operation on Xilinx FPGA architecture [3] (see Table 1).

Algorithm	Execution Time (ns)	Reconfiguration Time (ns)	Total (ns)
Standard	655360	20480	675840
Static	532480	17920	550400
Greedy	468010	56320	524330
DPMA	471160	33280	504440
DPMA-run	409600	15360	424960

Table 1: Execution times using various approaches

1.3.5 String Matching and Genetic Programming

An efficient design for string matching on multi-context FPGAs was derived. A novel technique for deriving data-dependent configurations was demonstrated. Based on this, speedups of the order of 10^6 over the conventional CAD tools design flow were obtained (including both the mapping time and the execution time on hardware) [11]. The speed-up obtained using the multi-context FPGA is illustrated in Table 2.

Approach	$T_d + T_c + T_e$			Speedup		
	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^4$	$n = 10^5$	$n = 10^6$
Multicontext FPGA	1.8 ms	18.3 ms	183.1 ms	1.0	1.0	1.0
CAD tool mapping	76.0 s	76.0 s	76.2 s	$\approx 10^5$	$\approx 10^4$	$\approx 10^3$
Software mapping	21.8 ms	39.3 ms	204.1 ms	12.1	2.1	1.1
Sun Ultra 1	30 ms	80 ms	680 ms	16.6	4.4	3.7

Table 2: Speedups for string matching different string sizes, n

The solution to string matching was extended to the area of genetic programming (GP). A fast, compact representation of the tree structures in FPGA logic was developed which can be evolved as well as executed without external intervention. The tree representation permits execution of all tree nodes in a parallel, pipelined fashion. Furthermore, the compact layout enables multiple trees to execute concurrently, dramatically speeding up the fitness evaluation phase. Compared with software implementations, a speedup of 19 for an arithmetic intensive problem and a speedup of almost three orders of magnitude for a logic operation intensive problem were achieved by implementations on a XC6264 FPGA device [10].

1.3.6 Instance-dependent Mapping Techniques

Configurable architectures can achieve performance improvement compared to ASICs by exploiting the structure in the algorithm and the input. Developing designs based on the structure of the input is Instance-dependent mapping. Mapping techniques for such an approach were developed and utilized in mapping graph problems to configurable hardware. High-level designs are synthesized for graph problems and adapted to the input graph instance at run-time. The proposed approach leads to reconfigurable solutions with superior time performance. The time performance metric includes both the mapping time and the execution time as defined in Section 1.2. For example, in the case of the single-source shortest path problem, the estimated run-time speed-up is 10^6 compared with the state-of-the-art. In comparison with software implementations, the estimated run-time speed-up is asymptotically 3.75 and can be improved by further optimization of the hardware design or improvement of the configuration time [7] (see Table 3).

Problem Size	Clock Rate		Execution Time		Mapping Time		Speed-up
vetices×edges	Current	Proposed	Current	Proposed	Current	Proposed	
16×64	1.79	15	8.94	21.42	4 hours	22 msec	6.5×10^6
64×256	1.14	15	56.14	79.02	4 hours	82 msec	1.7×10^6
128×515	0.78	15	164.10	199.72	8 hours	161 msec	1.8×10^6
256×1140	0.34	15	752.94	493.17	16 hours	319 msec	1.8×10^6

Table 3: Performance comparison with the state-of-the-art approach

1.3.7 Model-based ATR on Configurable Hardware

Model-based ATR uses geometric hashing as a technique for object recognition in occluded scenes. In this effort a design technique for parallelizing geometric hashing on an FPGA-based platform was developed. The hash table used in this approach is first transformed into a bit-level representation. By regularizing the data flow and exploiting bit-level parallelism in hardware, the proposed design achieves high performance. Using the proposed approach, given a scene consisting of 256 feature points, a probe can be performed in 1.65 milliseconds on an FPGA-based platform having 32 Xilinx 4062s. In earlier implementations, the same probe operation was performed in 240 milliseconds on a 32K-node CM2 and in 382 milliseconds on a 32-node CM5. Also, the same operation takes 40 milliseconds on a 32-node IBM SP-2. By parameterizing the application and the device characteristics, an area-time efficient design based on these parameters has been derived. Furthermore, the proposed approach can be applied to many geometric hashing methods and is portable to other FPGA devices [6].

1.3.8 Mapping Irregular Applications onto Configurable Hardware

Most intermediate and high-level vision tasks manipulate symbolic data. A kernel operation in these vision tasks is to search symbolic data satisfying certain geometric constraints. Such operations are data-dependent and their memory access patterns are irregular. In this effort a fast parallel design for symbolic search operations using configurable hardware has been developed. The symbolic data is manipulated using a pointer array and a bit-level index array. Depending on the input data, a corresponding search window is calculated and symbolic search operations are performed in parallel. Performance estimates using 16 Xilinx XC6216s and memory modules are very promising. Given 3519 line segments (extracted from an 1024×1024 pixel image), the operation can be performed in 1.11 milliseconds on an FPGA-based platform. On a Sun UltraSPARC Model 140, the same operation implemented using C takes 690 milliseconds [5].

1.4 DRIVE Software

Current simulation tools for reconfigurable architectures are based on existing CAD design flow and perform mapping of designs to low level hardware for simulation. Furthermore, there are very few tools which provide any ability to study the dynamic behavior of reconfigurable hardware. Most of the existing simulation environments are based on simulation of High-level Description Language(HDL) or schematic designs that implement an application.

As part of this effort, a novel interpretive simulation and visualization environment based on modeling and module level mapping approach was developed. The **D**ynamically **R**econfigurable systems **I**nterpretive simulation and **V**isualization **E**nvironment(**DRIVE**) can be utilized as a vehicle to study the system and application design space and performance analysis. Reconfigurable hardware is characterized by using a high level parameterized model. Applications are analyzed to develop an abstract application task model. *Interpretive* simulation measures the performance of the abstract application tasks on the parameterized abstract system model. This is in contrast to simulating the exact behavior of the hardware by using HDL models of the hardware devices.

The **DRIVE** framework can be used to perform interactive analysis of the architecture and design parameter space. Performance characteristics such as total execution time, data access bandwidth characteristics and resource utilization can be studied using the **DRIVE** framework. The simulation effort and time are reduced and systems and designs can be explored without time consuming low level implementations. The proposed approach reduces the semantic gap between the application and the hardware and facilitates the performance analysis of reconfigurable hardware. This approach also captures the simulation and visualization of dynamically reconfigurable architectures. The HySAM model (see Section 1.1) is currently utilized by the framework to map applications to a system model. The proposed approach can be utilized to analyze reconfigurable architectures and application performance and facilitate adoption of such architectures by a larger spectrum of users.

References

[1] K. Bondalapati. *Modeling and Mapping for Dynamically Reconfigurable Architectures.* PhD thesis, University of Southern California. Under Preparation.

[2] K. Bondalapati and V.K. Prasanna. Mapping Loops onto Reconfigurable Architectures. In *8th International Workshop on Field-Programmable Logic and Applications*, September 1998.

[3] K. Bondalapati and V.K. Prasanna. Dynamic Precision Management for Loop Computations on Reconfigurable Architectures. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1999.

[4] K. Bondalapati and V.K. Prasanna. Hardware Object Selection for Mapping Loops onto Reconfigurable Architectures. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1999.

[5] S. Choi, Y. Chung, and V.K. Prasanna. Configurable Hardware for Symbolic Search Operations. In *International Conference on Parallel and Distributed Systems*, December 1997.

[6] Y. Chung, S. Choi, and V.K. Prasanna. Parallel Object Recognition on an FPGA-based Configurable Computing Platform. In *International Workshop on Computer Architectures for Machine Perception*, October 1997.

[7] A. Dandalis, A. Mei, and V.K. Prasanna. Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices. In *Reconfigurable Architectures Workshop*, April 1999.

[8] A. Dandalis and V.K. Prasanna. Fast Parallel Implementation of DFT using Configurable Devices. In *International Workshop on Field-Programmable Logic and Applications*, September 1997.

[9] A. Dandalis and V.K. Prasanna. Space-Efficient Mapping of 2D-DCT onto Dynamically Configurable Coarse-Grained Architectures. In *International Workshop on Field-Programmable Logic and Applications*, September 1998.

[10] R.P.S. Sidhu, A. Mei, and V.K. Prasanna. Genetic Programming using Self-Reconfigurable FPGAs. In *International Workshop on Field Programmable Logic and Applications*, September 1999.

[11] R.P.S. Sidhu, A. Mei, and V.K. Prasanna. String Matching on Multicontext FPGAs using Self-Reconfiguration. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, February 1999.

Reconfigurable Architectures Workshop, International Parallel Processing Symposium, April 1997

Reconfigurable Meshes: Theory and Practice [1]

Kiran Bondalapati and Viktor K. Prasanna

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-2562

Abstract

Configurable computing has recently gained much attention with the promise of delivering an order of magnitude performance improvement over general purpose processors. In this paper we contrast the abstract models of reconfigurable architectures and actual hardware available for configurable computing systems.

There is a wealth of ideas related to abstract models of reconfigurable architectures and fast parallel algorithms which exploit the reconfiguration potential in non-trivial ways. We summarize these abstract models and illustrate the power of these models using several example algorithms. We identify the practical problems in implementing these models in VLSI and describe some prototype implementations. Commercial FPGA devices which are being touted as the solution for building configurable computing systems are also examined. The MAARC[2] project at USC endeavors to bridge this gap between the abstract and the real worlds.

[1] This work was supported by DARPA under contract DABT63-96-C-0049 monitored by Fort Hauchuca.

[2] Models, Algorithms and Architectures for Reconfigurable Computing, http://maarc.usc.edu

1 Introduction

Configurable computing has recently gained much attention with the promise of delivering an order of magnitude performance improvement over general purpose processors. The paradigm of computing in space, i.e., laying out a series of computations on several functional units, as opposed to computing in time, i.e., a series of computations executed in sequence on a single functional unit, is being actively explored. There are several directions in which research is being carried out to realize the potential of configurable computing.

The idea of a VLSI array of processors overlaid with a reconfigurable bus system and an abstract model based on this architecture was proposed in [23]. Several abstract models of reconfigurable architectures and fast parallel algorithms for many problems have been described in the literature. These models include the bus automaton [30], content addressable array parallel processor (CAAPP) [33], polymorphic processor array (PPA) [21], among others. Efficient algorithms for fundamental data movement operations [23, 24], sorting [2, 11, 27, 28], arithmetic [15, 29], graph problems [24], image processing [14, 16] and computational geometry [12] have been developed on reconfigurable meshes. There have been several research prototype implementations of reconfigurable architectures which are related to the abstract models. Such architectures include the GCN [33], YUPPIE [22], CLIP [10], PADDI [7], ABACUS [5], DPGA [8].

Currently the architectures which are being utilized to design reconfigurable systems have their root in Field Programmable Gate Array (FPGA). FPGAs consist of a matrix of fine grain computational elements, usually implemented using lookup tables, with a hierarchy of programmable interconnect. Traditionally, FPGAs have been used for logic design and hardware emulation. Their suitability as computing engines for reconfigurable architectures is being explored in SPLASH [6], DEC PeRLe [3], Teramac [1], among others. But FPGA architectures have been primarily designed to emulate random logic without frequent reconfiguration. Also, on-chip memory capacities are too small, reconfiguration times are relatively long (several milliseconds) and partial reconfiguration is difficult.

The advent of static RAM based FPGA devices has given rise to new opportunities in reconfigurable computing area. These devices provide features which allow changing the device configuration on the fly. But reconfiguration cost is still the prohibitive factor in using them for configurable computing. The other major factor is the lack of software tools which allow synthesis of applications exploiting dynamic reconfiguration. Research is also being carried out in designing coarser grain architectures which incorporate reconfigurable features such as MATRIX [25], BRASS Garp [36], RaPiD [9], CMU CVH [37], COLT [4].

This paper looks at the two extremes of the configurable computing world, the abstract models and actual devices. Though the abstract models have been shown to be very powerful, they are difficult to realize in VLSI. There have been several research prototypes of devices that show promise of implementing reconfigurability. But configurable computing cannot deliver the promise until commercial devices strive to deliver the reconfiguration potential possible with current VLSI technology.

In Section 2 we describe and characterize several variants of the reconfigurable mesh model. In Section 3 we illustrate the power of reconfiguration by describing algorithms for EXOR, Addition, Sorting, Prefix operations and Component labeling. We examine the technical issues in implementing these models and give brief descriptions of several implementations in Section 4. Some commercial devices which look promising for designing configurable systems are also explored in this section. Concluding remarks are made in Section 5.

2 Reconfigurable Meshes

A reconfigurable-bus architecture consists of a multi-dimensional array of processing elements (PEs) connected to a bus through a fixed number of I/O ports. This bus architecture is capable, on a per instruction basis, of configuring a topology that contributes to solving the problem at hand. Bus reconfiguration is achieved by locally configuring the switches within each PE. Different shapes of buses such as rows, columns, diagonals, zig-zag, and staircase can be formed by configuring the switches/ports.

A two dimensional processor array with a reconfigurable-bus system of size MN consisting of identical processors connected as a $M \times N$ rectangular mesh system is called a reconfigurable mesh. An example of a 4×4 reconfigurable mesh is shown in Figure 1. A set of four I/O ports labeled N, E, W and S, connect each PE to its four neighbors to the north, east, west and south, respectively. Each PE has locally controllable switches which configure the connection patterns between the four I/O ports. The switches allow the broadcast bus to be divided into *sub-buses*, providing smaller reconfigurable meshes. The bus and all I/O ports are assumed to be m-bit wide. The connection patterns are represented as $\{g_1, g_2, ...\}$, where each of g_i represents a group of switches connected together. For example {NS,E,W} represents the connection pattern with N and S connected and E and W unconnected.

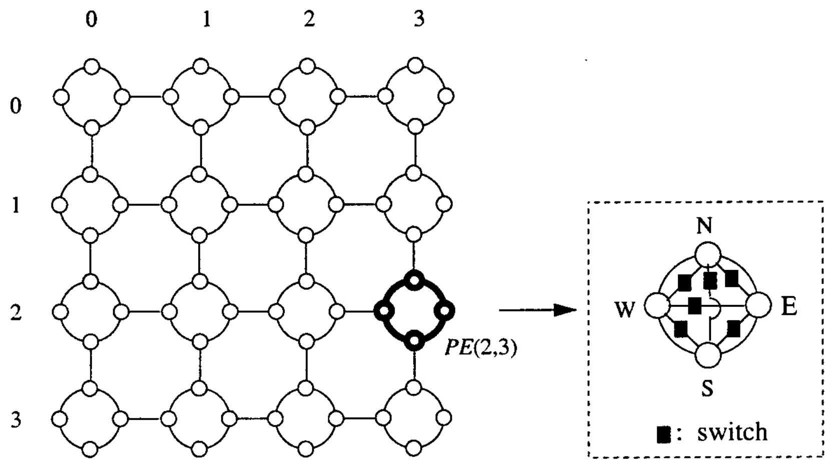


Figure 1: Reconfigurable Mesh.

The basic computational unit of the reconfigurable mesh is the Processing element (PE) which consists of a switch, local storage and an ALU (Fig. 1). In a unit time, a PE can perform:

1. Setting up of a connection pattern.
2. Read from or write onto a bus or local storage.
3. Logical or arithmetic operations on local data.

Various models of reconfigurable meshes have been proposed in the literature. Most of these models are synchronous in nature and permit unconditional global switch setting in addition to local switch control. Unconditional global switch setting is performed by the broadcast of a global instruction from a central controller. Reconfigurable mesh models can be characterized by the following parameters:

- **Width** It refers to the data width of the PE. The two classes of models which have been proposed are bit and word models. The main difference is the width of the input operands of the PE. Also, log n bits (where n is the size of the reconfigurable mesh) need to be accessed when the processor needs to know its position before setting its configuration. Note that the **Width** parameter is not directly related to the bus width of the reconfigurable mesh.

- **Delay** One critical factor in the analysis of reconfigurable algorithms is the time needed to propagate a signal. Some models assume this to be a unit-time operation no matter how far the signal has to travel, while other models assume this to be a function of the number of processors. Time analyses which assume constant time are called *unit-delay models* and logarithmic time are called *logarithmic-delay models.*

- **Bus Access** Each PE connects to the bus through its ports and will either read or write to it. Similar to shared memory machines the models can be classified as CRCW, CREW, ERCW, and EREW based on how the bus is accessed. The most common models are the ERCW models but the CRCW models have also been extensively studied. The CRCW models typically assume that a wired-or operation is performed on a concurrent write by multiple PEs onto the bus.

- **Connection Patterns** Each PE can set the connection between its four ports based on local data or global instruction. There are a total of 15 different connection patterns possible. Different models differ in the number of connection patterns(a subset of 15) which they allow. These models can also be classified based on whether they allow cross-over of the port connections. The models which allow cross-over of connections(such as N-S and E-W) have been shown to be more powerful than the non-cross-over models.

2.1 Various Models

Since the introduction of the reconfigurable mesh [23], several models have appeared in the literature. Following variations of the models have been studied extensively and efficient algorithms have been developed for several problems:

- **PARBS** The most general and the most powerful is the **PARBS** model [32]. In this model no restriction is placed on the allowed connections among the 4 I/O ports in each PE. Thus, all 15 connection patterns are possible and algorithms for a variety of applications have been developed on this model [11, 14, 29, 32].

- **RMESH** This consists of two-dimensional mesh of size $n \times n$, with each PE connected to a broadcast bus [23]. This bus, like the mesh, is also constructed as an $n \times n$ grid, where PEs are located at the intersection of the grid lines. Further each bus link between adjacent PEs has a switch embedded in it, where the two PEs at either end of the link can control the switch. When all the switches are closed, all the n^2 PEs are connected together. If all the PEs disconnect the switches to the north, then we obtain row buses. Similarly column buses can be obtained. The connection patterns allowed in RMESH are shown in Figure 2.

- **MRN/LRN** The Reconfigurable Network (RN) [2] is a general model in which no restriction is placed on the bus segments that connect the PE or on the placement of the

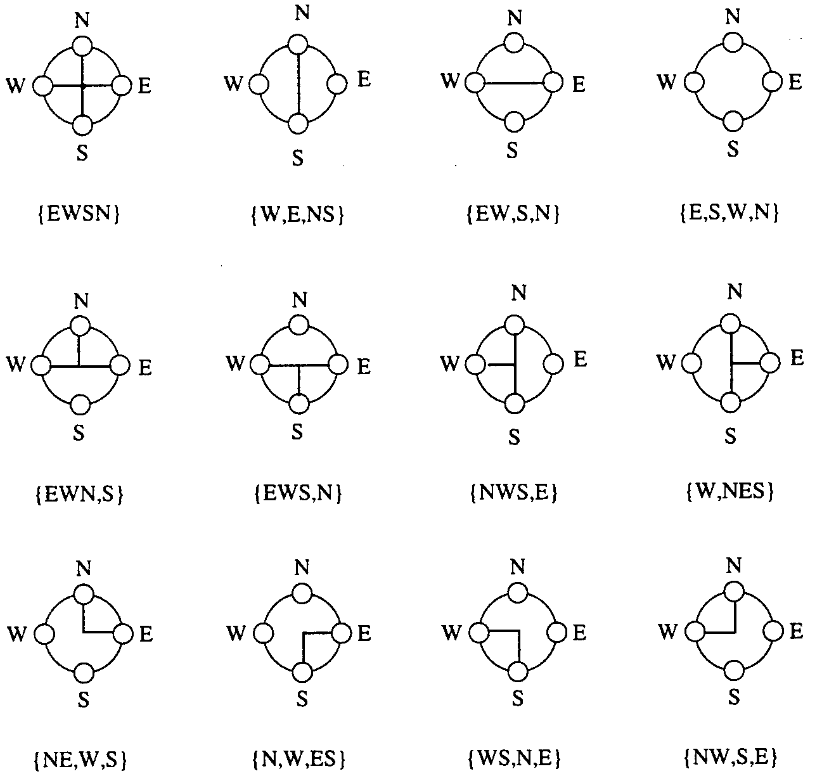


Figure 2: Connection patterns allowed in RMESH.

PEs. I.e. PEs may not lie at grid points and a bus segment may join an arbitrary pair of PEs. Variants of this model under the mesh restriction are the **MRN** and **LRN**. Connection patterns allowed in **MRN** are shown in Figure 3. In **LRN** a bus may consist of any connected path of edges. However, only *linear* buses are composed, so that a bus component is attached to at most one other bus component at each end.

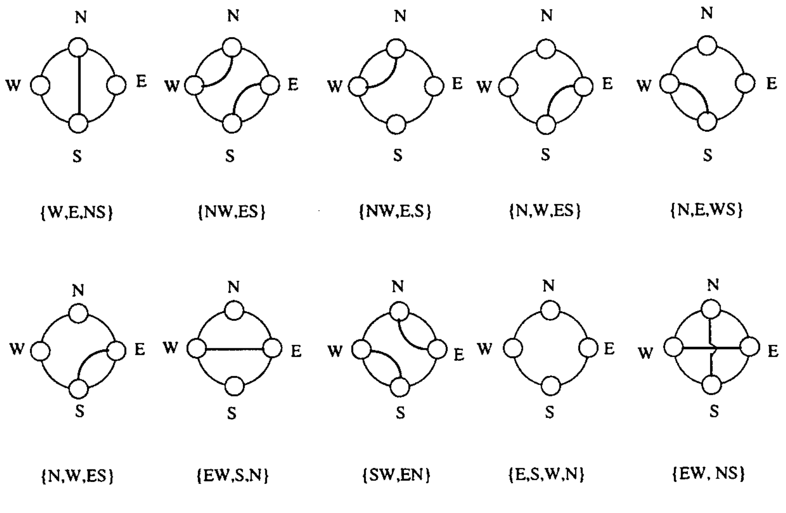


Figure 3: Connection patterns allowed in MRN.

- **Polymorphic Torus** A polymorphic torus architecture [18, 22] is identical to the PARBS architecture except that the rows and columns of the underlying mesh wrap around.

Jang *et. al.* proposed a **Bit Model** [13] of reconfigurable mesh which can simulate (asymptotically) most of the word based models of the reconfigurable mesh in the same amount of time using the same VLSI area. The basic PE in the Bit Model consists of a switch, local storage and a 1-bit ALU. The switch consists of six bit-level switches which can be closed or opened using local information within the PEs. The switch can realize any of the possible 15 connection patterns among its 4 I/O ports. The bus architecture is similar to the RMESH architecture and can carry $O(1)$ bits of data.

2.2 Related Models

- **REBSIS**

  In the reconfigurable buses with shift switching (REBSIS) [20] model, each word level switch consists of several bit level switches. In each connection pattern each of the bit level switches share a common connection pattern to control the bit level buses in a uniform way. Based on a control bit pattern the switch performs rotate-shift on the input bit pattern. It has been proved that that the REBSIS model is more powerful than several word models [20] but the Bit Model [13] of reconfigurable mesh has been shown to be able to simulate the REBSIS model using the same area.

- **RMBM** A more general reconfigurable network model called the reconfigurable multiple bus machine (RMBM) [31] was proposed to investigate effects of switch models on relative computational power of reconfigurable network models. This model separates the computational aspects from the connection configuration aspects. The RMBM model has processors, buses, fuse lines and sets of switches. Each processor has one write port and several read ports. The switches can be classiffied into connect switches, segment switches and fuse switches. The connect switches connect a particular port of a processor to one of the buses, the segment switches segment the bus and the fuse switches connect two or more buses together. There are restricted versions of this model which differ in the classes of switches which they allow.

3 Some Illustrative Algorithms

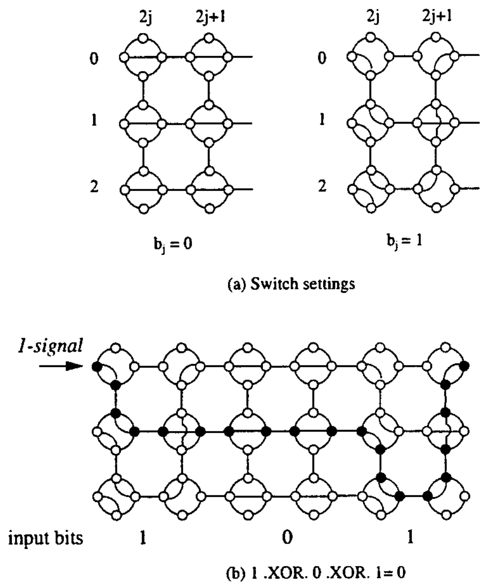
Lot of work has been done in exploiting the power of reconfigurable meshes. Algorithms for basic computations such as Or, And, Exor, Addition, Multiplication etc. have been designed and shown to be optimal on several variants of the reconfigurable mesh models. Using these basic data operations and additional non-trivial techniques of exploiting reconfiguration, algorithms for problems in image processing, computational geometry, graphs etc. have been designed. In this section some algorithms are described to illustrate the power of these architectures.

3.1 EXOR Computation

The EXOR of N bits of data can be computed on a reconfigurable mesh of size $2n \times 3$ in $\theta(1)$ time using the unit-time delay model and in $\theta(\log n)$ time using the log-time delay model [24]. The basic idea behind the algorithm is described here.

Based on a single input bit a 3×2 array of PEs set their local switch configurations to one of the two patterns as shown in Figure 4. If the input bit is 1 the top two rows cross-over and the 1-signal toggles to the other row and if the input bit is 0 then the 1-signal passes through the PEs, in the same row. When a 1-signal is applied to the top row input of the first processor of the system the EXOR of all the inputs appears at the last processor in the mesh. A 1-signal out of the top row indicates a result of 0 and a 1-signal out of the middle row indicates a result of 1.

An example EXOR computation of 3 input bits with 18 PEs is shown in Figure 4. The highlighted path shows the flow of the 1-signal from the left to the right. The result of the EXOR computation appears at the output after a constant delay in the unit-time delay model.



(a) Switch settings

(b) 1 .XOR. 0 .XOR. 1= 0

Figure 4: EXOR computation

3.2 Addition

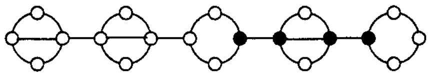
Addition of two n-bit numbers can be carried out in a similar way as EXOR computation. Each PE sets its switch pattern based on either a carry generate or a carry propagate configuration. If the two input bits a_i and b_i are different then the PE connects its West input to the East output port, which is a carry propagate configuration. If the input bits are the same then none of the switches are connected. The carry generate at a PE is implemented by the PE writing a 1 on its East port when both the bits a_i and b_i are 1.

An example addition of two 5-bit numbers is shown in Figure 5. The bits c_i indicate the intermediate carry bits and z_i are the result bits.

Using a similar idea and constructing a k-stage ripple carry adder it was shown that addition of n k-bit numbers ($1 \leq k \leq n$) can be performed in constant time using a $n \times nk$ bit model of reconfigurable mesh [11].

3.3 Parallel Prefix

Parallel Prefix is an important operation that can be used to sum values, broadcast data, solve problems in image processing and graph problems etc. [24]. Assume processor p_i, $0 \leq i \leq n-1$, initially contains the data element a_i. The parallel prefix problem requires p_i to compute $a_0 \otimes a_1 \otimes \ldots \otimes a_i$, where $\otimes$ is an associative operator such as addition (+).



i =	0	1	2	3	4
a_i	0	0	1	1	0
b_i	1	1	1	0	0
c_i	0	0	0	1	1
z_i	1	1	0	0	1

Figure 5: Addition of two 5-bit numbers.

The given n values are assumed to be distributed one per processor on a reconfigurable mesh of size n. The binary associative operation $\otimes$, is assumed to be a unit-time operation. First, parallel prefix is performed along the rows so that each processor knows the initial prefix of those values restricted to its row. Next, in the last column the parallel prefix is performed to determine row-wise prefix solutions. Finally, within each row, the prefix of previous of the previous rows is broadcast so that all the processors can update their entry appropriately. Parallel prefix can be computed in every row simultaneously in log $n^{1/2}$ iterations by appropriately setting switches, broadcasting and updating values at each iteration.

3.4 Sorting

There are several sorting algorithms on reconfigurable mesh models. We describe here the algorithm presented in [11]. Sorting of a sequence can be decomposed into sort of its subsequences and data movement between the sorted subsequences. The reconfigurable mesh algorithm uses a variation of Leighton's eight-stage column sort [17]. The stages are a combination of stages of $n^{1/4}$ sorters, each capable of sorting $n^{3/4}$ numbers, and $n^{1/4}$-shuffle network stages.

The input sequence of n numbers is assumed to be initially stored in the top row of the reconfigurable mesh. The sequence is partitioned into subsequences of $n^{3/4}$ numbers each. Sorting of a subsequence is done by computing the ranks of all the numbers and then storing each number according to its rank by using shuffle networks [11]. Sorting of $n^{3/4}$ numbers in constant time is carried out using a $n \times n^{3/4}$ reconfigurable mesh. In the first step, each of the $n^{3/4}$ PEs broadcast their numbers along each column of n PEs. Then the mesh is divided into $n^{3/4}$ submeshes each of size $n^{1/4} \times n^{3/4}$. The rank of number x_i is computed by submesh i using row broadcasts. The results of the comparisons made after this row broadcasts are added to give the rank of each number. The addition can be done in constant time as stated in Section 3.2. The $n^{1/4}$-shuffle stage can also be implemented in constant time using a sequence of broadcast operations.

3.5 Component Labeling

The problem is to label the connected components of a digitized image. Given an $n \times n$ image which is distributed as a pixel per processor onto the processors of a reconfigurable mesh of size $n \times n$, the connected components can be labeled in $\theta(\log n)$ time under the unit-time delay model [24].

In the first step each processor examines the pixels in each of its four neighbors and sets its four switches so that a connection is maintained only between neighboring black pixels. This $\theta(1)$ operation creates a subbus over each component. Given a linked list of processors overlaid by a reconfigurable subbus, the minimum(maximum) of the value stored in these processors can be computed in $O(\log n)$ iterations. Each iteration computes the local minima(maxima) and discards the other elements. Each iteration uses a constant number of broadcast steps and comparison operations, and hence the total running time is as stated above.

3.6 Summary of Results

We present a brief summary of algorithms on the reconfigurable mesh models. A comprehensive bibliography of results can be found in [26]. All results are with respect to the unit-time delay reconfigurable mesh model.

Problem	Mesh Size	Time
EXOR of n bits	$2n \times 3^*$	Constant
Prefix-And of n 1-bit numbers	$1 \times n^*$	Constant
Maximum(Minimum) of n log n-bit numbers	$n \times n$	Constant
Addition of n k-bit numbers, $1 \leq k \leq n$	$n \times nk^*$	Constant
Multiplication of two n-bit numbers	$n \times n^*$	Constant
Division of two n-bit numbers	$n \times n^*$	Constant
Histogram of an $n \times n$ image (h gray levels)	$n \times n$	$O(\min(\sqrt{h}+\log(\frac{n}{h}), n))$
Sort of n $O(\log n)$ bit numbers	$n \times n$	Constant
Convex Hull of n points	$n \times n$	Constant
Smallest enclosing rectangle of n points	$n \times n$	Constant
Triangulation of n planar points	$n^2 \times n$	Constant
All-pairs nearest neighbors of n points	$n \times n$	Constant
Two-set dominance counting of n points	$n \times n$	Constant
Connected components of an $n \times n$ image	$n \times n$	$O(\log n)$

* - *the bit model of reconfigurable mesh is used.*

4 Practical Considerations and Architectures

The choice of an architecture is strongly influenced by physical fabrication constraints. The reconfigurable mesh has nearly constant diameter and a dynamically reconfigurable bus system. It is very attractive in terms of implementation because of the two dimensional topology, low pin requirement and highly regular structure, which are well suited for today's VLSI and packaging technology.

There are several physical constraints that have to be overcome to successfully implement these architectures. Some of the features of the reconfigurable mesh models which should be examined in the context of hardware technology are:

- **Reconfiguration** The ability to set the local configurations of switches is one of the key aspects of reconfigurable meshes which is exploited in designing efficient algorithms. Assumptions made in the model impact the design since more flexibility in allowed switch patterns usually implies more area because of larger control memory etc. Most implementations support global control signals but implementing dynamic change of configuration based on local data is very expensive and is difficult to provide in general purpose implementations.

- **Signal Delay** There is potentially a large signal delay due to a long chain of shorted path, set up because of configuration. The signal propagation time grows linearly with the length of the wire carrying the signal. There are also unpredictable delays in VLSI because the wire capacitance is affected by the number of processors connected to the wire carrying the signal.

  Recent VLSI implementations have addressed these issues and suggest that the broadcast delay, although not constant, is very small. For example, only 16 machine cycles are required to broadcast on a 10^6 processor YUPPIE. GCN has shorter delays by adopting all-active and pre-charged circuit for local switches. ABACUS architecture propagates a signal through 18 PEs in a single 8ns clock cycle. Broadcast delay can be further reduced by using optical fibers for reconfigurable bus system and using electrically controlled directional coupler switches for connecting and disconnecting two fibers.

- **Clock Timing** In reconfigurable meshes variable length shorted path can be established based on the algorithm. If a fixed length clock is designed to accommodate the worst case shorted path, the clock for the system will be degraded. The constant time algorithms in the literature do not consider the clock implementation. Many clocking schemes are possible to accommodate the worst case path while not affecting the average clock performance. One such proposal is variable length clock that adjusts the length of clock to the length of the path. Global distribution of control signals also affects the clock signals. Detailed discussion of such issues is beyond the scope of this paper.

4.1 Architectures

We look at two variants of reconfigurable architectures. One class of architectures are based on the abstract models and try to approximate the features of the models. We describe the YUPPIE and the ABACUS architectures which are representative research prototypes. The other class consists of architectures which have evolved from commercial FPGA designs. We look at the features offered by two FPGAs, namely, XILINX 6200 and the NSC CLAy. Though these devices have not been designed for reconfigurable computing engines, they are a result of demand for fast reconfigurable components.

4.1.1 Polymorphic Torus Architecture - YUPPIE

The Polymorphic Torus [22] consists of a physical network (PNET) and a programmable internal network (INET) at each node of the PNET. The PNET is global while the INET is local. In a Polymorphic Torus consisting of $n \times n$ processors, the PNET is an $n \times n$ mesh with its boundary connected in either torus mode or spiral mode. Except for selection of torus or spiral mode, the PNET is a hard-wired, fixed, non-programmable network. In contrast INET is totally programmable. Each of the four ports of the INET can be connected to any port.

The VLSI implementation of the 2D Polymorphic Torus is called YUPPIE (Yorktown Ultra Parallel Polymorphic Image Engine). YUPPIE follows a regular SIMD model of computation with a central controller (CC) generating a stream of instructions. A processor array (PA), made up of many bit-serial PEs connected by a Polymorphic Torus receives the instruction stream from the CC and executes it. PEs can be selectively disabled based on local condition, but all enabled PEs carry out the same operations on their own data. Data Memory (DM) consists of on-chip 256 fast-access one-bit registers for each PE, termed local data memory (LDM), and off-chip external data memory (EDM). The YUPPIE chip consists of 16 nodes arranged as a 4×4 mesh. A programmable length clock generator (PLCG) generates the timing signals for YUPPIE, since it needs to be driven by a variable length clock.

The YUPPIE PE has a 1-bit ALU, carry register (CY), data registers (A, TR) and two control registers (EN, CCR). All registers are 1-bit wide and one of the data registers functions as the accumulator. The ALU can carry out basic addition and boolean operations with operands from physical links, local or external memory and/or the data registers. The INET switching is established by choosing one of two patterns broadcast by the CC. This choice is made depending on the data in one of the control registers, namely, CCR.

Implementation using a 2 micron CMOS technology with two metal layers has shown a less than 20% overhead for the programmable interconnect and the ability to propagate the signal through 16 PEs in a single clock cycle.

4.1.2 ABACUS

ABACUS [5] is a distributed bit-parallel (DBP) architecture based on the reconfigurable mesh. The ABACUS processing element (PE) contains 64 bits of dual-ported memory in two banks and two 3-input ALUs, each of which takes two inputs from its memory bank and one input from the other bank. Part of the memory bank is utilized as control registers for enabling PE and network operation.

At each PE, the network is composed of a wired-OR bus and four isolating switches. When all switches are open, two network control bits specify which of the four nearest neighbors is connected to the PE input. Each PE can also close the switch in the read direction, the other three switches remain open unless closed by a neighboring PE. Connected processors form a multiple-writer wired-OR bus. VLSI implementation of the network consists of a precharged bus which is pulled down by any PE writing a one. Additional delays are reduced by using local accelerator circuits.

There are additional circuits for reading and writing to on-chip distributed memory and interface circuitry to external data memory. A VLSI implementation in 0.8/1 micron technology is expected to sustain a 8 ns cycle time. Simulations show that in worst case the signal can propagate through 18 PEs in a single clock cycle. A single ABACUS IC is expected to deliver 1-5 giga-operations per second (GOPS) on 16-bit arithmetic operations. This is approximately 20 to 100 times faster than microprocessors implemented in comparable VLSI technology.

4.1.3 XILINX XC6200

The XC6200 FPGA [41] architecture from Xilinx is the first SRAM based FPGA architecture designed for implementing reconfigurable coprocessors. The XC6200 architecture features a fine-grained cell structure, abundant routing, built-in processor interface and supports fast partial reconfiguration.

The programmable logic of an XC6200 consists of large array of reconfigurable logic cells each of which contains both programmable logic and routing resources. Each cell contains a flip-flop and combinatorial logic capable of implementing any two-input function or any type of 2-to-1 multiplexer. Cells are arranged in 4-by-4 blocks and 16-by-16 tiles. The interconnection network consists of a hierarchy of programmable routing wires. Each cell can be used for logic or memory functions. When cells are configured as memory, each cell provides two bytes of ROM or RAM memory which can be accessed externally or internally.

The most important feature in the new XC6200 device is the FastMap interface, designed to connect directly to an external processor's system bus. The FastMap interface places the whole FPGA into the processors address space. The processor can read and write the logic and the configuration memory by using normal load and store. This parallel interface allows the entire configuration memory to be programmed in under 100 micro-seconds.

A random access feature allows arbitrary areas of the FPGA memory to be changed. This provides a fast partial reconfiguration capability. This partial reconfiguration can be performed without disturbing circuits running in other parts of the device. This facilitates sharing of hardware space by swapping in and out designs at runtime. A reconfigurable hardware platform based on the XC6200 architecture has been designed and is being offered as a commercial product by Virtual Computer Corporation [40].

4.1.4 NSC CLAy

The National Semiconductor CLAy [39] architecture is an SRAM based Configurable Logic Array. CLAy was designed to support real-time algorithm and logic sharing by using dynamic partial reconfiguration.

The logic cell layout is similar to existing FPGA devices, with a flip-flop and 5-input lookup tables. The interconnection network is made up of nearest neighbor connections, local and express bus wires. The full device can be configured in 640 micro-seconds. Larger designs are supported by an integrated Field Configurable Multi-Chip Module (FCMCM) which consists of a 2×2 array of CLAy devices.

CLAy supports partial reconfiguration by which a single cell's functionality can be changed. This is much faster than programming the complete device and reconfiguration time is the order of 1 micro-second. This partial reconfiguration can be done without functional interruption of the remaining parts of the device. These features of the CLAy devices have been exploited in designing novel applications [35].

5 Conclusion

Configurable computing holds lot of promise for the future. To realize this potential we need a variety of system architectures, algorithmic techniques and software tools. We have discussed the abstract models of reconfigurable architectures and algorithms using these models. The technology constraints in realizing these architectures have been examined and prototype implementations which try to overcome these constraints have been described.

Currently, the designs of configurable computing systems are based on fine-grained commercial devices like Field Programmable Gate Arrays. Since FPGAs were primarily designed for logic emulation there has not been much progress in trying to achieve fast reconfiguration times. Recent SRAM based devices have started addressing this issue and we examine some of these devices. We believe that the wealth of ideas in the abstract world can be leveraged in designing systems and algorithms which exploit the available reconfiguration potential. The MAARC [38] project at USC explores these opportunities.

6 Acknowledgment

Over the years, several students at USC have conducted research in reconfigurable architectures and algorithms. We would like to acknowledge the contributions of Ju-wook Jang, Heonchul Park and Dionisis Reisis. This paper summarizes many of their contributions in reconfigurable meshes and algorithms.

References

[1] R. Amerson, R. J. Carter, W. B. Culbertson, P. Kuekes and G. Snider, "Teramac - Configurable Custom Computing", *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 32-38, April 1995.

[2] Y. Ben-Asher, D. Peleg, R. Ramaswami and A. Schuster, "The Power of Reconfiguration", *Journal of Parallel and Distributed Computing*, V. 13, No. 2, pp. 139-153, 1991.

[3] P. Bertin, D. Roncin and J. Vuillemin, "Programmable active memories: a performance assessment", In F. Meyer auf der Heide, B. Monien, and A. L. Rosenberg, editors, *Parallel Architectures and their efficient use*, pp. 119-130, Lecture notes in Computer Science, Springer-Verlag, October 1992.

[4] R. Bittner and P. Athanas, "Wormhole Run-time Reconfiguration", *ACM International Symposium on Field Programmable Gate Arrays*, pp. 79-85, February 1997.

[5] M. Bolotski et. al., "Abacus: A High-Performance Architecture for Vision", *International Conference on Pattern Recognition*, 1994.

[6] D. A. Buell, J. M. Arnold and W. J. Kleinfelder, "Splash 2: FPGAs in a Custom Computing Machine", IEEE Computer Society Press, 1996.

[7] D. Chen and J. Rabaey, "A Reconfigurable Multiprocessor IC for Rapid Prototyping of Real Time Data Paths", *ISSCC*, Feb. 1992.

[8] A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century", *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1994.

[9] C. Ebeling, D. C. Cronquist, P. Franklin. "RaPiD - Reconfigurable Pipelined Datapath", *6th International Workshop on Field-Programmable Logic and Applications*, 1996.

[10] T. J. Fountain and V. Goetcherian, "CLIP4 Parallel Processing System", *Proc. of the Institute of Electrical Engineers*, Vol. 127E, pp. 219-224.

[11] J. Jang and V. K. Prasanna, "An Optimal Sorting Algorithm on Reconfigurable Mesh," *Journal of Parallel and Distributed Computing*, pp. 31-41, Vol. 25, 1995.

[12] J. Jang, M. Nigam, V. K. Prasanna and S. Sahni, "Constant Time Algorithms for Computational Geometry on the Reconfigurable Mesh," *IEEE Transactions on Parallel and Distributed Systems*, pp. 1-12, Vol. 8, January 1997.

[13] J. Jang, H. Park and V. K. Prasanna, "A Bit Model of Reconfigurable Mesh", *Proc. of Reconfigurable Architectures Workshop: International Parallel Processing Symposium*, April 1994.

[14] J. Jang, H. Park and V. K. Prasanna, "A Fast Algorithm for Computing the Histogram on Reconfigurable Mesh," *Proc. of Frontiers of Massively Parallel Computation*, pp. 244-251, October 1992.

[15] J. Jang, H. Park and V. K. Prasanna, "An Optimal Multiplication Algorithm on Reconfigurable Mesh," *Proc. of Symposium on Parallel and Distributed Processing*, pp. 384-391, December 1992.

[16] J. Jenq and S. Sahni "Reconfigurable Mesh Algorithms For Image Shrinking, Expanding, Clustering, and Template Matching", *Proc. of Fifth International Parallel Processing Symposium*, pp. 208-215, April 1991.

[17] F. T. Leighton, "Tight bounds on complexity of parallel sorting", *IEEE Transactions on Computers*, Vol. 34, pp. 344-354, 1985.

[18] H. Li and M. Maresca, "Polymorphic-Torus Network," *IEEE Trans. on Computers*, Vol. 38, No. 9, pp. 1345-1351, Sept. 1989.

[19] H. Li and Q. F. Stout, "Reconfigurable SIMD Massively Parallel Computers", *Proceedings of the IEEE*, Vol. 79, No. 4, pp. 429-443, April 1991.

[20] R. Lin and S. Olariu, "Reconfigurable Buses with Shift Switching: Concepts and Applications", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 1, pp. 93-102, January 1995.

[21] M. Maresca, "Polymorphic Processor Arrays", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 5, pp. 490-506, May 1993.

[22] M. Maresca and H. Li, "Connection Autonomy in SIMD Computers: A VLSI Implementation", *Journal of Parallel and Distributed Computing*, Vol. 7, pp 302-320, 1989.

[23] R. Miller, V. K. Prasanna Kumar, D. I. Reisis and Q. F. Stout, "Meshes with Reconfigurable Buses", *Proc. 5th MIT Conference On Advanced Research in VLSI*, pp. 163-178, March 1988.

[24] R. Miller, V. K. Prasanna Kumar, D. I. Reisis and Q. F. Stout, "Parallel Computations on Reconfigurable Meshes", *IEEE Trans. on Computers*, July 1993.

[25] E. Mirsky and A. Dehon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources", *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1996.

[26] K. Nakano, "A Bibliography of Published Papers on Dynamically Reconfigurable Architectures, *Parallel Processing Letters, Special Issue on Dynamically Reconfigurable Architectures*, 1995.

[27] K. Nakano, T. Msuzawa, N. Tokura, "A Fast Sorting Algorithm on a Reconfigurable Array", *The Institute of Electronics, Information and Communication Engineers*(Japanese), COMP 90-69, December 1990.

[28] M. Nigam and S. Sahni, "Sorting n numbers on $n \times n$ Reconfigurable Meshes with Buses", International Parallel Processing Symposium, pp. 174-181, April 1993.

[29] H. Park, V. K. Prasanna and J. Jang, "Fast Arithmetic on Reconfigurable Meshes", International Conference on Parallel Processing, August 1993.

[30] J. Rothstein, "Bus automata, brains, and mental models", *IEEE Transactions on Systems Man Cybernetics*, Vol. 18, pp. 522-531, 1988.

[31] R. K. Thiruchelvan, J. L. Trahan, and R. Vaidyanathan, "On the Power of Segmenting and Fusing Buses", *Proc. of International Parallel Processing Symposium*, pp. 79-83, April 1993.

[32] B. F. Wang and G.H. Chen, "Constant Time Algorithms for the Transitive Closure Problem and Some Related Graph Problems with Reconfigurable Bus Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 4, pp. 500-507, 1991.

[33] C. C. Weems, "The Image Understanding Architecture," *International Journal of Computer Vision*, V. 2, pp. 251-282, 1989.

[34] C. C. Weems and J. H. Burrill, "The Image Understanding Architecture and its Programming Environment," *Parallel Architectures and Algorithms for Image Understanding*, V. K. Prasanna Kumar(Ed.), Academic Press, 1991.

[35] M. J. Wirthlin and B. L. Hutchings, "Improving Functional Density Through Run-Time Constant Propagation", *ACM International Symposium on Field Programmable Gate Arrays*, pp. 86-92, February 1997.

[36] BRASS Homepage, *http://www.cs.berkeley.edu/projects/brass*.

[37] CMU Cached Virtual Hardware Homepage, *http://www.ece.cmu.edu/afs/ece/usr/herman/www/CVH/intro/intro.html*.

[38] MAARC Homepage, *http://maarc.usc.edu*.

[39] National Semiconductor, *Configurable Logic Array (CLAy) Data Sheet*, December 1993.

[40] Virtual Computer Corporation, "Reconfigurable Computing Products", *http://www.vcc.com/products/pci6200.html*.

[41] Xilinx, *XC6200 Field Programmable Gate Arrays*, 1996.

Computation Models for Reconfigurable Machines [1]

Reetinder P.S. Sidhu, Kiran Bondalapati, Seonil Choi, Viktor K. Prasanna

Department of Electrical Engineering Systems, EEB-244
University of Southern California
Los Angeles, CA 90089-2562

Contact Author
Viktor K. Prasanna
Email: prasanna@usc.edu
Tel: +1-213-740-4483
http://www.usc.edu/dept/prasanna/home.html

Abstract:

Currently, reconfigurable computing solutions are developed by writing High level Description Language (HDL) code and compiling it onto hardware. Though this approach is suitable for static reconfigurable devices, tools using this approach do not analyze the runtime behavior of the application. Hence designing tools which exploit dynamic reconfigurability is not an easy task. This paper presents a new approach to developing dynamically reconfigurable computing solutions. Computing models are developed which bridge the semantic gap between the algorithm and the actual hardware. A General Reconfigurable Computing Model (GRECOM) is used to capture the ability to change both the interconnections and the logic at runtime based on intermediate results. Two specific instances of GRECOM, the Reconfigurable Mesh and the FPGA Model are derived and applications are demonstrated using these models.

[1] This work was supported by DARPA under contract DABT63-96-C-0049 monitored by Fort Hauchuca.

1 Introduction

The advent of static RAM based Field Programmable Gate Arrays (FPGAs) has given rise to new opportunities in the reconfigurable computing area. An FPGA consists of an array of combinational logic blocks each with a flip-flop. The logic blocks are interconnected using a hierarchy of buses. The logic blocks at the periphery of the device also perform the I/O operations. The functions computed, the interconnection network and I/O block can be configured using external data. FPGAs also permit unlimited reconfiguration. These versatile devices have been used to build processors and coprocessors whose internal architecture as well as interconnections can be configured to match the needs of a given application. For a detailed architectural survey of FPGAs, see [4, 16].

FPGAs have been mostly used for rapid prototyping and emulation. Some of the designs based on reconfigurable logic have shown an order of magnitude price/performance advantage. But the prohibitive cost in using these devices as configurable computing engines has been the time for reconfiguration. Configuration of an FPGA is carried out by downloading the configuration information from a host processor often using bit-serial lines. The time required for this step is usually of the order of msec. An additional problem with existing FPGAs is that the complete device has to be reconfigured every time even if the new configuration is almost similar to the existing one.

Current and future generation devices such as CLAy, XC6200, DPGA etc. ameliorate the above cost by providing partial and dynamic reconfigurability [14, 21]. It is possible to modify the configuration of a part of the device while the configuration of the remaining part is retained. Some devices permit this partial reconfiguration even while other logic blocks are performing computations. Devices in which multiple contexts of the configuration of a logic block can be stored in the logic block and the context switched dynamically have also been proposed [5]. To distinguish the FPGAs which do not provide partial and dynamic reconfigurability we shall hereafter refer to them as static configurable devices.

Traditional approach to utilizing static configurable devices has been to use automated synthesis tools such as FPGA Express, OrCAD Express, Leonardo, Warp2, PL-Link etc. Designs are first specified using a hardware description language such as VHDL or Verilog, at the register transfer or gate level (Figure 1). This design is usually analyzed and verified using technology independent tools and is then submitted to logic synthesis for logic minimization and technology dependent mapping. Finally, physical design tools are used for placement and routing. To avoid designing solutions from scratch every time, components from a Library of Parameterized Modules (LPM) are used in the design. Some work is also being performed in mapping behavioral descriptions in high-level languages (C, C++, Occam etc.) to hardware.[8, 9].

This approach of *logic synthesis* as opposed to *algorithm synthesis* allows the user to specify the design using a behavioral model. But this abstraction is achieved at the cost of significant performance. To obtain better timing and layout characteristics, users fine tune the solution by editing the physical design [6]. Also, reconfiguration is performed by analyzing the application at compile time and the synthesis tools do not support analyzing the runtime behavior to take advantage of dynamic reconfigurability. By collapsing the abstraction layers we can expect to extract significant performance. Also, potentially this can lead to tools which exploit runtime

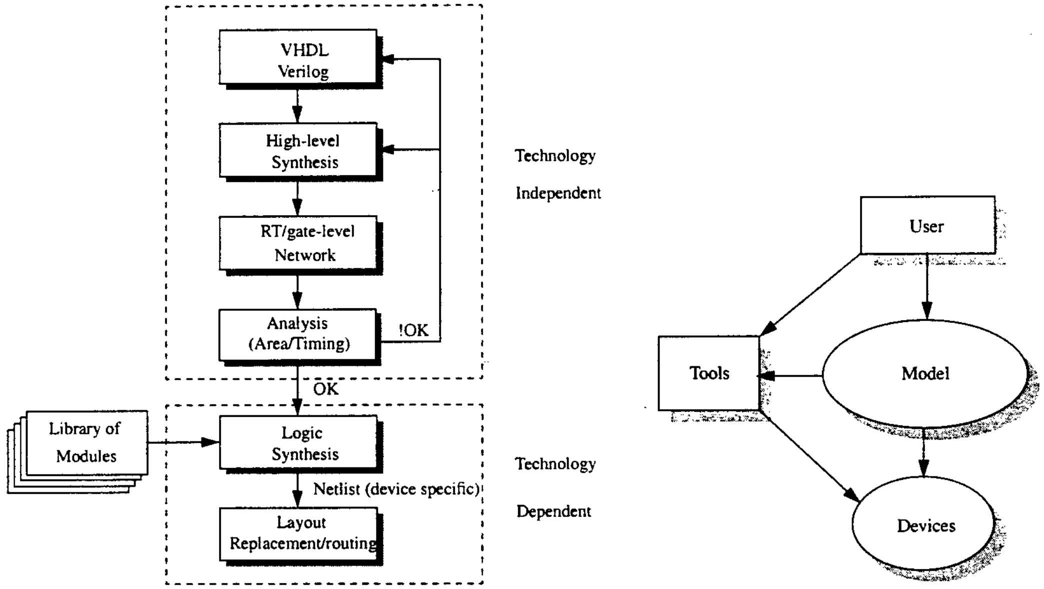


Figure 1: Traditional Design Synthesis Approach and the Modeling Approach

reconfiguration.

In our approach, we model the hardware features of reconfigurable devices which allows the user to think of dynamically reconfigurable computing at a high level during application development phase (Figure 1). Bridging the semantic gap between the algorithm and the hardware by using such a model allows the user to develop reconfigurable computing solutions in a natural manner. With our model the *algorithm synthesis* approach can be used and the algorithm can be specified as a computation performed on the input data. The computations performed by each element can be chosen from a set of operations supported by the model.

We first abstract the reconfigurable computing devices using General Reconfigurable Computing Model. We derive the Reconfigurable Mesh model and an FPGA based model and illustrate how algorithms can be mapped onto the latter model by using a simple example. This manuscript is a preliminary summary of ongoing experiment under the Models, Architecture and Algorithms for Reconfigurable Computing (MAARC) project.

2 Reconfigurable Machine Models

In this section we abstract various features of configurable computing devices which differentiate between various models. These parameters form the basis of the general model. By varying the values for these parameters we derive two specific variants of the general model.

2.1 A General Model

The General Reconfigurable Computer Model (GRECOM) covers most reconfigurable devices. It consists of a number of Processing Elements (PEs) linked together by an Interconnection

Network (IN) as in Figure 2. The operations performed by the PEs as well as the topology of the IN can be configured. PEs operate synchronously and have a fixed amount of local storage. The configuration information specifies the operation performed by each PE and the IN topology.

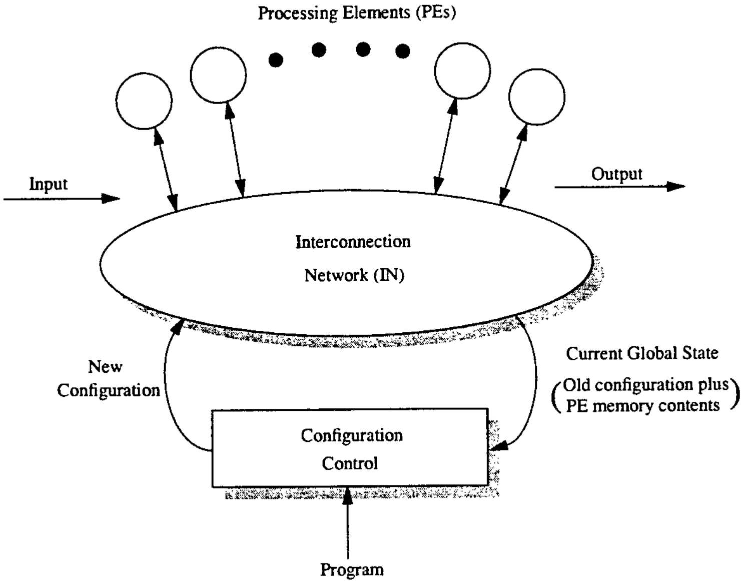


Figure 2: The General Reconfigurable Computer Model.

Since the PEs operate synchronously, the machine has a well defined global state. The global state is completely specified by the current configuration and contents of the local storage of all the PEs. The machine can also be reconfigured either partially or completely. In partial reconfiguration the existing configuration of some part of the device is retained while the configuration of the remaining part can be changed. This is modeled in GRECOM by the configuration control block. As shown in Figure 2, the new configuration is generated based on the program as well as the current state. The ability to use information about the current state to reconfigure is analogous to the conditional branch instructions of a traditional microprocessor.

The four basic parameters of the model are the PE granularity, IN topology, the method of reconfiguration and the reconfiguration time.

- **PE granularity**: This parameter is a measure of how coarse or fine grained the computations performed in the PE are. As an example, we can distinguish between word oriented models and bit oriented models.
- **IN topology**: The Interconnection Network is an important aspect of the model since the communication to computation ratio is very high in most configurable computing applications. This parameter models the organization of the connections between the PEs.
- **Method of reconfiguration**: The computation performed by the PE and/or the interconnection network can be reconfigured. This reconfiguration can be achieved either

by explicit transfer of control information from an external control unit or by using intermediate computational results. This parameter determines how the algorithm can be implemented on the model. When reconfiguration can be achieved only by using external control information, a dedicated unit such as a processor is needed to do this control information transfer.

- **Reconfiguration Time**: After a new configuration is specified by using the above method of reconfiguration, the delay before the next computation can start is a critical parameter. Reconfiguration Time determines how much the algorithm can make use of the reconfiguration potential. When Reconfiguration Time is very high usually the applications do a single reconfiguration at the beginning which does not change throughout the computation. If the model supports partial reconfiguration this time can be overlapped with computation to design efficient solutions.

By varying these parameters, we derive the Reconfigurable Mesh model and the FPGA model in the following sections.

2.2 Reconfigurable Mesh Model

The reconfigurable mesh model [13, 18] has its origins in SIMD machines. The reconfigurable mesh is a two dimensional variant of the multi-dimensional reconfigurable bus architecture. The model consists of an array of processing elements embedded in a very flexible interconnection network (Figure 3). The processing elements locally decide upon the IN configuration and operation to be performed. A number of efficient algorithms for diverse areas have been developed for this model of computation [10, 15, 19, 20]. Machines have been built based on the reconfigurable mesh models. These include the CLIP series [7], YUPPIE [12] based on the Polymorphic Torus Network model, Gated Connection Network [17], etc.

- **Processing Element**: Each PE can perform standard arithmetic and logic operations on one bit operands in unit time. Each PE has four ports, one each for the four connections to neighboring PEs, and fixed amount of local storage (Figure 3).

- **Interconnection Network**: The IN is a 2D mesh with each PE linked to its four nearest neighbors as in Figure 3. These links can be connected together to form buses of arbitrary shapes. This is achieved by using the link switches in each PE. By using the local switches a bus can be configured into several distinct components of varying shapes such as rows, columns, diagonals, zig-zag and staircase. All PEs connected to the same bus can read from or write to the bus in unit time.

- **Method of Reconfiguration**: Reconfiguration can be achieved by either using control signals broadcast globally or by using local state. Local state is defined by the four port configurations and the local storage.

- **Reconfiguration Time**: The Reconfiguration Time of the reconfigurable mesh model is of the order of the computation time. This allows for a potential reconfiguration every cycle.

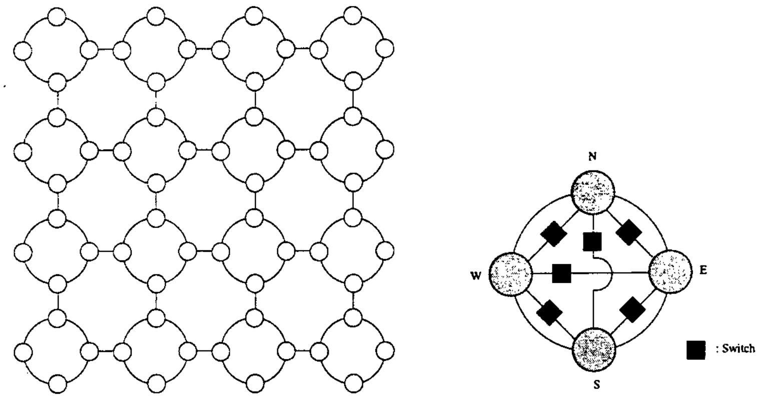


Figure 3: The reconfigurable mesh model and the PE architecture

This ability of the reconfigurable mesh model to allow single cycle reconfiguration, arbitrary IN topologies, and use of local data to determine configuration has led to the development of several efficient algorithms. Examples include sorting N numbers, multiplying two N bit numbers and $N \times N$ matrix multiplication all in $O(1)$ time on an $N \times N$ reconfigurable mesh [15, 20]. Efficient algorithms for this model of computation have been found for problems such as maze routing, Voronoi diagrams, histogram computation [10, 19].

2.3 FPGA Model

The FPGA model consists of a $N \times N$ array of processing elements embedded in an Interconnection Network. Most FPGA architectures have a similar structure [1, 2, 3, 14, 21]. Connections to I/O pins are provided along the perimeter of the array. The operations performed by the PEs as well as the IN topology can be configured.

- **Processing Element**: We model a PE as a configurable combinational logic block with an optional flip-flop at the single output. The combinational logic block can compute a function of X_{in} number of inputs producing a one bit result. In current devices X_{in} usually varies from 2 to 4. Almost all FPGAs employ this basic arrangement in their PE designs. The configurable logic is usually implemented using LUTs or the sea-of-gates approach. LUT based PEs can also be configured as a 1 bit word RAM; that is, the look-up tables can also be written, using the inputs as the memory address. The input as well as the outputs can be connected to various wires of the IN. These connections are controlled by multiplexers whose control signals are specified as a part of the configuration information. The Figure 4 shows the proposed PE model.
  - T_{comb}: It is the time taken for the signal to propagate from the input of the input multiplexer to the output of the output multiplexer.
  - T_{seq}: It is time taken for the signal to propagate from the output of combinational logic to the output of the output multiplexer.

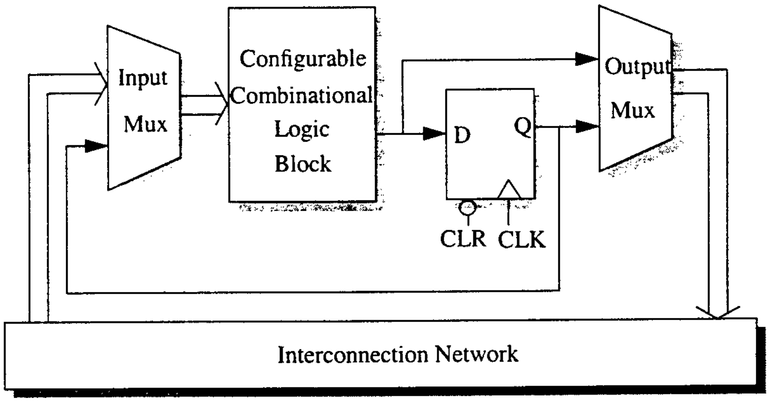


Figure 4: FPGA model.

- **Interconnection Network**: Typically, the IN topology in FPGA architectures is hierarchical. The IN consists of unit length wires connecting each PE to its four neighbors and wires of length N along each row and column. Most architectures also have wires of shorter length forming intermediate levels of the IN hierarchy. The connectivity offered by the IN is configured using switches situated at the end of each wire which can connect different wires, at possibly different levels in the hierarchy, and the PE input and output multiplexers.

  The precise propagation delay of a signal traveling along a path of the IN depends on various factors. The main factor is the number of switches and wires in the signal path. Also, as the device gets routed, the resources to make these connections may get used up, so depending on the design complexity and the architecture the delay can vary. Neighbor connections are almost always faster but also can be influenced by existing routing, i.e. logic block A is to the left of logic block B, and the route goes from left side of A to right side of B. The delay could be 4 times the delay from the right side of A to the left side of B. Fanout of the signals also affects the delays. Wires of long length are usually far less sensitive to multiple loads compared to the neighbor connections. To simplify the model and to retain it as a general model, we neglect these low level factors affecting the delay. For the purposes of algorithm mapping, these effects play a secondary role.

  In our model the IN has a two level hierarchy having two different delays, T_{local} and T_{global}.

  - T_{local}: The time taken for a signal to propagate from the output of one logic block to the input of the neighboring block.
  - T_{global}: The time taken for a signal to propagate from the output of one logic block to the input of the block which is at a distance of at most N along the same row or column. In current systems the ratio of T_{global} to T_{local} varies from 5 to 10.

- **Method of Reconfiguration**: The reconfiguration information stored in SRAM cells of the FPGA determines its functionality. This information specifies the logic performed by the PEs, the connectivity of the IN, and the operation of the I/O pins. Among current

FPGA architectures, the amount of information needed to configure a logic block is a few bytes. Additional bits are required for IN configuration. Thus, the total amount of information required to configure an entire array is a few hundred kilobits. In our FPGA model the configuration is only supported by external transfer of control information.

- **Reconfiguration Time**: We define the following parameters for the reconfiguration times in our model -
  - T_{cr}: The time required to reconfigure the complete device.
  - T_{sr}: The time required for reconfiguring a single logic block.
  - T_{lr}: The time required to reconfigure one line i.e. a row or a column of logic blocks.

  In current generation FPGAs the time required for complete reconfiguration is in the range of a few μsec to a few msec.

  Several FPGA architectures offer partial reconfiguration capability; that is, some part of the device can be reconfigured while the rest of keeps is performing computations based on prior configuration. The time required to reconfigure a single PE is in the range of 100nsec to 10μsec. Some vendors have recently announced devices which can be memory-mapped into the control processor address space and hence can efficiently support single block reconfiguration and reconfiguration of a complete row or column.

3 An Illustrative Application

This section illustrates the mapping of an algorithm onto the FPGA model and its analysis using the model parameters. The algorithm exploits partial reconfigurability to dynamically reconfigure parts of the FPGA at various stages of computation.

The algorithm that we consider is counting the number of 1s in an $n \times n$ bit image of 0s and 1s. This is a key operation required in image processing algorithms for median row [11] and histogram determination [19]. The operation is carried out in two phases. We first outline the required FPGA configuration and then describe the two phases.

3.1 FPGA configuration and algorithm overview

As described in Section 2.3, a PE can be configured as 1-bit word RAM. To store the $n \times n$ bits, $\lceil \frac{n}{k} \rceil$ rows of PEs in n columns are configured as RAM, k being the number of bits a single PE can store . In the next two rows are PEs are configured as a serial half adder (see Figure 5). A 2:1 multiplexer is used to select one of the two adder outputs that is to be written back to the RAM (see Figure 6). Based on the PE model described in section 2.3, the adder and multiplexer can be mapped into 4 PEs - two for the adder and flip-flops,one for the OR gate and one for the 2:1 multiplexer. Thus the RAM and adders can be configured using $n \times (\lceil \frac{n}{k} \rceil + 4)$ PEs.

Computation proceeds by selecting appropriate bits from of the RAM, adding them and storing back the result. The address and control generator is responsible for generating the appropriate control signals each clock cycle. This can be done using a synchronous counter and some additional logic, which can be implemented using a small number of PEs. These signals are applied to each 1-bit RAM column and adder, since they all operate in parallel.

The algorithm consists of two phases. In the first phase, the number of 1s in each column of the array are added. Next, the column sums are added to determine the total number of 1s. The following section describes the two phases.

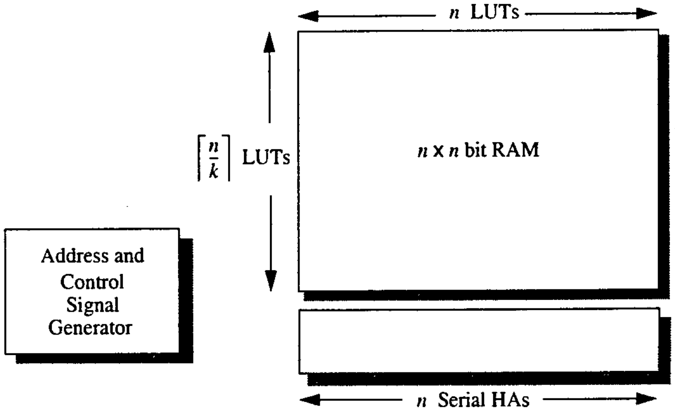

n LUTs
$\lceil \frac{n}{k} \rceil$ LUTs
n x n bit RAM
Address and Control Signal Generator
n Serial HAs

Figure 5: FPGA configuration.

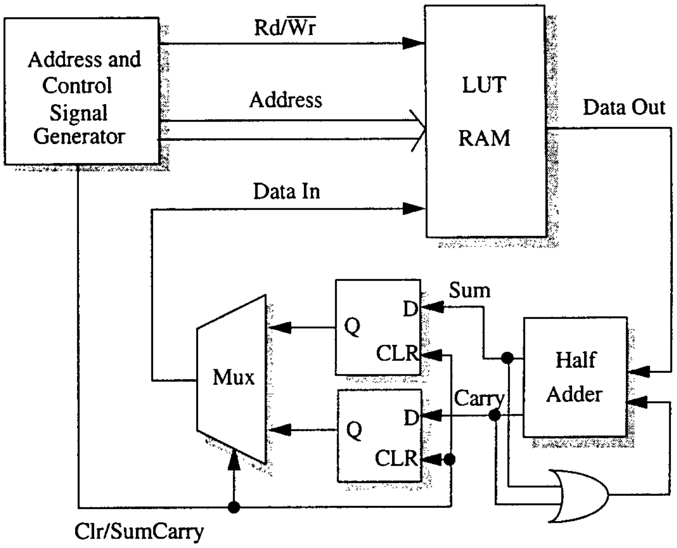


Figure 6: Configuration for a single RAM column.

3.2 Exploiting Partial and Dynamic Reconfiguration

Each serial adder is used to compute the sum of the bits stored in the column above it. The computation for a single column proceeds as follows. Initially, the column stores n 1-bit numbers. In the first iteration, bits i and $i+1$, $i = 0...n-1$ are added to obtain $\frac{n}{2}$ 2-bit numbers which are stored in the same positions. In general, in iteration j, $\frac{n}{2^j}$ j-bit numbers are added in

Clock cycle	Address	Rd/$\overline{\text{Wr}}$	Clear/CarrySum	Action
0	0	1	0	Read bit
1	1	1	0	Read bit
2	0	0	0	Write sum
3	1	0	1	Write carry
4	2	1	0	Read bit
5	3	1	0	Read bit
6	2	0	0	Write sum
7	3	0	1	Write carry
				Reconfigure
8	0	1	0	Read bit
9	2	1	0	Read bit
10	0	0	0	Write sum
11	1	1	0	Read bit
12	3	1	0	Read bit
13	1	0	0	Write sum
14	2	0	1	Write carry

Table 1: Address and Control signals for column addition.

pairs, $j = 0 \ldots \log n$. Thus, $\sum_{j=1}^{\log n} \frac{nj}{2^j} \approx O(n)$ bit addition operations are required to add the bits in one column. The above steps occur in parallel for all n columns resulting in the computation of the column sums.

Table 1 shows the addresses and control signals required for the first two iterations (assuming $n = 4$). The address and control signal generator would need to be reconfigured every iteration. This would involve reconfiguring only a few PEs; the PEs configured as RAM and adders would not be affected. Thus the dynamic reconfiguration required each iteration can be done quickly using the partial reconfiguration capability of FPGAs.

The next phase is the summation of the n column sums of $(\log n + 1)$ bits to obtain the $2 \log n + 1$ bit result of the number of 1s in the $n \times n$ array. At the beginning of this phase the adders are reconfigured as full adders so that they can receive one bit each from two separate RAM columns. Assuming the number of inputs to a PE, $X_{in} \geq 3$, no extra PEs would be required for full adder operation.

The summation is performed in $\log n$ iterations as follows. In the first iteration, adder $2i$ will add the two numbers ($\log n$ bits each) in columns $2i$ and $2i + 1$, $i = 0 \ldots n - 1$. In iteration j, adder $2^j i$ will add the two numbers ($\log n + j - 1$ bits each) $2^j i$ and $2^j(i + 1)$, $j = 1 \ldots \log n$, $i = 0 \ldots n - 1$ (the i additions in iteration j occur in parallel). Thus the total number of bit addition steps required is $\sum_{j=1}^{\log n} (\log n + j) = \frac{3}{2} \log^2 n + \frac{1}{2} \log n$.

Note that for each iteration, at least one of the memory columns feeding a serial adder is different from the one in the previous iteration. Thus the reconfiguration of the connections between RAM columns and adders would be required before each iteration of this phase of computation. $\frac{n}{2^j}$ connections need to be reconfigured in iteration j. Thus in all, $\sum_{j=1}^{\log n} \frac{n}{2^j} =$

$2n-1$ connection reconfigurations would be required. The partial reconfigurability feature can be used to make these selective changes to the IN at the beginning of every iteration.

The total number of steps (and hence the number of clock cycles) required in the two phases have been derived. We can use the FPGA model parameters to obtain an estimate of the clock period. The following operations occur each clock cycle. Address and control signals are generated (T_{seq}), the signals are propagated to the RAM columns (T_{global}), RAM data is read (T_{comb}), the data is propagated to the adders (T_{local}), sum and carry are computed and latched in flip-flops (T_{seq}), output passes through multiplexer (T_{comb}) to the RAM (T_{local}) and is finally stored in the RAM (T_{comb}). Thus the clock period is $T_{clock} = 3T_{comb} + 2T_{seq} + 2T_{local} + T_{global}$.

Let T_{acsg} be the time required for a single reconfiguration of the address and control signal generator and T_{con} be the reconfiguration time for a single RAM column to adder connection. Then the total reconfiguration time is $T_{reconf} = 2T_{acsg} \log n + T_{con}(2n-1)$.

4 Conclusions

In this paper we have presented a new approach to designing configurable computing solutions which is better suited to exploiting dynamic reconfiguration than traditional approach. Our approach of modeling allows the algorithm synthesis approach which is a more natural way of developing applications.

We described a general model of reconfigurable computing and derived two specific variants. The reconfigurable mesh model is a very powerful model with efficient algorithms for several computations. Restricted variants of this model have been implemented and the model provides interesting ideas as to the directions in which reconfigurable devices should evolve. The FPGA model abstracts current generation devices and can be used to map algorithms. We illustrate our approach by implementing an algorithm that exploits dynamic reconfiguration on the FPGA model.

References

[1] Actel Corporation. *ACT 3, Product Specification.*

[2] Altera Corporation. *FLEX 10K, Product Specification.*

[3] Atmel. *AT6000, Product Specification.*

[4] Stephen Brown and Jonathan Rose. FPGA and CPLD Architectures: A Tutorial. *IEEE Design & Test of Computers*, Summer 1996.

[5] Andre DeHon. *Reconfigurable Architectures for General Purpose Computing.* PhD thesis, MIT AI Lab, September 1996.

[6] Mike Donlin. Programmable logic and synthesis strive to get in sync. *Computer Design*, August 1996.

[7] T.J. Fountain and V. Goetcherian. CLIP4 parallel processing system. In *Proc. Institute of Electrical Engineering*, volume 127E, pages 219–224.

[8] M. B. Gokhale and A. Marks. Automatic Synthesis of Parallel Programs Targeted to Dynamically Reconfigurable Logic Arrays. In *Proceedings of the 1995 International Workshop on Field-Programmable Logic and Applications, Oxford, England*, September 1995.

[9] Scott Hauck. Survey of software for Reconfigurable Systems. Note in comp.arch.fpga, September 1996.

[10] J. Jang and V.K. Prasanna. Efficient parallel algorithms for some geometric problems on reconfigurable mesh. In *Proc. of International Conference on Parallel Processing*, August 1992.

[11] Viktor K. Prasanna Kumar and C.S. Raghavendra. Array Processor with Multiple Broadcasting. *Journal of Parallel and Distributed Computing*, 4:173–190, 1987.

[12] Massimo Maresca and Hungwen Li. Connection autonomy in SIMD computers. *Journal of Parallel and Distributed Computing*, 7:321–334, 1989.

[13] R. Miller, V.K. Prasanna Kumar, D.I. Reisis, and Q.F. Stout. Parallel Computations on reconfigurable Meshes. *IEEE Transactions on Computers*, July 1993.

[14] National Semiconductor. *CLAys, Advance Product Specification.*

[15] H. Park, V.K. Prasanna, and J. Jang. Fast arithmetic on reconfigurable meshes. In *Proc. of International Conference on Parallel Processing*, August 1993.

[16] Jonathan Rose, Abbas El Gamal, and Alberto Sangiovanni-Vincentelli. Architecture of Field Programmable Gate Arrays. *Proceedings of the IEEE*, July 1993.

[17] D.B. Shu and J.G. Nash. *The gated interconnection network for dynamic programming.* Plenum, 1988.

[18] Ju wook Jang, Heonchul Park, and Viktor K. Prasanna. A Bit Model of Reconfigurable Mesh. *Reconfigurable Architectures Workshop, 8th International Prarallel Processing Symposium*, April 1994.

[19] Ju wook Jang, Heonchul Park, and Viktor K. Prasanna. A Fast Algorithm for Computing a Histogram on Reconfigurable Mesh. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, February 1995.

[20] Ju wook Jang and Viktor Prasanna. An Optimal Sorting Algorithm on Reconfigurable Mesh. *Journal of Parallel and Distributed Computing*, 1995.

[21] Xilinx Inc. *XC6200, Advance Product Specification.*

International Workshop on Field Programming Logic and Applications, September 1997.

Fast Parallel Implementation of DFT Using Configurable Devices*

Andreas Dandalis and Viktor K. Prasanna
Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-2562, USA
{dandalis, prasanna}@usc.edu
http://ceng.usc.edu/~prasanna
Tel: +1-213-740-4336, Fax:+1-213-740-4418

Abstract. In this paper we propose a fast parallel implementation of Discrete Fourier Transform (DFT) using FPGAs. Our design is based on the Arithmetic Fourier Transform (AFT) using zero-order interpolation. For a given problem of size N, AFT requires only $O(N^2)$ additions and $O(N)$ real multiplications with constant factors. Our design emploies $2p+1$ PEs ($1 \leq p \leq N$), $O(N)$ memory and fixed I/O with the host. It is scalable over p ($1 \leq p \leq N$) and can solve larger problems with the same hardware by increasing the memory. All the PEs have fixed architecture. Our implementation is faster than most standard DSP designs for FFT. It also outperforms other FPGA-based implementations for FFT, in terms of speed and adaptability to larger problems.

1 Introduction

The Discrete Fourier Transform (DFT) plays a fundamental role in digital signal processing. The complexity and computation time of algorithmic approaches for forward computation of DFT, are essential issues in algorithms where many forward DFTs are required while one inverse Fourier transform must be performed at the end. For a problem of size N, the sequential computation time of a straightforward approach is $O(N^2)$ and is characterized mainly by the large number of complex multiplications and additions. This fact limits the computational performance of the approach as well as the algorithmic efficiency of implementations using Field Programmable Gate Arrays (FPGAs).

The FPGA based implementations for computing the DFT, proposed in [12, 8, 11], use the Fast Fourier Transform (FFT) to reduce the computation time and complexity to $O(N \log_2 N)$. The basic computation unit is the butterfly. Butterfly is a repetitive structure that has 2 inputs and 2 outputs. It involves

* This research was performed as part of the MAARC project (Models, Algorithms and Architectures for Reconfigurable Computing, http://maarc.usc.edu). This work is supported by DARPA Adaptive Computing Systems program under contract no. DABT63-96-C-00049 monitored by Fort Hauchuca.

one complex multiplication, one complex addition and one complex subtraction. For a problem of size N, the algorithm requires $\log_2 N$ stages with $N/2$ butterflies in each stage. Even though these designs optimize the structure of the butterfly, the complexity still remains high. All these designs are solutions optimized for a particular problem size. For larger problems, re-design is required resulting in area penalty. Parallelism is not exploited and the designs are not scalable except the one proposed in [11]. In [8], the idea of FPGAs with an external multiplier is used to overcome the critical issue of complex multiplication. This solution has still problems since it adds extra control/complexity and requires a large number of I/O pins for interfacing the multiplier chip. In spite of this, the computation time is not attractive. The implementation in [11] uses the CORDIC approach for optimizing the butterfly by eliminating multiplications. Again, the resulting performance is not attractive.

In this paper we propose a novel parallel, scalable, partitioned solution for computing the DFT using FPGAs, based on the Arithmetic Fourier Transform (AFT). Using this approach, we can solve larger problems with fixed hardware, simply by increasing the memory size. We can linearly speed-up the computation proportionally to the number of PEs employed and achieve superior performance compared with previous FPGA-based solutions. Also, it offers faster solution compared with most standard DSP designs for computing the DFT. The key idea of our design is the use of an algorithmic approach to the problem. Contrary to traditional approaches, we perform an algorithmic design for reconfigurable devices, based upon the architecture/features of the device. While known techniques map an algorithm for DFT onto the device and perform device dependent optimizations, our methodology emploies algorithm synthesis techniques instead of logic synthesis. This alleviates the FPGA's restriction of fast/compact adders vs slow/area-consuming multipliers. Complex multiplication is a critical issue in DSP applications and can lead to poor performance of FPGA-based solutions. AFT turns to be a suitable algorithmic approach for FPGAs since it is less complex than the FFT and performs real multiplications with constant factors instead of complex multiplications.

The Arithmetic Fourier Transform is based on the Möbius inversion formula of series and has been shown to be competitive with the conventional FFT in terms of accuracy, complexity and speed [9]. It needs $O(N^2)$ additions and $O(N)$ real multiplications by constant coefficients. It reduces the computation time of DFT to $O(N)$. In our design, two sets of p PEs ($1 \leq p \leq N$) and an additional PE are used for computing $2N+1$ Fourier coefficients [7]. Our design is scalable over p ($1 \leq p \leq N$), thus it can achieve $O(p)$ speed-up. It is also a partitioned solution since it can solve larger problems by increasing the memory size in proportion to the N the size of the problem. In each set, all PEs have the same architecture and perform additions and zero-order interpolation. The additional PE performs the scaling of the intermediate values by constant factors. All the PEs are cascaded using pipelining. The data as well as the control signals move from left to right. The complete design requires $O(N)$ memory and has fixed I/O bandwidth. External memory is used for storing the scaling factors as well as intermediate Fourier coefficients. Constant coefficients multiplier (KCM) [2], is used for performing the scaling operation. KCMs use the Distributed Arithmetic approach (DA) and turn out to be a very efficient choice for digital signal processing in terms of speed and area. The compact size and high performance of the KCMs compared with standard full multipliers, are promising features that make the AFT algorithm an efficient solution for computing the DFT using FPGAs. In addition, the parallel/modular structure, the regular architecture as well as the fixed, independent of the problem size I/O bandwidth, make our approach an attractive solution for implementation in FPGAs.

Preliminary estimations shows that our design achieves speed-up of 2-10 over most standard DSP designs for 256-FFT. Compared with the Fastest FFT in the West [12], the CORDIC approach [11] and the implementation in [8], our design outperforms these solutions in terms of speed and adaptability to larger size problems. Our preliminary implementation reported here using Xilinx devices, can be further optimized resulting in higher speed and less area.

This paper is organized as follows. In Section 2 we describe the Arithmetic Fourier Transform while in Section 3 we introduce our scalable architecture for AFT. In Section 4 the computation time and area estimations are shown. Finally in Section 5 comparisons are discussed and concluding remarks are made.

2 Arithmetic Fourier Transform

The Arithmetic Fourier Transform (AFT) is based on the Möbius inversion formula of series. Since it involves only additions and real multiplications by constant factors, it is computationally less complex than FFT while it achieves $O(\log_2 N)$ speed-up over it. An introduction to AFT is given below and detailed descriptions of it can be found in [7, 9].

Given $2N$ input samples $A(m), m = 0, 1, ..., 2N-1$, we compute an average and $2N$ alternating averages over them. All these averages are scaled by constant factors and then the Möbius inversion formula is applied for the computation of $2N+1$ Fourier coefficients. The Möbius inversion formula theorem [5] and the definition of the alternating average, are the key mathematical tools for the AFT algorithm.

Theorem (The Möbius inversion formula) Let $f(n)$ be a non vanishing function in the interval $1 \leq n \leq N$ and $f(n) = 0$ for $n > N$, where n, N are positive integers. If

$$g(n) = \sum_{m=1}^{\lfloor N/n \rfloor} f(mn)$$

then

$$f(n) = \sum_{k=1}^{\lfloor N/n \rfloor} \mu(k) g(kn)$$

where $\lfloor ... \rfloor$ denotes the integer part of a real number and $\mu(k)$ is the Möbius function.

The Möbius function is defined as

$$\begin{array}{ll} \mu(n) = 1 & \text{if } n = 1 \\ \mu(n) = (-1)^r & \text{if } n = p_1 p_2 ... p_r \text{ where } p_i\text{(i=1,2,...,r) distinct primes} \\ \mu(n) = 0 & \text{if } p^2 \mid n \text{ for some prime } p \end{array}$$

Definition (Alternating Average) The $2n$th alternating average $B(2n, \alpha)$ of the $2n$ values $A(mT/2n + \alpha T), 0 \leq m \leq 2n-1$, is defined as:

$$B(2n, \alpha) = \frac{1}{2n} \sum_{m=0}^{2n-1} (-1)^m A(mT/2n + \alpha T)$$

where α is a shifting factor, $-1 < \alpha < 1$. Assuming now a finite Fourier series $A(t)$ with period T, we can represent it as:

$$A(t) = a_0 + \sum_{n=1}^{N} a_n \cos 2\pi n f_0 t + \sum_{n=1}^{N} b_n \sin 2\pi n f_0 t$$

where $f_0 = 1/T$, a_n and b_n are the real and imaginary parts of the Fourier coefficients of the non vanishing function in the interval $-N \leq n \leq N$ and a_0 is the mean of $A(t)$. Applying the Möbius inversion formula to $A(t)$, we can compute the $2N+1$ Fourier coefficients in terms of the alternating averages as follows:

$$a_0 = \frac{1}{2n} \sum_{m=1}^{2N} A(m), \quad a_n = \sum_{l=1,3,..}^{\lfloor N/n \rfloor} \mu(l) B(2n, 0)$$

$$b_n = \sum_{l=1,3,...}^{\lfloor N/n \rfloor} \mu(l)(-1)^{\frac{l-1}{2}} B(2n, \frac{1}{4nl})$$

where $n = 1, 2, ..., N$ and $m = 0, 1, ..., 2N-1$.

For computing the alternating averages $B(2n, 0), B(2n, \frac{1}{4nl})$ from the input samples $A(m)$, we use zero-order interpolation for computational efficiency [9] . In this method we interpolate an unknown value $A(mT/2n + \alpha T)$ to a known input sample $A(i)$, where i is the integer part of $mT/2n + \alpha T$. The resulting error due to this approximation is shown to be tolerable [9, 10]. The AFT computation method presented above, requires $(2N+1)^2$ additions and $(2N+1)$ multiplications with constant factors, for computing $2N+1$ Fourier coefficients. The reduced complexity, the use of scaling by constant factors instead of complex multiplications and the amenability to parallel processing makes AFT more desirable computationally than FFT [10].

In Figure 1 we can show the structure of the AFT algorithm. In Part I the non-scaled alternating averages are computed while in Part II the scaling operation and the computation of a_0 take place. Finally, in Part III the $2N$ Fourier coefficients are computed. Multiplications with constant factors are performed only in Part II while in the other parts additions and zero-order interpolation are performed. The Möbius values in the last part define the sign of the alternating averages.

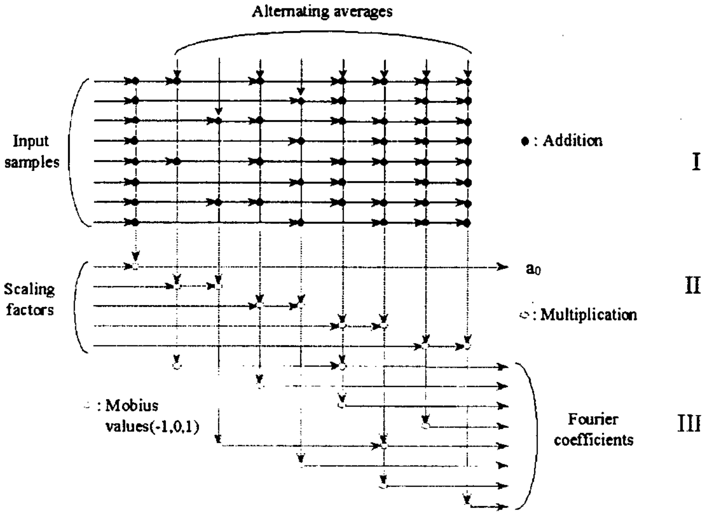


Fig. 1. Structure of AFT

3 A Scalable Architecture for AFT

In this section we show a scalable architecture to map the AFT algorithm (see Figure 2). Each part of the architecture corresponds to a part of the AFT structure in Figure 1. Data and control signals flow from left to right.

In Part I, p PEs are employed to compute the alternating averages. An input buffer B_I of size $2N \times w$ is used for storing the input window of $2N$ samples, where w denotes the number of bits in each input sample. Assuming that p divides N, each window is fed N/p times into the pipe. Let $PE_{1,i}$ denote the ith PE in Part I, $1 \leq p \leq N$ and $n = 1, 2, ..., N$. In $PE_{1,n\,modp}$, the alternating averages $B(2n, 0)$ and $B(2n, \frac{1}{4nl})$ are computed during the $\lceil n/p \rceil th$ feeding of the input data window into the pipe. Each PE checks if the received data is needed for its computation based on the zero-order interpolation. The interpolation is implemented using local registers and a comparator for checking the index of the received sample. Every $2N$ units, p alternating averages are computed. Thus, the total computation time for $2N$ alternating averages is $\frac{2N^2}{p} + 2p - 1$ units. All the PEs in this part have the same architecture and consist of one adder, one comparator and local registers. The local registers are used for performing the interpolation as well as for interconnection with other PEs.

In Part II, one PE is employed for scaling the averages computed in the previous part and for computing the mean a_0. This PE is denoted as PE_{mul}. Totally $2p$ scaled averages are computed every $2N$ time units. PE_{mul} employes

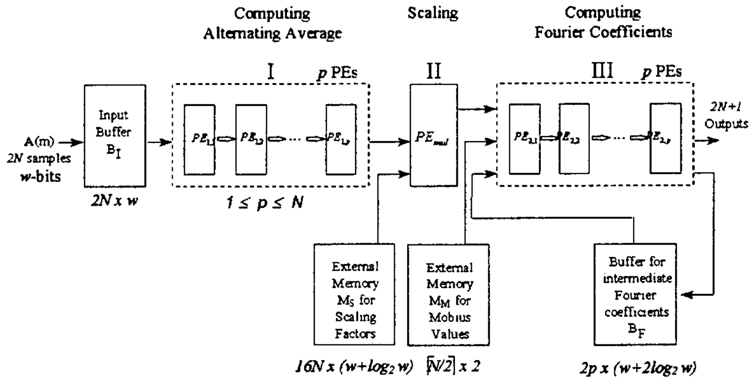


Fig. 2. Overall Architecture

one Constant Coefficient Multiplier (KCM) [2] for computing a_0 as well as for scaling $B(2n, 0)$, $B(2n, \frac{1}{4nl})$ by constant factors during each step. Using the hybrid technique described in [2], we have to precalculate 16 values for each scaling factor. The hybrid technique of multiplication is a hexademical equivalent of the long hand method. Since a single hex digit represents four bits, the look-up table for each constant factor has entries for 0 to $15(F)$. Thus, the size of the external memory M_S would be $16N \times (w + \log_2 w)$. The set of precalculated values of constant factor $\frac{1}{2i}$ is stored in 16 consecutive locations of the memory, starting from the ith memory location where $1 \leq i \leq N$. PE_{mul} also employes local registers for interconnection with other stages.

In Part III, p PEs are employed to compute the Fourier coefficients. Similar to Part I, the PEs in this part are denoted as $PE_{2,i}$. All the PEs have the same architecture. Each of them consists of one adder, one comparator and local registers. As in the first group, each processing element checks if the received average is needed in its partial sum. This checking is performed using the index of the incoming average. The Möbius values required for the computation of the partial sums are provided by an external memory M_M. The memory size is $\lceil \frac{N}{2} \rceil$ and the stored values are $\{-1, 0, 1\}$. Few local registers are employed for controlling the data flow between consecutive PEs. A buffer B_f is employed for storing the intermediate results of the computation. When a new set of $2p$ alternating averages are available to Part III, the intermediate results of the computation and the Möbius function values are fed back from the rightmost to the leftmost PE. The size of B_f is $2p \times (w + 2\log_2 w)$ where w denotes the number of bits of each input sample. In this part, only the Fourier coefficients $a_1, b_1, a_2, b_2, ..., a_{N/3}, b_{N/3}$ are computed since for $n > N/3$, a_n and b_n are equal to alternating averages $B(2n, \alpha)$.

Since the computation time of $2N$ alternating averages is $\frac{2N^2}{p} + 2p - 1$ units, the total computation time for $2N+1$ Fourier coefficients becomes $\frac{2N^2}{p} + 3p + \lceil \frac{p}{2} \rceil$ units. The throughput rate of KCM critically affects the overall performance since it determines the minimum time unit. The architecture employes $2p$ adders, 1 KCM, few local registers and external memories. The total size of the external memories is $O(N)$. A key advantage of the design is that it is scalable over p, $1 \leq p \leq N$, thus it can linearly speed-up the computation by increasing p. It can also solve larger size problems (at a lower throughput rate) by simply increasing the memory but still using the same number and structure of PEs.

4 Performance Estimates

Table 1 lists the estimated area for various components of our architecture. We estimated the area of each of the functional blocks and the total area for each PE was then derived. All the PEs in a group (I or II) have the same architecture. Thus, each of them occupies the same constant area. We have assumed 16-bit input data and that our design is mapped onto Xilinx XC4000 series of FPGAs.

CLBs per Function	$PE_{1,i}$	$PE_{2,i}$	PE_{mul}
Registers	110	140	50
16-bit comparator	2	2	-
16-bitadder	16	16	-
16-bit KCM	-	-	230
Control	2	2	2
Total CLBs	130	160	282

Table 1. PE area requirements in terms of number of CLBs to realize the function

Our design consists of p $PE_{1,i}$, p $PE_{2,i}$ and one PE_{mul}. Assuming $2N$ input samples, the total time for computing the $2N+1$ Fourier coefficients is $\frac{2N^2}{p} + 3p + \lceil \frac{p}{2} \rceil$ time units. The time unit is determined by the performance of the KCM since Part II is the most time-consuming stage of our design. The pipelined performance of a 16-bit operand KCM (after place and route) is $50MHz$ [2] using the -3 speed grade components. The performance of a 16-bit adder is in the range of $100MHz$ using -3 speed grade components [6]. Thus, there is enough time for the PEs in parts I and III to complete their operations using $50MHz$

clock rate. Currently, −2 speed grade devices are available. Thus, $50MHz$ system clock rate is an achievable goal for the entire design. Table 2 shows the area requirements and estimate of the computation time for two designs. We assume 256 input samples and $50MHz$ system clock rate.

Hardware	**Area Requirements**	**Computation Time**
$p = 8$	2602 **CLBs**, 3 XC4025	4124 **time units** (82.48 μsec)
$p = 16$	4922 **CLBs**, 5 XC4025	2104 **time units** (42.08 μsec)

Table 2. Area and performance estimates

5 Comparisons and Conclusions

In this paper, we have proposed a novel parallel, scalable, partitioned solution for computing the DFT using FPGAs. Our solution based on the AFT turns out to be more efficient than the FFT based approach in terms of area and speed. Our design is scalable over $1 \leq p \leq N$, where p is the number of PEs employed. We can also linearly speed-up the computation proportionally to p. The architecture of the PEs and the I/O bandwidth are fixed and independent of the problem size. The required memory is $O(N)$. Our design can solve larger problems (with reduced throughput) with fixed hardware.

In Figure 3 and in Table 3 the execution times of various designs for 256-FFT are shown. The input samples are 16-bit data for all the designs. Figure 3 shows the results of a benchmark evaluation [12] of DSP-based and Xilinx FPGA-based designs. Our design achieves speed-up of $2 - 10$ over most single chip DSP designs for 256-FFT.

Implementation	**Area Requirements**	**Computation Time**
Xilinx 3 nodes [12]	1 XC4025	102.4 μsec
Xilinx 70**MHz**[12]	1 XC4025	223 μsec
Xilinx 60**MHz** [12]	1 XC4025	312.5 μsec
PDSP16116/**A** [8]	2 Chips	61.4 μsec
CORDIC [1] [11]	10 XC4010	5000 μsec

Table 3. Performance of FPGA-based designs for 256-FFT

[1] 1000-FFT

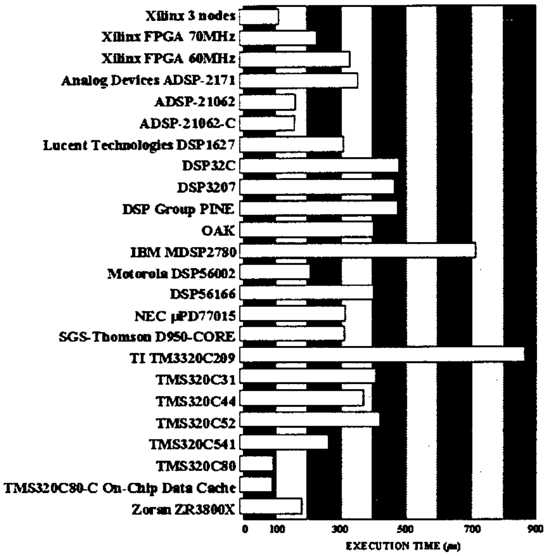


Fig. 3. Performance of DSP-based and FPGA-based designs for 256-FFT (from[12])

In Table 3 the performance of five FPGA-based implementations are shown. Three of them are from "The Fastest FFT in the West" [12]. In that work a radix-2 butterfly FFT design was used. The implementation in [8] makes use of one Altera FPGA and a PDS P16116/A 16-bit complex multiplier to overcome the critical problem of performing complex multiplication in FPGAs. A radix-4 design and necessary control were mapped onto the Altera FPGA. The implementation in [11] uses the CORDIC approach to eliminate the complex multiplications. Even though it computes a 1000-point FFT, the performance is not attractive compared with our approach. Table 3 also shows the area requirements of these implementations.

Our design is faster than the earlier FPGA-based implementations. The implementations in Table 3 are designs optimized for a particular problem size and device features and need to be redesigned for larger problems. Our design can also handle larger problems with the same fixed hardware by increasing the memory. Known implementations exploit the power of a single chip while we have developed a scalable and partitioned solution with high performance and adaptability to larger problems. Our performance estimation has been based on a preliminary implementation and no optimizations have been performed. Further improvement of the performance of our design is possible. Parallelism can be exploited; for example parallel I/O can improve the performance significantly. Many registers can be eliminated by efficiently performing zero-order interpolation. Other interpolation approaches (such as first-order) can also be exploited.

The work reported here is part of the USC MAARC project. This project is developing algorithmic techniques for realising scalable and portable applications using configurable computing devices and architectures. Contrary to traditional approaches to configurable computing, in our approach the user "sees" the architecture/device features and uses algorithm synthesis techniques instead of logic synthesis. We are developing computational models and algorithmic techniques based on these models to exploit dynamic reconfiguration. In addition, compilation onto reconfigurable hardware is also addressed. Some related results can be found in [1], [3], [4].

References

1. K. Bondalapati and V. K. Prasanna, "Reconfigurable Meshes: Theory and Practice", *Reconfigurable Architectures Workshop, Int. Parallel Processing Symposium (IPPS)*, April 1997.
2. K. Chapman, "Constant Coefficient Multipliers for the XC4000C", XILINX Application Note 054, Dec. 1996.
3. S. Choi, Y. Chung and V. K. Prasanna, "Configurable Hardware for Symbolic Search Operations", submitted to *Int. Conf. Parallel and Distributed Systems*, Dec. 1997.
4. Y. Chung S. Choi and V. K. Prasanna, "Parallel Object Recognition on an FPGA-based Configurable Computing Platform", submitted to *Int. Workshop on Computer Architecture for Machine Perception*, Oct. 1997.
5. L. K. Hua, "Introduction to Number Theory", New York: Springer-Verlag, 1982.
6. B. New, "Estimating the Performance of XC4000E Adders and Counters", XILINX Application Note 018, July 1996.
7. H. Park and V. K. Prasanna, "Modular VLSI Architectures for Computing the Arithmetic Fourier Transform", *IEEE Trans. Signal Processing*, vol. 41, no. 6, pp. 2236-2246, June 1993.
8. R. J. Petersen and B. L. Hutchings,"An Assessment of the Suitability of FPGA-Based Systems for use in Digital Signal Processing", in *Proc. Int. Workshop on Field-Programmable Logic and Applications*, Aug. 1995.
9. I. S. Reed, D. W. Tufts, X. Yu, T. K. Truong, M. Shih and X. Yin, "Fourier analysis and signal processing by use of the Möbius inversion formula", *IEEE Trans. Acoust., Signal Processing*, vol.38, no. 3, pp. 458-470, Mar. 1990.
10. I. S. Reed, M. T. Shih, T. K. Truong, E. Hendon and D. W. Tufts, "A VLSI architecture for simplified arithmetic Fourier transform algorithm", in *Proc. Int. Conf. Application Specific Array Processor*, 1990.
11. R. Wilson, "Reprogrammable FPGA-based techniques provide prototyping aid", *Electronic Engineering Times*, Mar. 11, 1996.
12. XILINX DSP Application notes, "The Fastest FFT in the West", http://www.xilinx.com/apps/displit.htm

This article was processed using the LaTeX macro package with LLNCS style

From Proceedings of Computer Architectures for
Machine Perception (CAMP '97)

Parallel Object Recognition on an FPGA-based Configurable Computing Platform *

Yongwha Chung

System Engineering Section
Electronics and Telecommunications Research Institute
Taejon, Korea

Seonil Choi and Viktor K. Prasanna

Department of EE-Systems, EEB-200C
University of Southern California
Los Angeles, CA 90089-2562
{yongwha + seonil + prasanna}*@halcyon.usc.edu*
http://maarc.usc.edu

Abstract

Object recognition involves identifying known objects in a given scene. It plays a key role in image understanding. Geometric hashing has been proposed as a technique for model-based object recognition in occluded scenes. However, parallel techniques are needed to realize real-time vision systems employing geometric hashing.

In this paper, we develop a design technique for parallelizing geometric hashing on an FPGA-based platform. We first transform the hash table which contains symbolic data into a bit-level representation. By regularizing the data flow and exploiting bit-level parallelism in hardware, our design achieves high performance. Using our approach, given a scene consisting of 256 feature points, a probe can be performed in 1.65 milliseconds on an FPGA-based platform having 32 Xilinx 4062s. In earlier implementations, the same probe operation was performed in 240 milliseconds on a 32K-node CM2 and in 382 milliseconds on a 32-node CM5. Also, the same operation takes 40 milliseconds on a 32-node IBM SP-2. By parameterizing the application and the device characteristics, we derive an area-time efficient design based on these parameters. Furthermore, our approach can be applied to many geometric hashing methods and is portable to other FPGA devices.

1 Introduction

Object recognition is a key step in computer vision. Most model-based recognition systems work by hypothesizing matches between scene features and model features, predicting new matches, and verifying or changing the hypotheses through a search process. *Geometric hashing* [12] has been proposed as an alternate approach for object recognition. However, parallel techniques are needed to use geometric hashing in real-time applications.

In geometric hashing, a set of models is specified using their feature points [12]. For each model, all possible pairs of feature points are designated as a *basis set.* The coordinates of the features points of each model are computed relative to each of its basis. These coordinates are then used to hash into a hash table. The entries in the hash table comprise of (*model, basis*) pairs and are precomputed as follows: using a chosen basis, if a feature point in a model hashes into a bin, then the model and the basis are recorded in the bin.

*This research was performed as part of the MAARC (*M*odels, *A*lgorithms, and *A*rchitectures for *R*econfigurable *C*omputing) project. This work is supported by DARPA Adaptive Computing Systems program under contract no. DABT63-96-C-0049 monitored by Fort Hauchuca.

In the recognition phase, an arbitrary pair of feature points in the scene is chosen as a basis and the coordinates of the feature points in the scene are computed relative to this basis. The new coordinates are used to hash into the hash table. Votes are accumulated for the (*model, basis*) pairs stored in the hashed location. The pair winning the maximum number of votes is chosen as a candidate for matching. The execution of the recognition phase corresponding to a basis pair is termed as a *probe*.

There have been some prior efforts in parallelizing geometric hashing on HPC platforms [4, 16, 18]. A major problem in these implementations is that their performance degrades significantly due to the irregular communication in accessing the hash bins and in voting because the hash table and the vote boxes are distributed among the processing nodes. We refer to the congestion in accessing the bins as well as in voting as "memory congestion problems".

Recently, configurable computing ideas [3, 8] have shown attractive speedups for many applications. They offer large scale parallelism and highly customized solutions. Field Programmable Gate Arrays (FPGAs) are becoming one of the major configurable computing devices which offer low development cost, rapid prototyping, and user controlled reconfigurability.

In this paper, we develop a design technique to parallelize the probe operation on an FPGA-based platform. We first transform the hash table which contains symbolic data into a "bit-level" representation. By regularizing the data flow and exploiting large scale bit-level parallelism in hardware, our design avoids the memory congestion problems. This leads to high performance. Since we employ a bit-level hash table, there is no hash bin access between the processing nodes although the hash table is distributed among the processing nodes. All operations are performed locally in each processing node except for finding a maximum value over the processing nodes. Furthermore, we parameterize the application as well as the device characteristics. Based on these parameters, we derive an area-time efficient design. The implementation is simplified using a modular approach.

We have synthesized our design using Xilinx 4062 devices. Using a clock rate of 10MHz, the execution time for the probe operation on a scene consisting of 256 feature points is estimated to be 1.65 *milliseconds* using 32 FPGAs and 128M bytes of memory. In this design, as in the earlier experiments, we assume that the model database has 1024 models and each model is represented using 16 feature points. For the sake of comparison, a parallel algorithm was implemented on a 32-node IBM SP-2. In our implementation, each processing node has the entire set of vote boxes to reduce the communication cost and to reduce memory congestion. However, the hash table was partitioned such that each hash bin is evenly distributed among the processing nodes. This balances the load on the processing nodes during the voting process. All operations are performed locally except for finding the global maximum. Using between 64 and 512M bytes of memory in each processing node operating at 66MHz, the execution time was about 40 *milliseconds*. In earlier implementations, the same probe operation was performed in 240 *milliseconds* on a 32K-node CM2 [16] and in 382 *milliseconds* on a 32-node CM5 [18].

The organization of the paper is as follows. In Section 2, configurable computing is briefly introduced. The geometric hashing technique is outlined in Section 3. Section 4 discusses parallelization of geometric hashing. In Section 5, implementation details are shown and the performance of the proposed design is compared with earlier results. Concluding remarks are made in Section 6.

2 Configurable Computing

Configurable computing has recently gained much attention with the promise of delivering an order of magnitude performance improvement over general-purpose processors. The paradigm of computing in space (i.e., a series of computations on several functional units), as opposed to computing in time (i.e., a series of computations executed in sequence on a single functional unit), is being actively explored. There are several directions in which research is being carried out to realize the potential of configurable computing.

The idea of a VLSI array of processors overlaid with a reconfigurable bus system, and an abstract model based on this architecture was proposed in [15]. Based on this initial work, several abstract models of reconfigurable architectures and fast parallel algorithms for many problems have been described in the literature. For example, efficient algorithms for fundamental data movement operations [15], sorting [11], and image processing [10] have been developed on the reconfigurable meshes. There have been several prototype implementations of such abstract models. Such architectures include Abacus [2] and YUPPIE [14].

Recently, the advent of Field Programmable Gate Arrays (FPGAs) has given rise to new opportunities in the configurable computing area. Traditionally, FPGAs have been used for rapid prototyping and emulation. The main bottleneck in using these devices as configurable computing engines has been the time for reconfiguration. Current and future generation devices such as CLAy, XC6200, DPGA etc. alleviate the above problem by providing partial and dynamic reconfigurability [8]. In these devices, it is possible to modify the configuration of a part of the device while the configuration of the remaining part is unchanged. Some devices permit this partial reconfiguration even while other logic blocks are performing computations. Unlike such fine-grain devices, coarse grain devices in which multiple contexts of the configuration can be stored in the logic block and the context is dynamically switched have been proposed (for example, see [8]). Also, there are efforts under way to develop coupled architectures in which a reconfigurable array and a processor core cooperate on a computational task, exploiting the strengths of both architectures (for example, see [9]). Wormhole runtime reconfiguration has been proposed in [1]. In this approach, as the stream of data moves through the reconfigurable hardware, it rapidly creates and modifies datapaths and computing resources along the way. There have been some efforts to exploit dynamic reconfiguration [3, 13]. In these, the connections are configured based on the input data or the intermediate result of the computation.

Configurable computing provides the ability to redefine the hardware/software boundary in computing systems. This paradigm change results in new computation models, new programming methods, and new approaches to implementation of applications. Some of the greatest gains in this field may well come from providing appropriate abstractions of this technology to algorithm developers and compiler designers to allow them control over hardware that has not been previously exploited [13].

3 Object Recognition Using Geometric Hashing

In a model-based recognition system, a set of objects is given and the task is to find instances of these objects in a given scene. The objects are represented as sets of geometric features, such as points or lines, and their geometric relations are encoded using a minimal set of such features. The task becomes more complex if the objects overlap in the scene and/or other occluded unfamiliar objects exist in the scene.

Many model-based recognition systems are based on hypothesizing matches between scene features and model features, predicting new matches, and verifying or changing the hypotheses through a search process. Geometric hashing, introduced by Lamdan and Wolfson [12], offers a different approach It can be used to recognize flat objects under weak perspective. Because of such robustness, geometric hashing has been employed in the DARPA next-generation, model-based ATR (Automatic Target Recognition) system [6]. In the following, for the sake of completeness, we briefly outline the geometric hashing technique. Additional details can be found in [12].

Figure 1 illustrates the geometric hashing algorithm. The algorithm consists of two procedures, *preprocessing* and *recognition.*

Preprocessing:
The preprocessing procedure is executed off-line and only once. In this procedure, the model features are encoded and are stored in a hash table. The information is stored in a highly redundant multiple-viewpoint way. Assume each model in the database has n feature points. For each ordered pair of feature points in the model chosen as a basis, the coordinates of all other points in the model are computed in the orthogonal coordinate frame defined by the basis pair. Then, (*model,basis*) pairs are entered into the hash table bins by applying a given hash function f to the transformed coordinates.

Recognition:
In the recognition procedure, a scene consisting of S feature points is given as input. An arbitrary ordered pair of feature points in the scene is chosen. Taking this pair as a basis, the coordinates of the remaining feature points are computed. Using the hash function on the transformed coordinates, a bin in the hash table (constructed in the preprocessing phase) is accessed. For every recorded (*model,basis*) pair in the bin, a vote is collected for that pair. The pair winning the maximum number of votes is taken as a matching candidate. The execution of the recognition phase corresponding to one basis pair is termed as *probe.* If no (*model,basis*) pair scores high enough, another basis from the scene is chosen and a probe is performed.

Note that, the basis set can be chosen as a set of single points, point pairs, or triple points depending on the required functionality for occlusion, rotation, translation, and perspective. The object features can also be represented by other geometric features such as *lines* [17].

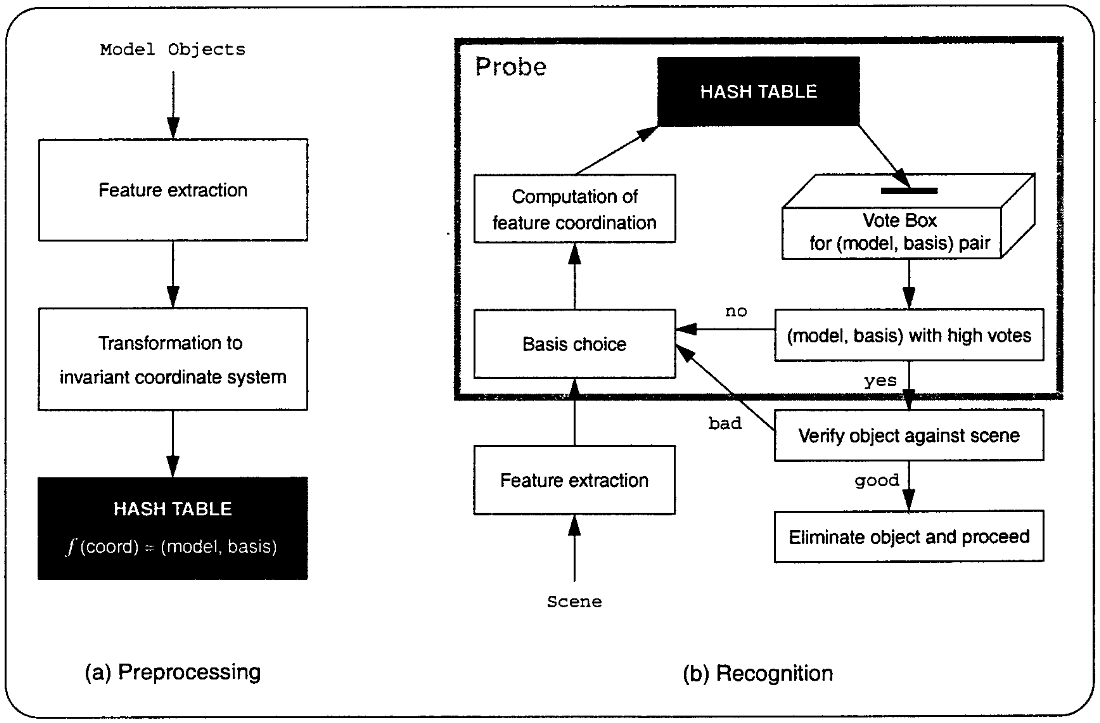

Model Objects
Feature extraction
Transformation to invariant coordinate system
HASH TABLE
f (coord) = (model, basis)
(a) Preprocessing
Probe
HASH TABLE
Computation of feature coordination
Vote Box for (model, basis) pair
no
Basis choice
(model, basis) with high votes
yes
bad
Verify object against scene
good
Feature extraction
Eliminate object and proceed
Scene
(b) Recognition

Figure 1: Illustration of the geometric hashing technique.

4 Parallel Geometric Hashing on an FPGA-based Configurable Computing Platform

In this section, we describe our parallel technique to implement the recognition phase. We first explain our bit-level hash table. Then, our parallel probe algorithm and its FPGA-based design are proposed. Finally, an analysis of our design is described.

We will not elaborate on parallelizing the preprocessing phase, since it is a one time process and can be carried out off-line. In the following, we ignore the initialization costs, such as loading the scene points to the processors and loading the hash table into the processor array. These assumptions were also made in the previous algorithms and in the implementations reported in [4, 16, 18].

The major difficulty in parallelizing the probe operation is that the performance depends on the partitioning and distribution of hash bins, the distribution of the votes generated, and the total number of votes generated. We refer to the congestion in accessing the bins as well as in voting as "memory congestion problems".

There have been several prior efforts in parallelizing the geometric hashing algorithms [4, 16, 18]. The implementations in [4, 16] have been performed on SIMD hypercube-based machines. A major problem in both the implementations is the requirement of large number of processing nodes. In [4], the number of processing nodes used is the same as the number of bins in the hash table. Thus, $O(Mn^3)$ processing nodes are needed. In implementations reported [16, 18], the $(model, basis)$ entries in a hash bin were represented as a linked list. Note that, the number of such entries in each hash bin can vary over the hash bins. By partitioning the hash table and the vote boxes statically among the processing nodes, the memory congestion in bin access as well as in voting significantly degrades the performance . In [18], these problems were solved using a sort-based approach. This approach handles congestion in bin access as well as in voting. However, additional overhead caused in implementing such a technique makes it attractive only if the computational cost associated with accessing the hash bin and processing the generated votes is high.

4.1 Bit-level Hash Table

In this paper, we propose a simple memory structure which can be accessed in parallel without memory congestion. By regularizing the data flow and exploiting a high degree of bit-level parallelism in hardware, large speedup is achieved. For the sake of explanation, we assume that there are no multiple entries of the same $(model, basis)$ pair in a hash bin. Note that, the number of $(model, basis)$ pairs recorded in a hash bin is upper bounded by $\frac{Mn(n-1)}{2}$.

The hash bin which has the linked list structure is converted to a bit-level hash bin. Figure 2 shows this hash table conversion. Let $UID(model, basis)$ denote an unique number, between 1 and $\frac{Mn(n-1)}{2}$, assigned to each $(model, basis)$ pair. Then, each hash bin is converted into our "bit-level" representation of size $\frac{Mn(n-1)}{2}$ bits as follows.

1. Initialize each of the $\frac{Mn(n-1)}{2}$ locations to "0".
2. For each $(model, basis)$ pair recorded in a hash bin, enter a "1" in the location $UID(model, basis)$.

The corresponding $(model, basis)$ pair in the hash table is marked as '1' in the bit-level hash table. Thus, a hash table in which the number of entries for each hash bin may not be uniformly distributed across the hash bins is mapped to a bit-level hash table having a regular structure. Using this bit-level hash table, we can exploit a high degree of parallelism and eliminate the congestion in hash bin access.

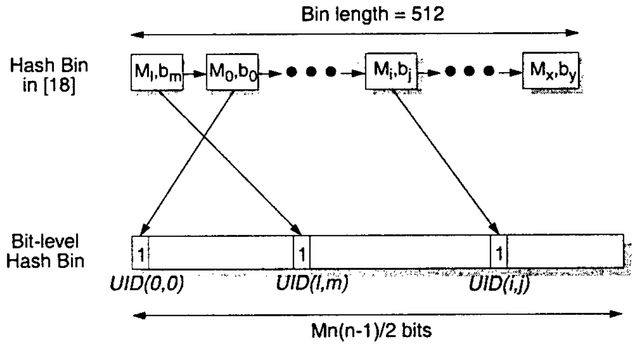


Figure 2: Hash table conversion.

4.2 Parallel Probe Algorithm

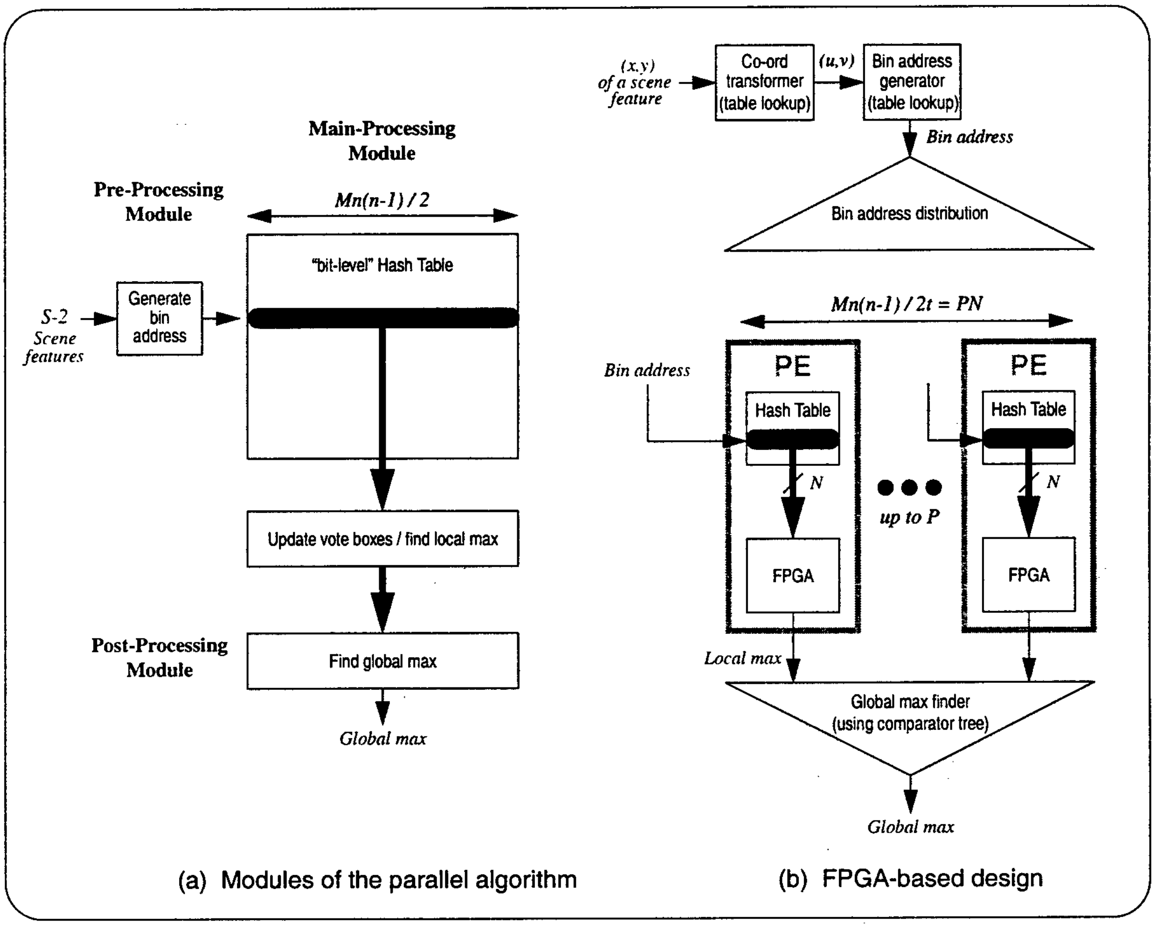
The basic strategy of our design is to access the $\frac{Mn(n-1)}{2}$ bit locations in a hash bin in parallel and then update the corresponding vote boxes in parallel (See Figure 3(a)). Thus, we can perform a probe operation without any memory congestion. Note that a single FPGA chip may not have enough number of Combinational Logic Blocks (CLBs) to handle the bit streams in parallel. Therefore, multiple FPGA chips are required. Finding the maximum among the vote boxes distributed in multiple FPGA chips is performed in 2 steps: find the *local maximum* in each FPGA chip and then find the *global maximum* across the FPGA chips.

To obtain a modular design, we partition our design into three modules: *pre-processing module*, *main-processing module*, and *post-processing module*. The pre-processing module generates the bin address of the bit-level hash table. Given (x, y), the coordinates of a scene point, the co-ord transformer first converts it into basis-relative coordinates (u, v). Then, the bin address generator converts (u, v) into a corresponding hash bin address. These two operations can be easily implemented using table look-up even though they involve complex arithmetic operations. Then, the main-processing module accesses the hash table using the computed bin address and detects *local* maximums in each FPGA chip. Finally, a *global* maximum across the FPGA chips is detected by the post-processing module using a comparator tree (See Figure 3(b)). Since parallelism is exploited in the main-processing module, in the following, we focus on designing that module.

A basic unit for the main-processing module consists of a pair of FPGA and local memory modules, and is denoted as *Processing Element* (PE). In general, trade-offs between area and time are possible when a specific function is implemented in hardware. Especially, the available sizes of Commercial Off-The-Shelf (COTS) devices may not match well with the basic unit of our design. Thus, the *virtual* PEs in our design need to be mapped onto *physical* PEs which can be implemented using COTS devices. To derive an area-time efficient design using COTS devices, we define parameters which characterize our design with respect to area and time (See Figure 4). Using P PEs, our design accesses PN bits of the hash table in parallel. Since $\frac{Mn(n-1)}{2}$ bits need to be accessed in each hash bin and since typically, $\frac{Mn(n-1)}{2} > PN$, the number of time multiplexing required to map virtual PEs to physical PEs is $\lceil t = \frac{Mn(n-1)}{2PN} \rceil$. Details of the time multiplexing to realize an area-time efficient design using COTS devices are described in the next section.

4.3 Design Analysis

The execution time for a probe using P PEs can be analyzed as follows. Throughout this paper, we assume that a memory access and an arithmetic or logic operation (such as **ADD**, **MUX**, **COMPARE**) can be



(a) Modules of the parallel algorithm

(b) FPGA-based design

Figure 3: Parallel probe algorithm and its FPGA-based design.

performed in unit time. The hash bin address can be generated in $O(1)$ times using table look-up. The generated hash bin address can be distributed to P PEs in $O(\log P$[1]$)$ time. The hash bin accesses for $S-2$ feature points can be performed concurrently with the operation to detect local maximums from the preceding hash bin accesses. Thus, the time to detect the local maximums over t multiplexing is $(S-2) \times t + \log N = O(\frac{SMn^2}{PN} + \log N)$. The hash bin access and the operation to detect a local maximum are pipelined. The final operation to detect a global maximum can be performed in $O(\log P)$ time.

Theorem 1 *Given a model database having M models where each model is represented using n feature points, a probe on a scene consisting of S feature points can be performed in $O(\frac{SMn^2}{PN} + \log N + \log P)$ time on an FPGA-based platform having P PEs, $1 < P \leq \frac{Mn(n-1)}{2N}$, where N denotes the width of FPGA-memory datapath in a PE.*

5 Implementation Details and Performance Estimate

In this section, we first discuss various issues in implementing the design technique developed in Section 4 on an FPGA-based platform. Then, we describe a design using Xilinx 4062 FPGA devices. Our design is motivated to achieve large speedup for typical size of images and models used by the vision community. We have chosen not to perform device dependent optimizations to improve performance. Figure 5 shows our development environment. We have synthesized our design using Synopsis FPGA compiler. Place and route was performed using Xilinx tools (XACT Devel-

[1] All logarithms in this paper are to base 2.

		Description
Application Parameters	M	Number of model objects
	n	Number of feature points in a model object
	S	Number of feature points in a scene
Design Parameters	P	Number of PEs required
	N	Width of FPGA-memory datapath in a PE
	t	Number of time multiplexing required to map virtual PEs to physical PEs

Figure 4: Parameters used for deriving an area-time efficient design.

opment System) to create a configuration file for an FPGA on a Sun Ultra Enterprise. Then, configuration file is downloaded onto the FPGA development board which consists of FPGAs and memory modules. The board is connected to a PC through PCI Local Bus.

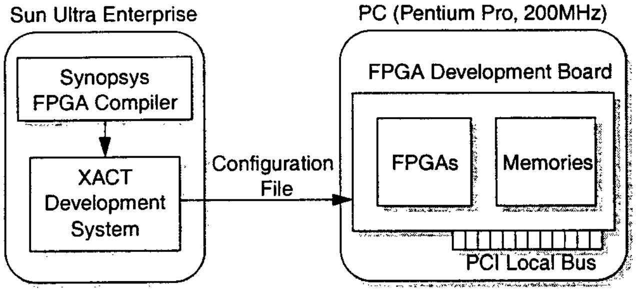


Figure 5: Our development environment

For the sake of illustration and evaluation of the resulting design, we consider a typical scenario as follows: We used a synthesized model database containing 1024 models. Each model consists of 16 randomly generated feature points in 2 dimensions. This results in a hash table having 4×2^{20} entries. These feature points were generated according to a Gaussian distribution with zero mean and unit standard deviation as in [16, 18]. Similarly, 256 scene points were synthesized using a normal distribution. The equalization technique in [16, 18] was applied to quantize the transformed coordinates, *i.e.*, for each transformed point (u, v), the following hash function (where σ denotes the standard deviation of the set of model points) is applied [18]:

$$f(u, v) = (1 - e^{-\frac{u^2+v^2}{3\sigma^2}}, \text{atan2}(v, u))$$

According to constraints imposed by Commercial Off-The-Shelf (COTS) devices, we first determine both the structure of each PE and the configuration of the PEs to synthesize an area-time efficient solution. Based on the configuration of the PEs in the main-processing module, the logic for the pre-processing and post-processing modules are determined.

5.1 Design Trade-offs

To illustrate feasibility of implementation and demonstrate resulting speedup, we assume that Xilinx 4062 FPGA chips and $512K \times 32$ bit memory modules are used for the implementation. The scenario considered above results in a hash table having $8K$ hash bins [16, 18]. To represent the same hash table using our bit-level design, we need $8K \times 120K$ bits (See Figure 6 (a)). For the sake of explanation, let MEM denote the number of memory modules needed to store the hash table. To implement this bit-level hash table with commercially available $512K \times 32$ bit memory modules, MEM, N, and t must satisfy the following constraints:

- $512K \times 32 \; bits \times MEM \geq 8K \times 120K \; bits$,
- $\frac{512K}{t} \geq 8K$, and
- $32 \; bits \times MEM \geq \frac{120K \; bits}{t} \geq PN \; bits.$

We can implement 64 vote boxes and the logic to find a local maximum among the vote boxes in an FPGA. Thus, the width of FPGA-memory datapath, N, becomes 64 and this amount of parallelism is achieved in a PE.

From the above constraints, a feasible configuration is $MEM = 60$, $t = 64$, and $P = 30$. Designs with large P (≥ 30) are not area efficient due to large number of memory modules. Also, note that the number of memory modules is 60 and the number of PEs is 30. Thus, two $512K \times 32$ bit memory modules are assigned to each PE to support 64-bit parallelism.

Figure 6 shows the actual mapping of the hash table onto the memory modules in our design. In this example, the size of the bit-level hash table is $8K \times 120K$ bits. A smaller ***sub-table*** whose size is $8K \times 4K$ bits is assigned to a PE (See Figure 6 (b)). Since the width of a hash bin in the sub-table is $4K$ bits and $N = 64$ bits can be read from the sub-table simultaneously, one hash bin access results in 64 reads to the memory module. When we map the sub-table into an actual memory module, a column major order is used. Thus, the first $8K \times 64$ bits are placed in the beginning of the memory module. The next $8K \times 64$ bits are placed immediately after this (See Figure 6 (c)). Figure 6 (c)).

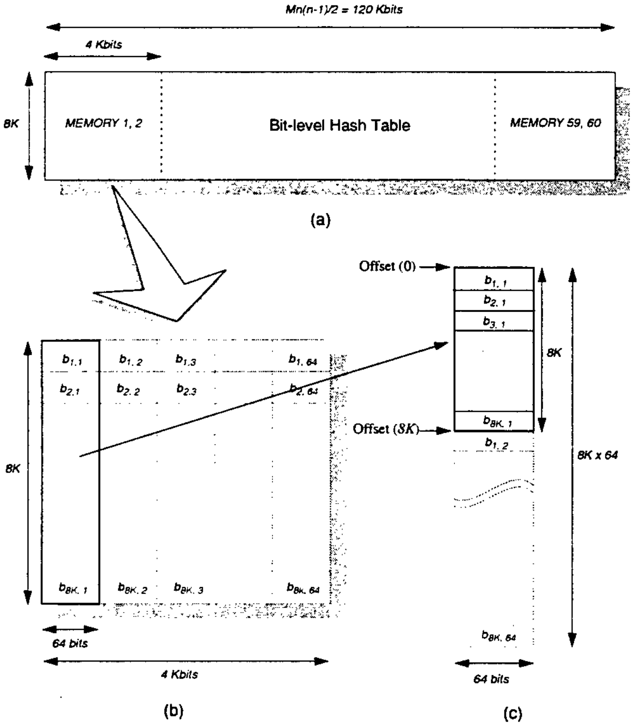


Figure 6: Memory organization. (a) Bit-level hash table, (b) Memory module (sub-table), and (c) The sequence of hash bins in a memory module.

Some details of the FPGA implementation of a PE is shown in Figure 7. It consists of 64 vote boxes, eight 8-to-1 multiplexors, and the logic to find a local maximum. The logic to find the local maximum consists of 8-input comparator tree and logic to update the local maximum. When a hash bin address is generated, 64 bits of the hash bin are fed into FPGAs from the memory modules. The corresponding vote boxes count the number of '1's. Once the voting operation is completed, the votes are stored in registers and are multiplexed to be fed into the comparator tree. Using the comparator tree, a maximum is found and is compared with the previous local maximum. If necessary, the local maximum is updated. In the design of the PE, we can also use a 64-input comparator tree to find the local maximum directly from 64 vote boxes. Since the execution time for the voting operation is longer than that for finding the local maximum, we multiplex the votes in the registers and feed them to a smaller comparator tree. Thus, we can reduce the amount of logic to find the local maximum.

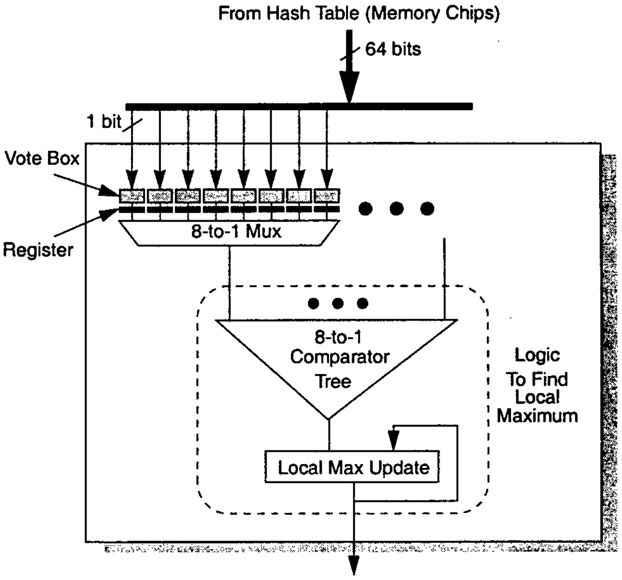
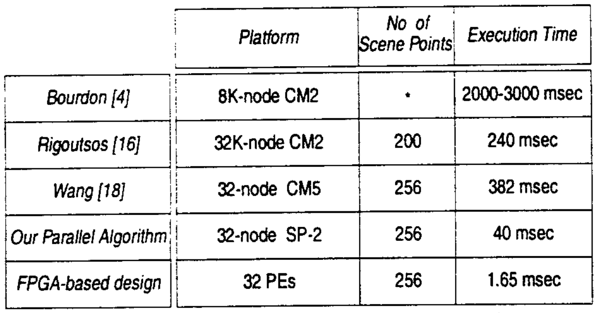


Figure 7: FPGA-based implementation of a PE.

For the above configuration, the logic for finding a maximum globally across the 30 PEs can be implemented in a single FPGA. Also, the generation of the transformed co-ordinates as well as hashing the addresses can be performed using 1 FPGA chip by using table look-up. In the above design using 30 PEs, since the total number of PEs is relatively small, the generated hash bin address is distributed directly over a bus to each PE rather than using a tree topology.

5.2 Performance Estimate

Our FPGA-based design has been developed using VHDL and synthesized using Synopsys synthesis tools to generate a gate-level design. Then, Xilinx development tools were used for placing and routing our design using FPGAs. Since all the PEs are identical



	Platform	No of Scene Points	Execution Time
Bourdon [4]	8K-node CM2	*	2000-3000 msec
Rigoutsos [16]	32K-node CM2	200	240 msec
Wang [18]	32-node CM5	256	382 msec
Our Parallel Algorithm	32-node SP-2	256	40 msec
FPGA-based design	32 PEs	256	1.65 msec

* Two real life models (a key and a shaver having 15 feature points and 14 feature points, respectively) and a scene consisting of 28 feature points were used.

Figure 8: Comparison of our FPGA-based implementation with previous parallel implementations on HPC platforms

except for the contents of memory, we synthesized a PE.

Our design runs at a clock rate of 10MHz and the estimated execution time for a probe is 1.65 *milliseconds* using 32 PEs. Each PE consists of an FPGA and a local memory. Two additional FPGA chips and memory modules are required to implement the pre- and post-processing modules.

For the sake of comparison, a sequential algorithm has been implemented using C. On an UltraSPARC Model 140 (143MHz clock, 128M byte memory, 7.44 *SPECint_95*, 10.40 *SPECfp_95*), it takes about 300 *milliseconds* to perform a probe operation. We implemented a parallel algorithm on a 32-node IBM SP-2. Each node of SP-2 had 66MHz processors and 64 up to 512M bytes of memory. The performance benchmarks of the processors were 3.14 *SPECint_95* and 7.50 *SPECfp_95*. In our parallel algorithm, we evenly distribute the hash table (which is vertically partitioned) to 32 processing nodes. However, unlike the previous implementations [18], each processing node has the complete set of vote boxes. Initially, all scene points are sent to each processing node. Each node performs a voting operation locally using the distributed hash table. The results of voting are sent to other processing nodes using an "all-to-all" communication so that all the data corresponding to a vote box is combined and stored in a single PE. Based on the collected votes, local maximums are computed in each processing node and a global maximum is computed over the 32 processing nodes. The implementation on SP-2 was performed using MPI. The execution time was about 40 *milliseconds*. Using one processing node of SP-2, the execution time was about 500 *milliseconds*. Therefore, our FPGA-based solution can achieve a speedup of close to 176 and 24, respectively. A comparison with parallel implementations on HPC platforms is shown in Figure 8.

Note that, the memory requirement per PE in our design is only 4M bytes, whereas the size of local memory in each node of SP-2 is between 64 and 512M bytes. Also, our design assumes 10MHz clock while the processors in SP-2 run at 66MHz.

Although we use 32 PEs, our design is scalable. Note that, as the number of PEs increases, the required number of time multiplexing decreases. However, the internal structure of each PE, such as the width of FPGA-memory datapath, is not affected by the number of PEs used. The upper limit on the number of PEs is obtained by setting $t = 1$ (no time multiplexing). As we mentioned earlier, however, designs with large P are not area efficient due to the number of memory modules used and low FPGA utilization.

Previously, we have assumed that in any hash bin there is no more than one $(model, basis)$ pair. Thus, we need only one bit to mark *UID*$(model, basis)$ in our bit-level hash bin. However, if there are more than one identical $(model, basis)$ pair in a hash bin, then our design can be easily modified to handle this. To allow K identical $(model, basis)$ pairs, $\lceil \log K \rceil$ bits are needed for each $UID(model, basis)$ in a bit-level hash bin. The memory size of the hash table also increases by a factor of $\lceil \log K \rceil$. To generate the bit-level hash bin, the number of identical $(model, basis)$ pairs is counted and is stored in the corresponding $UID(model, basis)$ location. The design for vote boxes needs to be modified. An additional $\lceil \log K \rceil$-bit adder is required for each vote box.

6 Conclusion

We have shown an area-time efficient FPGA-based design for the probe step in geometric hashing. In our design, we first transform a hash table which contains symbolic data into a bit-level representation. By regularizing the data flow and exploiting bit-level parallelism in hardware, our design avoids memory congestion. In addition, the implementation is simplified using a modular approach.

Performance estimates are very encouraging. Given a model database having 1024 models where each model is represented using 16 feature points, a probe operation on a scene consisting of 256 feature points can be performed in 1.65 *milliseconds* on an FPGA-based platform (32 FPGAs and 128M bytes of memory). This result does not assume any distribution of hash bin lengths or scene points. For the same probe operation, a parallel algorithm on a 32-node IBM SP-2 required 40 *milliseconds*, and the earlier implementation required 240 *milliseconds* on a 32K-node CM2 and 382 *milliseconds* on a 32-node CM5.

The work reported here is part of the USC MAARC (Models, Algorithms, and Architectures for Reconfigurable Computing) project for algorithmic configurable computing [13]. In this project, characteristics of state-of-the-art configurable hardware are abstracted to capture their capabilities. Using such representations of configurable devices, system-level models that allow the development of new algorithms for mapping applications on configurable systems are formulated. Using the models and metrics for configurable computing, high-performance algorithms for these architectures are designed.

Acknowledgment

We would like to thank Steve Casselman, David Franklin, and John Schewel of the Virtual Computer Corporation for their assistance in evaluating our designs using their development boards.

References

[1] R. Bittner and P. Athanas, "Wormhole Run-Time Reconfiguration," *International Symposium on Field Programmable Gate Arrays (FPGA'97)*, February 1997.

[2] M. Bolotski, R. Amirtharajah, W. Chen, T. Kutscha, T. Simon, and T. Knight Jr., "Abacus: A High-Performance Architecture for Vision," *International Conference on Pattern Recognition*, 1994.

[3] K. Bondalapati and V. K. Prasanna, "Reconfigurable Meshes: Theory and Practice," *Workshop on Reconfigurable Architectures, IPPS'97*, 1997.

[4] O. Bourdon and G. Medioni, "Object Recognition Using Geometric Hashing on the Connection Machine," *International Conference on Pattern Recognition*, vol. 2, pp. 596-600, 1990.

[5] Y. Chung, "Asynchronous Parallel Algorithms for Irregular Vision Problems on Distributed Memory Machines," *Ph.D. Thesis, University of Southern California*, August 1997.

[6] DARPA, "Moving and Stationary Target Acquisition and Recognition (MSTAR) Program," *http://yorktown.dc.isx.com/iso/battle/mstar.html.*

[7] A. Dandalis and V. K. Prasanna, "Fast Parallel Implementation on DFT using Configurable devices," *7th International Workshop on Field-Programmable Logic and Applications*, 1997.

[8] A. DeHon, "Reconfigurable Architectures for General Purpose Computing," *Technical Report, MIT AI Lab*, September 1996.

[9] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, April 1997,

[10] J. Jang, H. Park, and V. K. Prasanna, "A Fast Algorithm for Computing the Histogram on Reconfigurable Mesh," *Frontiers of Massively Parallel Computation*, pp. 244-251, October 1992.

[11] J. Jang and V. Prasanna, "An Optimal Sorting Algorithm on Reconfigurable Mesh," *Journal of Parallel and Distributed Computing*, vol. 25, pp. 31-41, 1995.

[12] Y. Lamdan and H. Wolfson, "Geometric Hashing: A General and Efficient Model Based Recognition Scheme," *International Conference on Computer Vision*, pp. 238-249, 1988.

[13] MAARC Homepage, *http://maarc.usc.edu.*

[14] M. Maresca and H. Li, "Connection Autonomy in SIMD Computer: a VLSI Implementation," *Journal of Parallel and Distributed Computing*, vol. 7, pp. 302-320, 1989.

[15] R. Miller, V. Prasanna Kumar, D. Reisis, and Q. Stout, "Meshes with reconfigurable Buses," *5th MIT Conference on Advanced Research in VLSI*, pp. 163-178, March 1988.

[16] I. Rigoutsos and R. Hummel, "Massively Parallel Model Matching: Geometric Hashing on the Connection Machine," *IEEE Computer*, pp. 33-42, 1992.

[17] F. Tsai, "Using Line Features for Object Recognition by Geometric Hashing," *Technical Report, New York University*, 1993.

[18] C. Wang, V. K. Prasanna, H. Kim, and A. Khokhar, "Scalable Data Parallel Implementations of Object Recognition Using Geometric Hashing," *Journal of Parallel and Distributed Computing*, vol. 21, pp. 96-109, 1994.

Configurable Hardware for Symbolic Search Operations *

Seonil Choi
University of Southern California

Yongwha Chung
Systems Engineering Section
Electronics and Telecommunications Research Institute
Taejon, Korea

Viktor K. Prasanna
Department of EE-Systems, EEB200C
University of Southern California
Los Angeles, CA 90089-2562
{seonil + yongwha + prasanna}*@halcyon.usc.edu*
`http://maarc.usc.edu`

Abstract

Most intermediate and high-level vision tasks manipulate symbolic data. A kernel operation in these vision tasks is to search symbolic data satisfying certain geometric constraints. Such operations are data-dependent and their memory access patterns are irregular.

In this paper, we propose a fast parallel design for symbolic search operations using configurable hardware. Using a pointer array and a bit-level index array, we manipulate the symbolic data and show high performance can be achieved. Depending on the input data, a corresponding search window is calculated and symbolic search operations are performed in parallel. Performance estimates using 16 Xilinx XC6216s and memory modules are very promising. Given 3519 line segments (extracted from an 1024×1024 *pixel image), the operation can be performed in 1.11 milliseconds on our FPGA-based platform. On a Sun UltraSPARC Model 140, the same operation implemented using C takes 690 milliseconds. Although we illustrate our design for a specific search operation, our design technique can be applied to related search operations with minor modifications. Also, it can be ported to other FPGA devices.*

1 Introduction

Vision systems consist of low, intermediate, and high-level tasks, each with different computational characteristics. Over the years, low-level vision tasks have been parallelized using parallel machines or custom VLSI. Since these tasks are characterized by regular, local, and pixel-based computations, such vision tasks can be easily parallelized [1, 2, 22]. However, in parallelizing intermediate- and high-level vision tasks, additional issues must be considered:

- The computations are performed on *symbolic data* (For example, the image data is represented by points, lines, and area).
- The computations are highly *data dependent* (For example, the size and the shape of a search window depend on the input data).

Many vision systems require real-time performance so that they can interact with humans or invoke other machines in real-time. For these vision systems, high performance computing machines such as Cray T3E, IBM SP-2, and Intel Paragon have been used [23]. However, because of the irregular nature of intermediate- and high-level vision tasks, the speedups achieved on these machines are low.

Recently, *configurable computing* ideas [5, 9] have shown attractive speed-ups for many applications. They offer large scale parallelism by exploiting customized hardware. Field Programmable Gate Arrays

[*] This research was performed as part of the MAARC (*M*odels, *A*lgorithms, and *A*rchitectures for *R*econfigurable *C*omputing) project. This work is supported by DARPA Adaptive Computing Systems program under contract no. DABT63-96-C-0049 monitored by Fort Hauchuca.

(FPGAs) are emerging as one of the major configurable devices which offers rapid prototyping, user reconfigurability. and low development cost. However, most research efforts have focused on mapping regular and non data-dependent applications such as convolution operations, median filtering, and FFT [2, 6. 20. 22] onto such devices. Parallelizing symbolic search operations which are irregular and data-dependent operations on FPGAs is challenging since non-trivial design techniques are required.

In this paper. we propose a parallel and configurable solution for a kernel operation in symbolic vision computations. We develop a design technique for *symbolic search*. a kernel operation, which search for symbolic data satisfying certain geometric constraints. For example, in perceptual grouping [12], a set of symbolic data satisfying certain geometric constraints are grouped to form structural hypotheses. In image matching, correspondences between symbolic data extracted from two different images are determined based on geometrical relationships [18].

In our design, we employ a pointer array and a bit-level index array to manipulate the symbolic data and to achieve high performance. Depending on the input data, a corresponding search window is generated and the symbolic search operations are performed in parallel. Although the symbolic search operation involves multiplication and division operations, we obtain an area-efficient design by employing a lookup table. Since the computations are highly data-dependent, we employ a sophisticated load distribution scheme to realize load balancing. Furthermore, in a typical vision system, various symbolic search operations are performed. Our design can be reconfigured dynamically to suit these operations. To the best of our knowledge, there has been no previous work in mapping symbolic vision computations onto FPGAs.

We have synthesized our design using Xilinx XC6216 devices. Using a 10MHz clock, the estimated execution time for symbolic search on an image consisting 3519 line segments (extracted from an 1024 × 1024 pixel image) is 1.11 milliseconds on a platform having 16 FPGAs. The same operation can be performed in 690 milliseconds on a Sun UltraSPARC Model 140 operating at 143MHz.

The rest of the paper is organized as follows. In Section 2, configurable computing is briefly introduced. In Section 3, characteristics of symbolic search operations in vision are explained. Mapping of such symbolic search operations onto configurable hardware is discussed in Section 4. In Section 5, implementation details are presented, and the performance is analyzed. Concluding remarks are made in Section 6.

2 Configurable Computing

Configurable computing has recently gained much attention (See, for example. Reconfigurable Architecture Workshop held annually at International Parallel Processing Symposium [13]). The paradigm of computing in space (i.e., a series of computations on several functional units), as opposed to computing in time (i.e., a series of computations executed in sequence on a single functional unit), is being actively explored. There are several directions in which research is being carried out to realize the potential of configurable computing [10].

The idea of a VLSI array of processors overlaid with a reconfigurable bus system, and an abstract model based on this architecture was proposed in [19]. Based on this initial work, several abstract models of reconfigurable architectures and fast parallel algorithms for many problems have been described in the literature. For example, efficient algorithms for fundamental data movement operations [19], sorting [14], and image processing [15] have been developed on the reconfigurable meshes. There have been several prototype implementations of such abstract models. Such architectures include Abacus [4] and YUPPIE [17].

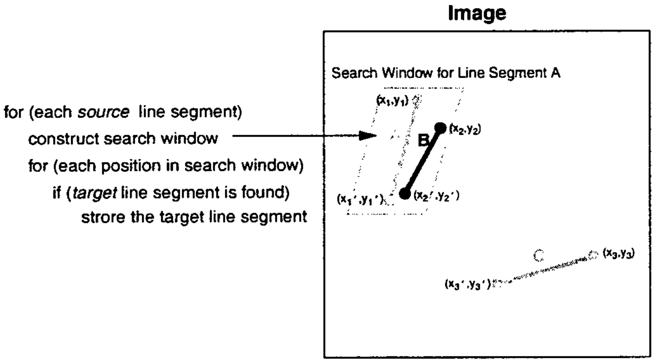
Recently, the advent of Field Programmable Gate Arrays (FPGAs) has given rise to new opportunities in the configurable computing area. Traditionally, FPGAs have been used for rapid prototyping and emulation. The main bottleneck in using these devices as configurable computing engines has been the time for reconfiguration. Current generation devices such as CLAy, XC6200, DPGA etc. alleviate the above problem by providing partial and dynamic reconfigurability [9]. In these devices, it is possible to partially modify the configuration of the device. Some devices permit this partial reconfiguration even while other logic blocks are performing computations. Unlike such fine-grain devices, coarse grain devices in which multiple contexts of the configuration can be stored in the logic block and the context is dynamically switched have been proposed (For example, see [9]). Also, there are efforts under way to develop coupled architectures in which a reconfigurable array and a processor core cooperate on a computational task, exploiting the strengths of both architectures (For example, see [11]). Wormhole run-time reconfiguration has been proposed in [3]. In this approach, as the stream of data moves through the reconfigurable hardware, it rapidly creates and modifies datapaths and computing resources along the way. There have been some efforts to exploit dynamic reconfiguration [5, 16]. In these, the connections are configured based on the input data or the intermediate result of the computation.

Configurable computing provides the ability to redefine the hardware/software boundary in computing systems. This paradigm change results in new computation models, new programming methods, and new approaches to implementation of applications. Some of the greatest gains in this field may well come from providing appropriate abstractions of this technology to algorithm developers and compiler designers to allow them control over hardware that has not been previously exploited [16].

3 Symbolic Search Operations in Vision

Searching symbolic data satisfying certain geometric constraints is a kernel operation used in many intermediate and high-level vision tasks. Such an operation can be modeled as a search operation within a window of the image plane. We assume that the symbolic data is already stored in the image plane before performing the search operations.

To illustrate our idea, we use line segments extracted from raw images as symbolic data [18]. The line segments are represented by their end-point coordinates, lengths, and orientations. Note that low-level processing ensures that line segments do not cross.



```
for (each source line segment)
    construct search window
    for (each position in search window)
        if (target line segment is found)
            strore the target line segment
```

(a) Example code

(b) Search Window

Figure 1: Typical symbolic search operation.

In Figure 1, we show a typical search operation for line segment A. A search operation is performed within a region on both sides of a *source line segment* to find *target line segments* for further processing. The search operation can refer to either a *grouping* process or an *image matching* process. Details of the grouping process and the image matching process can be found in [12, 18].

Each source line segment has a unique line number, $LID(i)$, associated with it. The end-point coordinate, the orientation, the length, and $LID(i)$ of a source line segment are given as input. Also, the width of search window is specified. The search operations produce the target line segments in the search window. For each target line, a record is output which consists of $LID(i)$ and the mid-point coordinates of the target line segments in the window.

Note that the definition of search windows is different for each vision task. For instance, in Line Folding [12], a region on both sides of the source line segment is searched to find target line segments approximately parallel to it as shown in Figure 1. In Corner Detection [12], however, a fixed size region near two end-points of the source line segment is searched to find target line segments which may jointly form right-angled corners.

4 Parallel Symbolic Search on a Configurable Computing Platform - Key Ideas

In this section, we describe our technique for mapping symbolic search operations onto an FPGA-based platform. To illustrate our idea, in the rest of the paper, the search operation refers to the operation shown in Figure 1(b). The major issues in parallelizing the symbolic search operation are: 1) symbolic data from an image must be manipulated, and 2) data-dependent and irregular memory access pattern must be efficiently handled. We first describe the data structures used in our design to manipulate the symbolic data. Based on these data structures, our architecture for symbolic search operation is shown. Finally, a performance analysis of our design is described.

4.1 Data Structures for Symbolic Search

We employ two data structures: a pointer array and a "bit-level" index array. For an $N \times N$ pixel image , there is a corresponding $N \times N$ pointer array. For every line segment, its mid-point coordinates, (x_m, y_m), in the pixel image is mapped onto the pointer array. If (x, y) is a mid-point of a line segment, then a pointer which points the symbolic data structure of the line segment is stored in the corresponding entry . Such symbolic data structure contains information for the line segment as shown in Figure 2. All other locations in the pointer array are set to "null".

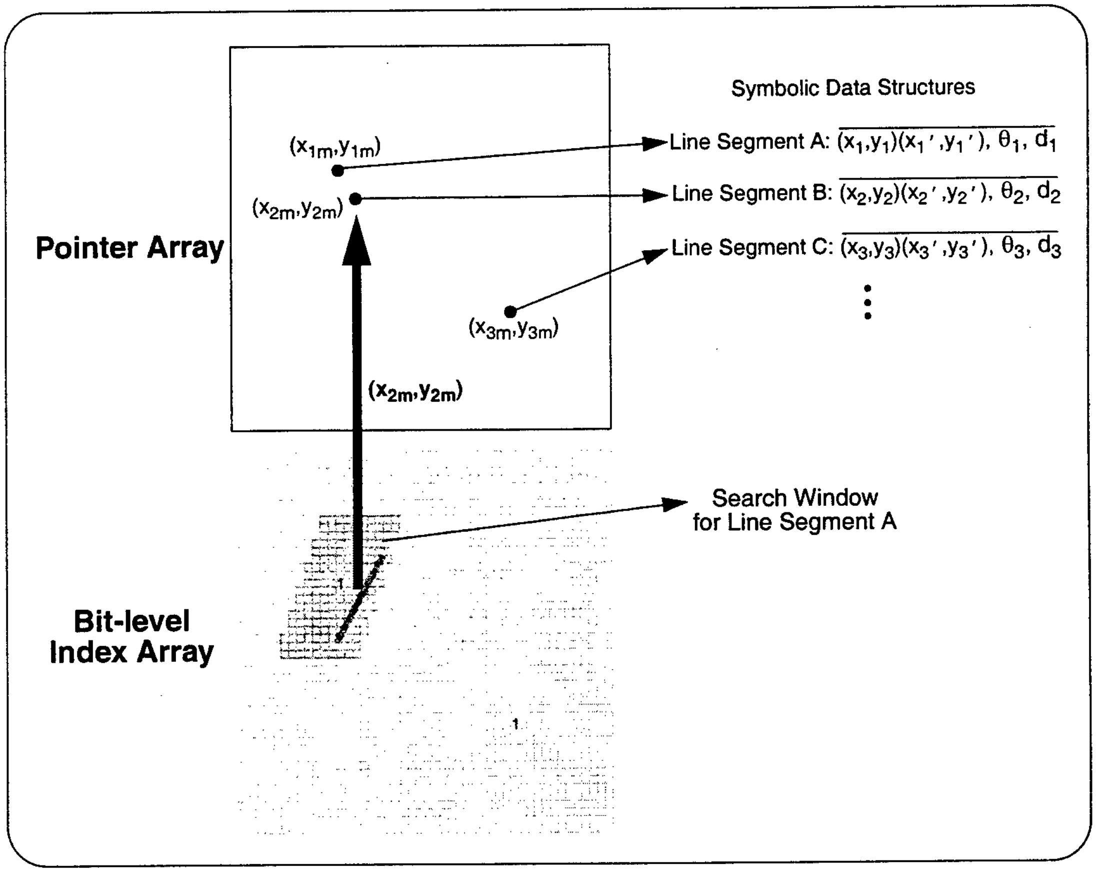

Symbolic Data Structures
Line Segment A: $\overline{(x_1,y_1)(x_1',y_1'), \theta_1, d_1}$
Line Segment B: $\overline{(x_2,y_2)(x_2',y_2'), \theta_2, d_2}$
Line Segment C: $\overline{(x_3,y_3)(x_3',y_3'), \theta_3, d_3}$
(x_{1m},y_{1m})
(x_{2m},y_{2m})
(x_{3m},y_{3m})
(x_{2m},y_{2m})
Pointer Array
Bit-level Index Array
Search Window for Line Segment A
1
1

Figure 2: Symbolic search operation using a pointer array and a bit-level index array.

A search window is constructed by using an end-point (x, y), the orientation (θ), and the length (d) of the source line segment. To perform the search, we initially create a bit-level index array which contains the same data as the pointer array does except pointers. Instead of pointers in the pointer array, the bit-level index array contains a "1" for the existing pointers (i.e., a line segment exists there) and a "0" for all null pointers. Since the mapping between the pointer array and the bit-level index array is one-to-one, all 1's found in the bit-level index array have corresponding pointers to the symbolic data in the pointer array.

4.2 An Architecture for Symbolic Search

In the following, we ignore the initialization cost such as loading the bit-level index array since it is a one time process. We assume that the pointer array is maintained in a host, while the bit-level index array is maintained in the Processing Elements (PEs). Such PE consists of FPGAs and memory modules (See Figure 3). Each local memory has a copy of the bit-level index array. Between the host and all the PEs, there are a configurable network and a FPGA (X_{net}). X_{net} can be connected to any of the PEs through the configurable network. The configurable network consists of Field Programmable Interconnect Devices (FPIDs). X_{net} performs two operations: 1) configure the network, and 2) control the data transfer between the host and the PEs through the network.

The basic strategy of our design is as follows. The host sends the source line segments to X_{net}. It has a FIFO which stores the received source line segments from the host. X_{net} configures the network and sends the line segments to PEs through the network.

Using the source line segment, each PE performs a symbolic search. It generates the corresponding search window, accesses the bit-level index array, and converts the 1's found in the search window into corresponding mid-point coordinates of target line segments. Then the mid-point coordinates are stored in a

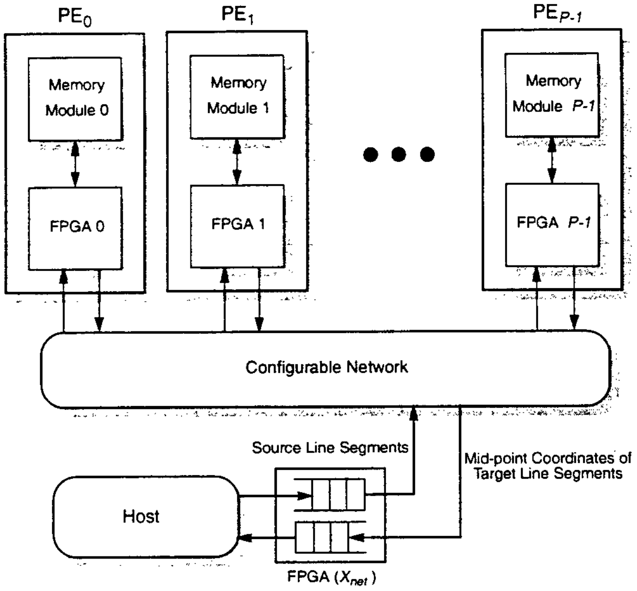


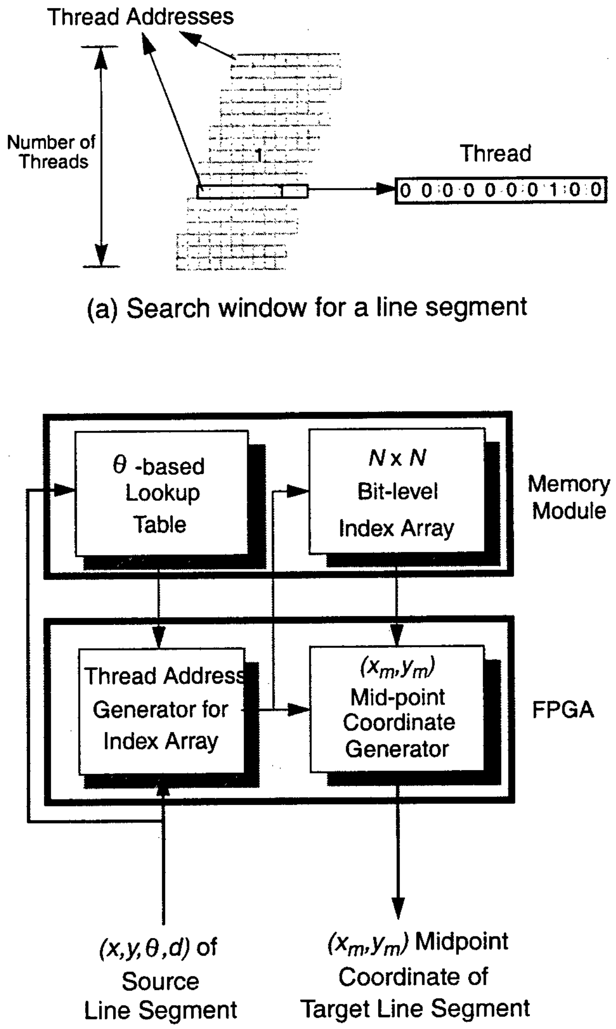
Figure 3: Overall architecture for symbolic search operation.

FIFO of the PE and are sent to X_{net} through the configurable network. X_{net} also has a FIFO for incoming data from the PEs. The host then takes the mid-point coordinates from X_{net} and performs further processing depending on the vision task to be executed.

Figure 4(a) shows a search window. It is a parallelogram with a four-pixel width on both sides of each source line segment. The search window consists of several rows of equal length. The search operation is performed for each row. We define each such row to be a *thread* and its starting coordinate in the bit-level index array to be the *thread address*. The search operation is iterated for each thread in the search window and it consists of the following four steps:

1. Calculate the thread address in the bit-level index array.
2. Read the data corresponding to the thread from the bit-level index array.
3. Search for "1"s in the thread.
4. Calculate the mid-point coordinates for the detected "1"s in a thread.

Figure 4(b) shows the organization of the PEs The input data, (x, y, θ, d), is sent to a PE (e.g. source line segment A in Figure 1). The number of threads in the search window is calculated from the values of θ and d. To calculate the thread address for the first row, a θ-based lookup table is used. The table also contains information needed to generate thread addresses for subsequent rows (i.e., amount of shift per subsequent row). Since the symbolic search operation depends on (x, y, θ, d) of the source line segment, arithmetic computations such as multiplication or division are required to calculate the thread addresses. However, the size and shape of each search window is a function of θ and d of the source line segment. Therefore, instead of implementing complex logic to perform arithmetic using FPGAs, the thread address generation logic is


Thread Addresses
Number of Threads
1
Thread
0 0 0 0 0 0 0 1 0 0
θ -based Lookup Table
N x N Bit-level Index Array
Memory Module
Thread Address Generator for Index Array
(x_m, y_m) Mid-point Coordinate Generator
FPGA
(x, y, θ, d) of Source Line Segment
(x_m, y_m) Midpoint Coordinate of Target Line Segment

(a) Search window for a line segment

(b) Organization of the Processing Element

Figure 4: Search window and processing element for symbolic search operation.

implemented by the θ-based lookup table and a simple logic to perform arithmetic. The implementation details are explained in Section 5.

Once the thread address is generated, the thread is read from the bit-level index array. Using the thread and the thread address, the mid-point coordinate of the target line segment (e.g., the mid-point coordinate (x_{2m}, y_{2m}) of line segment B in Figure 2) is computed.

Since the workload of a PE depends on the length of the line segment, a search operation assigned to a PE can be completed earlier or later than that of other PEs. In order to handle this, we employ the following load distribution scheme:

1. The host scans the list of search operations and finds the smallest task in terms of the area of the search window. The smallest task (measured in terms of the area of the search window) is considered as one operation unit, OU.

2. The search windows are partitioned and assigned to PEs using OU as a basic unit. For example, if $OU = 100$ pixels and a search window is 350 pixels in area, the search is assigned to 4 PEs.

The execution time for a symbolic search operation using P PEs can be analyzed as follows. We assume that a memory access and an arithmetic or logic operation (such as **ADD, MUX, COMPARE**) can be performed in unit time. Let S denote the total number of source lines. Let A_i denote the area of the search window for a source line segment. Let $A_{min} = min\{A_i\}$. The serial execution time is $O(\sum_{i=1}^{S} A_i) = O(\sum_{i=1}^{S} \lceil \frac{A_i}{A_{min}} \rceil) \times t_{min}$. In a multiple PE configuration, Step 1 above takes $O(S)$ time. Therefore, the execution time for a symbolic search operation of the proposed solution using P PEs is

$$O\left(\left[S + \frac{1}{P} \times (\sum_{i=1}^{S} \lceil \frac{A_i}{A_{min}} \rceil)\right] \times t_{min}\right),$$

where t_{min} is the time to execute an operation unit in a PE.

The above analysis assumes that the PEs do not starve due to overheads in load distribution. Also, the above time does not include the time for collecting the target line segments.

5 Implementation Details and Performance Estimate

In this section, we first discuss various issues in implementing the design developed in Section 4 on an FPGA-based platform. Then, we describe a design using Xilinx 6216 FPGAs. Our design is motivated to achieve large speed-ups for typical size of images used by the vision community. We have chosen not to perform device dependent optimizations to improve performance. Figure 5 shows our development environment. The design was synthesized using Synopsis FPGA compiler. Place and route was performed using Xilinx tools (XACT Development System) to create a configuration file for an FPGA on a Sun Ultra Enterprise server. The configuration file is downloaded onto the FPGA development board which consists of FPGAs and memory modules. The board is connected to a PC through PCI Local Bus.

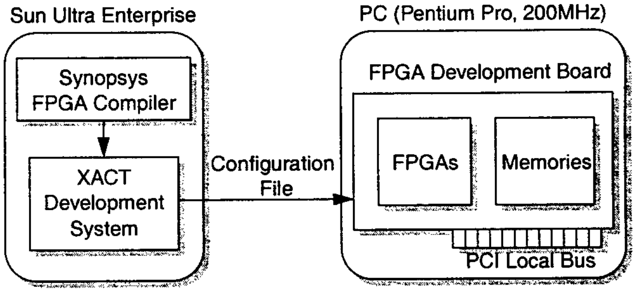


Figure 5: Our development environment.

Figure 6 shows the source line segments extracted from an 1024 × 1024 modelboard image and the corresponding search windows for each line segment. The number of source line segments was 3519, and the symbolic search operation was performed within the region on both sides of each line segment with a four-pixel width to find target line segments.

Both the address generator for the bit-level index array and the mid-point coordinate generator fit in one XC6216. A 512KB memory module was used to store the bit-level index array and the θ-based lookup table.

5.1 Implementation Details

In this section, we discuss various issues in implementing the design. We first show organization of a bit-level index array and a θ-based lookup table. Finally, a design for a PE using an FPGA is shown.

When the bit-level index array is stored in the local memory of a PE, the shape of the search window is considered. Figure 7 shows the shapes of the search windows depending on θ. To ease the memory access patterns of the search windows, two bit-level index arrays are employed in each PE: one of

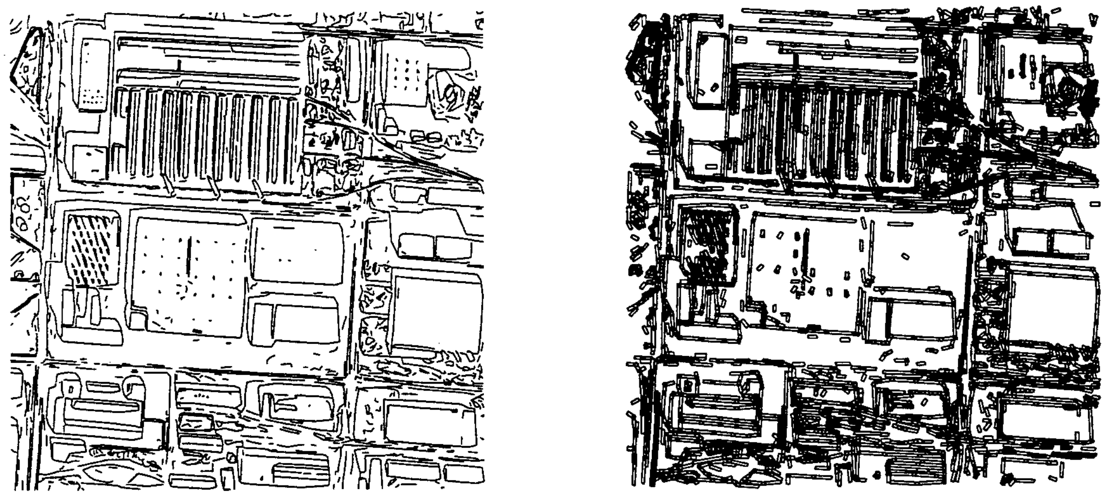


Figure 6: Extracted line segments from an 1024 × 1024 Modelboard image (left) and search windows generated by a symbolic search operation (right).

them, BLI_R, stores the index array in row-major order and the other, BLI_C, stores it in column-major order. If $45° \leq \theta \leq 135°$, the row-major index array is used to access the threads. If $0° \leq \theta < 45°$ or $135° < \theta \leq 180°$, the column-major index array is used. Note that the thread which was defined in Figure 4(a) is defined to be a vertical line segment in the case of column-major index array.

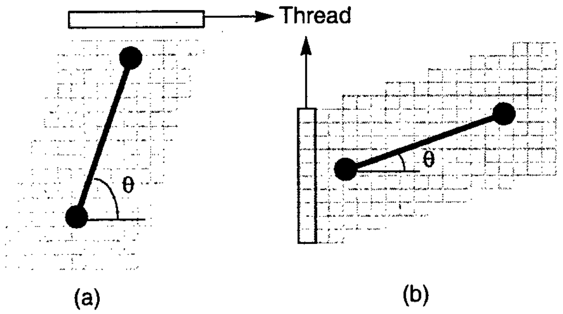
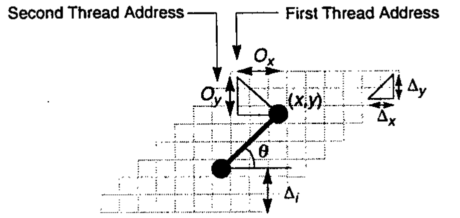


Figure 7: Shapes of search windows depending on θ, (a) $45° \leq \theta \leq 135°$, (b) $0° \leq \theta < 45°$ or $135° < \theta \leq 180°$.

Another design issue is organization of the θ-based lookup table. For a given line segment, the height and the width of its search window is determined using θ and the length of the line segment. The search window is stored in the index array. To read the threads in a search window, thread addresses are calculated using the θ-based lookup table and a simple logic for arithmetic.



	Description
O_x	x axis offset from the end-point of a line segment to the beginning of the first thread
O_y	y axis offset from the end-point of a line segment to the beginning of the first thread
Δ_x	Amount of shift per subsequent thread along x axis
Δ_y	Amount of shift per subsequent thread along y axis
$2\Delta_i$	Distance from the end-point of a line segment to the window boundary
M	Mask

Figure 8: Parameters stored in the θ-based lookup table.

Figure 8 shows the parameters of the θ-based lookup table. For a given θ, the first thread address is calculated by adding the end-point coordinate, (x, y) to the offset, (O_x, O_y). This offset is stored in the lookup table. Note that thread addresses can be computed by simply adding the amount of shift per subsequent thread, (Δ_x, Δ_y). Since (Δ_x, Δ_y) depends on the value of θ, we also keep the values of (Δ_x, Δ_y)

in the θ-based lookup table. We can notice that for a thread in the row-major index array, $\Delta_y = 1$ and Δ_x depends on θ (See Figure 7(a)). However, for the thread in the column-major index array, $\Delta_x = 1$ and Δ_y depends on θ (See Figure 7(b)). To compute the number of threads (i.e., the height of the search window), the height of a line segment, d, and Δ_i are used.

Finally, in the lookup table, a mask, M, is stored. A 32-bit data bus was used between the FPGAs and the memory modules in our design. However, the length of the threads varies depending on θ. For a given thread address, a 32-bit data is read from the bit-level index array and a thread is extracted by masking the data using M.

Using the θ-based lookup table and two bit-level index arrays, the implementation details of a PE are shown in Figure 9.

A source line segment comes from X_{net} through the configurable network. The end-point coordinate (x, y), the orientation (θ), and the length (d) of the source line segment are fed to a PE. To compute the first thread address, the offset, (O_x, O_y), is added to the end-point coordinate, (x, y). The θ test unit generates an additional bit for the first thread address to decide on one of the two bit-level index arrays to be accessed. Using this thread address, the bit-level index array is accessed and the data is fed to the mask operation unit. To obtain a thread, the data from the bit-level index array is masked out by using M. If the thread has a '1', the distance between the thread address and the '1' in the thread, (Δ_{adj}), is produced. Note that to obtain the mid-point of a target line segment, the thread address is used. By adding Δ_{adj} to the thread address, the mid-point coordinate, (x_m, y_m), is finally extracted and is fed to a FIFO.

To read the next thread, its address is generated by adding the amount of shift per subsequent thread, (Δ_x, Δ_y), to the previous thread address. For a given search, the amount of computation depends on the number of threads. It can be computed by adding the length of the line segment, d, and the distance from the end-point of a line segment to the boundary of the search window, $(2\Delta_i)$. The number of threads determines the number of iterations.

We have explained our design in the case of using the threads in the row-major index array. For a thread in the column-major index array, x and y of the end-point coordinate are exchanged depending on θ. If $0° \leq \theta < 45°$ or $135° < \theta \leq 180°$, we need to exchange them to access the correct address in the column-major index array. The θ test unit gives a control signal for the exchanger.

5.2 Performance Estimate

Our FPGA-based design has been developed using VHDL and synthesized using Synopsys synthesis tools to generate a gate-level design. Then, Xilinx development tools were used for placing and routing our design using FPGAs. Since all the PEs are identical, we synthesized a PE. Xilinx XC6216 devices were used. Our design operates using a 10MHz clock and the execution time for the symbolic search operation with 3519 source line segments is estimated as 1.11 milliseconds on our 16-PE FPGA-based platform. Each PE consists of one XC6216 and one memory module of size 512KB. One additional FPGA is used for controlling the configurable network and the FIFOs. We have assumed that the θ-based lookup table and the bit-level index arrays are already stored in the memory module.

For the sake of comparison, a sequential algorithm was implemented using C. On a Sun UltraSPARC Model 140 (143MHz clock, 128MB memory, 7.44 *SPECint*_95, 10.40 *SPECfp*_95), it took 690 milliseconds to perform the search operation. We used "gcc" compiler with level-2 optimization (-O2) for compiling our sequential code. Therefore, the speed-ups of close to 622 can be achieved (See Figure 10). The same operation took 1820 milliseconds on an IBM RS/6000 (67MHz clock, 32MB memory, 3.14 *SPECint*_95 and 7.50 *SPECfp*_95).

Note that, the memory requirement in our design is only 8MB, whereas the size of the main memory in a Sun UltraSPARC was 128MB. Also, our design operates at 10MHz while the UltraSPARC operates at 143MHz.

Platform	*Execution Time* *(for 3519 Line Segments)*
IBM RS/6000 (67 MHz, 32 MB)	1820 msec
Sun UltraSPARC (143 MHz, 128 MB)	690 msec
Our Design using 16 PEs (10 MHz, 8 MB)	1.11 msec

Figure 10: Performance comparison between our FPGA-based implementation and a serial implementation on general-purpose machines.

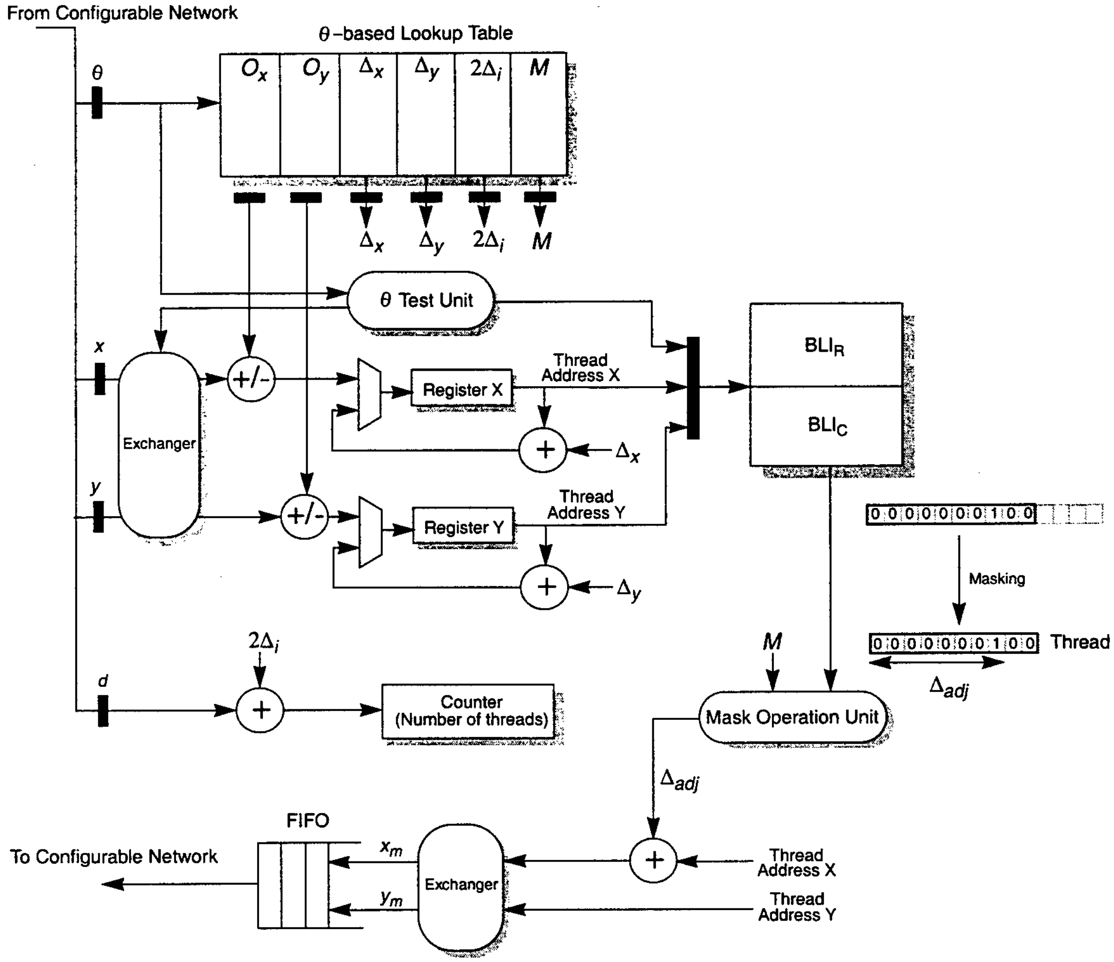


Figure 9: Design for a PE.

6 Conclusion

We have shown a configurable hardware design for parallelizing a symbolic search operation which is a kernel operation in intermediate- and high-level vision.

We first transformed the symbolic data structure into a bit-level representation. Then, depending on the input data, search windows were generated and the search operation was performed in parallel using multiple FPGAs and memory modules. Performance estimates of our design are very encouraging. Given 3519 line segments, speed-ups for the symbolic search operation using 16 PEs was close to 622 over a sequential implementation on Sun UltraSPARC Model 140.

Although we illustrated our design for a specific search operation, our design technique can be applied to many symbolic search operations in intermediate and high-level vision to satisfy real-time performance requirements. Examples of such vision tasks are hypothesis verification, image matching, perceptual grouping, and stereo matching, among others.

The work reported here is part of the USC MAARC (Models, Algorithms, and Architectures for Reconfigurable Computing) project for algorithmic configurable computing [16]. In this project, characteristics of state-of-the-art configurable hardware are abstracted to capture their capabilities. Using such representations of configurable devices, system-level models that allow the development of new algorithms for mapping applications onto configurable systems are formulated. Using the models and metrics for configurable computing, high-performance algorithms for these architectures are designed.

Acknowledgment

We would like to thank Steve Casselman, David Franklin. and John Schewel of the Virtual Computer Corporation for their assistance in evaluating our designs using their development boards.

References

[1] P. Athanas and A. Abbott, "Addressing the Computational Requirements of Image Processing with a Custom Computing Machine: An Overview." *Workshop on Reconfigurable Architectures, IPPS '95*, pp. 1-15, 1995.

[2] M. Barros and M. Akil, "Low Level Image Processing Operators on FPGA: Implementation Examples and Performance Evaluation," *International Conference on Pattern Recognition*, pp. 262-267, 1994.

[3] R. Bittner and P. Athanas, "Wormhole Run-Time Reconfiguration," *International Symposium on Field Programmable Gate Arrays*, 1997.

[4] M. Bolotski, R. Amirtharajah, W. Chen, T. Kutscha, T. Simon, and T. Knight Jr., "Abacus: A High-Performance Architecture for Vision," *International Conference on Pattern Recognition*, 1994.

[5] K. Bondalapati and V. K. Prasanna, "Reconfigurable Meshes: Theory and Practice," *Workshop on Reconfigurable Architectures, IPPS'97*, 1997.

[6] D. Buell, J. Arnold, and W. Kleinfelder, *Splash 2*, IEEE Computer Society Press, 1996.

[7] Y. Chung, S. Choi, and V. K. Prasanna, "Parallel Object Recognition on an FPGA-based Configurable Computing Platform," *IEEE Conference on Computer Architectures for Machine Perception*, 1997.

[8] A. Dandalis and V. K. Prasanna, "Fast Parallel Implementation on DFT using Configurable Devices," *7th International Workshop on Field-Programmable Logic and Applications*, 1997.

[9] A. DeHon, "Reconfigurable Architectures for General Purpose Computing," *Technical Report, MIT AI Lab*, September 1996.

[10] R. Hartenstein and V. K. Prasanna (Eds.), "High Performance by Configware," *Workshop on Reconfigurable Architectures, IPPS'97*, ITpress Verlag, 1997.

[11] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97)*, April 1997,

[12] A. Huertas, C. Lin, and R. Nevatia, "Detection of Buildings from Monocular Views of Aerial Scenes using Perceptual Grouping and Shadows," *Image Understanding Workshop*, pp. 253-260, 1993.

[13] IPPS Homepage, *http://www.ippsxx.org.*

[14] J. Jang and V. K. Prasanna, "An Optimal Sorting Algorithm on Reconfigurable Mesh," *Journal of Parallel and Distributed Computing*, Vol. 25, pp. 31-41, 1995.

[15] J. Jang, H. Park, and V. K. Prasanna, "A Fast Algorithm for Computing the Histogram on Reconfigurable Mesh," *Frontiers of Massively Parallel Computation*, pp. 244-251, October 1992.

[16] MAARC Homepage, *http://maarc.usc.edu.*

[17] M. Maresca and H. Li, "Connection Autonomy in SIMD Computer: a VLSI Implementation," *Journal of Parallel and Distributed Computing*, Vol. 7, pp. 302-320, 1989.

[18] G. Medioni and R. Nevatia, "Matching Images Using Linear Features," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 6, No. 6, pp. 675-685, 1984.

[19] R. Miller, V. Prasanna Kumar, D. Reisis, and Q. Stout, "Meshes with reconfigurable Buses," *5th MIT Conference on Advanced Research in VLSI*, pp. 163-178, March 1988.

[20] G. Panneerselvam, P. Graumann, and L. Turner, "Implementation of Fast Fourier Transforms and Discrete Cosine Transforms on FPGAs," *Field-Programmable Logic and Applications*, pp. 272-281, 1995.

[21] V. K. Prasanna, *Parallel Architectures and Algorithms for Image Understanding*, Academic Press, 1991.

[22] N. Ratha, A. Jain, and D. Rover, "Convolution on Splash 2," *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 204-213, 1995.

[23] C. Wang, P. Bhat, and V. K. Prasanna "High-Performance Computing for Vision," *Proceedings of the IEEE*, pp. 931-946, July 1996.

IEEE Symposium on Field-Programmable Custom Computing Machines, April 1998.

Mapping Homogeneous Computations onto Dynamically Configurable Coarse-Grained Architectures*

Andreas Dandalis and Viktor K. Prasanna
Department of Electrical Engineering-Systems
University of Southern California, Los Angeles, CA 90089-2562
{dandalis, prasanna}@usc.edu, http://maarc.usc.edu/

The Problem

Conventional FPGAs are fine-grained architectures, mainly designed for implementing bit-level tasks and random logic functions. Their performance is limited for computationally demanding applications over large word length data. A highly promising avenue that is being explored by many research groups is coarse-grained configurable architectures. These architectures are datapath-oriented structures and consist of a small number of powerful, word-based configurable processing elements (*PEs*). Such architectures can result in greater computational efficiency and high throughput for coarse-grained computing tasks.

The key for achieving high performance solutions is efficient mapping of tasks onto above architectures. In addition to achieving high computational rates, partitionability is a desirable characteristic of the mapping. Moreover, the computational efficiency must scale with the size of the architecture. Finally, it must result in a simple *PE* structure, regular/balanced dataflow and sustainable I/O requirements so that it can be realized in hardware.

In this paper we show a methodology for deriving dynamic computation structures for *2 dimensioned* homogeneous computations. Homogeneous computations lead to all *PEs* having the same functionality. The derived dynamic structures match the datapath-oriented nature of coarse-grained architectures and lead to efficient mapping schemes. Our solutions require constant I/O and smaller amount of local memory/*PE* compared with known solutions.

Our Approach

Our design methodology is based on a simple model of typical coarse-grained configurable architectures. It is a configurable linear array of identical powerful *PEs*. Adjacent *PEs* are connected in a pipelined fashion with word parallel links. The data/control channels can be configured to communicate with each other at different speeds (*datapath configuration*). The *PEs* can also be configured to have different internal structures (*functional configuration*). This can be exploited to map heterogeneous computations [3] where different computations are performed by different sets of *PEs*. The parameters of the model include p, the number of *PEs*, m the amount of total memory in each *PE* and w the data word width. An external controller/memory system provides the required data and control signals and can store the results computed by the array. *I/O* operations can only be performed at the left and right boundary of the array.

The key idea of our approach is dynamic datapath configuration. By configuring the datapaths, we essentially schedule the dataflow along the array and the computations that each data stream participates in. The data operands are transported and aligned through the array via differential speed data channels. Furthermore, the functionality of the *PEs* is changed by reconfiguring the connectivity (datapaths) among the functional units and local memories. The design methodology consists of three major steps:

- *Step 1 (Dynamic datapath configuration):* First, we derive a full size solution for the given algorithm. The solution does not depend on the parameters p and m. The derived computation structure is a linear array and determines:
  – Basic *PE* structure - Speed of Data/Control channels.
  – Control/Communication scheme for the array.
  – Schedule of Data/Control streams.
- *Step 2 (Memory management):* Efficient utilization of the local memory in the *PEs* can improve the solution derived in Step 1 with respect to time performance and/or the needed resources. Again, at this stage the solution does not depend on the parameters p and m.
- *Step 3 (Partitioning):* Finally, partitioned schemes for the solution in *Step 2* are derived. The solution now depends on the parameters p and m. This is a critical step that "fits" the solution derived in *Step 2* into the target architecture.

An Example: Matrix Multiplication

To illustrate our ideas we consider $N \times N$ matrix multiplication. Due to space limitations, we show the final partitioned solution. Although we assume $N \geq p$, similar solutions can be derived for smaller problem sizes.

For performing $C = A \times B$, where each matrix is of size $N \times N$, p *PEs* with $m = 2p$ storage in each *PE* are

[*] This research was performed as part of the MAARC project. This work is supported by the DARPA Adaptive Computing Systems program under contract no. DABT63-96-C-0049 monitored by Fort Hauchuca.

required. Using such an array, a subproblem of size ($p \times N$) $\times$ ($N \times p$) can be solved. We can perform a $N \times N$ matrix multiplication using at most $\lceil N/p \rceil^2$ iterations of the subproblem. In each *PE* (see Figure 1a), p rows from a ($p \times N$) submatrix of A commute with exactly one column from a ($N \times p$) submatrix of B resulting in p elements of matrix C.

Submatrices $A_{p \times N}$ are fed into the array through a *slow* data channel (2 clock cycles delay per *PE*) in column major order. Submatrices $B_{N \times p}$ are fed through a *fast* data channel (1 clock cycle delay per *PE*) in row major order. A *fast* output data channel is used to carry the results from the local memories out of the *PEs*.

Two p-word banks of local memory are used for storing the intermediate results. During each iteration, the contents of one memory bank are uploaded onto the *OUT* data channel, while the intermediate results are stored in the other one. The uploading mechanism is performed in a repetitive manner along the array, starting from the leftmost *PE*.

Using the speed differential between the data channels and the uploading mechanism, regular data flow and full utilization of the array are achieved. The regular structure of the computation makes the control of the array simple and uniform. The control signals travel through the array via *fast* and *slow* channels as well.

The clock cycle is determined by the multiply-add-update operation performed in each *PE* (see Figure 1a). By pipelining the datapath for this operation, the clock cycle time can be decreased. $\lceil N/p \rceil^2 pN + p^2 - 1$ clock cycles are required to perform $N \times N$ matrix multiplication (p results are computed per clock cycle on the average). On the average, the derived structure requires 2 external memory accesses and p local memory accesses per clock cycle (i.e. one local access in each *PE*).

For the sake of illustration, we apply the above approach to perform matrix multiplication on RaPiD (a coarse-grained configurable architecture) [2]. Our solution results in full utilization of the *PEs* and asymptotically the same time performance as in [2]. However, our mapping uses $m = 2p$ memory/*PE* for storing intermediate results. In [2] additional local memory is required for storing data operands as well. It is easy to verify that our solution reduces the total number of local memory accesses. In addition, it can result in potential area savings.

Figure 1b compares the memory performance of the two approaches. The difference in the asymptotic average # of memory accesses per cycle is drawn as a function of $\alpha = \frac{m}{p}$. The value $m = \alpha p$ is the amount of local memory in each of the p *PEs*. Note that, $\alpha = 2$ and 6 represent the minimum and the available size of the local memory in [2]. Figure 1c shows the ratio of the local memory used in [2] over that needed in our approach as a function of α. Our approach reduces the overhead due to local memory accesses. For $\alpha > 3$, the solution in [2] requires less external memory accesses than our approach. However, in order to achieve this, larger amount of local memory is used (see Figure 1c) resulting in increased area and control overheads. In addition, the time performance remains the same since the execution time depends on the number of available multipliers in both the approaches.

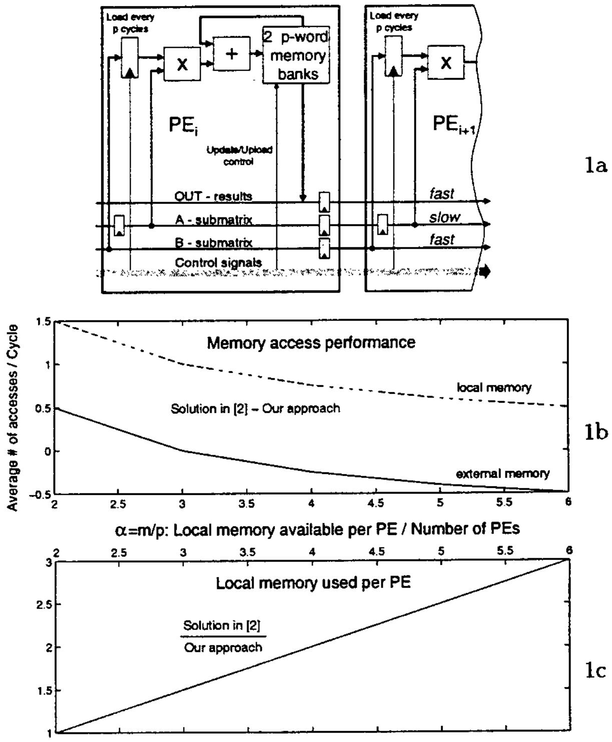


Figure 1: PE Organization and memory performance.

Conclusions

By using our design methodology, scalable mapping schemes having high computational rates can be derived. These schemes do not depend on the problem size and require constant I/O. These require lower amount of local memory/*PE*. Also, it results in lower number of local memory accesses compared with known solutions.

Our methodology can also lead to efficient mapping for various other matrix-oriented computations, including *2-D* DCT and *2-D* FIR. Similar techniques can be applied to *1-D* problems as well [1]. A technique to perform *DFT* using the Arithmetic Fourier Transform (which uses less number of multiplications) is shown in [1]. Techniques to map heterogeneous computations are shown in [3].

References

[1] A. Dandalis and V. K. Prasanna, "Fast Parallel Implementation of DFT using Configurable Devices", *Int. Workshop on Field Programmable Logic and Applications*, Sep. 1997.

[2] C. Ebeling, D. C. Cronquist, P. Franklin and C. Fisher, "RaPiD - A configurable computing architecture for compute-intensive applications", *Technical Report UW-CSE-96-11-03*, Nov. 1996.

[3] A. Dandalis and V. K. Prasanna, "Mapping Heterogeneous Computations onto Dynamically Configurable Coarse-Grained Architectures", *Manuscript*, Jan. 1998.

International Workshop on Field Programming Logic and Applications, September 1998.

Space-efficient Mapping of 2D-DCT onto Dynamically Configurable Coarse-Grained Architectures [*]

Andreas Dandalis and Viktor K. Prasanna

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, CA 90089-2562, USA
Tel: +1-213-740-4483, Fax: +1-213-740-4418
{dandalis, prasanna}@usc.edu
http://maarc.usc.edu/

Abstract. This paper shows an efficient design for 2D-DCT on dynamically configurable coarse-grained architectures. Such coarse-grained architectures can provide improved performance for computationally demanding applications as compared to fine-grained *FPGAs*. We have developed a novel technique for deriving computation structures for *two dimensional* homogeneous computations. In this technique, the speed of the data channels is dynamically controlled to perform the desired computation as the data flows along the array. This results in a space efficient design for 2D-DCT that fully utilizes the available computational resources. Compared with the state-of-the-art designs, the amount of local memory required is reduced by 33% while achieving the same high throughput.

1 The Problem

Coarse-grained configurable architectures consist of a small number of powerful configurable units. These units form datapath-oriented structures and can perform critical word-based operations (e.g. multiplication) with high performance. This can result in greater efficiency and high throughput for coarse-grained computing tasks.

The *2D-DCT* of a $N \times N$ image U is defined as [8]:

$$v(k,l) = \sum_{m,n=0}^{2N-1} \sum_{m,n=0}^{2N-1} u(m,n) c_{k,l}(m,n)$$

[*] This research was performed as part of the MAARC project (Models, Algorithms and Architectures for Reconfigurable Computing, http://maarc.usc.edu). This work is supported by the DARPA Adaptive Computing Systems program under contract no. DABT63-96-C-00049 monitored by Fort Hauchuca.

where $C = c_{k,l}(m,n)$ is the $N \times N$ cosine transform matrix and $0 \leq k, l < N$.

The 8×8 *2D-DCT* is a fundamental computation kernel of still-picture and video compression standards. Efficient solutions (mapping schemes) must achieve high computational rates. In addition, since in coarse-grained configurable architectures, the functional units are word-based, the amount of chip area available for local storage is limited. Hence, the designs should be space-efficient as well.

In this work we derive computation structures for 8×8 *2D-DCT*. These structures match the datapath-oriented nature of the target architectures and lead to efficient mapping. The characteristics of the derived structures are:

- Scalability with the size of *2D-DCT*,
- Partitionability with the image size $N \times N$,
- Maximum utilization of computational resources, and
- Space efficiency.

2 Our Approach

The design methodology is based on a model of a configurable linear array. The array consists of identical powerful *PEs*, connected in a pipeline fashion with word parallel links between adjacent *PEs*. The data/control channels can be configured to communicate with each other at different speeds. The *PEs* can also be configured to have various internal organization (*functionality*). The parameters of the model include p, the number of *PEs*, m the amount of total memory in each *PE* and w the data width. An external controller/memory system is assumed to provide the required data/control signals and can store results computed by the array. *I/O* operations can only be performed at the left and right boundary of the array. Several research groups are currently building such configurable coarse-grained architectures [2, 6, 7, 9].

The key idea of our approach is dynamic interaction between data streams. The dataflow is determined by the speed and the connectivity of the datapaths. The configuration of the data paths schedules the computations to be performed onto a data stream along the array. Furthermore, the functionality of the *PE*s can be changed by reconfiguring the connectivity among their functional units, local memories and data channels.

The parameters of the target architecture p, m, and w are given as input and are assumed to be independent of the problem size. The three major steps of the approach are:
Step 1 (Algorithm selection): Selection of an appropriate algorithm for the considered task.
Step 2 (Primitive structure): Derivation of a "primitive" computation structure which is independent of the parameters p and m. The derived multirate linear array determines:

- Internal structure of the *PE*,
- Control/communication scheme,
- Schedule of Data/Control streams, and
- Speed of Data/Control channels.

Step 3 (Partitioning): "Fitting" the solution obtained in *Step 2* into the target architecture. Partitioned schemes are derived by efficient utilization of the local memory in the *PEs*. The new solution depends on the parameters p and m.

3 2D-DCT on Coarse-grained Architectures

Given a $N \times N$ image U, we partition it into 8×8 submatrices ($U_{8\times8}$) and then compute the *2D-DCT* of each of them as $V_{8\times8} = CU_{8\times8}C^T = [C(CU_{8\times8})^T]^T$ [8]. C is the 8×8 cosine transform matrix. This approach reduces the complexity of the problem from $O(N^4)$ to $O(N^3)$ [8]. It also leads to decomposition of the *2D-DCT* into two *1D-DCT* blocks. Each block performs the *1D-DCT* transform and transposes the computed matrix as well.

Correspondingly, the *2D-DCT* array (Fig. 1a) consists of two identical computational blocks of 8 *PEs* each. Each block computes and transposes the result of a 8×8 matrix multiplication (Fig. 1a). Data and control travel through the *PEs* via differential speed channels (*fast*/*slow* channels). The regular nature of the computation makes the control of the array uniform and simple.

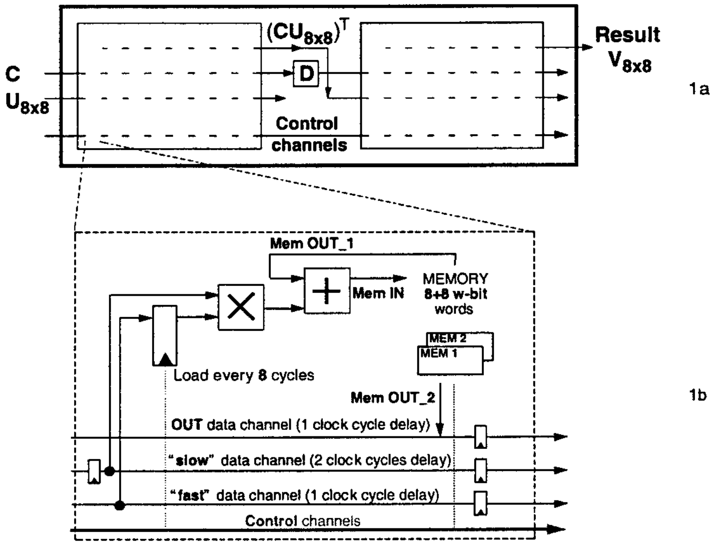


Fig. 1. The 2D-DCT array (1a) and the PE organization (1b).

Figure 1b shows the *PE* structure. In each *PE*, one column of the input submatrix commutes with all the 8 rows of the cosine transform matrix C. Thus, 8 results are computed per *PE*. Two 8-word banks of local memory are used per *PE* for storing the intermediate results. The contents of each memory bank are updated for 64 cycles alternatively. The memory contents are read in order 8 times during this time period via *Mem OUT_1*. The values read by *Mem OUT_1* get updated by the incoming data values and are stored back via

Mem IN. The memory bank that is not updated, uploads in order its contents on *OUT* via *Mem OUT_2*. Each memory bank is *flushed* every 128 cycles for 8 consecutive cycles. The memory contents are uploaded in a repetitive manner along the array, starting from the leftmost to the rightmost *PE*.

The cosine transform matrix C is fed into the slow data channel (Fig. 1a) in column-major order. The submatrix $U_{8\times8}$ is fed into the fast data channel in row-major order. Figure 1a shows the way in which the data and control streams are transferred between the two computational blocks. The matrices C and $(CU_{8\times8})^T$ are fed to the second block via the *slow* and *fast* data channels respectively. In addition, a delay (D) of 56 clock cycles is added to the datastream of matrix C. By inserting this delay, the datastreams of the two matrices are synchronized at the input of the second block. This synchronization can also be performed by using a new data channel. This new channel ***transports*** matrix C from the leftmost *PE* of the array to the second block. The delay is now distributed among the first 8 *PEs* (9 clock cycles per *PE*). Similar delays are inserted in the control channels as well.

The order in which the results are computed in each block, assures that the output matrix is the desirable one. No additional block for transposing the result of matrix multiplication is needed. Moreover, the uploading mechanism leads to full utilization of the computational resources of the array. *I/O* operations are performed only at the right/left boundary of the array. Thus, the required *I/O* bandwidth is constant.

The latency of the resulting array is 144 clock cycles while its throughput is same as the clock rate of the array. The multiply-add-update operation inside the *PEs* (Fig. 1b), is the most time consuming part of the computation and determines the clock rate of the array. By pipelining the datapath of this operation, the clock rate can be increased up to the computational rate of the slowest functional unit (multiplier, adder, memory). By replacing matrix C with its transpose C^T, the same structure can compute the inverse *2D-DCT* transform without any additional changes. On the average, the array requires 3 external memory accesses and 18 local memory accesses per clock cycle (i.e. 1.13 local accesses in each *PE* on the average).

For the sake of illustration, we compare our solution with that proposed for RaPiD (a coarse-grained configurable architecture) [6]. Both solutions are based on the row/column decomposition of *2D-DCT* to two *1D-DCTs*. The time performance is asymptotically the same. The key difference is the amount of local memory required and the number of local memory references.

In [6], matrix C is stored locally among the *PEs* and additional memory is needed for matrix-transpose operations. A total of 384 words of memory is used (i.e. $m = 24$ per *PE*). On the average, 20 local memory accesses/cycle are required. Our solution uses 256 words of local memory for storing intermediate results. On the average, it requires 18 local memory accesses/cycle. However, it also requires 3 (compared with 2 in [6]) external memory accesses/cycle. This is not a limiting factor since RaPiD can support at most two reads and one write to the external memory per cycle [6].

4 Conclusion

Coarse-grained configurable architectures offer the potential for high computational efficiency and throughput for coarse-grained computing tasks. In this paper, a *2D-DCT* structure for such architectures was derived, using our dynamic data path interaction technique. Space efficiency, high throughput and constant I/O requirements, are the main advantages of the derived array.

Our technique is based on dynamic interaction of data streams via differential speed data channels. It also leads to scalable and partitioned mapping schemes for similar matrix-oriented computations (e.g. matrix-multiplication [4]). These schemes achieve high computational rates while the required I/O bandwidth is constant (independent of the size of the array). Moreover, their space efficiency makes them an attractive solution for coarse-grained configurable architectures.

The work reported here is part of the USC MAARC project. This project is developing algorithmic techniques for realising scalable and portable applications using configurable computing devices and architectures. Computational models and algorithmic techniques based on these models are being developed to exploit dynamic reconfiguration. In addition, partitioning and mapping issues in compiling onto reconfigurable hardware are also addressed [1, 3, 4, 5].

References

1. K. Bondalapati and V. K. Prasanna, "Mapping Loops onto Reconfigurable Architectures", *Int. Workshop on Field Programmable Logic and Applications*, Sep. 1998.
2. D. C. Chen and J. M. Rabaey, "A Reconfigurable Multiprocessor IC for Rapid Prototyping of Algorithmic-Specific High-Speed DSP Paths", *IEEE Journal of Solid-State Circuits*, 27(12):1985-1904, Dec. 1992.
3. Y. Chung, S. Choi and V. K. Prasanna, "Parallel Object Recognition on an FPGA-based Configurable Computing Platform", *Int. Workshop on Computer Architectures for Machine Perception*, Oct. 1997.
4. A. Dandalis and V. K. Prasanna, "Mapping Homogeneous Computations onto Dynamically Configurable Coarse-Grained Architectures", *IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 1998.
5. A. Dandalis and V. K. Prasanna, "Fast Parallel Implementation of DFT using Configurable Devices", *Int. Workshop on Field Programmable Logic and Applications*, Sep. 1997.
6. C. Ebeling, D. C. Cronquist, P. Franklin and C. Fisher, "RaPiD - A configurable computing architecture for compute-intensive applications", *Technical Report UW-CSE-96-11-03*, Nov. 1996.
7. R. W. Hartenstein, R. Kress, H. Reinig, "A Scalable, Parallel, and Reconfigurable Datapath Architecture", *6th Int. Symposium on IC Technology, Systems & Applications*, Sept. 1995.
8. A. K. Jain, "Fundamentals of Digital Image processing", Prentice-Hall Inc., Englewood Cliffs, NJ, 1989.
9. A. Agarwal et al., "Baring it all to Software: The Raw Machine", *MIT/LCS Technical Report TR-709*, March 1997.

This article was processed using the LaTeX macro package with LLNCS style

High-Performance Embedded Computing Workshop, September 1998

Mapping Signal Processing Loops onto Reconfigurable Hardware[1]

Kiran Bondalapati and Viktor K. Prasanna
Department of Electrical Engineering Systems
University of Southern California
Los Angeles, CA 90089-2562
{kiran,prasanna}@ceng.usc.edu

Introduction

Configurable systems have evolved from logic emulators and special purpose logic circuits to embedded system components and general purpose application accelerators. Various reconfigurable architectures are being explored by several research groups to develop a general purpose configurable system. Reconfigurable architectures vary from systems which have FPGAs and glue logic attached to a host computer to systems which include configurable logic on the same die as a microprocessor. Such systems-on-a-chip have enormous application potential in high performance embedded computing.

Application development using such configurable hardware still necessitates expertise in low level hardware details. In this paper, we address some of the issues in the development of techniques for automatic compilation of applications. We develop algorithmic techniques for mapping applications in a platform independent fashion.

Reconfigurable architectures with their regular structure, adaptive functionality and fine granularity are well suited for signal processing applications. Regular and repetitive byte-wise or bit-wise operations which occur in signal processing can be mapped onto reconfigurable architectures to achieve high speed-up. Most signal processing applications consist of core routines such as FFT, DCT and QRD among others. Loop constructs which occur in such routines provide an opportunity to develop effective mapping techniques.

We address the problem of mapping a loop construct onto reconfigurable architectures. We define problems, based on the Hybrid System Architecture Model (HySAM), which address the issue of minimizing reconfiguration overheads by scheduling the configurations. A polynomial time solution for generating the optimal configuration sequence for one important variant of the mapping problem is presented.

Loop Synthesis

Computations which operate on a large set of data using the same set of operations are most likely to benefit from configurable computing. Hence, loop structures will be the most likely candidates for performance improvement using configurable logic. Configurations which execute each task can be generated for the operations in a loop. Since each operation is executed on a dedicated hardware configuration, the execution time for the task is expected to be lower than that in software.

Hybrid System Architecture Model (HySAM)

We have developed a parameterized model of a configurable computing system, which consists of configurable logic attached to a traditional microprocessor. The *Hybrid System Architecture* is a general architecture consisting of a traditional RISC microprocessor with additional Configurable Logic Unit(CLU). The architecture consists of a traditional RISC microprocessor, standard memory, configurable

[1] This work was supported by the DARPA Adaptive Computing Systems Program under contract DABT63-96-C-0049 monitored by Fort Hauchuca.

logic, configuration memory and data buffers communicating through an interconnection network. The parameterized HySAM can model a wide range of systems from board level architectures to systems on a chip. Such systems include SPLASH, Berkeley Garp and NSC NAPA1000 among others.

Linear Loop Synthesis

Scheduling a general sequence of tasks, with a set of dependencies, to minimize the total execution time is known to be an NP-complete problem. We consider the problem of generating this sequence of configurations for loop constructs which have a sequence of statements to be executed in linear order. There is a linear data or control dependency between every pair of adjacent tasks. Most loop constructs, including those occurring in signal processing applications, fall into such a class.

Find an optimal sequence of configurations to execute a linear sequence of statements in a loop.

Problem: Given a sequence of tasks of a loop, T_1 through T_p to be executed in linear order ($T_1\ T_2 \ldots T_p$), for N number of iterations, find an optimal sequence of configurations S ($=C_1C_2 \ldots C_q$), where $S_i \in \{C_1,C_2,\ldots,C_m\}$ which minimizes the execution time cost E, where $E = {}_{i=1}\Sigma^{q}$ $(t_{S_i} + \Delta_{i\,i+1})$. t_{S_i} is the execution time in configuration S_i and $\Delta_{i\,i+1}$ is the reconfiguration cost which is given by $R_{i\,i+1}$.

Optimal Solution for Loop Synthesis

The input consists of a sequence of statements $T_1 \ldots T_p$ and the number of iterations N. We can compute the execution times t_{ij} for executing each of the tasks T_i in configuration C_j. The reconfiguration costs R_{ij} can be pre-computed since the configurations are known beforehand. In addition there is a loop setup cost which is the cost for loading the initial configuration, memory access costs for accessing the required data and the costs for the system to initiate computation by the Configurable Logic Unit. Though the memory access costs are not modeled in this work, it is possible to statically determine the loop setup cost. We present the mapping results without the proofs below:

Lemma 1
Given a sequence of tasks $T'_1\ T'_2 \ldots T'_r$ and the set of possible configurations $\{C_1,\ldots,C_m\}$ an optimal sequence of configurations for executing these tasks **once** can be computed in $O(rm^2)$ time.

Lemma 2
An optimal configuration sequence for the tasks for N iterations can be computed by unrolling the loop only m times.

Theorem 1
The optimal sequence of configurations for N iterations of a loop statement with p tasks, when each task can be executed in one of m possible configurations, can be computed in $O(pm^3)$ time.

The complexity of the algorithm is $O(pm^3)$ which is better than fully unrolling the loop ($O(Npm^2)$) by a factor of $O(N/m)$. This solution can be used even when the number of iterations N is not known at compile time and is determined at runtime. The decision to use this sequence of configurations to execute the loop can be taken at runtime from the statically known loop setup and single iteration execution costs and the runtime determined N.

Illustrative Example

The Discrete Fourier Transform(DFT) is a very important component of many signal processing systems. Typical implementations use the Fast Fourier Transform(FFT) to compute the DFT in $O(N \log N)$ time. The basic computation unit is the butterfly unit which has 2 inputs and 2 outputs. It involves one complex multiplication, one complex addition and one complex subtraction.

We describe an analysis of the implementation to highlight the key features of our mapping technique and model. The aim is to highlight the technique of mapping a sequence of operations onto a sequence of configurations. This technique can be utilized to map onto any configurable architecture. We use the timing and area information from BRASS Garp architecture as representative values. The Garp architecture has a traditional RISC CPU (MIPS variant) attached to a reconfigurable array of logic blocks on the same die.

For the given architecture we first determine the model parameters. We calculated the model parameters from published values and have tabulated them in Table 1 below.

Operation	Configuration	Reconfiguration Time	Execution Time
Multiplication (Fast)	C_1	14.4 μs	37.5 ns
Multiplication (Slow)	C_2	6.4 μs	52.5 ns
Addition	C_3	1.6 μs	7.5 ns
Subtraction	C_4	1.6 μs	7.5 ns
Shift	C_5	3.2 μs	7.5 ns

Table 1: Representative Model Parameters for Garp Reconfigurable Architecture ($m = 5$).

The input application, which is the FFT innermost loop, is analyzed and decomposed. First, the loop statements have to be decomposed into functions which can be executed on the CLU, given the list of functions in Table 1. One complex multiplication consists of four multiplies, one addition and one subtraction. Each complex addition and subtraction consist of two additions and subtractions respectively. The statements in the loop are mapped to multiplication, addition and subtraction and linearized resulting in a task sequence T_m, T_m, T_m, T_m, T_a, T_s, T_a, T_a, T_s, T_s. Here, T_m is the multiplication task, T_a is the addition task and T_s is the subtraction task.

When we find the optimal sequence of configurations for this task sequence using our algorithm, the solution is the configuration sequence C_1,C_3,C_4,C_3,C_4 repeated for all the iterations. The most important aspect of the solution is that the multiplier configuration in the solution is actually the slower configuration. The reconfiguration overhead is lower for C_2 and hence the higher execution cost is amortized over all the iterations of the loop. The total execution time is given by N*13.055 μs where N is the number of iterations.

International Workshop on Field Programmable Logic, September 1998.

Mapping Loops onto Reconfigurable Architectures *

Kiran Bondalapati and Viktor K. Prasanna

Department of Electrical Engineering Systems, EEB-200C
University of Southern California
Los Angeles, CA 90089-2562, USA
{kiran, prasanna}@usc.edu
http://maarc.usc.edu

Abstract. Reconfigurable circuits and systems have evolved from application specific accelerators to a general purpose computing paradigm. But the algorithmic techniques and software tools are also heavily based on the hardware paradigm from which they have evolved. Loop statements in traditional programs consist of regular, repetitive computations which are the most likely candidates for performance enhancement using configurable hardware. This paper develops a formal methodology for mapping loops onto reconfigurable architectures. We develop a parameterized abstract model of reconfigurable architectures which is general enough to capture a wide range of configurable systems. Our abstract model is used to define and solve the problem of mapping loop statements onto reconfigurable architectures. We show a polynomial time algorithm to compute the optimal sequence of configurations for one important variant of the problem. We illustrate our approach by showing the mapping of an example loop statement.

1 Introduction

Configurable systems are evolving from systems designed to accelerate a specific application to systems which can achieve high performance for general purpose computing. Various reconfigurable architectures are being explored by several research groups to develop a general purpose configurable system. Reconfigurable architectures vary from systems which have FPGAs and glue logic attached to a host computer to systems which include configurable logic on the same die as a microprocessor.

Application development onto such configurable hardware still necessitates expertise in low level hardware details. The developer has to be aware of the intricacies of the specific reconfigurable architecture to achieve high performance. Automatic mapping tools have also evolved from high level synthesis tools. Most tools try to generate hardware configurations from user provided descriptions of

* This work was supported by the DARPA Adaptive Computing Systems Program under contract DABT63-96-C-0049 monitored by Fort Hauchuca.

circuits in various input formats such as VHDL, OCCAM, variants of C, among others.

Automatic compilation of applications involves not only configuration generation, but also configuration management. CoDe-X [8] is one environment which aims to provide an end-to-end operating system for applications using the Xputer paradigm. General techniques are being developed to exploit the characteristics of devices such as partial and dynamic reconfiguration by using the concepts of Dynamic Circuit Switching [11], Virtual Pipelines [10] etc. But there is no framework which abstracts all the characteristics of configurable hardware and there is no unified methodology for mapping applications to configurable hardware.

In this paper we address some of the issues in the development of techniques for automatic compilation of applications. We develop algorithmic techniques for mapping applications in a platform independent fashion. First, we develop an abstract model of reconfigurable architectures. This parameterized abstract model is general enough to capture a wide range of configurable systems. These include board level systems which have FPGAs as configurable computing logic to systems on a chip which have configurable logic arrays on the same die as the microprocessor.

Configurable logic is very effective in speeding up regular, repetitive computations. Loop constructs in general purpose programs are one such class of computations. We address the problem of mapping a loop construct onto configurable architectures. We define problems based on the model which address the issue of minimizing reconfiguration overheads by scheduling the configurations. A polynomial time solution for generating the optimal configuration sequence for one important variant of the mapping problem is presented.

Our mapping techniques can be utilized to analyze application tasks and develop the choice of configurations and the schedule of reconfigurations. Given the parameters of an architecture and the applications tasks the techniques can be used statically at compile time to determine the optimal mapping. The techniques can also be utilized for runtime mapping by making static compile time analysis. This analysis can be used at runtime to make a decision based on the parameters which are only known at runtime.

Section 2 describes our Hybrid System Architecture Model(HySAM) in detail. Several loop mapping problems are defined and the optimal solution for one important variant is presented in Section 3. We show an example mapping in Section 4 and discuss future work and conclusions in Section 5.

1.1 Related Work

The question of mapping structured computation onto reconfigurable architectures has been addressed by several researchers. We very briefly describe some related work and how our research is different from their work. The previous work which addresses the related issues is Pipeline Generation for Loops [17], CoDe-X Framework [8], Dynamic Circuit Simulation [11], Virtual Pipelines [10], TMFPGA [14]. Though most of the projects address similar issues, the framework of developing an abstract model for solving general mapping problems is not fully addressed by any specific work.

2 Model

We present a parameterized model of a configurable computing system, which consists of configurable logic attached to a traditional microprocessor. This model can be utilized for analyzing application tasks, as regards to their suitability for execution on configurable logic and also for developing the actual mapping and scheduling of these tasks onto the configurable system.

We first describe our model of configurable architectures and then discuss the components of the model and how they abstract the actual features of configurable architectures.

2.1 Hybrid System Architecture Model(HySAM)

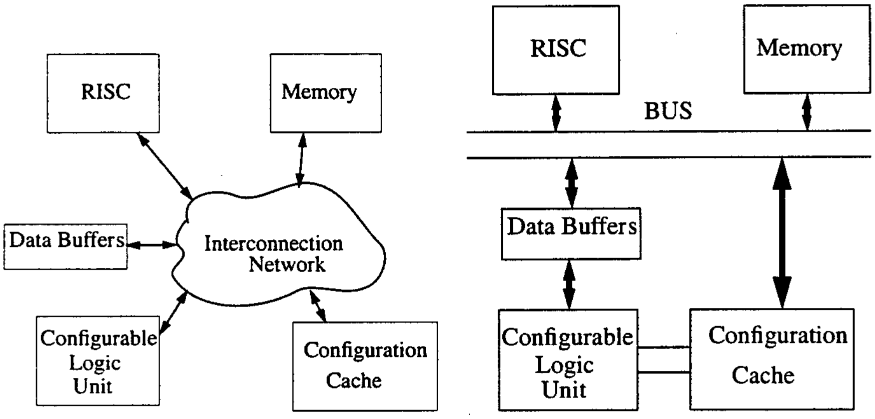


Fig. 1. Hybrid System Architecture and an example architecture

The *Hybrid System Architecture* is a general architecture consisting of a traditional RISC microprocessor with additional Configurable Logic Unit(CLU). Figure 1 shows the architecture of the HySAM model and an example of an actual architecture. The architecture consists of a traditional RISC microprocessor, standard memory, configurable logic, configuration memory and data buffers communicating through an interconnection network.

We outline the parameters of the Hybrid System Architecture Model(HySAM) below.

F : Set of functions $F_1 \ldots F_n$ which can be performed on configurable logic.
C : Set of possible configurations $C_1 \ldots C_m$ of the Configurable Logic Unit.
t_{ij} : Execution time of function F_i in configuration C_j.
R_{ij} : Reconfiguration cost in changing configuration from C_i to C_j.

N_c : The number of configuration contexts which can be stored in the configuration cache.

k, K : The reconfiguration time spent in configuring from the cache and external memory respectively.

W, D : The Width and Depth of the configurable logic which describe the amount of configurable logic available.

w : The granularity of the configurable logic which is the width of an individual functional unit.

S : The schedule of configurations which execute the input tasks.

E : Execution time of a sequence of tasks, which is the sum of execution time of tasks in the various configurations and the reconfiguration time.

The parameterized HySAM which is outlined above can model a wide range of systems from board level architectures to systems on a chip. Such systems include SPLASH [3], DEC PeRLE [16], Oxford HARP [9], Berkeley Garp [7], NSC NAPA1000 [15] among others. The values for each of the parameters establish the architecture and also dictate the class of applications which can be effectively mapped onto the architecture. For example, a system on a chip would have smaller size configurable logic(lower W and D) than an board level architecture but would have potentially faster reconfiguration times(lower k and K).

The model does not encompass the memory access component of the computation in terms of the memory access delays and communication bandwidth supported. Currently, it is only assumed that the interconnection network has enough bandwidth to support all the required data and configuration access. For a detailed description of the model and its parameters see [2].

3 Loop Synthesis

It is a well known rule of thumb that 90% of the execution time of a program is spent in 10% of the code. This code usually consists of repeated executions of the same set of instructions. The typical constructs used for specifying iterative computations in various programming languages are DO, FOR and WHILE, among others. These are generally classified as LOOP constructs.

Computations which operate on a large set of data using the same set of operations are most likely to benefit from configurable computing. Hence, loop structures will be the most likely candidates for performance improvement using configurable logic. Configurations which execute each task can be generated for the operations in a loop. Since each operation is executed on a dedicated hardware configuration, the execution time for the task is expected to lower than that in software. Each of the operations in the loop statement might be a simple operation such as an addition of two integers or can be a more complex operation such as a square root of a floating point number. The problems and solutions that we present are independent of the complexity of the operation.

3.1 Linear Loop Synthesis

The problem of mapping operations(tasks) of a loop to a configurable system involves not only generating the configurations for each of the operations, but also reducing the overheads incurred. The sequence of tasks to be executed have to be mapped onto a sequence of configurations that are used to execute these tasks. The objective is to reduce the total execution time.

Scheduling a general sequence of tasks with a set of dependencies to minimize the total execution time is known to be an NP-complete problem. We consider the problem of generating this sequence of configurations for loop constructs which have a sequence of statements to be executed in linear order. There is a linear data or control dependency between the tasks. Most loop constructs, including those which are mapped onto high performance pipelined configurations, fall into such a class.

The total execution time includes the time taken to execute the tasks in the chosen configurations and the time spent in reconfiguring the logic between successive configurations. We have to not only choose configurations which execute the given tasks fast, but also have to reduce the reconfiguration time. It is possible to choose one of many possible configurations for each task execution. Also, the reconfiguration time depends on the choice of configurations that we make. Since reconfiguration times are significant compared to the task execution times, our goal is to minimize this overhead.

Problem : Given a sequence of tasks of a loop, T_1 through T_p to be executed in linear order(T_1 T_2 ... T_p), where $T_i \in F$, for N number of iterations, find an optimal sequence of configurations S (=C_1 C_2 ... C_q), where $S_i \in C$ (=$\{C_1,C_2,\ldots,C_m\}$) which minimizes the execution time cost E. E is defined as

$$E = \sum_{i=1}^{q}(t_{S_i} + \Delta_{ii+1})$$

where t_{S_i} is execution time in configuration S_i and Δ_{ii+1} is the reconfiguration cost which is given by R_{ii+1}.

3.2 Optimal Solution for Loop Synthesis

The input consists of a sequence of statements $T_1 \ldots T_p$, where each $T_i \in F$ and the number of iterations N. We can compute the execution times t_{ij} for executing each of the tasks T_i in configuration C_j. The reconfiguration costs R_{ij} can be pre-computed since the configurations are known beforehand. In addition there is a loop setup cost which is the cost for loading the initial configuration, memory access costs for accessing the required data and the costs for the system to initiate computation by the Configurable Logic Unit. Though, the memory access costs are not modeled in this work, it is possible to statically determine the loop setup cost.

A simple greedy approach of choosing the best configuration for each task will not work since the reconfiguration costs for later tasks are affected by the choice of configuration for the current task. We have to search the whole solution space by considering all possible configurations in which each task can be executed. Once an optimal solution for executing up to task T_i is computed the cost for executing up to task T_{i+1} can be incrementally computed.

Lemma 1. *Given a sequence of tasks $T'_1 T'_2 \ldots T'_r$, an optimal sequence of configurations for executing these tasks* **once** *can be computed in $O(rm^2)$ time.*

Proof: Using the execution cost definition we define the optimal cost of executing up to task T'_i ending in a configuration C_j as E_{ij}. We initialize the E values as $E_{0j} = 0, \ \forall\, j : \ 1 \leq j \leq m$.

Now for each of the possible configurations in which we can execute T'_{i+1} we have to compute an optimal sequence of configurations ending in that configuration. We compute this by the recursive equation:

$$E_{i+1j} = t_{i+1j} + min_k(E_{ik} + R_{kj}) \quad \forall\, j : \ 1 \leq j \leq m$$

We have examined all possible ways to execute the task T'_{i+1} once we have finished executing T'_i. If each of the values E_{ik} is optimal then the value E_{i+1j} is optimal. Hence we can compute an optimal sequence of configurations by computing the E_{ij} values. The minimum cost for the complete task sequence$(T'_1 T'_2 \ldots T'_r)$ is given by $min_j[E_{rj}]$. The corresponding optimal configuration sequence can be computed by using the E matrix.

We can use dynamic programming to compute the E_{ij} values. Computation of each value takes $O(m)$ time as there are m configurations. Since there are $O(rm)$ values to be computed, the total time complexity is $O(rm^2)$. $\odot$

Lemma 1 provides a solution for an optimal sequence of configurations to compute one iteration of the loop statement. But repeating this sequence of configurations is not guaranteed to give an optimal execution for N iterations. Figure 2 shows the configuration space for two tasks T_1 and T_2 and four possible configurations C_1, C_2, C_3, C_4. T_1 can be executed in C_1 or C_3 and task T_2 can be executed in C_2 or C_4. The edges are labeled with the reconfiguration costs and cost for the edges and configurations not shown is very high. We can see that an optimal sequence of execution for more than two iterations will be the sequence $C_1\, C_4\, C_3\, C_2$ repeated $N/2$ times. The repeated sequence of $C_1\, C_4$ which is an optimal solution for one iteration does not give an optimal solution for N iterations.

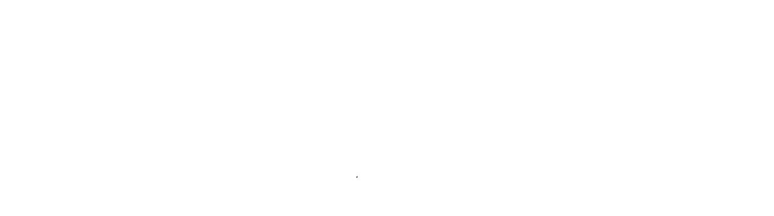


Fig. 2. Example reconfiguration cost graph and optimal configuration sequence

One simple solution is to fully unroll the loop and compute an optimal sequence of configurations for all the tasks. But the complexity of algorithm will be $O(Npm^2)$, where N is the number of iterations. Typically the value of N would be very high(which is desirable since higher value of N gives higher speedup compared to software execution). We assume $N \gg m$ and $N \gg p$. We show that an optimal configuration sequence can be computed in $O(pm^3)$ time.

Lemma 2. *An optimal configuration sequence can be computed by unrolling the loop only m times.*

Proof: Let us denote the optimal sequence of configurations for the fully unrolled loop by $C_1C_2 \ldots C_x$. Since there are p tasks and N iterations $x = N * p$. Configuration C_1 executes T_1, C_2 executes T_2 and so on. Now after one iteration execution, configuration C_{p+1} executes task T_1 again. Therefore, task T_1 is executed in each of configurations C_1, C_{p+1}, C_{2*p+1}, ..., C_{x-p+1}. Since there are at most m configurations for each task, if the number of configurations in C_1, C_{p+1}, C_{2*p+1}, ..., C_{x-p+1} is more than m then some configuration will repeat. Therefore, $\exists\ y\ s.t.\ C_{y*p+1} = C_1$.

Let the next occurrence of configuration C_1 for task T_1 after C_{y*p+1} be C_{z*p+1}. The subsequence $C_1\ C_2\ C_3 \ldots\ C_{y*p+1}$ should be identical to $C_{y*p+1}\ C_{y*p+2}\ \ldots\ C_{z*p+1}$. Otherwise, we can replace the subsequence with higher per iteration cost by the subsequence with lower per iteration cost yielding a better solution. But this contradicts our initial assumption that the configuration sequence is optimal. Hence the two subsequences should be identical. This does not violate the correctness of execution since both subsequences are executing a fixed number of iterations of the same sequence of input tasks. Applying the same argument to the complete sequence $C_1C_2 \ldots C_x$, it can be proved that all subsequences should be identical.

The longest possible length of such a subsequence is $m * p$(p possible tasks each with m possible configurations). This subsequence of $m * p$ configurations is repeated to give the optimal configuration sequence for $N * p$ tasks. Hence, we need to unroll the loop only m times. $\odot$

Theorem 3. *The optimal sequence of configurations for N iterations of a loop statement with p tasks, when each task can be executed in one of m possible configurations, can be computed in $O(pm^3)$ time.*

Proof: From Lemma 2 we know that we need to unroll the loop only m times to compute the required sequence of configurations. The solution for the unrolled sequence of $m * p$ tasks can be computed in $O(pm^3)$ by using Lemma 1. This sequence can then be repeated to give the required sequence of configurations for all the iterations. Hence, the total complexity is $O(pm^3)$. $\odot$

The complexity of the algorithm is $O(pm^3)$ which is better than fully unrolling ($O(Npm^2)$) by a factor of $O(N/m)$. This solution can also be used when the number of iterations N is not known at compile time and is determined at runtime. The decision to use this sequence of configurations to execute the loop can be taken at runtime from the statically known loop setup and single iteration execution costs and the runtime determined N.

4 Illustrative Example

The techniques that we have developed in this paper can be evaluated by using our model. The evaluation would take as input the model parameter values and the applications tasks and can solve the mapping problem and output the sequence of configurations. We are currently building such a tool and show results obtained by manual evaluation in this section.

The Discrete Fourier Transform(DFT) is a very important component of many signal processing systems. Typical implementations use the Fast Fourier Transform(FFT) to compute the DFT in $O(N \log N)$ time. The basic computation unit is the butterfly unit which has 2 inputs and 2 outputs. It involves one complex multiplication, one complex addition and one complex subtraction.

There have been several implementations of FFT in FPGAs [12, 13]. The computation can be optimized in various ways to suit the technology and achieve high performance. We describe here an analysis of the implementation to highlight the key features of our mapping technique and model. The aim is to highlight the technique of mapping a sequence of operations onto a sequence of configurations. This technique can be utilized to map onto any configurable architecture. We use the timing and area information from Garp [7] architecture as representative values.

For the given architecture we first determine the model parameters. We calculated the model parameters from published values and have tabulated them in Table 1 below. The set of functions(F) and the configurations(C) are outlined in Table 1 below. The values of n and m are 4 and 5 respectively. The Configuration Time column gives the reconfiguration values R. We assume the reconfiguration values are same for same target configuration irrespective of the initial configuration. The Execution Time column gives the t_{ij} values for our model.

Function	Operation	Configuration	Configuration Time	Execution Time
F_1	Multiplication(Fast)	C_1	14.4 μs	37.5 ns
	Multiplication(Slow)	C_2	6.4 μs	52.5 ns
F_2	Addition	C_3	1.6 μs	7.5 ns
F_3	Subtraction	C_4	1.6 μs	7.5 ns
F_4	Shift	C_5	3.2 μs	7.5 ns

Table 1. Representative Model Parameters for Garp Reconfigurable Architecture

The input sequence of tasks to be executed is is the FFT butterfly operation. The butterfly operation consists of one complex multiply, one complex addition and one complex subtraction. First, the loop statements were decomposed into functions which can be executed on the CLU, given the list of functions in Table 1. One complex multiplication consists of four multiplications, one addition and one subtraction. Each complex addition and subtraction consist of two additions and subtractions respectively. The statements in the loop were mapped to multiplications, additions and subtractions which resulted in the task sequence T_m. T_m, T_m, T_m, T_a, T_s, T_a, T_a, T_s, T_s. Here, T_m is the multiplication task mapped to function F_1, T_a is the addition task mapped to function F_2 and T_s is the subtraction task mapped to function F_3.

The optimal sequence of configurations for this task sequence, using our algorithm, was C_1,C_3,C_4,C_3,C_4 repeated for all the iterations. The most important aspect of the solution is that the multiplier configuration in the solution is actually the slower configuration. The reconfiguration overhead is lower for C_2 and hence the higher execution cost is amortized over all the iterations of the loop. The total execution time is given by $N * 13.055$ μs where N is the number of iterations.

5 Conclusions

Mapping of applications in an architecture independent fashion can provide a framework for automatic compilation of applications. Loop structures with regular repetitive computations can be speeded-up by using configurable hardware. In this paper, we have developed techniques to map loops from application programs onto configurable hardware. We have developed a general Hybrid System Architecture Model(HySAM). HySAM is a parameterized abstract model which captures a wide range of configurable systems. The model also facilitates the formulation of mapping problems and we defined some important problems in mapping of traditional loop structures onto configurable hardware. We demonstrated a polynomial time solution for one important varíant of the problem. We also showed an example mapping of the FFT loop using our techniques. The model can be extended to solve other general mapping problems. The application development phase itself can be enhanced by using the model to develop solutions using algorithm synthesis rather than logic synthesis.

The work reported here is part of the USC MAARC project. This project is developing algorithmic techniques for realizing scalable and portable applications using configurable computing devices and architectures. We are developing computational models and algorithmic techniques based on these models to exploit dynamic reconfiguration. In addition, partitioning and mapping issues in compiling onto reconfigurable hardware are also addressed. Some related results can be found in [1], [4], [5], [6].

References

1. K. Bondalapati and V.K. Prasanna. Reconfigurable Meshes: Theory and Practice. In *Reconfigurable Architectures Workshop, RAW'97*, Apr 1997.

2. Kiran Bondalapati and Viktor K. Prasanna. The Hybrid System Architecture Model (HySAM) of Reconfigurable Architectures. Technical report, Department of Electrical Engineering-Systems, University of Southern California, 1998.
3. D. A. Buell, J. M. Arnold, and W. J. Kleinfelder. *Splash 2: FPGAs in a Custom Computing Machine.* IEEE Computer Society Press, 1996.
4. S. Choi and V.K. Prasanna. Configurable Hardware for Symbolic Search Operations. In *International Conference on Parallel and Distributed Systems*, Dec 1997.
5. Y. Chung and V.K. Prasanna. Parallel Object Recognition on an FPGA-based Configurable Computing Platform. In *International Workshop on Computer Architectures for Machine Perception*, Oct 1997.
6. A. Dandalis and V.K. Prasanna. Fast Parallel Implementation of DFT using Configurable Devices. In *7th International Workshop on Field-Programmable Logic and Applications*, Sept 1997.
7. J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, April 1997.
8. R. Kress, R.W. Hartenstein, and U. Nageldinger. An Operating System for Custom Computing Machines based on the Xputer Paradigm. In *7th International Workshop on Field-Programmable Logic and Applications*, pages 304–313, Sept 1997.
9. A. Lawrence, A. Kay, W. Luk, T. Nomura, and I. Page. Using reconfigurable hardware to speed up product development and performance. In *5th International Workshop on Field-Programmable Logic and Applications*, 1995.
10. W. Luk, N. Shirazi, S.R. Guo, and P.Y.K. Cheung. Pipeline Morphing and Virtual Pipelines. In *7th International Workshop on Field-Programmable Logic and Applications*, Sept 1997.
11. P. Lysaght and J. Stockwood. A Simulation Tool for Dynamically Reconfigurable FPGAs. *IEEE Transactions on VLSI Systems*, Sept 1996.
12. Xilinx DSP Application Notes. The Fastest FFT in the West, http://www.xilinx.com/apps/displit.htm.
13. R.J. Petersen and B. Hutchings. An Assessment of the Suitability of FPGA-Based Systems for use in Digital Signal Processing. In *5th International Workshop on Field-Programmable Logic and Applications*, 1995.
14. S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A Time-Multiplexed FPGA. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 22–28, April 1997.
15. NSC NAPA 1000 URL. http://www.national.com/appinfo/milaero/napa1000/.
16. J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, March 1996.
17. M. Weinhardt. Compilation and Pipeline Synthesis for Reconfigurable Architectures. In *Reconfigurable Architectures Workshop RAW'97.* ITpress Verlag, Apr 1997.

This article was processed using the LaTeX macro package with LLNCS style

String Matching on Multicontext FPGAs using Self-Reconfiguration

Reetinder P. S. Sidhu
sidhu@halcyon.usc.edu

Alessandro Mei
amei@halcyon.usc.edu

Viktor K. Prasanna
prasanna@ganges.usc.edu

Department of EE-Systems, University of Southern California,
Los Angeles CA 90089

Abstract

FPGAs can perform better than ASICs if the logic mapped onto them is optimized for each problem instance. Unfortunately, this advantage is often canceled by the long time needed by CAD tools to generate problem instance dependent logic and the time required to configure the FPGAs.

In this paper, a novel approach for runtime mapping is proposed that utilizes self-reconfigurability of multicontext FPGAs to achieve very high speedups over existing approaches. The key idea is to design and map logic onto a multicontext FPGA that in turn maps problem instance dependent logic onto other contexts of the same FPGA. As a result, CAD tools need to be used just once for each problem and not once for every problem instance as is usually done.

To demonstrate the feasibility of our approach, a detailed implementation of the KMP string matching algorithm is presented which involves runtime construction of a finite state machine. We implement the KMP algorithm on a conventional FPGA (Xilinx XC 6216) and use it to obtain accurate estimates of performance on a multicontext device. Speedups in mapping time of $\approx 10^6$ over CAD tools and more than 1800 over a program written specifically for FSM generation were obtained. Significant speedups were obtained in overall execution time as well, including a speedup ranging from 3 to 16 times over a software implementation of the KMP algorithm running on a Sun Ultra 1 Model 140 workstation.

1 Introduction

By exploiting the reconfigurability of FPGAs, significant performance improvements have been obtained over other modes of computation for several applications. However, there are two serious problems that prevent FPGAs from being utilized to their fullest potential:

- long mapping time;
- long reconfiguration time.

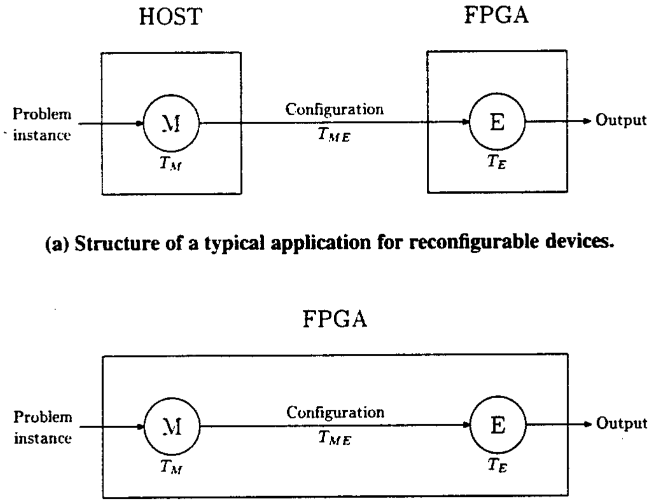
Mapping time refers to the time to compile, place and route the logic to be used on the FPGA; reconfiguration time is the time needed to load the configuration data into the FPGA. Mapping computation onto FPGAs is typically done using CAD tools. It is a time consuming process and can take anywhere from a few minutes to a few days. In order to take advantage of the reconfigurability of FPGAs, a new mapping should be created for every problem instance. As a result, the mapping time becomes very critical and it is extremely important to reduce it.

The time required to completely reconfigure an FPGA is typically about 1 ms. Since reconfiguration time needs to be amortized over computation time, frequent run-time reconfiguration is not possible. It should be noted that even partial reconfiguration is not a complete solution to this problem. Since reconfigurability is the key advantage of FPGAs over other modes of computation, reduction of reconfiguration time is very important.

In this paper we show how to significantly reduce both mapping and reconfiguration times through *self-reconfiguration*. By self-reconfiguration we mean that not only does the FPGA load the configuration information itself, but also that it generates the configuration. We show how self-reconfiguration can be efficiently implemented using *multicontext FPGAs* (FPGAs having more than one configuration context on-chip). Although, such devices have been primarily designed to reduce reconfiguration times, we show how they can be used for self-reconfiguration as well.

Self-reconfiguration reduces mapping time because all logic to be configured is generated by previously configured logic. The mapping logic is designed to generate highly specific mapped logic and is therefore much simpler than general purpose CAD tools. Also, it executes on an FPGA. For these reasons, the mapping time is considerably lesser than mapping via software running on a host machine. Self-reconfiguration reduces reconfiguration time because configurations are generated and stored on-chip which is much faster than loading it from an external source. Also, multicontext FPGAs can very quickly switch between stored configurations. As a result of these improvements, self-reconfiguration allows runtime generation of logic and its use to be interwoven in ways that would be impractical otherwise. We demonstrate this power and flexibility by a string matching algorithm implementation. Even though our early results are very promising, a deep investigation is needed to fully understand what can be achieved by using this approach

This work was supported by the DARPA Adaptive Computing Systems Program under contract DABT63-96-C-0049 monitored by Fort Hauchuca. Alessandro Mei is with the Department of Mathematics of the University of Trento, Italy. This work has been done during his stay at the USC.



(a) Structure of a typical application for reconfigurable devices.

(b) Mapping and execution on a self-reconfigurable device.

Figure 1:

to reconfigurable computing, and it seems to be a challenging and wide open research area.

In the first part of the paper, we introduce self-reconfiguration and its advantages (Section 2) and how it is achieved using multicontext FPGAs (Section 3). In the second part, we introduce the KMP algorithm (Section 4) and present detailed implementation description and performance analysis (Section 5). The conclusion is in Section 6.

2 Introduction to Self-Reconfiguration

2.1 Problem instance dependence and hardware compiling

The effectiveness of reconfigurable computing is better exploited by building hardware solutions for each single instance of a given problem. That essentially means that a good application for reconfigurable devices should read the input of the problem (the *instance*), compute *instance dependent* logic, i.e. logic optimized for that particular instance, and load it into a reconfigurable device to solve the problem. Applications which produce *instance independent* logic to be loaded onto a reconfigurable device are simply not exploiting the power of reconfiguration. In that case the logic mapped is static, depends only on the algorithm used, and is not conceptually different from ASIC approach.

A large class of applications developed for reconfigurable devices can thus be modeled in the following way (see Figure 1(a)). A process M reads the input problem instance. Depending on the instance a logic E, ready to be loaded, is computed such that it is optimized to solve that single problem instance. This process is usually executed by the host computer. Let T_M denote the time to perform this. After reconfiguring the device, E is executed. Let T_{ME} denote the time to reconfigure. The time T_E required for the execution includes the time needed for reading the inputs from the memory and producing the output. Therefore, the time required by the execution of a single iteration of the computation described above is $T_I = T_M + T_{ME} + T_E$.

The actual execution time on the reconfigurable device is T_E. It is often very low compared with the time needed to solve the same problem by using a software solution, due to hardware efficiency. This has been used to claim that very high speed-up can be achieved by reconfigurable computing. It should be clear that this is not a fair way to compare the performance of reconfigurable systems. However, this is frequently done. We believe that all times involved in computing the solution to a given instance of a problem should be taken into account.

The time T_M required by M varies considerably among applications, and usually ranges from a few minutes to several hours, and, for some particularly complex logic, even days! The reason lies in the fact that usually CAD tools are used. CAD tools are very powerful and general applications, but their flexibility is obtained at the expense of large computing time. In fact, what is actually done, is to *compile*, using a CAD tool, each single instance to derive the logic E to be used to solve the problem.

The fact that T_M is usually large limits the effectiveness of reconfigurable computing. In [1], for example, a shortest paths algorithm is implemented. In that case, the execution time T_E for a problem instance is order of microseconds, while the mapping time T_M is order of hours. Also in [15], the proposed algorithm for SAT usually takes hours to be mapped. SAT is NP-complete, and thus a good candidate to make T_M affordable since T_E is usually very high. In spite of that, when mapping time is taken into account, only modest speedups are obtained. The time T_{ME} depends on to the device used. For FPGAs, for example, it is typically around 1 millisecond, and it is related to the bottle-neck represented by the bus connecting the host computer to the FPGA board. Even if the reconfiguration time T_{ME} is often much lower than the mapping time, it can still be unacceptable for most real-time applications.

Some efforts have been made to overcome these problems. For example, in [7] CAD Tools are used only once to compute a generic skeleton logic. Then, for each problem instance, some limited changes are made by the host computer to build an instance dependent circuit and load it into the FPGA board. This is an interesting technique that can be useful to lower the mapping time T_M, but cannot avoid the bottle-neck represented by the bus connecting the host computer to the FPGA board. In [7], $T_M + T_{ME}$ is around 3 seconds, still too high for a large class of applications.

This paper presents a novel approach to reconfigurable computing which is able to dramatically reduce T_M and T_{ME}. Since M has to be speeded up, what we propose is to let fast reconfigurable devices to be able to execute it (see Figure 1(b)). In case a single FPGA is being used, this essentially means that the FPGA should be able to read from a memory the problem instance, configure itself, or a part of it, and execute the logic built by it to solve the problem instance. Evidently, in this case M is itself a logic circuit, and cannot be as complex and general as CAD tools are.

Letting FPGA system execute both M and E on the same chip gives the clear advantage that CAD tools are used only *once*, in spite of classical solutions where they are needed for computing a logic for each problem instance. This is possible since the adaptations, needed to customize the circuit to the requirements of the actual input, are performed dynamically by the FPGA itself, taking advantage of hardware efficiency.

Another central point is that the bus connecting the FPGA system to the host computer is now only used to input the problem instance, since the reconfiguration data are generated locally. In this way, the bottle-neck problem is also handled.

These ideas are shown to be realistic and effective by presenting a novel implementation of a string matching algorithm. String matching is one of the most important problems in Computer Science, both from a theoretical and from a practical point of view. In Section 5, a detailed implementation is described, and $T_M + T_{ME}$ is shown to be around 28μs, for patterns 16 character long, achieving a dramatic speed-up over classical FPGA computations.

2.2 Self-reconfiguration

The main feature needed by an FPGA device to fulfill the requirements needed by the technique shown in the previous section is self-reconfigurability. This concept has been mentioned few times in the literature on reconfigurable architectures in the last few years [6][5]. In spite of that, to the best of our knowledge not only no one devised an application that actually used that feature, but no one even investigated to understand how self-reconfiguration could be used to achieve superior performance.

The concept of self-reconfiguration was earliest presented in [6], where a small amount of static logic is added to a reconfigurable device based on an FPGA in order to build a self-reconfiguring processor. Being an architecture oriented work, no application of this concept is shown.

The recent Xilinx XC6200 is also a self-reconfiguring device, and this ability has been used in [5] to define an abstract model of virtual circuitry, the Flexible URISC. This model still has a self-configuring capability, even though it is not used by the simple example presented in [5].

All these devices are potentially capable of self-reconfiguring, and are thus able of implementing the ideas presented in this paper. However, moving the process of building the reconfigurable logic into the device itself requires a larger amount of configuration memory in the device with respect to traditional approaches. For this reason, multi-context FPGAs seem to answer better to these requirements, since they have been shown to be able to store a large amount of different contexts (see [12], for example, where a self-reconfiguring 256-context FPGA is presented).

3 Multicontext FPGAs

As described in the Introduction, the time required to reconfigure a traditional FPGA is very high. To reduce the reconfiguration time, a device having more than one configuration context was proposed in [4]. Several such *multicontext* FPGAs have been recently proposed [13][11][14] [8][3].

These devices have on-chip RAM to store a number of configuration contexts, varying from 8 to 256. At any given time, one context governs the logic functionality and is referred to as the *active* context. Switching contexts takes 5–100 ns. This is several orders of magnitude faster than the time required to reconfigure a conventional FPGA (≈1 ms).

For self-reconfiguration to be possible, the following two additional features are required of multicontext FPGAs:

- The active context should be able to initiate a context switch—no external intervention should be necessary.
- The active context should be able to read and write the configuration memory corresponding to other contexts.

The multicontext FPGAs described in [13][11][14] satisfy the above requirements and hence are capable of self-reconfiguration[1].

4 The KMP Algorithm for String Matching

The String Matching problem consists of finding all occurrences of a *pattern* P, of length m, in a *text* T, of length n, $m \leq n$, with P and T being strings over a finite alphabet Σ.

Besides being a fundamental problem in Computer Science from a theoretical point of view, String Matching is of paramount practical relevance. Important examples of its application can be easily found in the areas of Text Processing, Pattern Recognition, Image Understanding, Databases, and Biology, to name a few. In particular, applications of String Matching in Biology are of utmost importance, since finding patterns of DNA inside longer sequences is becoming central in the analysis of human genome.

A naive algorithm that can be used to solve String Matching consists in trying to match the pattern at each position in the text by a "brute force" search. Meaning that for each position i in the text, we perform a do-loop operation to check whether all m characters of P match m characters of T starting from position i. If we found a mismatch, say at position $i + h$, we can stop this search and try at position $i + 1$. This leads to a simple, but slow, algorithm, whose time complexity is $O(mn)$, in the worst case, and thus quite far from optimality.

It can be remarked, however, that if we find a mismatch at position i, it makes sense to try at position $i + 1$ only if the pattern is such that its first $h - 1$ characters, which are equal to the $h - 1$ characters starting at position i in the text, are exactly equal to the $h - 1$ characters starting at position 2 in the pattern itself. If this is not the case, we waste our time looking for a match at position $i + 1$; moreover, if this is the case, we also waste time comparing the first $h - 1$ characters of the pattern, from position $i + 1$ to position $i + h$ excluded in the text, since we already know that we are going to find all matches.

More generally, after finding a mismatch at position $i + h$, we can jump in the pattern at the end of the longest prefix that is also a suffix of the first h character in the pattern, and keep on comparing the character at position $i + h$ in the text. There is no way to find an

[1] The string matching implementation described later also requires configuration memory writes to take only a few clock cycles. At least one of the devices [11] allows this and others may also.

```
Procedure TextSearch($P, T$)

  $n = length(T)$;
  $m = length(P)$;
  $\pi =$ ComputePrefixFunction($P$);
  $q = 0; i = 0$;
  while ($i < n$) do
    if ($T[i] \neq P[q]$)and($q == 0$) then
      ++i;
    else if ($T[i] \neq P[q]$)and($q \neq 0$) then
      $q = \pi[q]$;
    else if ($T[i] == P[q]$)and($q \neq m - 1$) then
      $+ + i; + + q$;
    else if ($T[i] == P[q]$)and($q == m - 1$) then
      print "match found";
      $+ + i; + + q$;
    end if
  end while
```

```
Function ComputePrefixFunction($P$)

  $m = length(P)$;
  $\pi[1] = 0$;
  $i = 1; q = 0$;
  while ($i < m$) do
    if ($P[i] \neq P[q]$)and($q == 0$) then
      ++i;
      $\pi[i] = 0$;
    else if ($P[i] \neq P[q]$)and($q \neq 0$) then
      $q = \pi[q]$;
    else if ($P[i] == P[q]$) then
      $+ + i; + + q$;
      $\pi[i] = q$;
    end if
  end while
```

Figure 2: KMP algorithm Phase 2 (Text search) and Phase 1 (Prefix function Computation).

occurrence of the pattern before that point, and, at the same time, we can take advantage of internal symmetries of the pattern avoiding checking characters in the text more than needed.

This is the key idea of the Knuth-Morris-Pratt algorithm, which computes, for each position h in the pattern, the longest prefix that is also a suffix of the first h character of the pattern itself. This information is encapsulated in a function π such that $\pi[h] = j$ if and only if the first j characters of P are the longest proper prefix that is also a suffix of the first h characters of P. Note that π does not depend on the text, and can be thus precomputed by looking at the pattern only.

The KMP algorithm is a classical 2-phase computation. It takes in input the pattern P, performs a precomputation on P to get the function π, and then, in the second phase, uses π to speed-up the search inside the text. Using terminology introduced earlier, $T_M + T_{ME}$ is the time taken by Phase 1 while the time taken by Phase 2 is T_E. The algorithms used for Phase 1 (Prefix function computation) and Phase 2 (Text search) are shown in detail in Figure 2. The algorithms shown have been written such that they correspond closely to their hardware implementation. It can be proved that KMP is optimal, requiring $O(m + n)$ to perform both phases (see [2] for a proof and a detailed description of the KMP algorithm).

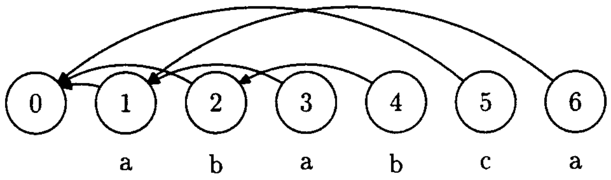


Figure 3: Example of π function for a pattern $p = ababca$. The index q of the algorithms in Figure 2 can be implemented as a pointer to a node, and an edge from the node h to the node j is present if and only if $\pi[h] = j$.

KMP seems to be an ideal candidate to be implemented on reconfigurable devices. Indeed, thanks to reconfigurability, the function π, depending on the input of each single instance of the problem, can be implemented in hardware, thus considerably speeding-up the searching phase. A good way to visualize the function π is given in Figure 3, where each node indicate a position in the pattern, and an edge is present between nodes h and j if and only if $\pi[h] = j$. In this way, the value of the index q in the the KMP-Matcher, shown in Figure 2, can be stored as a pointer to a node, and at each step of computation the pointer q moves either to the next node $q + 1$, if a match is found, or to the node $\pi[q]$ indicated by the edge starting from node q, otherwise. This behavior is very similar to that of a finite state machine, and it is well suited for hardware implementation, as will be shown in the next section.

Our implementation is devised to handle an *on-line* version of String Matching. Meaning that our FPGA system is able to read an incoming pattern, configure itself depending on it, and solve the problem on an incoming text. Moreover, it is possible to change the problem instance by furnishing a new pattern to the system. In this case, the FPGA reconfigures itself to optimize depending on the new pattern, and is ready to solve the new instance on an incoming text. All these operations (including reconfiguration) are performed inside the FPGA system itself, without involving the host computer.

5 Implementation of the KMP algorithm

In this section, we present the details of how the KMP algorithm exploiting self-reconfiguration would be implemented on a multicontext FPGA. Unfortunately, multicontext FPGAs are not commercially available. Therefore, we implement the logic on a conventional FPGA and simulate self-reconfiguration via software. We begin by describing in Section 5.1 how the algorithm is realized in hardware without discussing any FPGA specific features. Since the FSM is the most important component, its structure and runtime construction are described in detail. Section 5.2 presents the details of how it would be implemented on a multicontext FPGA. The actual implementation on a conventional FPGA (Xilinx XC6216) is presented in Section 5.3. Finally, performance is evaluated in Section 5.4.

5.1 Hardware Realization

We describe Phase 2 of the algorithm first. Logic is constructed at runtime in Phase 1 and used in Phase 2. Knowing what the constructed logic looks like and how it works makes it easier to understand the subsequent description of Phase 1.

5.1.1 Phase 2: Text Search

The datapath used for Phase 2 (see Figure 2) is shown in Figure 4. The text T is stored in external memory. The index i in the algorithm is essentially an address counter used to fetch the next text character. The entire pattern P is stored on-chip. The comparator is used to compare the appropriate text and pattern characters. The last major logic block implements the prefix function π .

The operation of the datapath can be easily understood by looking at Figure 4. Each clock cycle, the four **if** conditions are evaluated in parallel but only one of the statements is executed. The values of the signals `char_match`, `state_zero` and `state_final` determine which of the four paths is selected. The controller generates appropriate values for the signals `inc_i`, `inc_match`, `next_state` and `inc_state`. If `next_state` is 0, q remains unchanged for the clock cycle. Otherwise, $++q$ (`state_inc=1`) or $q = \pi[q]$ (`state_inc=0`) is performed. To improve performance, the implementation overlaps fetching $T[i]$ with datapath operation.

Prefix Function FSM As described in Section 4, the prefix function π can be implemented as a FSM. The FSM contains m states, 0 to $m - 1$. The state corresponding to the value of q is the current state.

There are two standard techniques for implementing FSMs using programmable logic [9]. One way is using a LUT that stores the FSM states in a (typically binary) encoded form. As the FSM size increases, the speed decreases and area required increases because of the wider and deeper decoding logic and the associated routing. Also, in the our case two comparators would be required for generating the `state_zero` and `state_final` signals.

The other approach is to use the One-Hot Encoding (OHE) scheme—one flip-flop is associated with each state. At anytime exactly one flip-flop has a 1 bit signifying the current state. This approach is simpler and more efficient as it requires lesser decoding logic and suits the flip-flop rich architecture of FPGAs.

We exploit properties of π to develop a particularly compact and simple implementation of the FSM. There are exactly two possible transitions from each state. One of these is to the following state (forward edge) and the other is to one of the previous states (backward edge). These properties simplify the routing considerably. In addition, the signals `initial_state` and `final_state` are simply the outputs of the initial and final state flip-flops respectively, eliminating the need for any comparators.

5.1.2 Phase 1: Prefix Function Construction

As can be seen from Figure 2, Phase 1 is similar to Phase 2. Two minor differences are that i is initialized to 1 and the pattern P is compared with itself instead of text T. The only major difference is additional steps for constructing the prefix function π through assignments to $\pi[i]$. In terms of logic, these assignments translate to constructing the back edges of all the states of the FSM. Construction of the FSM at runtime and the logic required to do so are described below.

Online FSM Construction The FSM for the given pattern is constructed using a preconfigured *template*. The FSM template, shown in Figure 5 is independent of the pattern and constructed beforehand. Flip-outputs go to the next flip-flop (forward edges) and to horizontal wires (which runtime back edge construction described below). At any time during execution, only the flip-flop for state q has a 1-bit. The template also has storage for the pattern P with $P[q]$ available as the output of the rightmost mux.

At runtime, the first step is to customize the template for the input pattern size m. This is done by connecting the output of flip-flop for state $m - 1$ to the horizontal wire that is the lower input to the state 0 flip-flop. This is followed by loading and storing the pattern on-chip. Next Phase 1 starts, and the execution of statements $\pi[i + 1] = q$ and $\pi[i + 1] = 0$ in the Phase 1 algorithm results in the construction a back edge from state $i + 1$ to state q or state 0 respectively. As can be seen from Figure 6, this is only a matter of inserting an OR gate at the appropriate position. The piece of logic that constructs back edges takes q and i as inputs and computes the position (i^{th} row and q^{th} column) at which the OR gate is to be inserted. See Section 5.3 for implementation details of this logic.

In this manner, problem instance dependent logic is mapped within clock cycles, instead of minutes or hours that would be required if software was in the loop. Another interesting feature is that in FSM construction alternates with FSM use (whenever π is read in Phase 1). Such a fine grained interleaving would not be possible without self-reconfiguration.

5.2 Proposed Implementation on a multicontext FPGA

Before computation begins, the pattern P, pattern length m, text T and text length n are stored in external memory that can be accessed by the multicontext FPGA. The following logic is configured onto four contexts of the FPGA. Context 0 contains control logic that governs overall execution of the algorithm. Context 1 has logic for customizing the FSM for given m. Context 2 contains datapath for Phase 1 of the KMP algorithm as well as logic for runtime FSM construction. Hardwired into this logic are configuration bits for the OR-gate and its connections (referred to as *or_gate*). The number of configuration memory writes needed for OR-gate insertion is s_{or_gate}. The FSM is constructed on context 3 in Phase 1. During Phase 2 it includes the datapath required for Phase 2 as well.

Figure 7 shows the computation performed in each context (computation done in context 3 during Phase 1 and Phase 2 is shown as context 3a and context 3b respectively). At the end of each statement is the time required by the logic to execute it. The times are expressed in terms of t_{cm} (configuration memory read or write time), t_{em} (external memory read or write time), t_{clk} (one clock cycle time), t_{cs} (time required to switch contexts) and s_{or_gate}. Computation starts with context 0 switching to context 1 which customizes the FSM size. The FSM is constructed on a separate context since

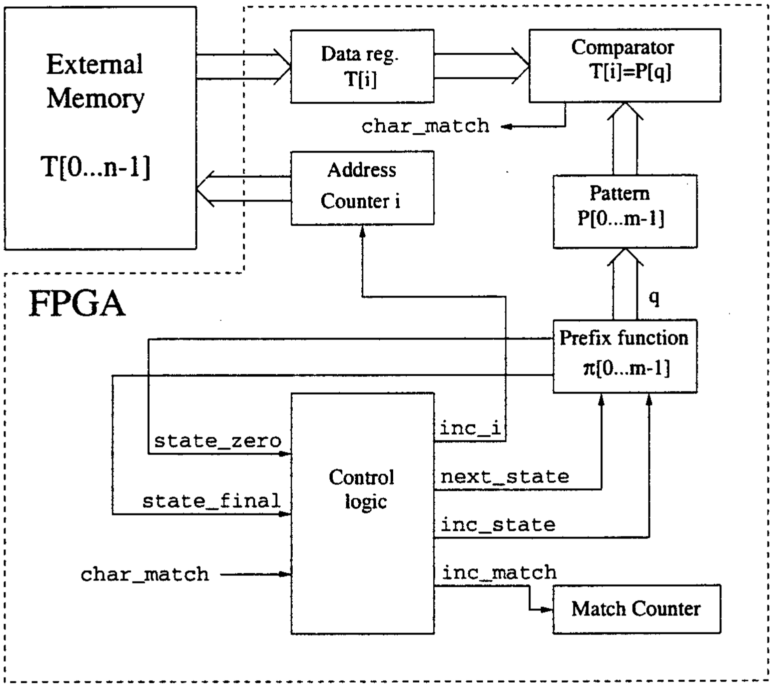


Figure 4: Datapath for Phase 2.

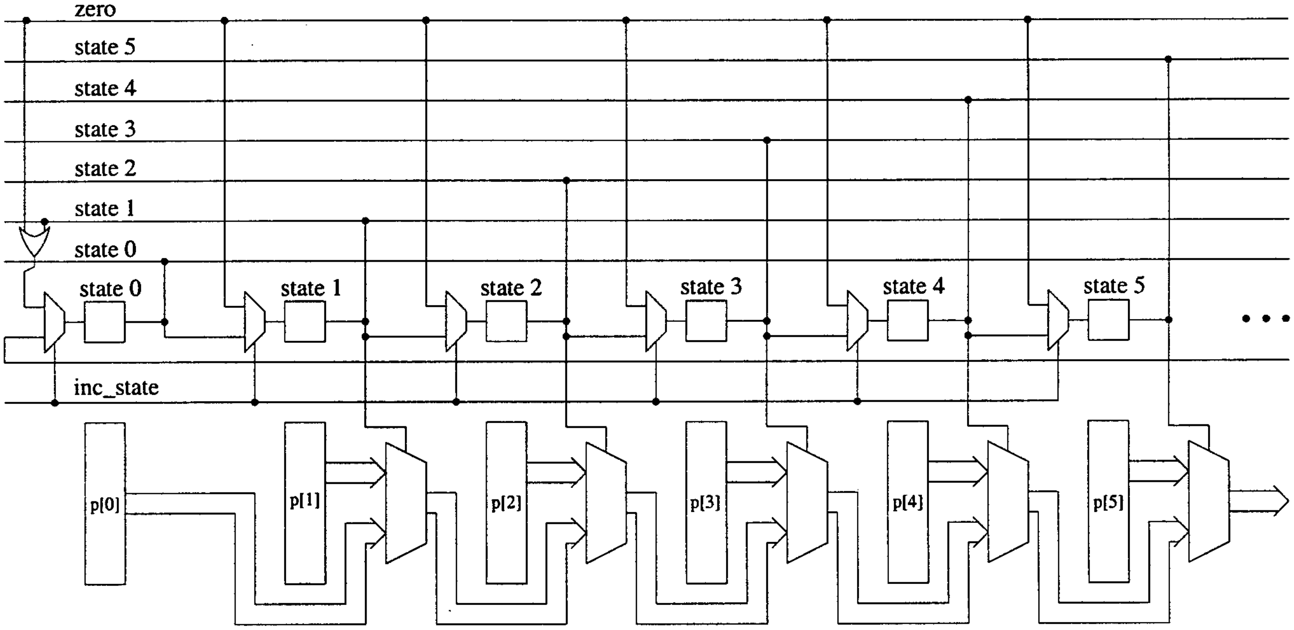


Figure 5: FSM template. The OR-gate implements $\pi[1] = 0$.

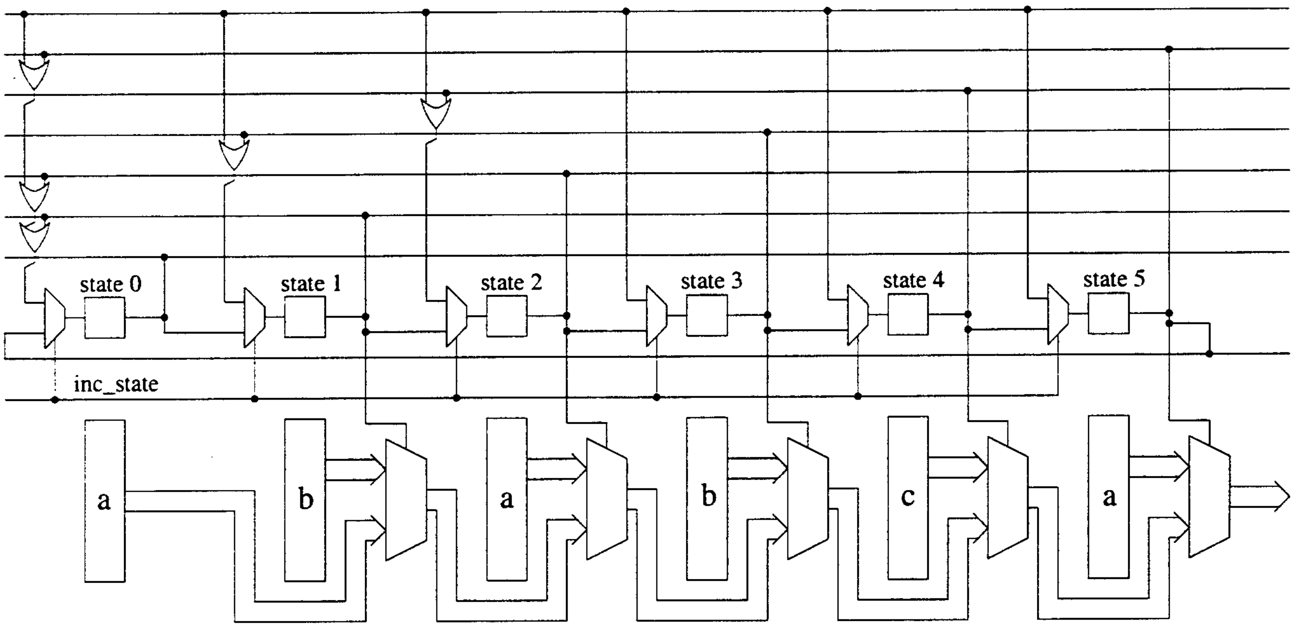


Figure 6: Back edges built through OR-gate insertion. Corresponds to FSM in Figure 3.

the currently executing context cannot modify itself. Doing do requires data sharing between contexts which is possible on multicontext FPGAs [13]. In Figure 7, *read_em* and *write_em* refer to an external memory access while *read_cm* and *write_cm* refer to a context memory access. Note that no external intervention by the host machine is required in constructing the FSM.

Next, the logic on context 2 performs Phase 1 of the KMP algorithm. Self-reconfiguration is performed via configuration memory writes to construct the appropriate back edges. Note how the FSM back edge construction alternates with use of the partially constructed FSM (by switching to context_3) alternates every few clock cycles. Finally context 0 connects the FSM to the text search datapath already present on context 2. Since their positions are fixed beforehand, the datapath can be interfaced with the runtime generated FSM to form the complete logic required for performing Phase 2 of the KMP algorithm.

The context switching is similar to context switching of processes on a uniprocessor. At a time only one of the FPGA contexts executes and switching to a context resumes its execution from where it had stopped earlier due to a context switch. This is possible because the state of the active context (bits stored in all the flip-flops) are saved before switching to a different context.

We now derive T_M, T_{ME} and T_E in terms of the times in Figure 7. T_{ME} is the time spent in write_cm operations. From the times in Figure 7,

$$T_{ME} = (m-1)s_{or\_gate}t_{cm} \qquad (1)$$

The remaining time spent in contexts 1, 2 and 3a is T_M, the time required to compute the FSM mapping and is given by [2]

$$T_M = (4m-2)t_{cs} + (m+1)t_{em} + (7m-4)t_{clk} \qquad (2)$$

Finally, the execution time T_E is the time spent in Phase 2 which is[3]

$$T_E = (2n - \frac{n}{m})t_{clk} \qquad (3)$$

A few remarks on how the above times were determined— read_em P and write_cm P are pipelined and take $(m+1)t_{em}$ time. In context 3b, only one if statement is executed each iteration taking t_{clk} time. Similarly context 3a also takes t_{clk} time. The execution time of context 2 depends upon the input pattern and the worst case occurs when all characters are identical and the last if statement is executed each iteration. The worst case time is used in T_m above.

5.3 Actual implementation on a conventional FPGA

We implement logic described for contexts 2, 3a and 3b in the previous section on a Xilinx XC 6216 device. From the implementation we determine t_{clk} and t_{em} and t_{cm}[4]. And by using a t_{cs} value based on published context switching times, we obtain using equations 1, 2 and 3, an accurate performance estimate of the KMP algorithm implemented on an abstract multicontext version of the XC 6216. The feasibility of such a device should not be in doubt since the extensions we assume have been demonstrated in various multicontext devices built so far.

The VCC Hotworks board was used for the implementation. Required logic was specified in structural VHDL and translated to EDIF format using velab. XACT 6000 was used for place, route and configuration file generation. For debugging and runtime support, XC 6200 Inspector and PCI Test were used. The 128 KB of SRAM (referred to as external memory henceforth) on the VCC board was used to simulate the configuration memory of a multicontext device.

[2] This is the worst case T_M which corresponds to a pattern containing all identical characters except the last one.

[3] This is the worst case T_E which corresponds to text containing m character repetitions in each of which the first $m-1$ characters match the pattern and the last one does not.

[4] We make the conservative assumption that $t_{cm} = t_{em}$.

```
context_0
/*Stage 1 of FSM construction.*/
switch context_1; $t_{cs}$
/*Stage 2 and Phase 1.*/
switch context_2; $t_{cs}$
/*Phase 2.*/
connect Phase 2 datapath; $t_{clk}$
switch context_3; $t_{cs}$
```

```
context_1
read_em $m$; $t_{em}$
/*Connect final state output to state 0 input.*/
connect flip-flop $m - 1$; $t_{clk}$
/*Store pattern characters in pattern registers.*/
read_em $P$; $mt_{em}$
write_cm $P$; $mt_{cm}$
switch context_0; $t_{cs}$
```

```
context_2
$i = 1; q = 0;$ 0
read_em $m$; $t_{em}$
while ($i < m$) do
  /* One if statment executed every iteration.*/
  if ($P[i] \neq P[q]$)and($q == 0$) then
    ++i; $t_{clk}$
    /*Create back edge for $\pi[i] = 0$.*/
    compute OR gate insertion position; $t_{clk}$
    write_cm $or\_gate$; $s_{or\_gate}t_{cm}$
  end if
  if ($P[i] \neq P[q]$)and($q \neq 0$) then
    /*Switch to FSM context and perform $q = \pi[q]$.*/
    state_inc=0; $t_{clk}$
    switch context_3; $t_{cs}$
  end if
  if ($P[i] == P[q]$) then
    /*Switch to FSM context and perform $++q$.*/
    $++i$; $inc\_state = 1$; $t_{clk}$
    switch context_3; $t_{cs}$
    /*Create back edge for $\pi[i] = q$.*/
    compute OR gate insertion position; $t_{clk}$
    write_cm $or\_gate$; $s_{or\_gate}t_{cm}$
  end if
end while
switch context_0; $t_{cs}$
```

```
context_3a
if ($inc\_state == 1$) then
  if ($q == m - 1$) then $q = 0$; else $++q$; $t_{clk}$
end if
if ($inc\_state \neq 1$) then
  $q = \pi[q]$; $t_{clk}$
end if
switch context_0; $t_{cs}$
```

```
context_3b
$i = 0; q = 0;$ 0
read_em $n$; $t_{em}$
while ($i < n$) do
  /* One if statment executed every iteration.*/
  if ($T[i] \neq P[q]$)and($q == 0$) then
    ++i; $t_{clk}$
  end if
  if ($T[i] \neq P[q]$)and($q \neq 0$) then
    $q = \pi[q]$; $t_{clk}$
  end if
  if ($T[i] == P[q]$)and($q \neq m - 1$) then
    $++i$; $++q$; $t_{clk}$
  end if
  if ($T[i] == P[q]$)and($q == m - 1$) then
    /*Pattern match found.*/
    $++i$; $++q$; $++matches$; $t_{clk}$
  end if
end while
```

Figure 7: KMP algorithm implementation on a multicontext FPGA.

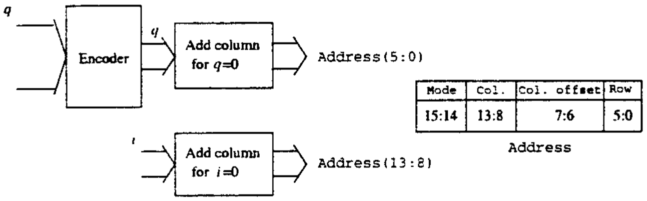


Figure 8: Generating configuration memory address for OR-gate insertion. Address bits 15:14 and 7:6 are constant and known beforehand.

For Phase 1 we implement on the XC 6216 the Phase 1 datapath, OR-gate construction logic and the FSM template. All this logic corresponds to contexts 2 and 3a in Figure 7. For each back edge, the address in configuration memory where the OR-gate is to be inserted is written out to external memory (in one clock cycle). This information is used to modify the configuration file which is used to reconfigure the FPGA for computing the next back edge. Knowing row and column of a logic cell, it is trivial to compute the corresponding configuration addresses since the row and column numbers directly form a part of the 6200 address. The logic for OR-gate computation is thus quite simple and is shown in Figure 8. Inserting the OR-gate and making the appropriate connections needs just 24 bits of configuration data which is embedded in the logic itself. Three separate writes are required however since each byte needs to be written to a separate address. Thus $s_{or\_gate} = 3$. For Phase 2 we implement logic corresponding to context 3b on the XC 6216. The logic searches through text stored in the external memory just as a multicontext FPGA would since no context switching is involved in this phase.

5.4 Performance Evaluation

From the implementation description in Section 5.3 it should be clear that $t_{cm} = t_{em} = t_{clk}$. Based on published literature, we make the conservative assumption that $t_{cs} = 100$ns. We determine t_{clk} as follows. For a given pattern size, we increase the clock frequency till any further increase makes the implemented logic stop working correctly. The corresponding clock period is the value of t_{clk}. t_{clk} increases somewhat with pattern size since the corresponding FSM is bigger and hence the critical path is longer. Plugging all the above values into equations 1, 2 and 3 for pattern size m varying from 4 to 16, and text size $n = 10^4$ characters, we obtain the results shown in Table 1.

m	t_{clk}	T_M	T_{ME}	T_E	Total time
4	81.6 ns	3.7 μs	0.7 μs	1428 μs	1432 μs
8	97.6 ns	9.0 μs	2.1 μs	1830 μs	1841 μs
16	129.6 ns	22.4 μs	5.8 μs	2511 μs	2539 μs

Table 1: Performance of the implementation for various values of m with $n = 10^4$.

We now compare the mapping time ($T_M + T_{ME}$) of the proposed multicontext FPGA approach with other approaches. Consider the case where CAD tools are used to perform the FSM construction. To find T_M for this approach, we determine the time taken to compile a structural VHDL description[5] for $m = 8$ using velab (4 s) and route it using XACT 6000 (68 s) giving $T_M = 72$ s. $T_{ME} = 1$ ms is the time required to download the configuration onto the XC 6216 via the PCI bus. To make T_M as small as possible, we explicitly specify placement of logic and use XACT 6000 only for routing. Even then, as can be seen from row 2 Table 2, the proposed approach is six orders of magnitude faster than the naive use of CAD tools. Of course a multicontext FPGA is needed to obtain the speedup. A smarter approach would be to write a program that directly modifies the binary configuration file based on the input pattern. This approach is essentially doing in software what we do on the FPGA itself. Row 3 of Table 2 shows the performance of this approach[6]. Although much faster than the CAD tools approach, it is still more than 1800 times slower than the proposed approach.

Table 3 shows the total execution time speedups over other approaches. We also compare the performance with a software implementation of the KMP algorithm running on a Sun Ultra 1 Model 140. As can be seen from row 4 of Table 3, reasonable speedups are obtained. A key point to note is that the multicontext FPGA is better than others for all values of n. This is in contrast to most reported results where the problem size must be very large to amortize the high mapping time.

Comparison of the implementation with other FPGA based string matching implementations is unfortunately not possible due to differences in the FPGA architectures and the algorithms used. We note however, that in [7] $T_M = 0.16s$ and $T_{ME} = 3.05s$. These times are for a naive string matching implementation on 16 CAL1024 FPGAs that runs at 20 MHz. Thus, in [7], speedups will be obtained only for very large problem sizes due to the high $T_M + T_{ME}$.

6 Conclusion

We have shown dramatic speedups in the time required to map logic at runtime onto FPGAs. This is done by the novel approach of developing logic that maps logic and putting the former on the FPGA itself. As a result CAD tools need to be used just once for each problem (to build logic that builds logic and some template logic) and not once for every problem instance as is usually done. The reduction in mapping time achieved is extremely important because FPGAs can do better than ASICs only if the mapping is problem instance dependent, which means that the runtime mapping time is a part of the overall execution time.

We show how self-reconfiguration can be performed using multicontext FPGAs and how to efficiently realize the above approach through self-reconfiguration. We demonstrate our approach by presenting a detailed implementation of the KMP string matching algorithm which utilizes the above approach to construct a FSM at runtime. An interesting feature of the implementation is that FSM construction and use of the FSM alternate every few clock cycles. Such a fine grained interleaving of mapping logic and using it would not be possible with software in the loop.

Finally, we implement the KMP algorithm on a conventional FPGA and use it to obtain accurate estimates of performance on a multicontext device. Our results show high speedups in mapping time

[5] We ignore the time required to generate the VHDL code for the given input pattern as it would be quite small. In any case, accounting for this time would only improve our speedup. The times are obtained on an IBM PC with a 200 MHz Pentium Pro and 64 MB RAM.

[6] The time T_M is for a C program running on a Sun Ultra 1 Model 140.

Approach	T_M	T_{ME}	$T_M + T_{ME}$	Speedup
Multicontext FPGA	9.0 μs	2.1 μs	11.1 μs	1.0
CAD tool mapping	76 s	1 ms	76 s	$\approx 6 \times 10^6$
Software mapping	20 ms	1 ms	21 ms	1892

Table 2: Speedup in mapping time (m = 8).

Approach	$T_M + T_{ME} + T_E$			Speedup		
	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^4$	$n = 10^5$	$n = 10^6$
Multicontext FPGA	1.8 ms	18.3 ms	183.1 ms	1.0	1.0	1.0
CAD tool mapping	76.0 s	76.0 s	76.2 s	$\approx 10^5$	$\approx 10^4$	$\approx 10^3$
Software mapping	21.8 ms	39.3 ms	204.1 ms	12.1	2.1	1.1
Sun Ultra 1	30 ms	80 ms	680 ms	16.6	4.4	3.7

Table 3: Speedups over other approaches for various values of n, with m=8.

and reasonable speedups in overall execution time over various existing approaches.

This work has been done as a part of the MAARC (Models, Algorithms and Architectures for Reconfigurable Computing) project. The MAARC project is developing a framework of algorithmic techniques for reconfigurable computing and exploiting this technology for embedded signal and image processing applications. Please see [10] for more information.

References

[1] Babb, J., Frank, M., and Agarwal, A. Solving graph problems with dynamic computation structures. In *Proceedings of the SPIE - The International Society for Optical Engineering* (Boston,MA, 1996).

[2] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachussets, 1990.

[3] DeHon, A. Multicontext Field-Programmable Gate Arrays. http://HTTP.CS.Berkeley.EDU/ãmd/CS294S97/papers/dpga_cs294.ps.

[4] DeHon, A. DPGA-coupled microprocessors: Commodity ICs for the early 21st century. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1994), D. A. Buell and K. L. Pocek, Eds., pp. 31–39.

[5] Donlin, A. Self modifying circuitry - a platform for tractable virtual circuitry. In *Eighth International Workshop on Field Programmable Logic and Applications* (1998).

[6] French, P. C., and Taylor, R. W. A self-reconfiguring processor. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1993), D. A. Buell and K. L. Pocek, Eds., pp. 50–59.

[7] Gunther, B., Milne, G., and Narasimhan, L. Assessing document relevance with run-time reconfigurable machines. In *Proceedings of the 4th IEEE Symposium on FPGAs for Custom Computing Machines* (Napa, California, Apr 1996).

[8] Jones, D., and Lewis, D. A time-multiplexed FPGA architecture for logic emulation. In *Proceedings of the 1995 IEEE Custom Integrated Circuits Conference* (May 1995), pp. 495–498.

[9] Knapp, S. K. Accelerate FPGA macros with one-hot approach. *Electronic Design 38*, 17 (1990), 65–71.

[10] MAARC project. http://maarc.usc.edu.

[11] Motomura, M., Aimoto, Y., Shibayama, A., Yabe, Y., and Yamashina, M. An embedded DRAM-FPGA chip with instantaneous logic reconfiguration. In *1997 Symposium on VLSI Circuits Digest of Technical Papers* (June 1997), pp. 55–56.

[12] Motomura, M., Aimoto, Y., Shibayama, A., Yabe, Y., and Yamashina, M. An embedded DRAM-FPGA chip with instantaneous logic reconfiguration. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1998). Poster paper.

[13] Scalera, S. M., and Vázquez, J. R. The design and implementation of a context switching FPGA. In *IEEE Symposium on Field-Programmable Custom Computing Machines* (Apr. 1998), pp. 495–498.

[14] Trimberger, S., Carberry, D., Johnson, A., and Wong, J. A time-multiplexed FPGA. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1997), J. Arnold and K. L. Pocek, Eds., pp. 22–28.

[15] Zhong, P., Martonosi, M., Ashar, P., and Malik, S. Accelerating boolean satifiability with configurable hardware. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1998).

6th IEEE Reconfigurable Architectures Workshop, April 1999.

Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices*

Andreas Dandalis, Alessandro Mei**, and Viktor K. Prasanna

University of Southern California
{dandalis, prasanna, amei}@halcyon.usc.edu
http://maarc.usc.edu/

Abstract. Conventional mapping approaches to Reconfigurable Computing (RC) utilize CAD tools to perform the technology mapping of a high-level design. In comparison with the execution time on the hardware, extensive amount of time is spent for compilation by the CAD tools. However, the long compilation time is not always considered when evaluating the time performance of RC solutions. In this paper, we propose a domain specific mapping approach for solving graph problems. The key idea is to alleviate the intervention of the CAD tools at mapping time. High-level designs are synthesized with respect to the specific domain and are adapted to the input graph instance at run-time. The domain is defined by the algorithm and the reconfigurable target. The proposed approach leads to predictable RC solutions with superior time performance. The time performance metric includes both the mapping time and the execution time. For example, in the case of the single-source shortest path problem, the estimated run-time speed-up is 10^6 compared with the state-of-the-art. In comparison with software implementations, the estimated run-time speed-up is asymptotically 3.75 and can be improved by further optimization of the hardware design or improvement of the configuration time.

1 Introduction

Reconfigurable Computing (RC) solutions have shown superior execution times for several application domains (e.g. signal & image processing, genetic algorithms, graph algorithms, cryptography), compared with software and DSP based approaches. However, an efficient RC solution must achieve not only minimal execution time, but also minimal time for mapping onto the hardware [5, 7].

Conventional mapping approaches to RC (see Fig. 1) utilize CAD tools to generate hardware designs optimized with respect to execution time and area.

* This research was performed as part of the MAARC project. This work is supported by the DARPA Adaptive Computing Systems program under contract no. DABT63-96-C-0049 monitored by Fort Hauchuca.

** A. Mei is with the Department of Mathematics of the University of Trento, Italy. This work was performed while he was visiting the University of Southern California.

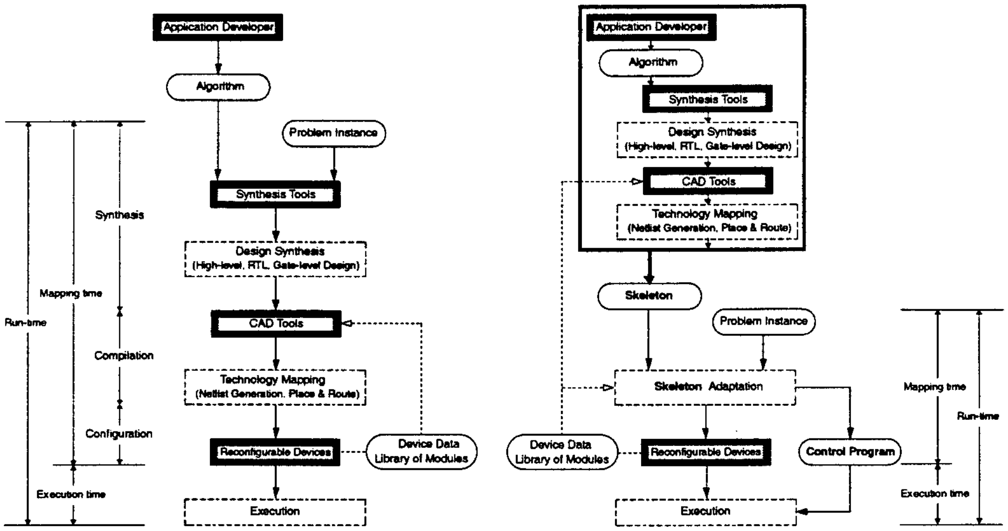


Fig. 1. Conventional (left) and Domain Specific (right) Mapping Approaches

The resulting mappings incur several overheads due to the dominant role of the CAD tools. The clock rate and the required area of the RC solution depend heavily on the CAD tools used and cannot be estimated reliably at compile time. Moreover, the technology mapping phase requires extensive compilation time. Usually, several hours of compilation time is required to achieve execution time in the range of *msec* [1,7]. In the case of mappings that are reused over time, compilation occurs only once and is not a performance bottleneck. But, for mappings that depend on the input problem instance, the mapping time cannot be ignored and often becomes a serious performance limitation.

In this paper we propose a novel RC mapping approach for solving graph problems. For each input graph instance, a new mapping is derived. The objective is to derive RC solutions with superior time performance. The time performance metric is the running time which is defined as the sum of the mapping time and the execution time. The key idea of the proposed approach is to reduce the intervention of the CAD tools at mapping time. A technology dependent design is synthesized based on the specific domain [algorithm,target]. The reconfigurable target is now visible to the application developer (see Fig. 1). At run-time, the derived design is adapted to the input graph instance. The proposed mapping approach eliminates the dominant role of the CAD tools and leads to “real-time” RC solutions. Furthermore, the time performance and the area requirements can be accurately estimated before compilation. This is particularly important in run-time environments where the parameters of the problem are not known *a priori* but time and area constraints must be satisfied.

In Section 2, the proposed mapping approach is briefly described. In Section 3, to illustrate our approach, we demonstrate a solution for the single-source shortest path problem. Finally, in Section 4, concluding remarks are made.

2 Domain Specific Mapping

A simple and natural model is assumed for application development. It consists of a host processor, an array of FPGAs, and external memory. The FPGAs are organized as a 2D-mesh. The memory stores the configuration data to be downloaded to the array and the data required during execution. The role of the memory is analogous to the role of the cache memory in a memory system in terms of providing a high-speed link between the host and the array. To illustrate our ideas, we consider an adaptive logic board from Virtual Computer Corporation to map our designs. This board is based on the XC6200 architecture.

In this paper, we consider graph problems as the application domain. Each graph instance leads to a different mapping. Thus, the mapping overhead cannot be ignored. Given a specific domain [algorithm,target], the objective is to derive a working implementation with superior time performance. The time performance metric is the running time which is defined as the sum of the mapping time and the execution time (see Fig. 1). To obtain the hardware implementation, an algorithm specific *skeleton* is synthesized based on the specific domain and is dynamically adapted to the input graph instance at run-time. The proposed mapping approach consists of three major steps (see Fig. 1):

1. **Skeleton design** For a given graph problem, a general structure (*skeleton*) is derived based on the characteristics of the specific domain. The *skeleton* consists of modules that correspond to elementary features of the graph (i.e. graph vertex). The modules are optimized hardware designs and their functionality is determined by the algorithm. Configurations for the modules and their interconnection are derived based on the target architecture.
   The interconnection of the modules is fixed and is defined to be general enough to capture the individual connectivity of different graph instances. Hence, the placement and routing of the modules are less optimized than in conventional CAD tools based approaches. The *skeleton* is derived before compilation and its derivation does not affect the running time. In addition, the *skeleton* exploits low-level hardware details of the reconfigurable target in terms of logic, placement, and routing.
2. **Adaptation to graph input instance** Functional and structural modifications are performed to the *skeleton* at run-time. Such modifications are dictated by the characteristics of the problem instance based on which the configuration of the final layout is derived.
   The functional modifications dynamically add or alter module logic to adapt the modules to the input data precision and problem size. The structural modifications shape the interconnection of the skeleton based on the characteristics of the problem instance.
   A software program (Control Program) is also derived to manage the execution in the FPGAs. This program schedules the operations and the on-chip data flow based on the computational requirements of the problem instance. In addition, it coordinates the data flow to/from the hardware implementation. Since the interconnection of the *skeleton* is well established in Step

1, the execution scheduling essentially corresponds to a software routing for the adapted *skeleton*.

3. **Configuration** Finally, the reconfigurable target is configured based on the adapted structure derived in Step 2. After the completion of the configuration, the control program is executed on the host to initiate and manage the execution on the hardware.

The proposed approach leads to RC solutions with superior time performance compared with conventional mapping approaches. Furthermore, in our approach, the *skeleton* mainly determines the clock rate and the area requirements. Hence, reliable time and area estimates are possible before compilation.

3 The Single-Source Shortest Path problem

To illustrate our ideas, we demonstrate a mapping scheme for the single-source shortest path problem. It is a classical combinatorial problem that arises in many optimization problems (e.g. problems of heuristic search, deterministic optimal control problems, data routing within a computer communication network) [2]. Given a weighted, directed graph and a source vertex, the problem is to find a shortest path from the source to every other vertex.

3.1 The Bellman-Ford Algorithm

For solving the single-source shortest path problem, we consider the Bellman-Ford algorithm. Figure 2 shows the pseudocode of the algorithm [3]. The edge weights can be negative. The complexity of the algorithm is $O(ne)$, where n is the number of vertices and e is the number of edges.

```
The Bellman-Ford algorithm
Initialize G (V,E)
  FOR each vertex i ε V
    DO label(i) ← ∞
  label(source) ← 0
Relax edges
  FOR k = 1.. n-1
    DO FOR each edge (i,j) ε E
      DO label(j) ←min {label(j), label(i) + w(i,j)}
Check for negative-weight cycles
  FOR each edge (i,j) ε E
    DO IF label(j) > label(i) + w(i,j)
      THEN return FALSE
  return TRUE
```

Problem Siz e	# iter ations
# vertices x # edges	m* (average)
16 x 64	4.52
16 x 128	4.40
16 x 240	4.31
64 x 256	4.13
64 x 512	4.06
64 x 1024	4.03
128 x 512	5.32
128 x 1024	4.96
128 x 2048	4.97
256 x 1024	6.04
256 x 2048	5.75
256 x 4096	5.86
512 x 2048	7.02
512 x 4096	6.71
512 x 8192	6.76
1024 x 4096	7.40
1024 x 8192	7.77
1024 x 16384	7.80

Fig. 2. The Bellman-Ford Algorithm and experimental results for m^*

For graphs with no negative-weight cycles reachable from the source, the algorithm may converge in less than $n-1$ iterations [2]. The number of required iterations m^*, is the height of the shortest path tree of the input graph. This height is equal to the maximum number of edges in a shortest path from the source. In the worst case $m^* = n-1$, where n is the number of the vertices.

We performed extensive software simulations to determine the relation between m^* and $n-1$ for graphs with no negative-weight cycles. Note that known RC solutions [1] always perform $n-1$ iterations of the algorithm, regardless the value of m^*. Figure 2 shows the experimental results for different problem sizes. For each problem size, $10^5 - 10^6$ graph instances were randomly generated. Then, the value of m^* for each graph instance was found and the average over all graph instances was calculated. For the considered problem sizes, the number of required iterations grows logarithmically as the number of vertices increases. For values of e/n smaller than those in the table in Fig. 2, m^* starts converging to $n-1$.

3.2 Mapping the Bellman-Ford algorithm

The skeleton The *skeleton* corresponds to a general graph G(V,E) with n vertices and e edges. A weight $w(i,j)$ is assigned for each edge $(i,j) \in$ E (i.e. edge from vertex i to vertex j). The derived structure (see Fig. 3) consists of n modules connected in a pipelined fashion. An index $id = 0, 1, .., n-1$ and a *label* are uniquely associated with each module. Module i corresponds to vertex i. The weight of the edges is stored in the memory. No particular ordering of the weights is required. Each memory word consists of the weight $w(i,j)$ and the associated indices i and j.

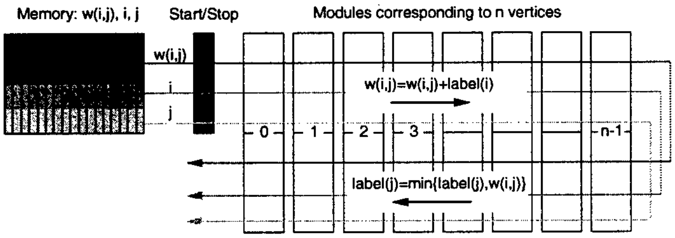


Fig. 3. The *skeleton* architecture for the Bellman-Ford algorithm

The Start/Stop module initiates execution on the hardware. An iteration corresponds to the e cycles needed to feed once the contents of the memory to the modules. The weights $w(i,j)$ are repetitively fed to the modules every e cycles. The algorithm terminates after m^* iterations. One extra iteration is required for the Start/Stop module to detect this termination. If no labels are modified during an iteration and $m^* \leq n$, the graph contains no negative-weight cycles reachable from the source and a solution exists. Otherwise, the graph contains a negative-weight cycle reachable from the source and no solution exists.

In each module (see Fig. 4), the values *id* and *label* are stored and the relaxation of the corresponding edges is performed. In the upper part, the *label* is added to each incoming weight $w(i,j)$. The index i is compared with *id* to determine if the edge (i,j) is incident from vertex *id*. The weight $w(i,j)$ is updated only if $i = id$. In the lower part, the weight $w(i,j)$ is relaxed according to the *min* operation of the algorithm as shown in Figure 2. The index j is compared with *id* to determine if the edge (i,j) is incident to the vertex *id*. The *label* of the vertex *id* is updated only if $j = id$ and $w(i,j) < label$. When *label* is updated, a flag U is asserted.

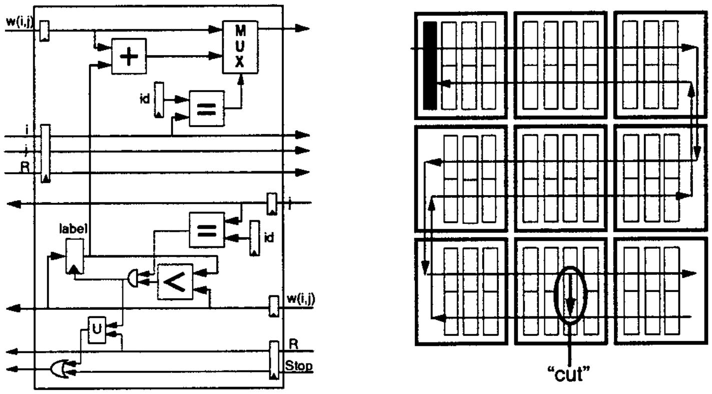


Fig. 4. The structure of the modules (left) and the placement of the *skeleton* into the FPGA array (right)

At the beginning of each iteration, the signal R is set to 1 by the Start/Stop module to reset all the flags. In addition, R resets a register that contains the signal *Stop* in module $n-1$. The signal *Stop* travels through the modules and samples all the flags. At the end of each iteration, the *Stop* signal is sampled by the Start/Stop module. If $Stop = 0$ and $m^* \leq n$, the execution terminates and a solution exists. Otherwise, if $Stop = 1$ after n iterations, no solution exists.

The *skeleton* placement and routing onto the FPGAs array (see Fig. 4) is simple and regular. The communication between consecutive modules is uniform and differs only at the boundaries of the array. Depending on the number of the required modules, a "cut" (Fig. 4) is formed that corresponds to the communication links of the last module (Fig. 3). During the adaptation of the *skeleton* the "cut" is formed and the labels in the allocated modules are initialized. The execution is managed by a control program executed on the host. This program controls the memory for feeding the required weights to the array. In addition, it initiates and terminates the execution via the Start/Stop module.

Area and running time estimates The above module was created based on the parametrized libraries for Xilinx 6200 series of FPGAs [9]. The footprint of each module was $(p + \lceil \log n \rceil) \times (4p + 2\lceil \log n \rceil + 10)$, where p denotes the number of bits in each weight/label, and n is the number of vertices. For $p = \log n = 16, 4$ modules can be placed in the largest device of the XC6200 family. The memory space required was $(p + 2\lceil \log n \rceil) \times e$ bits, where e is the number of edges. The needed memory-array bandwidth is $p + 2 \log n$ bits/cycle to support the execution. To fully utilize the benefits of the FastMAP^{TM} interface, 140 MB/sec bandwidth is required. Under this assumption, the largest XC6200 device can be configured in 165 μsec using wildcards [8].

The algorithm terminates after $(m^* + 1) \times e + 2n$ cycles, where m^* is the number of required iterations for a given graph. One cycle corresponds to the clock period of the *skeleton*. The clock rate for the *skeleton* was estimated to be at least 15 MHz for p=16 bits, and at least 25 MHz for p=8 bits. In the clock rate analysis, all the overheads caused by the routing were considered. The clock rate was determined mainly by the carry-chain adder of the modules. By using a faster adder, improvements in the clock rate are possible. The mapping time was in the range of *msec*. The above mapping time analysis is based on the timings for the FastMAP^{TM} interface in the Xilinx 6200 series of FPGAs databook [8].

3.3 Performance Comparison

In [1], the shortest path problem is solved by using Dynamic Computation Structures (DCSs). The key characteristic of the solution is the mapping of each edge onto a physical wire. The experiments considered only problems with an average out-degree of 4 and a maximum in-degree of 8. For the instances considered, the compilation time was 4-16 hours assuming that a network of 10 workstations was available. Extensive time was spent for placement and routing. Hence, the resulting mapping time eliminated any gains achieved by fast execution time. To make fair comparisons with our solution, we assumed that the available bandwidth for configuring the array is 4 MB/sec as in [1]. Even though, the mapping time for our solution was estimated to be in the *msec* range (see Fig. 5).

	Check for negative-weight cycles	Mapping time	Execution time			Area requirements
			# of iterations	# of cycles	Clock rate	
Solution in [1]	NO	> 4 hours $^{+}$	n-1	n-1	$\Omega(1/n^2)$	$\Omega(n^4)$
Our Solution	YES	~ 100 msec $^{++}$	m*	(m*+1)e+2n	independent of n	O(n+e)

+ a network of 10 workstations was used

++ memory-array bandwidth 4MB/sec is assumed as in [1]

Fig. 5. Performance comparisons with the solution in [1]

Besides the mapping overhead, the mapping of edges into physical wires resulted in several limitations in [1], with respect to the clock rate and the area requirements. The clock rate depended on the longest wire which is $\Omega(n^2)$ in the worst case, where n is the number of vertices. This remark is supported by well-known theoretical results [6] which show that in the worst case, a graph takes $\Omega(n^4)$ area to be laid out, where n is the number of vertices, and that the longest wire is $\Omega(n^2)$ long. Therefore, as n increases, the execution time of their solution drops dramatically. For $n = 16, 64, 128$, the execution time was on the average 1.5-2 times faster than our approach while it became 1.3 times slower for $n = 256$. For larger n, the degradation of performance in [1] is expected to be more severe. Considering both the execution and the mapping time, the resulting speed-up comparing with the solution in [1] was 10^6.

Also, in [1], $n-1$ iterations were always executed and negative-weight cycles could not be detected. If checking for algorithm convergence and negative-weight cycles were included in the design, the resulting longest wire would increase further drastically affecting the clock rate and the excution time. Finally, the time performance and the area requirements in [1] are determined completely by the efficiency of the CAD tools and no reliable estimates can be made before compilation.

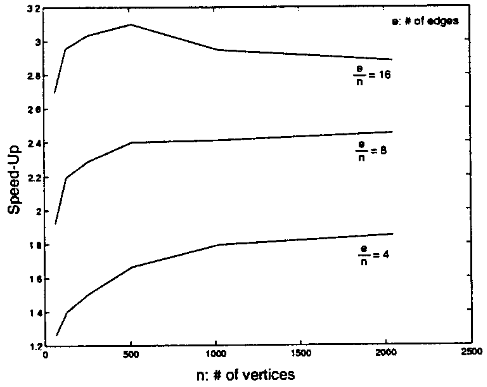


Fig. 6. Comparison of running time: our approach v.s. software implementation.

Area comparisons are difficult to make since different FPGAs were used in [1]. Furthermore, the considered graph instances in [1] were not indicative of the entire problem space since $e/n = 4$. For the considered instances in [1], one XC4013 FPGA was allocated per vertex but, as e/n increases, the area required grows rapidly. In our solution, $O(n)$ area for FPGAs and $O(e)$ memory were required. Moreover, our design is a modular design and can be easily adapted to different graph instances without complete redesign.

Software simulations were also performed to make time performance comparisons with uniprocessor-based solutions. The algorithm that was mapped onto the hardware was also implemented in C language. The software experiments were performed on a Sun ULTRA 1 with 64 MB of memory and a clock rate of 143 MHz. No limitations on the in/out-degree of the vertices were assumed. For each problem instance, $10^4 - 10^6$ graph instances were randomly generated and the average running time was calculated. The compilation time on the uniprocessor to obtain the executable was not considered in the comparisons. Moreover, the data were assumed to be in the memory before execution and no cache effects were considered. Under these assumptions, on the average, an edge was relaxed every 250 *nsec*.

For the hardware implementation, it was assumed that p=16 bits. The mapping time was proportional to the number of vertices of the input graph. Both the mapping and the execution time were considered in the comparisons. The achieved run-time speed-up was asymptotically 3.75. However, for the considered problem sizes (see Fig. 6), lower speed-up was observed. As e/n increases, the mapping time overhead is amortized over the corresponding execution time. Hence, shorter configuration time would result in convergence to the speed-up bound (3.75) for smaller e/n and n.

4 Conclusions

In this paper, a *domain specific* mapping approach was introduced to solve graph problems on FPGAs. Such problems depend on the input graph instance and constitute a suitable application domain to exploit reconfigurability. The proposed approach reduces the dominant role of CAD tools and leads to RC solutions with superior time performance. For example, for the single-source shortest path problem, a speed-up of 10^6 was shown compared with [1]. In comparison with software solutions, an asymptotic speed-up of 3.75 was also shown.

Future work includes more graph problems examples to further validate our mapping approach. In addition, we will focus on specific instances of NP-hard problems where the execution time is comparable to the corresponding mapping time provided by general purpose CAD tools based approaches. We believe that the proposed approach combined with a software/hardware co-design framework can efficiently attack specific instances of NP-hard problems.

References

1. J. Babb, M. Frank, and A. Agarwal, "Solving graph problems with dynamic computation structures", *SPIE '96: High-Speed Computing, Digital Signal Processing, and Filtering using Reconfigurable Logic*, 1996.
2. D. P. Bertsekas, J. N. Tsitsiklis, "Parallel and Distributed Computations: Numerical Methods", Athena Scientific, Belmont, Massachusetts, 1997.
3. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, "Introduction to Algorithms", The Massachusetts Institute of Technology, 1990.
4. B. L. Hutchings, "Exploiting Reconfigurability Through Domain-Specific Systems", *Int. Workshop on Field Programmable Logic and Applications*, Sep. 1997.
5. A. Rashid, J. Leonard, and W. H. Mangione-Smith, "Dynamic Circuit Generation for Solving Specific Problem Instances of Boolean Satisfiability", *IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 1998.
6. J. D. Ullman, "Computational Aspects of VLSI", Computer Science Press, Rockville, Maryland, 1984.
7. P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating Boolean Satisfiability with Configurable Hardware", *IEEE Symposium on Field-Programmable Custom Computing Machines*, Apr. 1998.
8. The XC6200 Field Programmable Gate Arrays Databook, http://www.xilinx.com/apps/6200.htm, April 1997.
9. Parameterized Library for XC6200, H.O.T. Works Ver. 1.1, Aug. 1997.

Field Programmable Custom Computing Machines, FCCM '99

Dynamic Precision Management for Loop Computations on Reconfigurable Architectures*

Kiran Bondalapati and Viktor K. Prasanna
University of Southern California, Los Angeles
{kiran,prasanna}@ceng.usc.edu
http://maarc.usc.edu

Abstract

Reconfigurable architectures promise significant performance benefits by customizing the configurations to suit the computations. Variable precision for computations is one important method of customization for which reconfigurable architectures are well suited. The precision of the operations can be modified dynamically at run-time to match the precision of the operands. Though the advantages of reconfigurable architectures for dynamic precision have been discussed before, we are not aware of any work which analyzes the qualitative and quantitative benefits which can be achieved. This paper develops a formal methodology for dynamic precision management. We show how the precision requirements can be analyzed for typical computations in loops by computing the precision variation curve. We develop algorithms to generate optimal schedules of configurations using the precision variation curves. Using our approach, we demonstrate 25%-37% improvement in the total execution time of an example loop computation on the XC6200 device.

1 Introduction

Reconfigurable hardware has the potential to enhance the performance of many computer applications. The hardware resources can be tuned to the algorithm and the software overhead can be avoided to achieve superior performance compared to conventional microprocessors. Reconfigurable hardware also possesses more flexibility than ASIC hardware and can be utilized for a more diverse set of computations. There are several methods of generating custom hardware configurations suited to the computations to be performed. The ability to perform variable precision arithmetic is one of the significant advantages of reconfigurable hardware.

Reconfigurable hardware such as FPGAs [14, 16] and various custom computing machines (CCMs) [2, 4, 9, 15] contain fine-grained configurable resources. Such fine-grained configurable logic can be utilized to build computing modules of various sizes. The modules can be built to perform computations on various bit-widths. For example, it is possible to build a standard 16-bit×16-bit multiplier or a 8-bit×12-bit multiplier using reconfigurable hardware. The 8-bit×12-bit multiplier would consume less area and execute faster than the standard 16-bit×16-bit multiplier. In configurable hardware, using higher precision usually results in wastage of resources such as logic area, time and power. For example, performing 32-bit multiplications when the operands have only 8 significant bits will typically require 16 times more area and 4 times more execution time. Redundant computations also expend more clock cycles and increase the power consumption. The ability to construct modules of required precision is one of the key advantages of reconfigurable hardware. Variable precision computations can be implemented by using a *static* approach. In the *static* approach, the precision of the operands and operation is fixed at compile time and can be different from the standard precision(e.g. 8-bit, 16-bit, 32-bit, etc.) used on microprocessors. Reconfigurable architectures also support *dynamic precision*, which is the ability of the hardware to change its precision at run-time in response to variant precision demands of the algorithm.

Applications are typically developed to perform operations on standard 32-bit variables. The precision of the operands and the operations is sufficient to guarantee the correctness of the operations in the worst

*This work was supported by the DARPA Adaptive Computing Systems Program under contract DABT63-96-C-0049 monitored by Fort Hauchuca.

case. But in most applications, the actual precision required for computations is usually much lower than the precision implemented. This is typically the case in computations which accumulate values as the computations progress, as in iterative computations such as loops. The precision of the operands increases as the iterations of the loops progress. Loop computations offer the most potential for pipelining and parallelizing in most applications. Configurable hardware is an excellent match for computations with fine-grain pipelining and parallelism. In addition to the performance benefits obtained by mapping of computations in a loop onto configurable hardware, loops can also take advantage of variable precision.

Applications are currently mapped to reconfigurable hardware either by high level behavioral compilers or exhaustive hand-tooled designs. To extract the performance advantages of configurable hardware for variable precision, the trade-offs in performing computations using a very high precision versus changing the precision of computations as the execution progresses need to be evaluated. Performing this analysis by hand and tuning the implementation to the requirements of the application entails significant effort on the part of the designer. Dynamic precision management can result in implementations with lower execution times, logic area and power consumption compared to previous approaches.

For managing dynamic precision in loop computations, intelligent choices on the use of appropriate modules from the available set of modules with different precision need to be made. These configurations then have to be scheduled to achieve optimal execution schedule. We consider a schedule to be optimal if the schedule has minimum *total execution time*, which includes both the execution time in various configurations and the reconfiguration time between configurations. Automatic computation of the actual precision and configurations to be utilized in the computations is the focus of this paper. Currently, a framework for managing dynamic precision computations for any class of computations does not exist. We develop such a framework for loop computations in this paper.

In Section 2 we give an overview of our approach to the *dynamic precision* management problem. Each of the steps in our approach are then described in detail in the later sections. Analysis of the required precision for loop computations is discussed in Section 4. Section 5 describes our Hybrid System Architecture Model(HySAM) of reconfigurable architectures. The variable precision loop mapping problem is defined and our Dynamic Precision Management Algorithm(DPMA) for computing the optimal schedule is presented in Section 6. We illustrate the utility of our approach by showing an example mapping in Section 7. Conclusions and some related problems are discussed in Section 8.

2 Overview of Our Approach

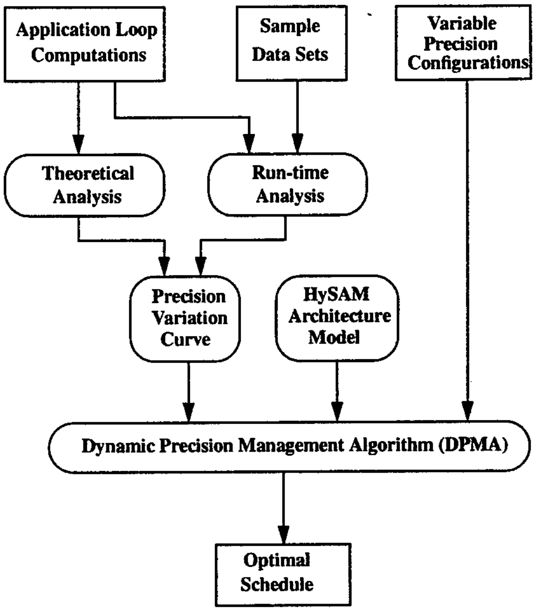

Application Loop Computations
Sample Data Sets
Variable Precision Configurations
Theoretical Analysis
Run-time Analysis
Precision Variation Curve
HySAM Architecture Model
Dynamic Precision Management Algorithm (DPMA)
Optimal Schedule

Figure 1: Overview of our approach for dynamic precision management in loops(shaded and rounded regions indicate our contributions)

This paper details an approach to managing the task of adapting the precision of the implementation to that of the application. An overview of our approach is shown in Figure 1. We focus our efforts on dynamic precision management for loop computations since they are the most compute intensive tasks in typical applications. For the loop computations in applications, we describe an approach to determine the required precision using theoretical analysis and run-time instrumentation. The required precision for the computations in a loop can be expressed as the variation in precision as the iterations of the loop progress. We introduce the concept of the *precision variation curve* to represent this variation. The *precision variation curve* for the operations and operands in the loop can be identified either by theoretical analysis or by run-time analysis as described in Section 4.

Given the required precision for the iterations of the loop, we need to determine the mapping of the iterations to a set of configurations which are used to execute the operations in the loop. For each iteration the precision of the configuration which executes the iteration should be equal to or greater than the required precision for that iteration. The configurations are chosen from the set of library components or parameterized modules that are provided for the architecture.

Given the requirements for the precision of the computations and the available module configurations, we compute the set of configurations and the schedule of reconfigurations. We compute these by developing algorithmic techniques for precision management. First, we develop an abstract model of reconfigurable architectures, the Hybrid System Architecture Model(HySAM). This parameterized abstract model is general enough to capture a wide range of configurable systems. We define the precision management problem in loop computations using our model. A dynamic precision management algorithm is then developed to compute the optimal sequence of configurations for minimizing the total execution time including the reconfiguration time.

3 Related Work

There has been significant research in the area of mapping applications to configurable computing in the last decade [2, 5, 8, 15]. Customizing configurable hardware to suit the computations has been acknowledged as the most significant advantage of such architectures. Some researchers have adapted the hardware to perform computations with exactly the required precision for the computations [11, 13]. Such *static* approaches do not exploit the ability of configurable hardware to be adapted to the exact required precision as the computations progress. The maximum possible precision of variables which is determined in the *static* approach can still involve execution with superfluous precision and unnecessary overheads. Several efforts have also focused on developing parameterized libraries and components, precision being one of the parameters. Most FPGA device vendors provide such highly optimized parameterized libraries for their architectures. Efforts have also been made to generate such modules using high level descriptions [3, 6].

We are not aware of any formal framework to study and analyze the dynamic precision variation in applications. Algorithmic techniques to utilize configurable computing to dynamically vary the precision of computations have not been demonstrated previously.

4 Precision Requirement Analysis

The precision required for the computations in an application might not only vary with the specific operation but also change as the execution progresses. For iterative computations in which values are accumulated over the execution time of the application, the precision varies as the iterations progress. Loop computations are the most typical iterative computations which show such behavior. In addition to the varying precision, loops are the most compute intensive tasks in a program. In this paper we focus on the varying precision of operations in loop computations. This variation can be measured by analyzing the variation of the precision of the operands and the operations as the iterations progress. We represent this variation in terms of the loop iterations by using the *precision variation curve.*

4.1 Precision Variation Curve

The *precision variation curve* facilitates the representation of the notion of the variation in the precision of the operands and the operation as the execution of the loop progresses. A simple method to represent such a variation is to indicate the precision of the operand for each iteration so that the precision is defined for the whole iteration space. But as we shall show in the subsequent sections, the precision usually varies very slowly as the iterations progress. Thus the *precision variation curve* can be represented by specifying the points where the precision of the operands or the operation changes.

Definition: The *precision variation curve* for a given operation or operand in a loop computation can be represented by the sequence $<L_i, P_i>$, where $1 \leq i \leq u+1$ and $L_{u+1} = N+1$. For $1 \leq i \leq u$, P_i is the minimum precision required for the computing the iterations $L_i \ldots L_{i+1}-1$. Note that the hardware has to support at least a precision of P_i to execute the iterations $L_i \ldots L_{i+1}-1$ and produce the correct result.

Examples of *precision variation curves* are shown in Figure 3. We develop theoretical and run-time instrumentation methods for determining the *precision variation curve* in the next two sections.

4.2 Theoretical Analysis of Loops

We can theoretically determine the *precision variation curve* for the operations in a given computation. The precision of computed variables in a loop is determined by the precision of the variables before the iteration, the number of iterations and the operations

```
-----------------------------------------
 DO 10 I=1,N
   DO 20 J=1,N
     RSQ(J) = RSQ(J)+XDIFF(I,J)*YDIFF(I,J)
20   IF (MAXQ.LT.RSQ(J)) THEN
       MAXQ = RSQ(J)
   POVERR = POVERR / MAXQ
10 VIRTXY = VIRTXY + MAXQ * SCALE(I)
-----------------------------------------
```

Figure 2: Example code for simulations

performed on the variable. For each type of arithmetic operation, the maximum possible precision of the result can be expressed using the above values. For example, the precision of a variable X(initially 0) after N iterations of a loop which contains the statement $X = X + C$ is bounded by

$$Pr(X) \leq Pr(C) + \log(N+1)$$

where $Pr(X)$ denotes the bit size of the variable X. The analysis is not limited to simple expressions, but extends to complex arithmetic expressions in loops. For recursive expressions in loops where the value of the variable X in iteration i is given by X_i, if

$$X_i = c_1 * X_{j_1} + c_2 * X_{j_2} + \ldots + c_k * X_{j_k} = \Sigma_{l=1}^{l=k} c_l * X_{j_l}$$

then the upper bound on the precision of X_i is given by

$$Pr(X_i) \leq (i-1) * \log C + (i-1) * \log k + Pr(X_1)$$

where $C = max[c_1, c_2, \ldots, c_k]$, the maximum of the constant coefficients. Similarly, for the expression $X = X \times C$, the upper bound of precision for X with an initial value 1 and after N iterations is given by

$$Pr(X) \leq N * Pr(C)$$

The *precision variation curve* can be computed theoretically for all expressions in loops which are polynomials of variables and constants. Since most scientific applications consist of many such computations, theoretical analysis can be performed for all such computations. It is to be noted however, that such an analysis is not entirely feasible for floating point computations. But the analysis can be performed for integer and fixed point data and computations. This does not limit the applicability of the analysis or the algorithms we present later as many signal and image processing computations and several benchmark problems operate on integer and fixed point data. The remaining computations can be implemented with their default maximum precision.

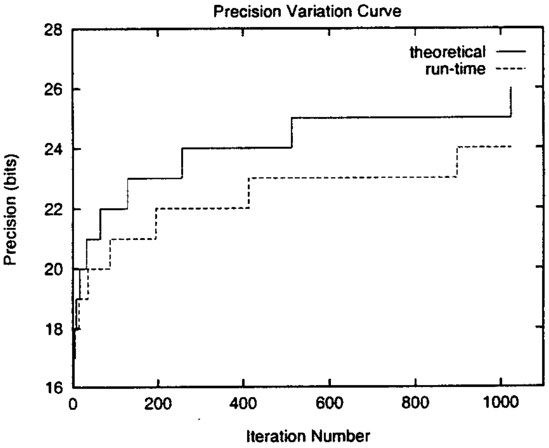


Figure 3: Precision Variation Curves for *RSQ* using theoretical and run-time analysis

4.3 Run-time Analysis

Theoretical analysis of expressions in loops computes the upper bounds on the precision of the variables and computations. This determines the minimum precision required to represent these variables. The estimates using theoretical analysis are conservative and can usually be much higher than the actual precision of the operands. For example, using the above analysis for the Fibonacci series $X_i = X_{i-1} + X_{i-2}$, we obtain $Pr(X_i) = i - 1$ and hence, $Pr(X_{15}) = 14$. But, $X_{15} = 610$ which needs only 10 bits. Even in the case when the bound is actually tight for expressions, the actual precision might be lower than theoretical estimate. This can occur when the data inputs are assumed to have maximum precision, but are actually randomly distributed over the complete input range. Using theoretical analysis can provide significant performance benefits by dynamic precision management. We discuss below how these benefits can be augmented by using profiling based analysis.

For example, consider the code segment shown in Figure 2. We performed simulations with uniformly distributed random values for the 8-bit non-negative data inputs $XDIFF$ and $YDIFF$. The precision of the *RSQ* variable was measured by tracing the earliest iteration in which a new higher significant bit was set. Since the maximum bits in the result of $XDIFF(I,J) * YDIFF(I,J)$ are 16, the iteration in which the kth most significant bit of the result is set is given by 2^{k-16}. The *precision variation curves* obtained using the theoretical and run-time analysis are plotted in Figure 3. The actual precision required for the computations is significantly lower than the theoretical estimate as evident from the graph.

This run-time measurements illustrate a very important advantage in exploiting variable precision computations. The actual $XDIFF$ and $YDIFF$ values have significantly lower precision than the maximum possible precision of 8 bits. The assumption of maximum precision for all the input $XDIFF$ and $YDIFF$ values has a rolling effect on precision of other operands and operations. The repeated accumulation of the product of these numbers results in a precision difference in the final values which is much larger than the precision difference for one value. It is clearly revealed in simulations where the actual required precision is much lower than the theoretical precision.

For computations which do not have a tight bound on the precision and for computations with complex control flow, computing the required precision by using run-time statistics is a viable alternative. The application can be instrumented to measure the precision of the different variables and the knowledge can be utilized by the mapping tool or the compiler to identify the required precision at various program points. Though we do not address the run-time mapping issues in this paper, it is also possible to determine the precision of the operands and the operations by examining the values at run-time and modifying the precision of the operations on the fly. In this paper we focus on run-time precision management based on the knowledge of the required precision at compile(mapping) time. The required precision can either be analyzed automatically or can be user specified.

5 Hybrid System Architecture Model (HySAM)

To realize a formal framework for algorithm development, we developed the Hybrid System Architecture Model(HySAM) of reconfigurable architectures. The *Hybrid System Architecture* is a general architecture consisting of a conventional microprocessor with additional Configurable Logic Unit(CLU). Figure 4 shows the architecture of the HySAM model. The architecture consists of a conventional microprocessor, standard memory, configurable logic, configuration memory and data buffers communicating through an interconnection network.

Key parameters of the Hybrid System Architecture Model(HySAM) are outlined below.

F : Set of functions $F_1 \ldots F_n$ which can be performed on configurable logic.

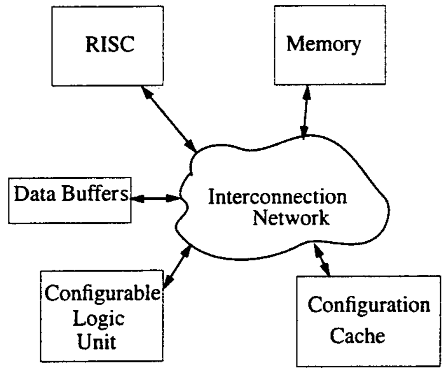


Figure 4: Hybrid System Architecture Model(HySAM)

C : Set of possible configurations $C_1 \ldots C_m$ of the Configurable Logic Unit.

$Pr(C_j)$: Precision of the configuration C_j.

t_{ij} : Execution time of function F_i in configuration C_j.

R_{ij} : Reconfiguration cost in changing configuration from C_i to C_j.

The parameterized HySAM models a wide range of systems from board level architectures to systems on a chip. Such systems include SPLASH [2], DEC PeRLE [15], Oxford HARP [5], Berkeley Garp [4], NSC NAPA1000 [9], Sanders CSRC [10] among others. The values for each of the parameters establish the architecture and also dictate the class of applications which can be effectively mapped onto the architecture. For example, a system on a chip architecture would have potentially faster reconfiguration times(lower k and K) than a board level architecture.

The set of functions(F) is the set of modules or library components which are available or implemented for the given architecture. Configurations(C) are developed by mapping one or more of such functions onto the available hardware architecture. A single function can have multiple configurations which can potentially execute the function. Each of the configurations might have different algorithm, area, precision, time and power characteristics. For example, a function such as division can be implemented using different algorithms such as iterative multiplication or iterative subtraction in different configurations. The execution time of a function F_i in a configuration C_j is given by t_{ij}. The cost of reconfiguring the hardware from a configuration C_i to a configuration C_j is given by R_{ij}. The reconfiguration cost includes the cost of memory access for the configurations, the configuration data transfer cost and the cost of activating the configuration on the hardware.

5.1 Configurations for Variable Precision

Efficient modules are being developed by hand design, by automatic mapping and by generators [3, 6]. Modules for executing computations with a specified precision have also been explored. Some of the modules are parameterized which facilitate the construction of a configuration which can execute a computation of any given precision within a range of values. They are usually either statically developed designs such as the Xilinx LogicBlox or dynamically constructed using generators [3, 6]. The modules are usually optimized to exploit the nature of the computation for any given precision. Modules designed for specific architectures also exploit the hardware features which are available to enhance performance. For example, addition and multiplication modules exploit the carry chains available at nibble or byte boundaries in many FPGA architectures.

In this paper we assume that the set of modules which can execute the required arithmetic operations are available. Each function(such as multiplication) can have several configurations, each of which executes the operation with different precision. It is not necessary that a given operation have configurations which execute the operation with all the possible precision values. Note that each configuration is limited to the execution of one function in this paper though the HySAM model is actually more powerful. Hence, we represent t_{ij} as t_{C_j} is the rest of the paper.

6 Dynamic Precision Management

Given the *precision variation curve* for the loop, we need to determine the mapping of the iterations to a set of configurations which are used to execute the operations in the loop. For each iteration, the precision of the corresponding configuration which executes the iteration should be equal to or greater than the required precision for that iteration. But, reconfiguring the hardware whenever the required precision changes can result in significant reconfiguration overheads. For architectures in which the reconfiguration times are much higher than the execution times, the reconfiguration overhead might be prohibitive. Thus, it is necessary to identify the optimal set of configurations which result in minimization of the overall execution cost, including the reconfiguration cost. Also, the set of configurations which are available for executing an operation might not encompass all the possible precision values that are required. Some of the operations will have to be executed with more precision than is necessary in the absence of configurations with the exact precision.

We present the Precision Management Problem and the Dynamic Precision Management Algorithm based on the following assumptions:

- Higher precision computations require more resources such as power, logic area and computation time(t_{C_j}).
- The required precision for the computations varies monotonically. This is true for most computations which accumulate values as the loop iterations progress. The algorithms we describe can be applied to monotonic subsequences with optimal schedules for each subsequence individually.
- The algorithm determines the optimal schedule for a *given* precision variation curve. When the actual variation is different from the precision variation curve, the schedule might not be optimal.

Precision Management Problem(PMP)

Input: An operation in a loop with N iterations of the loop body and the *precision variation curve* for the operation. The *precision variation curve* is given as a sequence of pairs $<L_i, P_i>$, where $1 \leq i \leq u+1$ and $L_{u+1} = N+1$. For $1 \leq i \leq u$, P_i is the minimum precision required for computing the operation in iterations $L_i \ldots L_{i+1} - 1$.

Output: An optimal schedule of configurations $S = <Q_j, C_j>$, where $1 \leq j \leq v+1$ and $Q_{v+1} = N+1$. For $1 \leq j \leq v$, C_j is the configuration used for iterations $Q_j \ldots Q_{j+1} - 1$.

A schedule S is said to be valid if it satisfies the precision requirement for all the iterations of the loop, i.e., $\forall K \; s.t. \; 1 \leq K \leq N$, if

$$\begin{aligned} Pr_I &= P_i, \; for \; some \; i \; s.t. \; L_i \leq K < L_{i+1} \\ Pr_O &= Pr(C_j) \; for \; some \; j \; s.t. \; Q_j \leq K < Q_{j+1} \end{aligned}$$

then $Pr_I \leq Pr_O$ (see Figure 5).

An optimal schedule has the minimum total execution cost E which includes the reconfiguration cost among

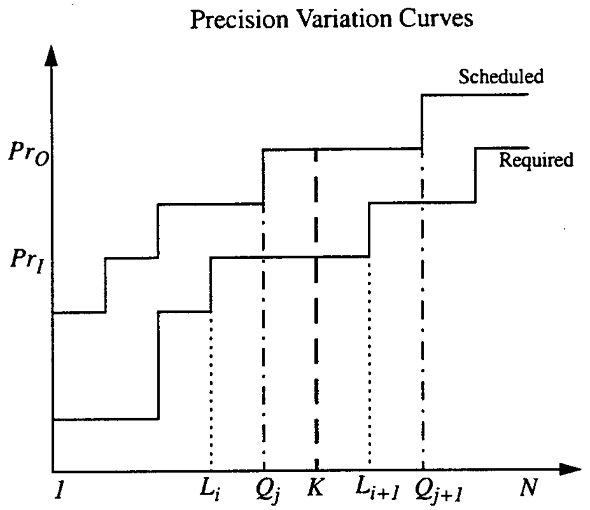


Figure 5: Constraint on required and scheduled Precision Variation Curves

all valid schedules. The cost of a schedule is given by

$$E = \sum_{j=1}^{v} [(Q_{j+1} - Q_j) \times t_{C_j} + R_{j-1j}]$$

where t_{C_j} is time for executing one iteration of the loop in configuration C_j and R_{j-1j} is the reconfiguration cost between configurations C_{j-1} and C_j. ⊙

To minimize the total execution cost, both the execution cost and the reconfiguration cost have to be examined. The set of configurations and the schedule of reconfigurations need to be determined. We first show that the points of reconfiguration are the subset of the points where the required precision changes, i.e., $Q \subseteq L$, where $Q = \{Q_1, \ldots, Q_v\}$ and $L = \{L_1, \ldots, L_u\}$.

Lemma 1. *Given the definitions in the* **PMP** *problem, the schedule S of configurations satisfies the property $Q \subseteq L$.*

Proof: Assume that $Q \not\subseteq L$ in the optimal schedule S. Then there exists at least one point of reconfiguration which is not a point of change of required precision.

$$\exists i : Q_i \notin L$$

Without loss of generality,

$$\exists j : Q_{i-1} \leq L_{j-1} < Q_i < L_j \leq Q_{i+1}$$

Consider the schedule S' where the configurations are the same as S but the reconfiguration points are different:

$$S = [Q_1 \ldots Q_{i-1} Q_i Q_{i+1} Q_{i+2} \ldots Q_n]$$

$$S' = [Q_1 \ldots Q_{i-1} L_j Q_{i+1} Q_{i+2} \ldots Q_n]$$

t_{C_i} is the cost of executing one iteration in configuration C_i. Since we assume precision variation to be monotonic, $Pr(C_{i+1}) > Pr(C_i)$ and $t_{C_{i+1}} > t_{C_i}$. The difference in execution cost of the two schedules is

$$\begin{aligned} S' - S &= (t_{C_i}(L_j - Q_{i-1}) + t_{C_{i+1}}(Q_{i+1} - L_j)) \\ &\quad -(t_{C_i}(Q_i - Q_{i-1}) + t_{C_{i+1}}(Q_{i+1} - Q_i)) \\ &= t_{C_i}(L_j - Q_{i-1} - Q_i + Q_{i-1}) \\ &\quad +t_{C_{i+1}}(Q_{i+1} - L_j - Q_{i+1} + Q_i) \\ &= (t_{C_i} - t_{C_{i+1}})(L_j - Q_i) \\ &< 0 \end{aligned}$$

Since $L_j > Q_i$ and $t_{C_i} < t_{C_{i+1}}$, $S' - S < 0$. The new schedule has lower cost and hence a schedule with reconfiguration points which is the subset of the precision change points has lower execution cost. Since S is the optimal schedule our assumption must be incorrect. Hence, $Q \subseteq L$. ⊙

6.1 Precision Management Algorithms

To determine the choice of configuration at each L_i, we can use a greedy approach where the best configuration with the required precision is chosen at each L_i. The best configuration $C_j (C_j \in C_1, \ldots, C_m)$ is given by the configuration which has the lowest execution cost t_{C_j}. But the greedy algorithm will not provide the optimal solution due to two reasons:

- The greedy approach does not consider the reconfiguration costs which are incurred at future reconfiguration points. A configuration with higher execution cost might have a lower reconfiguration cost at the next step, making it a better choice for executing the given iterations.

- With significant reconfiguration costs, it is possible that we use a higher precision configuration than required(even if exact precision configuration is available in C), to avoid a reconfiguration step in future. The greedy approach does not consider this case and thus can result in non-optimal schedule.

In the following, we present an algorithm based on dynamic programming which computes an optimal schedule having the minimum execution cost including the reconfiguration cost.

Dynamic Precision Management Algorithm (DPMA)

Let E_{ij} be the execution cost for executing up to L_i iterations with C_j being the last configuration. The initial values of E are assigned as $E_{0j} = 0, 1 \leq j \leq m$. For each of the possible configurations C_j which can execute iterations from L_i we have to compute the optimal sequence of configurations ending in C_j. For $1 \leq j \leq m$. we compute E_{ij} by using the recursive equation:

$$\begin{aligned} E_{i+1j} &= (L_{i+1} - L_i) \times t_{C_j} + min_k(E_{ik} + R_{kj}) \\ & \qquad\qquad\qquad\qquad\qquad 1 \leq k \leq m \\ & \quad if\ Pr(C_j) \geq P_i \\ &= \propto\ otherwise \end{aligned}$$

For each configuration, we have examined all the possible paths in executing the iterations $L_i \ldots L_{i+1} - 1$ once we have executed iterations $1 \ldots L_i - 1$. Note that we examine all configurations such that $Pr(C_j) \geq P_i$ which assures that we consider the case of using a higher precision than required$(Pr(C_j) = P_i)$. If each of the values E_{ik} is optimal then the value E_{i+1j} is optimal. Hence we can compute the optimal schedule of configurations S by computing the E_{ij} values. The minimum cost for execution of the loop is given by $min_j[E_{uj}]$.

We can use dynamic programming to compute the E_{ij} values. Computing one E_{ij} value takes $O(m)$ time since there are m configurations. The total number of values to be computed is $O(um)$, therefore the total time complexity of the algorithm is $O(um^2)$. $\odot$

7 An Illustrative Example

We illustrate our approach by mapping the multiplication operation from the example code segment presented in Figure 2.

```
-----------------------------------------
 DO 10 I=1,N
   ...
10 VIRTXY = VIRTXY + MAXQ * SCALE(I)
-----------------------------------------
```

The input data $SCALE(I)$ is an 8-bit integer. The precision of $MAXQ$ has been analyzed in Section 4.3. We present the same result in the form of a table in Table 1.

We have abstracted the Xilinx XC6200 series device by using our model. The parameters specified are for the HySAM model and have been evaluated from XC6200 documentation [16, 7]. The footprint of each precision is given by the equation $4 \times row \times col$, where row and col are the precisions of the two inputs. For the configurations relevant to mapping the given operation, row is 8. Reconfiguration times are based on a 32-bit data bus running at 50MHz. It is possible to design modular configurations which can be reconfigured in lesser time using partial reconfiguration. For this mapping, we assumed that complete reconfiguration is needed for each configuration. The parameters for various multiplier configurations with different precisions are listed in Table 2.

P_i	L_i	L'_i	P_i	L_i	L'_i
Pr	Theoretical	Simulated	Pr	Theoretical	Simulated
16	1	1	22	64	195
17	2	2	23	128	412
18	4	5	24	256	897
19	8	14	25	512	-
20	16	35	26	1024	-
21	32	87			

Table 1: Theoretical and simulated iteration numbers for $N = 1024$

Configuration C_i	Precision $Pr(C_i)$	Time t_{C_i} (ns)	Reconfig. R_{0i} (ns)
C_1	8×8	140	5120
C_2	8×16	250	10240
C_3	8×20	300	12800
C_4	8×24	400	15360
C_5	8×28	520	17920
C_6	8×32*	*640	20480

Table 2: HySAM model parameters for XC6200 multiplier configurations(* values are estimates based on XC6264 device)

We measured the total execution time for the loop computations using five different approaches. The first two approaches do not exploit the *dynamic precision* by varying the precision of the operation at run-time. The different approaches and the schedule of configurations(<Q_j,C_j>) in each approach are described below.

- **Raw**: The first approach uses a static configuration of $8bit \times 32bit$ precision for all the iterations of the loop.
  Schedule: <1,C_6>

- **Static**: We utilize the theoretical analysis where we determine that the highest precision required for 1024 iterations is only $8bit \times 28bit$. But the configuration is still static and is used for all the iterations.
  Schedule: <1,C_5>

- **Greedy**: We used the greedy algorithm (see Section 6.1) to compute the schedule of configurations to be utilized for the computations. The precision of the operation is varied dynamically but the greedy choice is based on the lowest execution time for each configuration.
  Schedule: <1,C_2>,<2,C_3>,<32,C_4>,<512,C_5>
- **DPMA**: Our dynamic precision management algorithm was utilized to compute the optimal schedule using the *precision variation curve*. This approach uses higher execution cost configurations for some of the computations but reduces the overall execution cost by performing lesser number of reconfigurations.
  Schedule: <1,C_4>,<512,C_5>
- **DPMA-run**: In this approach we performed run-time analysis of the loop and utilized the *precision variation curve* from the run-time analysis as the input to the algorithm. This approach can be implemented easily by adding a run-time check of the precision, which needs very small amount of additional logic and no extra clock-cycles if the precision remains within the run-time statistics.
  Schedule: <1,C_4>

Algorithm	Execution Time (*ns*)	Reconfiguration Time (*ns*)	Total (*ns*)
Raw	655360	20480	675840
Static	532480	17920	550400
Greedy	468010	56320	524330
DPMA	471160	33280	504440
DPMA-run	409600	15360	424960

Table 3: Execution times using different approaches

The execution times including the reconfiguration times are summarized in Table 3. The approaches using *dynamic precision* achieve significantly lower execution times compared to the Raw and Static approaches. We noticed that our DPMA algorithm executed all the iterations of the loop in the minimum time for the theoretical and run-time *precision variation curves*. The DPMA-run achieves significant speed-up by exploiting the fact that 28-bit precision is never required.

8 Conclusions

This paper has developed a framework for dynamic precision management for loop computations. We have shown how the variable precision in computations can be captured by using the *precision variation curve*. The paper described our approach to computing the *precision variation curve* using theoretical and run-time analysis. The information obtained from these analyses is used to develop optimal schedules for dynamic precision management. The DPMA algorithm that we have developed can compute the required optimal schedule for a given operation in a loop using the *precision variation curve* and the set of variable precision configurations. Our Hybrid System Model(HySAM) of reconfigurable architectures facilitates the development of these algorithms using a high level abstract model. The paper illustrated the performance benefits achievable for an example loop computation using our approach. We expect that the proposed approach can lead to significant improvement in performance and automatic mapping of variable precision computations on reconfigurable architectures.

The dynamic precision management framework gives rise to a wealth of issues which can potentially provide enormous benefits to mapping computations onto configurable hardware. Bit-serial and digit-serial computations are one class of computations which can exploit dynamic precision without large overheads. The control component of the design needs to execute the configurations for a variable number of steps based on the required precision. Run-time precision management where the control modifies the precision of the computations are being explored. Configurable logic can be utilized to execute multiple iterations of loops in parallel in the absence of dependencies. Reduction of the logic resources due to dynamic precision management can be exploited to execute more number of iterations in parallel. Multi-context devices and configuration caches can be utilized to reduce the reconfiguration overheads by storing variable precision configurations.

The work reported here is part of the USC MAARC project(http://maarc.usc.edu). This project is developing algorithmic techniques for realizing scalable and portable applications using configurable computing devices and architectures. We are developing computational models and algorithmic techniques based on these models to exploit dynamic reconfiguration. Some recent related results can be found in [1, 12].

9 Acknowledgments

We would like to thank Steve Casselman, David Franklin, and John Schewel of the Virtual Computer Corporation for their interest and support.

References

[1] K. Bondalapati and V.K. Prasanna. Mapping Loops onto Reconfigurable Architectures. In *8th International Workshop on Field-Programmable Logic and Applications*, September 1998.

[2] D. A. Buell. J. M. Arnold, and W. J. Kleinfelder. *Splash 2: FPGAs in a Custom Computing Machine.* IEEE Computer Society Press, 1996.

[3] M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek. Object Oriented Circuit-Generators in Java. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[4] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, April 1997.

[5] A. Lawrence, A. Kay, W. Luk, T. Nomura, and I. Page. Using reconfigurable hardware to speed up product development and performance. In *5th International Workshop on Field-Programmable Logic and Applications*, 1995.

[6] O. Mencer, M. Morf, and M.J. Flynn. PAM-Blox: High Performance FPGA Design for Adaptive Computing. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[7] Xilinx Application Notes. A Fast Constant Coefficient Multiplier for the XC6200.

[8] R.J. Petersen and B. Hutchings. An Assessment of the Suitability of FPGA-Based Systems for use in Digital Signal Processing. In *5th International Workshop on Field-Programmable Logic and Applications*, 1995.

[9] C.R. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J.M. Arnold, and M. Gokhale. The NAPA Adaptive Processing Architecture. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

[10] S.M. Scalera and J.R. Vázquez. The design and implementation of a context switching FPGA. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.

[11] N. Shirazi, P.M. Athanas, and A.L. Abbott. Implementation of a 2-D Fast Fourier Transform on an FPGA-Based Custom Computing Machine. In *International Workshop on Field-Programmable Logic and Applications*, September 1995.

[12] R.P.S. Sidhu, A. Mei, and V.K. Prasanna. String Matching on Multicontext FPGAs using Self-Reconfiguration. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Feb 1999.

[13] A.F. Tenca and M.D. Ercegovac. A Variable Long-Precision Arithmetic Unit Design for Reconfigurable Coprocessor Architectures. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.

[14] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A Time-Multiplexed FPGA. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 22–28, April 1997.

[15] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, March 1996.

[16] Xilinx. XC6200 Field Programmable Gate Arrays, 1996.

International Conference on Parallel and Distributed Processing Techniques and Applications, June 1999.

Hardware Object Selection for Mapping Loops onto Reconfigurable Architectures*

Kiran Bondalapati and Viktor K. Prasanna

Department of Electrical Engineering Systems, EEB-200C
University of Southern California
Los Angeles, CA 90089-2562, USA
{kiran, prasanna}@usc.edu
http://maarc.usc.edu

Abstract

Reconfigurable circuits and systems have evolved from application specific accelerators to a general purpose computing paradigm. Reconfiguring the logic is still an expensive operation and precludes frequent configuration changes. To reduce the overheads involved in reconfiguration, devices with configuration caches and multiple contexts are being designed. Reconfigurable computing solutions are typically designed by composing lower level modules or library components. Each operation in an application can be implemented by using any one among several of these modules or hardware objects. This gives rise to the problem of choosing an optimal set of modules for utilizing the cache or the multiple contexts. This paper develops a formal methodology for selection of these modules to minimize the total execution time. The total execution time includes the reconfiguration time and the computation time in various configurations. We focus on loop computations since they are the most compute intensive parts of applications. We utilize a parameterized abstract model of reconfigurable architectures which is general enough to capture a wide range of configurable systems. Our abstract model is used to define the problem of mapping loop statements onto reconfigurable architectures. We show a polynomial time algorithm to compute the optimal sequence of configurations(modules) for one important variant of the problem.

1 Introduction

Configurable systems are evolving from systems designed to accelerate a specific application to systems which can achieve high performance for general purpose computing. Various reconfigurable architectures are being explored by several research groups to develop a general purpose configurable system. Reconfigurable architectures vary from systems which have FPGAs and glue logic attached to a host computer to systems which include configurable logic on the same die as a microprocessor.

The performance achievable on reconfigurable architectures is limited by the costs involved in reconfiguring the logic. Currently, this overhead is very high and discourages the reconfiguration of the logic during the execution of a single application. To address this problem architectures which support configuration caches and multiple contexts on the devices have been proposed [7, 5, 6, 8, 9]. In devices with configuration caches, the cost of

*This work was supported by the DARPA Adaptive Computing Systems Program under contract DABT63-96-C-0049 monitored by Fort Hauchuca.

loading a configuration from the cache is much lower than loading a configuration from off-chip memory. In multi-context devices, the overhead for switching between contexts is very low. In some devices this can be done in a few clock cycles.

Development of reconfigurable computing solutions is typically based on hierarchical designs. Modules or library components are utilized to compose and construct larger designs. Utilizing such hardware *objects* makes the design development easier and promotes reuse of optimized modules. For executing a given operation, various modules can be utilized. These modules can differ in their performance characteristics such as area, execution time, power consumption, reconfiguration time etc.

In this paper we address the problem of automatic selection of optimal modules or hardware objects to be utilized in cached-configuration or multi-context devices. We focus our efforts on loop statements since they provide the maximum opportunity for performance improvement. Loop statements have regular and repetitive computations which are well-suited to reconfigurable architectures. We had previously developed an abstract model of reconfigurable architectures, the Hybrid System Architecture Model [1, 2, 3, 4]. This parameterized abstract model is general enough to capture a wide range of configurable systems. We define the problem of optimal module selection using the HySAM model. We consider one variation of the problem when the multiple configurations in the cache or the contexts can be pre-loaded but cannot be modified during execution. We present an efficient algorithm to compute the solution for this variant.

Section 2 described multi-context devices and their operation. Section 3 describes our Hybrid System Architecture Model(HySAM). The optimal module selection problem for a loop is defined and the optimal solution is presented in Section 4. We present conclusions and future research in Section 5.

2 Reconfigurable Architectures

Typical reconfigurable devices have high reconfiguration times in the order of milli-seconds. Reconfiguration in such devices involves downloading the bit stream for the complete device configuration. Some reconfigurable devices permit partial and dynamic reconfiguration [10]. These devices permit reconfiguration of a part of the device while the configuration of the remaining device is unchanged. Many of the reconfigurable devices are based on SRAM controlled configuration of the logic and the interconnection network. Configuration of a device involves configuring the SRAM cells in the device.

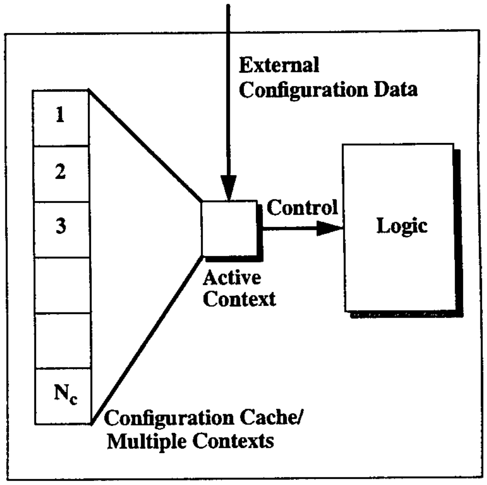


Figure 1: Device logic control model

In devices with configuration caches or multiple contexts, the SRAM cells controlling the functionality of the logic cell can be configured using any one of the multiple configurations. Figure 1 illustrates the abstract view of such devices. Configuration of the multi-context device is performed by loading the configurations of the various contexts onto the device. Loading all the contexts onto the device takes reconfiguration time similar to the reconfiguration times of typical reconfigurable devices. But, once the configurations are loaded onto the chip, switching between the configurations is very inexpensive. Switching times for such devices are expected to be in the range of 5-

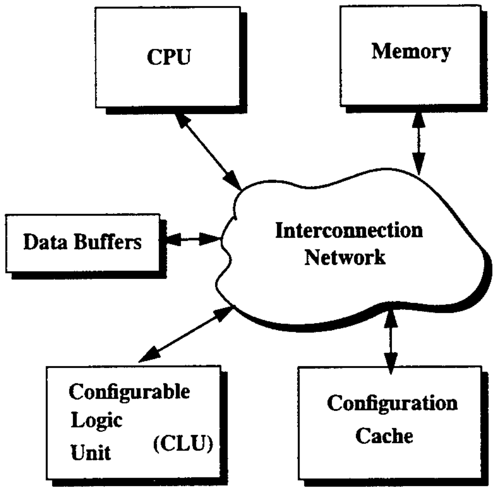


Figure 2: Hybrid System Architecture Model

100 ns. This is several orders of magnitude faster than configuring the active context by using external data.

3 Hybrid System Architecture Model(HySAM)

A high level model of reconfigurable hardware is needed to abstract the low level details. Existing models supplied by the CAD tools have either multiple abstraction layers or are very device specific. We present a parameterized model of a configurable computing system, which consists of configurable logic attached to a traditional microprocessor. Our model cleanly partitions the *capabilities* of the hardware from the *implementations* and presents a very clean interface to the user. We describe the model below briefly since it is not the main focus of the paper. Details of the HySAM model and some prior algorithms for mapping based on the model are available in [1, 2, 3, 4].

The *Hybrid System Architecture Model* is a general model consisting of a traditional microprocessor with additional Configurable Logic Unit(CLU). Figure 2 shows the architecture of the HySAM model and Figure 3 shows an example of an actual architecture. The architecture consists of a traditional microprocessor, standard memory, configurable logic, configuration memory and data buffers communicating through an interconnection network.

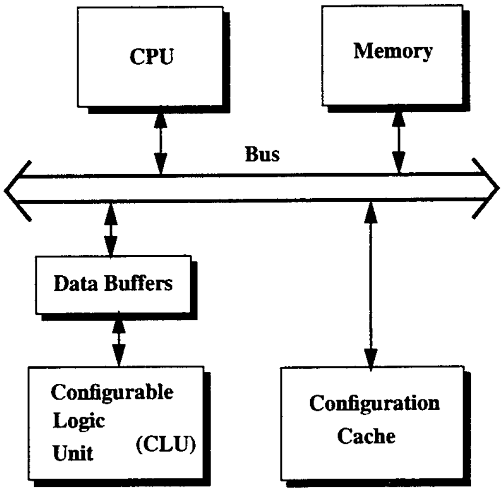


Figure 3: Example hybrid system architecture

We outline some of the parameters of the Hybrid System Architecture Model(HySAM) below.

- F : Set of functions $F_1 \ldots F_n$ which can be performed on configurable logic. (*capabilities*)
- C : Set of possible configurations $C_1 \ldots C_m$ of the Configurable Logic Unit. (*implementations*)
- t_{ij} : Cost of executing function F_i in configuration C_j.
- R_{ij} : Reconfiguration cost in changing configuration from C_i to C_j.
- N_c : The number of configuration which can be stored in the cache or the multiple contexts in the CLU.
- k_c : The cost of switching to one of the context from among those resident on the CLU.

The hardware objects or modules are represented by the Functions and the Configurations. The functions F and configurations C have a *many-to-many* relationship. Each configuration C_i, can potentially contain more than one function F_j. For example, a configuration can contain both addition and logical OR, given enough logic resources. The execution cost of a function F_i in configuration C_j is specified as one of t_{ij}. In the HySAM model, only function can be active in a configuration at any given time. Each function F_i can be executed by using any one configuration from a subset of the configurations.

The different configurations might be generated by different tools, libraries or algorithms. These configurations might have different area, time, reconfiguration, precision, power, etc. characteristics. For example, it is possible to design multipliers of various area/time characteristics by choosing various degrees of pipelining and carry look ahead techniques. The multiplier can have different values for the area, pipeline stages, cycle time and number of cycles for finishing the computation. Similarly, floating point operation configurations can be designed with various degrees of precision.

The execution model that we consider contains N_c configurations resident on the chip in the cache or the multiple contexts. There is one active context which can be based on one of the N_c configurations or can be configured from external memory. Switching to a configuration C_j from a configuration C_i takes k_c time if C_j is one of the N_c configurations or R_{ij} if the configuration has to be fetched from outside the chip. We assume that only the active context can be configured externally during the execution of the application. Before the application has started execution, the multiple configurations can be loaded onto the device.

The reconfiguration costs R define the costs involved in changing the configuration of the CLU between two configurations. This cost can be statically evaluated based on the configuration information for different configurations. The cost can also be computed dynamically when the configurations are constructed dynamically. The cost defines the amount of logic reconfigured and the time spent in reconfiguring the logic between any two configurations belonging to C. This cost incorporates the factors when partial and dynamic reconfiguration is exploited.

4 Mapping Configurations onto Contexts

Computations which operate on a large set of data using the same set of operations are most likely to benefit from configurable computing. Hence, loop structures will be the most likely candidates for performance improvement using configurable logic. Configurations which execute each task can be generated for the operations in a loop. Since each operation is executed on a dedicated hardware configuration, the execution time for the task is expected to lower than that in software. We solve the restricted version of the problem which imposes a linear order on the list of tasks to be executed in a loop. Any given list of tasks with directed acyclic dependencies can be converted to a linear list by using topological sorting.

Each of the operations in the loop statement might be a simple operation such as an addition of two integers or can be a more complex operation such as a square root of a floating point number. The problems and solutions that we present are independent of the complexity of the operation. As we described in Section 3, a single operation can be implemented using various optimizations to provide several implementations. These different configurations can have different performance characteristics.

The mapping problem is to select the configuration to be utilized for each function and the configurations which are stored in the contexts. To select the configuration for executing a given function we can employ the greedy strategy. The greedy algorithm chooses the best possible configuration for executing a given function, i.e., the configuration with the lowest execution cost. But this configuration might have a large reconfiguration cost which increases the total execution time and gives a sub-optimal solution. For selecting the configurations to be pre-loaded the greedy strategy is still sub-optimal. Pre-loading the configuration with the highest reconfiguration cost gives a sub-optimal solution. Selecting a different configuration to be pre-loaded and using a configuration with lower execution cost can give a better solution. We assume the following regarding the model as explained in Section 3:

1. The N_c configurations are loaded on to the device at the start of the computation.
2. The active context can be configured from any of the N_c configurations with a cost k_c.
3. The pre-loaded configurations can not be modified during the execution of the complete application. Only the active context can be reconfigured externally.

Hardware Object Selection Problem

Input : A sequence of tasks of a loop, T_1 through T_p to be executed in linear order(T_1 $T_2 \dots T_p$), where $T_i \in F$, for N number of iterations, and the number of configurations which can be cached or stored in contexts N_c.

Output : An optimal schedule of configurations S (=C_1 C_2 $\dots$ C_q), and the set X of configurations to be stored in the N_c contexts. An optimal schedule has the minimum total execution cost E, which includes the reconfiguration cost. The cost of a schedule is given by

$$E = \sum_{j=1}^{q} t_{C_j} + R'_{j-1j}]$$

where t_{C_j} is time for executing one iteration of the loop in configuration C_j and R'_{j-1j} is the reconfiguration cost between configurations C_{j-1} and C_j. R'_{ij} is defined as

$$\begin{aligned} R'_{ij} &= k_c \ if \ C_j \in X \\ &= R_{ij} \ otherwise \end{aligned}$$

Solution: We compute the optimal schedule S and the set of contexts X by using a dynamic programming approach. We first discuss how the optimal solution can be computed for a fully unrolled loop. All the iterations of the loop are unrolled to give a linear task sequence. We define the following variables:

- E_{ij}, $1 \leq j \leq m$: the cost of executing tasks T_1 to T_i with T_i being executed using configuration C_j and the configuration C_j is added to the contexts in X if not already in X.
- E_{ij}, $m+1 \leq j \leq 2*m$: the cost of executing tasks T_1 to T_i with T_i being executed using configuration C_j and the configuration C_j is **not** added to the contexts in X if not already in X.
- X_{ij}, $1 \leq j \leq 2*m$: the set of contexts which are added to X for executing tasks T_1 to T_i with T_i being executed using configuration C_j.
- $|X_{ij}|$: the number of contexts in set X_{ij}.

The E_{ij} and the X_{ij} values are computed using dynamic programming. The recursive equations for computing them are given below:

$$mink = 1 \leq k \leq 2*m : min[E_{ik} + \delta_{kj}]$$

δ_{kj} denotes the reconfiguration cost and can be evaluated based on the various possible scenarios:

- Configuration C_j is already in cache. The reconfiguration cost is the cost of performing a context switch, k_c.
- Configuration C_j has not been cached. The reconfiguration cost is based on the set $|X_{ik}|$. If there is space in this set of configurations to be pre-loaded, then the configuration C_j is added to the set and reconfiguration cost is k_c. If the cache is already full then the full reconfiguration cost R_{ij} is incurred.

The value of δ_{kj} is computed at each step as

```
$if \ (C_j \in X_{ik})$
    $\delta_{kj} = k_c$
$else \ if \ (|X_{kj}| < N_c \ and \ 1 \leq j \leq m)$
    $\delta_{kj} = k_c$
$else$
    $\delta_{kj} = R_{ij}$
```

Given the value of $mink$, the E_{i+1j} and the X_{i+1j} values are computed as follows:

$$E_{i+1j} = t_{i+1j} + E_{i\ mink} + \delta_{mink\ j}$$

$$\begin{aligned} X_{i+1j} &= X_{i\ mink} \cup C_j \\ &\qquad if\ |X_{imink}| < N_c\ \ and\ \ 1 \leq j \leq m) \\ &= X_{i\ mink} \quad otherwise \end{aligned}$$

The minimum execution cost E and the corresponding set of contexts X for executing tasks T_1 to T_z for any z are given by:

$$jmin = 1 \leq j \leq 2 * m : min[E_{zj}]$$

$$E = E_{z\ jmin}$$

$$X = X_{z\ jmin}$$

The required optimal schedule and the set of contexts can be computed by fully unrolling the loop and computing E and X for $z = p*N$ where N is the number of the iterations and p is the number of tasks in the loop. ⊙

5 Conclusions

Mapping of applications in an architecture independent fashion can provide a framework for automatic compilation of applications. Loop structures with regular repetitive computations can be speeded-up by using configurable hardware. In this paper, we have developed techniques to map loops from application programs onto configurable hardware. The low reconfiguration costs of multi-context devices are exploited to reduce the reconfiguration overheads in mapping. We described an efficient algorithm to select modules to be mapped onto the available contexts.

The problem that we solve assumes that the pre-loaded configurations can not be modified during the application execution. A more general version of the problem to be addressed is optimizing the execution time when the configurations can be replaced and the replacement can overlap with execution in a configuration. The work reported here is part of the USC MAARC project. This project is developing algorithmic techniques for realizing scalable and portable applications using configurable computing devices and architectures. Some related results can be found at http://maarc.usc.edu.

References

[1] K. Bondalapati and V.K. Prasanna. Mapping Loops onto Reconfigurable Architectures. In *8th International Workshop on Field-Programmable Logic and Applications*, September 1998.

[2] K. Bondalapati and V.K. Prasanna. DRIVE: An Interpretive Simulation and Visualization Framework for Dynamically Reconfigurable Architectures. In *Under Review, http://maarc.usc.edu/*, 1999.

[3] K. Bondalapati and V.K. Prasanna. Dynamic Precision Management for Loop Computations on Reconfigurable Architectures. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1999.

[4] K. Bondalapati, V.K. Prasanna, and P.A. Beerel. The Hybrid System Architecture Model for Reconfigurable Architecture Analysis. In *Under Preparation, http://maarc.usc.edu*, 1999.

[5] A. DeHon. DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1994.

[6] J. Hauser and J. Wawrzynek. Garp: A MIPS Processor with a Reconfigurable Coprocessor. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 12–21, April 1997.

[7] CMU Cached Virtual Hardware Homepage. http://www.ece.cmu.edu/research/piperench/.

[8] S.M. Scalera and J.R. Vázquez. The design and implementation of a context switching FPGA. In *IEEE Symposium*

on Field-Programmable Custom Computing Machines, April 1998.

[9] S. Trimberger, D. Carberry, A. Johnson, and J. Wong. A Time-Multiplexed FPGA. In *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 22–28, April 1997.

[10] Xilinx. XC6200 Field Programmable Gate Arrays, 1996.

International Workshop on Field Programmable Logic and Applications, FPL '99

DRIVE: An Interpretive Simulation and Visualization Environment for Dynamically Reconfigurable Systems*

Kiran Bondalapati and Viktor K. Prasanna

Department of Electrical Engineering Systems
University of Southern California
Los Angeles, CA 90089-2562, USA
{kiran, prasanna}@ceng.usc.edu
http://maarc.usc.edu/

Abstract. Current simulation tools for reconfigurable systems are based on low level simulation of application designs developed in a High-level Description Language(HDL) on HDL models of architectures. This necessitates expertise on behalf of the user to generate the low level design before performance analysis can be accomplished. Most of the current simulation tools also are based on static designs and do not support analysis of dynamic reconfiguration.

We propose a novel interpretive simulation and visualization environment which alleviates these problems. The **D**ynamically **R**econfigurable systems **I**nterpretive simulation and **V**isualization **E**nvironment(**DRIVE**) framework can be utilized for performance evaluation and architecture and design space exploration. *Interpretive* simulation measures the performance of an application by executing an abstract application model on an abstract parameterized system architecture model. The simulation and visualization framework is being developed in *Java* language and supports modularity and extensibility. A prototype version of the **DRIVE** framework has been implemented and the complete framework will be available to the community.

1 Introduction

Reconfigurable systems are evolving from rapid prototyping and emulation platforms to a general purpose computing platforms. The systems being designed using reconfigurable hardware range from FPGA boards attached to a microprocessor to systems-on-a-chip having programmable logic on the same die as the microprocessor. Reconfigurable systems have been utilized to demonstrate large speed-ups for various classes of applications. Architectures are being designed which support partial and dynamic reconfiguration. The reconfiguration overhead to change the functionality of the hardware is also being diminished by the utilization of configuration caches and multiple contexts on the same device.

* This work was supported by the DARPA Adaptive Computing Systems Program under contract DABT63-96-C-0049 monitored by Fort Hauchuca.

Compilation of user level programs onto reconfigurable hardware is also being explored.

The general purpose computing area is the most promising to achieve significant performance improvement for a wide spectrum of applications using reconfigurable hardware. But, research in this area is hindered by the absence of appropriate techniques and tools. Current design tools are based on ASIC CAD software and have multiple layers of design abstractions which hinder high level optimizations based on reconfigurable system characteristics. Existing frameworks are either based on simulation of HDL based designs [1, 11, 13] or they are tightly coupled to specific architectures [5, 9, 14](See Section 1.1). It is also difficult to incorporate dynamic reconfiguration into the current CAD tools framework. Simulation tools provide a means to explore the architecture and the design space in real time at a very low resource and time cost. The absence of mature design tools also impacts the simulation environments that exist for studying reconfigurable systems and the benefits that they offer. System level tools which analyze and simulate the interactions between various components of the system such as memory and configurable logic are limited and are mostly tightly coupled to specific system architectures.

In this paper we present a novel interpretive simulation and visualization environment based on modeling and module level mapping approach. The **D**ynamically **R**econfigurable systems **I**nterpretive simulation and **V**isualization **E**nvironment(**DRIVE**) can be utilized as a vehicle to study the system and application design space and performance analysis. Reconfigurable hardware is characterized by using a high level parameterized model. Applications are analyzed to develop an abstract application task model. *Interpretive* simulation measures the performance of the abstract application tasks on the parameterized abstract system model. This is in contrast to simulating the exact behavior of the hardware by using HDL models of the hardware devices.

The **DRIVE** framework can be used to perform interactive analysis of the architecture and design parameter space. Performance characteristics such as total execution time, data access bandwidth characteristics and resource utilization can be studied using the **DRIVE** framework. The simulation effort and time are reduced and systems and designs can be explored without time consuming low level implementations. Our approach reduces the semantic gap between the application and the hardware and facilitates the performance analysis of reconfigurable hardware. Our approach also captures the simulation and visualization of dynamically reconfigurable architectures. We have developed the Hybrid System Architecture Model(HySAM) of reconfigurable architectures. This model is currently utilized by the framework to map applications to a system model.

An overview of our framework is given in Section 2. Various aspects of the simulation and visualization framework including our Hybrid System Architecture Model(HySAM) are described in detail in Section 3. Conclusions and future work are discussed in Section 4.

1.1 Related Work

Several simulation tools have been developed for reprogrammable FPGAs. Most tools are device based simulators and are not system level simulators. The most significant effort in this area has been the Dynamic Circuit Switching(DCS) based simulation tools by Lysaght et.al. [13]. Luk et.al. describe a visualization tool for reconfigurable libraries [11]. They developed tools to simulate behavior and illustrate design structure. CHASTE [5] was a toolkit designed to experiment with the XC6200 at a low level. There are other software environments such as CoDe-X [9], JHDL [1], HOTWorks [7], Riley-2 [14], etc.

These tools study the dynamically reconfigurable behavior of FPGAs and are integrated into the CAD framework. Though the simulation tools can analyze the dynamic circuit behavior of FPGAs, the tools are still low level. The simulation is based on CAD tools and requires the input design of the application to be specified in VHDL. The parameters for the design are obtained only after processing by the device specific tools. Most of the software frameworks do not support system level analysis and are utilized for for low level hardware design and evaluation.

2 DRIVE Overview

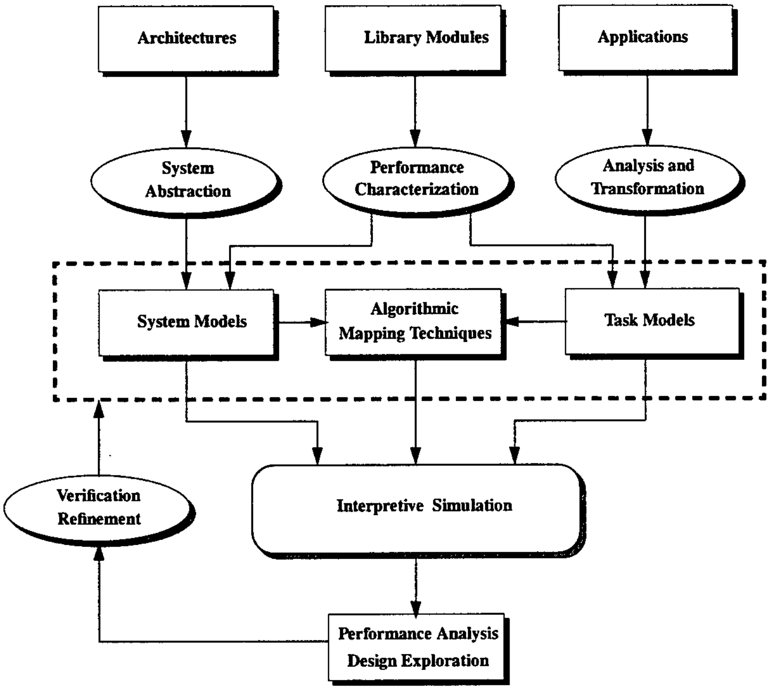


Fig. 1. DRIVE framework

Figure 1 shows an overview of our framework. The system architecture can be characterized to capture the parameter space which affects the performance. The implementations of various optimized modules can be encapsulated by characterizing the performance of the module with respect to the architecture. This characterization is partitioned into the *capabilities* of the system and the actual *implementations* of these *capabilities.* The application is not mapped onto a low level design but is analyzed to develop an application task model. The application model can exploit the knowledge available in the form of the system *capabilities* provided by the module characterization. Algorithmic techniques are utilized to map the application task model to the system models, to perform interpretive simulation and obtain performance results for a given set of parameter values.

Interpretive simulation is performed on the system model which permits a higher level abstract simulation. The application does not need to be actually executed by using device level simulators like HDL models of the architectures. The performance measures can be obtained in terms of the application and model parameters and system characteristics. An interpretive simulation framework will permit design exploration in terms of the architectural choices, application algorithm options, various mapping techniques and possible problem decomposition onto the system components. Development of all the full blown designs which exercise these options is a non-realizable engineering task. Simulation, estimation and visualization tools can be designed to automate this exploration and obtain tangible results in reasonable time.

The abstractions and the techniques that are developed are enclosed in the dashed box in Figure 1. Verification of the models, mapping techniques and simulation framework can be performed by mapping some designs onto actual architectures. This verification process can be utilized to expand on the abstraction knowledge and refine the various models and techniques that are developed. The verification and refinement process completes the feedback loop of the design cycle to result in final accurate models and efficient techniques for optimal designs.

3 Simulation Framework

The simulation framework consists of abstractions and algorithmic techniques as discussed in Section 2(Fig. 1). A high level model of reconfigurable hardware is needed to abstract the low level details. Existing models supplied by the CAD tools have either multiple abstraction layers or are very device specific. We have developed a parameterized model of configurable computing system, which consists of configurable logic attached to a traditional microprocessor. Our model cleanly partitions the *capabilities* of the hardware from the *implementations* and presents a very clean interface to the user. The algorithmic techniques for mapping are not the focus of this paper. Some algorithms for mapping based on the HySAM model are described in our prior work [3, 4].

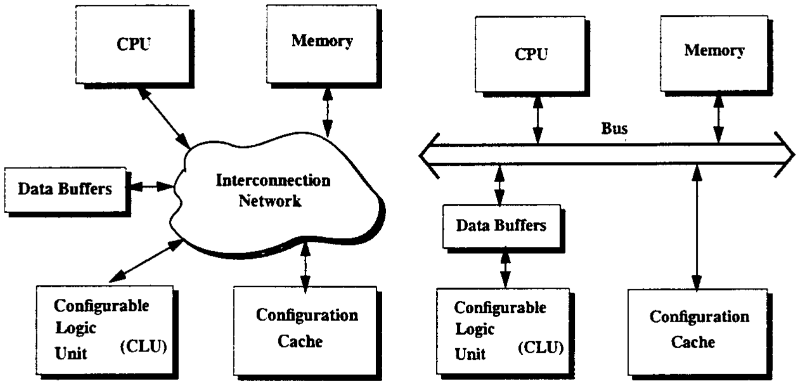


Fig. 2. Hybrid System Architecture and an example architecture

3.1 Hybrid System Architecture Model(HySAM)

The *Hybrid System Architecture Model* is a general model consisting of a traditional microprocessor with additional Configurable Logic Unit(CLU). Figure 2 shows the architecture of the HySAM model and an example of an actual architecture. The architecture consists of a traditional microprocessor, standard memory, configurable logic, configuration memory and data buffers communicating through an interconnection network.

We outline some of the parameters of the Hybrid System Architecture Model(HySAM) below.

F : Set of functions $F_1 \ldots F_n$ which can be performed on configurable logic. (*capabilities*)

C : Set of possible configurations $C_1 \ldots C_m$ of the Configurable Logic Unit. (*implementations*)

A_{ij} : Set of attributes for implementation of function F_i using configuration C_j.

R_{ij} : Reconfiguration cost in changing configuration from C_i to C_j.

G : Set of generators which abstract the composition of configurations to generate more configurations.

B : Bandwidth of the interconnection network (bytes/cycle).

N_c : The number of configuration contexts which can be stored in the configuration cache.

k_c, K_c : The cost of accessing configuration data from the cache and external memory respectively (cycles/byte).

k_d, K_d : The cost of accessing data from the cache and external memory respectively (cycles/byte).

The functions F and configurations C have a *many-to-many* relationship. Each configuration C_i, can potentially contain more than one function F_j. In the HySAM model, only function can be active in a configuration at any given time. Each function F_i can be executed by using any one configuration from a subset of the configurations. The different configurations might be generated by

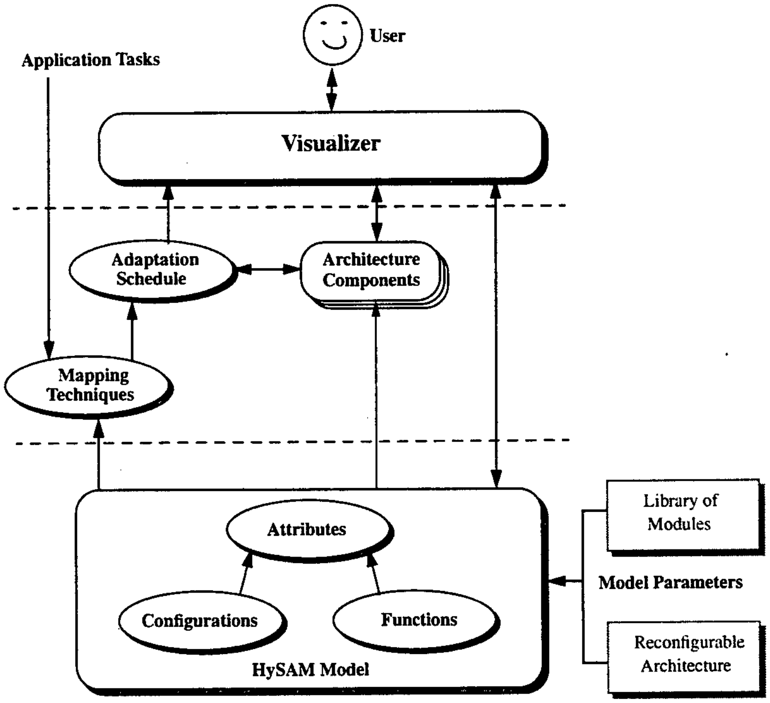


Fig. 3. Major components in the **DRIVE** framework and the information flow

different tools, libraries or algorithms. These configurations might have different area, time, reconfiguration cost, precision, power, etc. characteristics.

The attributes A define the relationship between the functions and the configurations. The attributes define values such as the execution time and the data accessed during execution of a function in a configuration etc. For example, the different execution times and the different data input patterns when a multiplier is implemented as a bit parallel versus a bit serial multiplier are defined by the attributes. The reconfiguration costs R define the costs involved in changing the configuration of the CLU between two configurations. This cost can be statically evaluated based on the configuration information for different configurations. The cost can also be computed dynamically when the configurations are constructed dynamically.

3.2 DRIVE Framework Implementation

An overview of the major components in the **DRIVE** framework and their interactions is given in Figure 3. The framework utilizes high level models of reconfigurable hardware. The current prototype uses the HySAM model described in Section 3.1.

The main input requirements to the **DRIVE** framework are the model parameters and the application tasks. The model parameters supply information about the Functions, Configurations, Attributes and the Reconfiguration costs.

The user can visualize and update any of the instantiated parameters to explore the design space. For a given model parameters, performance results can be obtained for any set of application tasks with various algorithmic mapping techniques.

The high level model partitions the description of the hardware into two components: the Functions(*capabilities*) of the hardware and the Configurations(*implementations*). For example, ability of the hardware to perform multiplication is a capability. The implementations are the different multiplier designs available with varying characteristics such as area, time, precision, structure, etc. Components from a library or modules form the *implementations* in the model and can be determined for different architectures. Vendors and researchers have developed parameterized libraries and modules optimized for a specific architectures. The proposed framework can exploit the various efforts in design of efficient and portable modules [6, 12, 15]. The framework can incorporate such knowledge as the parameters for the HySAM model.

The user only needs to have a knowledge of the *capabilities*. The application task model consists of specification of the application in terms of the Functions(*capabilities*). The input to the framework consists of a directed acyclic graph of the application tasks specified with the Functions as the nodes of the graph. The edges denote the dependencies between the tasks. This technique reduces the effort and expertise needed on the part of the user. The application need not be implemented as an HDL design by the user to study the performance on various reconfigurable architectures. Automatic compilation efforts [2] can be leveraged to generate the Functions from high level language application programs.

Algorithmic mapping techniques are then utilized to map the application specification to actual implementations. These techniques map the *capabilities* to the *implementations* and generate a sequence of configuration, execution, and reconfiguration steps. This is the *adaptation schedule* which specifies how the hardware is adapted during the execution of the application. The schedule contains a sequence of configurations($C_1 \ldots C_q$) where each configuration $C_i \in C$. This *adaptation schedule* can be computed statically for some applications by using algorithmic techniques. Also, the simulation framework can interact with the model and the mapping algorithms to determine the *adaptation schedule* at run-time.

The interpretive simulation framework is based on module level parameterization of the hardware. The user can analyze the performance of the architecture for a given application by supplying the parameters of the model and the application task. Typically the architectural parameters for the model are supplied by the architecture designer and the library designer. But, the user can modify the model parameters and explore the architecture design space. This provides the ability to study design alternatives without the need for actual hardware. The simulation and the performance analysis are presented to the user through a Graphical User Interface. The framework supports incorporation of additional information in the configurations(C) which can be utilized for actual execution or simulation. It can contain configuration bitstreams or class descriptions which can be utilized to perform actual configuration of hardware or simulation using low level models. Using this information, it is possible to link the abstract definitions to actual implementations to verify and refine the abstract models.

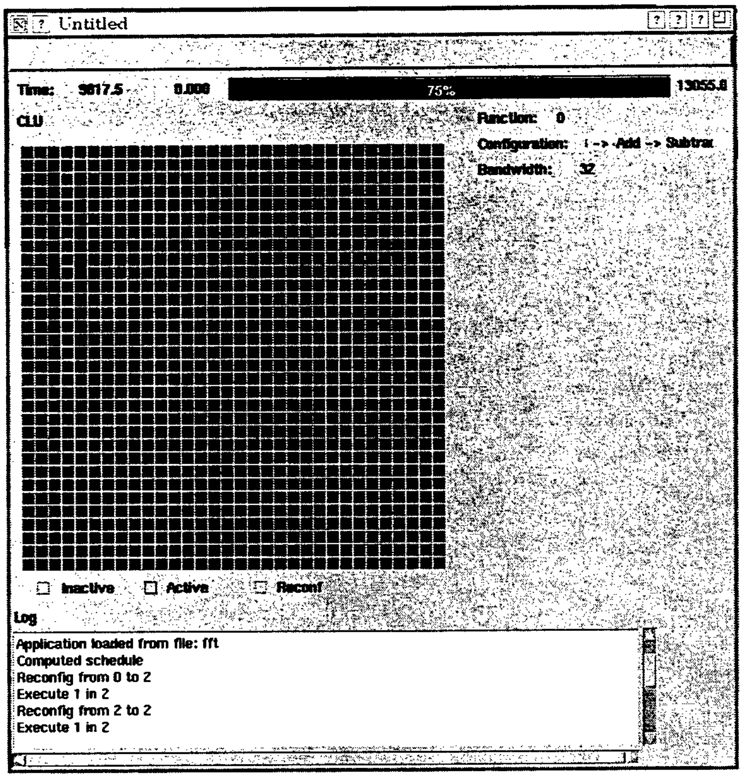
The parameters and attributes of the model can also be evaluated and adapted at run-time to compute the required information for scheduling and visualization. For example, reconfiguration costs can be determined by computing the difference in the configuration information and configurations can even be generated dynamically by future integration of tools like JBits [10]. It is assumed currently that the attributes for configurations are available a priori. It is easy to integrate simulation tools which evaluate the attributes such as execution time by performing simulations as in various module generators [1, 6, 15]. These simulations are based on module generators which do not require mapping using time consuming CAD tools. Once the attribute information for low level modules are obtained by initial simulations and implementations, the attributes for higher level modules can be simulated or computed without the intervention of CAD tools.

The **DRIVE** framework has been designed using object-oriented methodology to support modification and addition to the existing components. The framework facilitates addition of new architectural models, algorithmic mapping techniques, performance analysis tools, etc. in a seamless manner. The framework can also be interfaced to existing tools such as parameterized libraries(Xilinx XBLOX, Luk et. al. [12]), module generators(PAM-Blox [15], Berkeley Object Oriented Modules [6], JHDL [1]), configuration generators(JBits [10]), module interfaces(FLexible API for Module-based Environments [8]), etc. The components of the framework will be made available to the community to facilitate application mapping and modular extensions.

3.3 Visualization

The visualizer for the framework has been developed using the *Java* language AWT toolkit. A previous version of the visualizer was developed using Tcl/Tk. The C programming language was utilized for implementing the simulation engine. The current prototype has been developed in *Java* to utilize the object oriented framework and make the framework modular and easily extensible. Implementing the visualizer and the interpretive simulation in the same language provides for a clearer interface between the components. *Java* is becoming the language of choice for several research and implementation efforts in hardware design and development [1, 6, 10]. Incorporating the results and abstractions from other research efforts is simplified using the current version.

The visualizer acts as a graphical user interface to support the full functionality of the framework. It is implemented as a separate *Java* class communicating with the remaining classes. Any component of the simulation or visualizer framework can be completely replaced with a different component supporting the same interface. The visualizer is oblivious of the algorithmic techniques and implementation details. It accesses information from the different components



Time: 9817.5 0.000 75% 13055.8

Function: 0

Configuration: I --> Add --> Subtrac

Bandwidth: 32

```
Application loaded from file: fft
Computed schedule
Reconfig from 0 to 2
Execute 1 in 2
Reconfig from 2 to 2
Execute 1 in 2
```

Fig. 4. Sample **DRIVE** visualization

in the simulation framework on an event by event basis and displays the state of the various architecture components and the performance characteristics. A sample view of the visualizer is shown in Figure 4.

4 Conclusions

Software tools are an important component of reconfigurable hardware development platforms. Simulation tools which permit performance analysis and design space exploration are needed. The utility of current tools for reconfigurable hardware design is limited by the required user expertise in multiple domains. We have proposed a novel *interpretive* simulation and visualization environment which supports system level analysis. The **DRIVE** framework supports a parameterized system architecture model. Algorithmic mapping techniques have been incorporated into the framework and can be extended easily. The framework can be utilized for performance analysis, design space exploration and visualization. It is implemented in the *Java* language and supports flexible extensions and modifications. A prototype version has been implemented and is currently available. The USC Models, Algorithms and Architectures project is developing algorithmic techniques for realizing scalable and portable applications using configurable computing devices and architectures. Details on **DRIVE** and related research results can be found at http://maarc.usc.edu.

References

1. P. Bellows and B. Hutchings. JHDL - An HDL for Reconfigurable Systems. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.
2. K. Bondalapati, P. Diniz, P. Duncan, J. Granacki, M. Hall, R. Jain, and H. Ziegler. DEFACTO: A Design Environment for Adaptive Computing Technology. In *Reconfigurable Architectures Workshop, RAW'99*, April 1999.
3. K. Bondalapati and V.K. Prasanna. Mapping Loops onto Reconfigurable Architectures. In *8th International Workshop on Field-Programmable Logic and Applications*, September 1998.
4. K. Bondalapati and V.K. Prasanna. Dynamic Precision Management for Loop Computations on Reconfigurable Architectures. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1999.
5. G. Brebner. CHASTE: a Hardware/Software Co-design Testbed for the Xilinx XC6200. In *Reconfigurable Architectures Workshop, RAW'97*, April 1997.
6. M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek. Object Oriented Circuit-Generators in Java. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.
7. Virtual Computer Corporation. Reconfigurable Computing Products, http://www.vcc.com/.
8. A. Koch. Unified access to heterogeneous module generators. In *ACM International Symposium on Field Programmable Gate Arrays*, February 1999.
9. R. Kress, R.W. Hartenstein, and U. Nageldinger. An Operating System for Custom Computing Machines based on the Xputer Paradigm. In *7th International Workshop on Field-Programmable Logic and Applications*, pages 304–313, Sept 1997.
10. D. Levi and S. Guccione. Run-Time Parameterizable Cores. In *ACM International Symposium on Field Programmable Gate Arrays*, February 1999.
11. W. Luk and S. Guo. Visualising reconfigurable libraries for FPGAs. In *Asilomar Conference on Signals, Systems, and Computers*, 1998.
12. W. Luk, S. Guo, N. Shirazi, and N. Zhuang. A framework for developing parametrised FPGA libraries. In *Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, 1996.
13. P. Lysaght and J. Stockwood. A Simulation Tool for Dynamically Reconfigurable FPGAs. *IEEE Transactions on VLSI Systems*, Sept 1996.
14. P.I. Mackinlay, P.Y.K. Cheung, W. Luk, and R. Sandiford. Riley-2: A Flexible Platform for Codesign and Dynamic Reconfigurable Computing Research. In *7th International Workshop on Field-Programmable Logic and Applications*, September 1997.
15. O. Mencer, M. Morf, and M.J. Flynn. PAM-Blox: High Performance FPGA Design for Adaptive Computing. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.

This article was processed using the LaTeX macro package with LLNCS style

International Workshop on Field Programming Logic and Applications, August 1999.

Genetic Programming using Self-Reconfigurable FPGAs *

Reetinder P. S. Sidhu[1], Alessandro Mei[2], and Viktor K. Prasanna[1]

[1] Department of EE-Systems, University of Southern California,
Los Angeles CA 90089, USA
sidhu@halcyon.usc.edu, prasanna@ganges.usc.edu
[2] Department of Mathematics, University of Trento
38050 Trento (TN), Italy
mei@science.unitn.it

Abstract. This paper presents a novel approach that utilizes FPGA *self-reconfiguration* for efficient computation in the context of Genetic Programming (GP). GP involves evolving programs represented as trees and evaluating their fitness, the latter operation consuming most of the time.

We present a fast, compact representation of the tree structures in FPGA logic which can be *evolved as well as executed* without external intervention. Execution of all tree nodes occurs in parallel and is pipelined. Furthermore, the compact layout enables multiple trees to execute concurrently, dramatically speeding up the fitness evaluation phase. An elegant technique for implementing the evolution phase, made possible by self-reconfiguration, is also presented.

We use two GP problems as benchmarks to compare the performance of logic mapped onto a Xilinx XC6264 FPGA against a software implementation running on a 200 MHz Pentium Pro PC with 64 MB RAM. Our results show a speedup of 19 for an arithmetic intensive problem and a speedup of *three orders of magnitude* for a logic operation intensive problem.

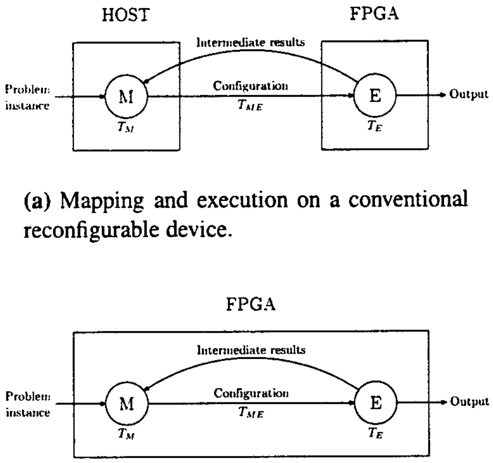
1 Introduction to Self-Reconfiguration

1.1 Problem Instance Dependence and Hardware Compiling

Building logic depending on a single problem instance is the key advantage of reconfigurable computing versus ASICs. That essentially means that a good application for reconfigurable devices should read the input of the problem (the *instance*), compute *instance dependent* logic, i.e. logic optimized for that particular instance, and load it into a reconfigurable device to solve the problem. Applications which produce *instance independent* logic to be loaded onto a reconfigurable device are simply not exploiting the power of reconfiguration. In that case the logic mapped is static, depends only on the algorithm used, and is not conceptually different from ASIC approach.

* This work was supported by the DARPA Adaptive Computing Systems Program under contract DABT63-96-C-0049 monitored by Fort Hauchuca. Alessandro Mei is with the Department of Mathematics of the University of Trento, Italy.

A large class of applications developed for reconfigurable devices can thus be modeled in the following way (see Figure 1(a)). A process M reads the input problem instance. Depending on the instance a logic E, ready to be loaded, is computed such that it is optimized to solve that single problem instance. This process is usually executed by the host computer. Let T_M denote the time to perform this.



(a) Mapping and execution on a conventional reconfigurable device.

(b) Mapping and execution on a self-reconfigurable device.

Fig. 1. Problem instance dependent mapping.

After reconfiguring the device, E is executed. Let T_{ME} denote the time to reconfigure. The time T_E required for the execution includes the time needed for reading the inputs from the memory and producing the output and/or intermediate results sent back to the mapping module. Therefore, the time required by the execution of a single iteration of the computation described above is $T_I = T_M + T_{ME} + T_E$. This process can be iterated. The intermediate results returned by E can be used by M to compute and map new logic toward the final solution of the problem instance.

A large number of applications fit this model. In some of them, a small amount of parallelism can be obtained by running M and E in parallel. However, the best speedup that can be obtained this way is a factor of 2. Thus, we can suppose that only one of the two modules runs at a given time, without loss of generality. Of course, this factor cannot be ignored in performance analysis.

Self-Reconfiguration is a novel approach to reconfigurable computing presented in [12]. It has been shown to be able to dramatically reduce T_M and T_{ME} with respect to classical CAD tool approach. Since M has to be speeded up, the basic idea is to let fast reconfigurable devices to be able to execute it (see Figure 1(b)). In case a single FPGA is being used, the FPGA should be able to read from a memory the problem instance, configure itself, or a part of it, and execute the logic built by it to solve the problem instance. Evidently, in this case M is itself a logic circuit, and cannot be as complex and general as CAD tools.

Letting FPGA system execute both M and E on the same chip gives the clear advantage that CAD tools are used only *once*, in spite of classical solutions where they are needed for computing a logic for each problem instance. This is possible since the adaptations, needed to customize the circuit to the requirements of the actual input, are performed dynamically by the FPGA itself, taking advantage of hardware efficiency.

Another central point is that the bus connecting the FPGA system to the host computer is now only used to input the problem instance, since the reconfiguration data are generated locally. In this way, the bottle-neck problem is also handled.

These ideas have been shown to be realistic and effective by presenting a novel implementation of a string matching algorithm in [12]. In that paper, however, a simpler version of the above model was introduced which consists of a single iteration of the map-execute loop. Nevertheless, speedups in mapping time of about 10^6 over CAD tools were shown.

Since self-reconfiguration has been proved to be very effective in reducing mapping and host to FPGA communication time, we expect that mapping and communication intensive applications can get the maximum advantage from this techniques. A very important example of this kind of an application is Genetic Programming. Section 3 briefly introduces GP and shows how GP applications can fit our model, proving this way they can be dramatically speeded up by using reconfigurable computing enhanced with Self-Reconfiguration.

1.2 Previous Work Related to Self-Reconfiguration

The main feature needed by an FPGA device to fulfill the requirements needed by the technique shown in the previous section is self-reconfigurability. This concept has been mentioned few times in the literature on reconfigurable architectures in the last few years [5][4].

In [5], a small amount of static logic is added to a reconfigurable device based on an FPGA in order to build a self-reconfiguring processor. Being an architecture oriented work, no application of this concept is shown. The recent Xilinx XC6200 is also a self-reconfiguring device, and this ability has been used in [4] to define an abstract model of virtual circuitry, the Flexible URISC. This model still has a self-configuring capability, even though it is not used by the simple example presented in [4]. The concept of self-reconfiguration has also been used in the reconfigurable mesh [6]—a theoretical model of computation—to develop efficient algorithms. However, there has been no demonstration (except in [12]) of a practical application utilizing self-reconfiguration of FPGAs to improve performance. This paper shows how self-reconfiguration can be used to obtain significant speedups for Genetic Programming problems.

Devices like the XC6200 can self-reconfigure and are thus potentially capable of implementing the ideas presented in this paper. However, moving the process of building the reconfigurable logic into the device itself requires a larger amount of configuration memory in the device compared to traditional approaches. For this reason, multi-context FPGAs are better suited since they can store several contexts (see [10], for example, where a self-reconfiguring 256-context FPGA is presented).

2 Multicontext FPGAs

As described in the Introduction, the time required to reconfigure a traditional FPGA is very high. To reduce the reconfiguration time, several such *multicontext* FPGAs have been recently proposed [11][10][13] [7][3].

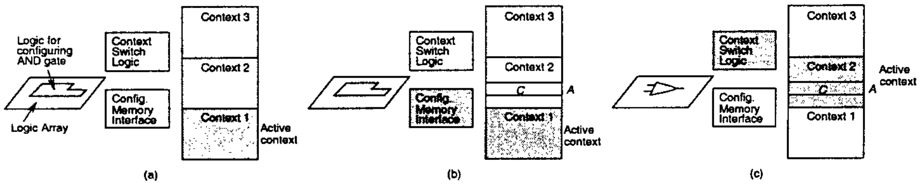


Fig. 2. Self Reconfiguration and context switching in a Multicontext FPGA.

These devices have on-chip RAM to store a number of configuration contexts, varying from 8 to 256. At any given time, one context governs the logic functionality and is referred to as the *active* context. Switching contexts takes 5–100 ns. This is several orders of magnitude faster than the time required to reconfigure a conventional FPGA (≈1 ms). For self-reconfiguration to be possible, the following two additional features are required of multicontext FPGAs:

- The active context should be able to initiate a context switch—no external intervention should be necessary.
- The active context should be able to read and write the configuration memory corresponding to other contexts.

The multicontext FPGAs described in [11][10][13] satisfy the above requirements and hence are capable of self-reconfiguration. Figure 2 illustrates how a multicontext FPGA with above features can modify its own logic. As shown in Figure 2(a), the active context initially is context 1 which has logic capable of configuring an AND gate. Figure 2(b) shows this logic using the configuration memory interface to write bits corresponding to an AND gate at appropriate locations in the configuration memory corresponding to Context 2. Finally the logic on the context 1 initiates a context switch to Context 2 which now has an AND gate configured at the desired location.

3 Introduction to Genetic Programming

Genetic Programming [8] is an adaptive and learning system evolving a *population* of individual computer programs. The evolution process generates a new population from the existing one using analogs of Darwinian principle and genetic operations such as mutation and sexual recombination. In Genetic Programming, each *individual* is obtained by recursively composing *functions* taken from a set $F = \{f_1, \ldots, f_{N_{\text{func}}}\}$, and *terminals* from $T = \{a_1, \ldots, a_{N_{\text{term}}}\}$. Each of the individuals has an associated *fitness* value, usually evaluated over a set of *fitness cases*.

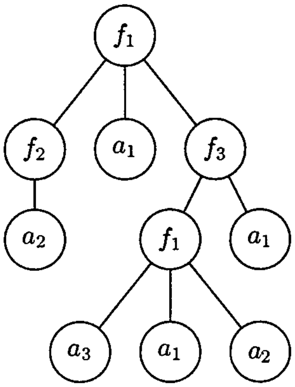


Fig. 3. Example of individual tree structure in Genetic Programming.

A natural way of representing an individual is thus as a *tree*, where a leaf contains a terminal and an internal node a function whose arity is exactly equal to the number of its children (see Figure 3). The evolution process, starting from a randomly generated population of individuals, iteratively transforms it into a new population by applying the following genetic operations:

reproduction Reproduce an existing individual by copying it into the new population.

crossover Create two new individuals by genetically recombining two existing ones. This is done by exchanging the subtrees rooted at two randomly chosen crossover points, one per parental tree.

mutation Create a new individual from an existing one by randomly changing a randomly chosen subtree.

The genetic operations are applied to individuals in the population selected with a probability based on their fitness value, simulating the driving force of Darwinian natural selection: survival and reproduction of the fittest. Computing the fitness value of each individual is a central computational task of GP applications, usually taking around 95-99% of the overall computation time.

It is thus not surprising that the main effort aimed to speedup a Genetic Programming application is focused on the fitness evaluation. For example, in [2] an FPGA is used to accelerate the computation of the fitness value of a population of sorting networks achieving much faster execution.

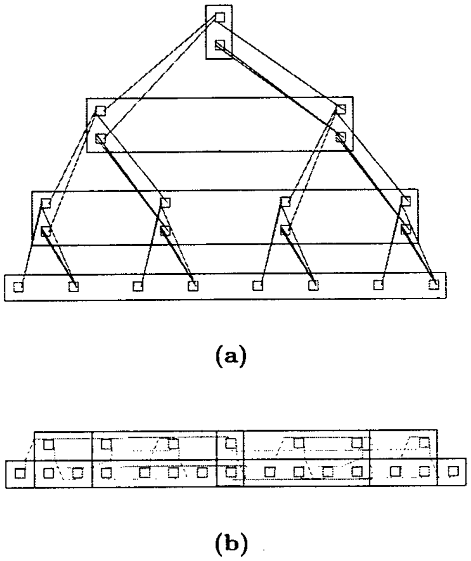
It is worth noting that the reconfigurable computing application presented in [2] nicely fits our model shown in Figure 1(a). Indeed, M is the process responsible for managing and storing the population, computing the logic E to fitness test each individual, mapping it onto the device, and reading the output value. This operation is repeated for each individual in the population and for each generation in the evolutionary process, resulting in a considerable mapping and host to FPGA communication overhead. Our performance evaluation (see Section 7) shows that reconfiguration time (T_{ME}) is greater than the fitness evaluation time (T_E) and thus self-reconfiguration is essential.

Moreover, in [2] only a rather specific application is shown to benefit from FPGA computing, and it is not clear how the same approach can be extended to an arbitrary GP application.

This paper presents important improvements toward in directions. First, it is shown how a generic GP application can be mapped onto an FPGA system, taking advantage of the massive parallelism of contemporary devices in several ways. Second, how Self Reconfiguration can dramatically speed it up, by handling long mapping and reconfiguration times, and by allowing the evolution phase, as well as the fitness evaluation phase, to be mapped onto the FPGA. The FPGA executes the complete GP algorithm and does not require any external control.

We begin by describing the mapping of the program trees onto FPGA logic in the following section. Section 5 presents the proposed operation of a GP algorithm on FPGAs. The two GP problems used as benchmarks are discussed in Section 6 and the results obtained are presented in Section 7. We summarize the contributions of this paper in Section 8.

4 Tree Template


(a)

(b)

Fig. 4. Compact tree layout using hierarchical interconnect structure.

Before execution begins, a number of *tree templates* are configured onto various contexts of the FPGA. Each template holds the tree representing an individual program throughout its lifetime. As evolution progresses, the nodes of the tree template are appropriately configured to represent the program—the interconnection remain fixed. By configuring the nodes to reflect the actual program, efficient execution results through pipelined execution of all nodes of the tree in parallel (see Section 5.2). By employing a template with static interconnect, fitness evaluation is speeded up and implementation of the mutation, reproduction and crossover operators is simplified (see Sections 5.2 and 5.3). The template tree is a complete binary tree of height k (having $n = 2^k - 1$ nodes). Number of levels of the tree is restricted to the number of levels of the template tree. (Restricting the number of levels is a common technique used in GP implementations to limit tree size.) Below we discuss its mapping onto FPGA logic cells and interconnect.

We map the nodes of the tree along a single row or column of logic cells. The sequence of nodes is the same as obtained through an in-order traversal of the binary tree. The width of each node is a power of 2 while its height is arbitrary—it depends upon the complexity of the functions in the function set. All nodes have the same size.

Figure 4(a) shows a complete 15 node binary tree. Also shown in Figure 4(b) is the mapping of the template tree edges onto wires of the interconnect of the Xilinx XC6200 FPGA architecture[1]. The compact mapping of the tree structure is possible because the interconnect of the XC6200 FPGAs, like that of most other FPGA architectures, is hierarchical. Moreover, most newer generation FPGA architectures (including the Xilinx XC4000 and Virtex, and the Atmel AT40K) have richer and more flexible interconnects than the XC6200. Thus the tree template can be easily mapped onto such FPGAs.

5 Operation

5.1 Initialization

A number of tree templates (equal to the required population size) are configured on one or more contexts of the FPGA. These templates are then initialized with trees generated using standard GP techniques [1]. The size of the nodes in the template is chosen to accommodate the largest area occupied by a function implementation. The size of the template itself is chosen to accommodate the desired maximum number of levels in the trees. Also configured is logic required for the fitness evaluation and evolution phases (explained in the following two sections).

5.2 Fitness Evaluation Phase

Figure 5 shows the datapath configured onto each context. The test case generator iterates through all the test cases. It uses a part of the context memory to store the test cases. For each case it also generates the expected output. A set of values corresponding to the members of the terminal set (described in Section 3) forms a test case. The crossbars are required to map these terminal set values onto leaf nodes of the tree templates. The crossbars are configured using self-reconfiguration in the evolution phase. All nodes of the template trees process the test case in parallel. There is concurrent execution of all nodes in each tree level and pipelined execution along each path. The test case generation and fitness computation are also pipelined and thus do not incur additional overhead. As shown in Figure 5, the fitness computation logic compares the output of the tree for a test case with the corresponding expected value. The resulting measure of fitness is accumulated in the cumulative fitness register. The two benchmark GP problems described in Section 7 give concrete examples of test cases, test case generation logic and fitness computation logic.

We now compute the time required to perform fitness evaluation using the above approach. The total time required to evaluate the fitness of a single generation is:

$$T_{FE} = (t_{\text{node}} \times n_{\text{tests}} + t_{\text{testgen}} + t_{\text{crossbar}} + kt_{\text{node}} + t_{\text{fitcomp}} + t_{\text{fitreg}}) \left\lceil \frac{n_{\text{trees}}}{n_{\text{tcontext}}} \right\rceil \quad (1)$$

where

$$k = \text{number of levels in the tree templates}$$
$$t_{\text{node}} = \text{latency of a node}$$

[1] It should be noted that the XC6200 is used purely for illustration and the proposed mapping in no way depends on any XC6200 specific features. Its choice here was motivated by the authors' familiarity with its CAD tools rather than any architectural considerations.

n_{trees} = total number of trees to be evaluated
n_{tcontext} = number of trees per context
n_{tests} = total number of fitness tests
t_{testgen} = test generator latency
t_{crossbar} = crossbar latency
t_{fitcomp} = fitness computation latency
t_{fitreg} = fitness register latency

The first term within the parenthesis represents the time required for pipelined execution of the test cases while the remaining terms account for the pipeline fill time. The sum within the parenthesis is the time in each of the $\left\lceil \frac{n_{\text{trees}}}{n_{\text{tcontext}}} \right\rceil$ contexts.

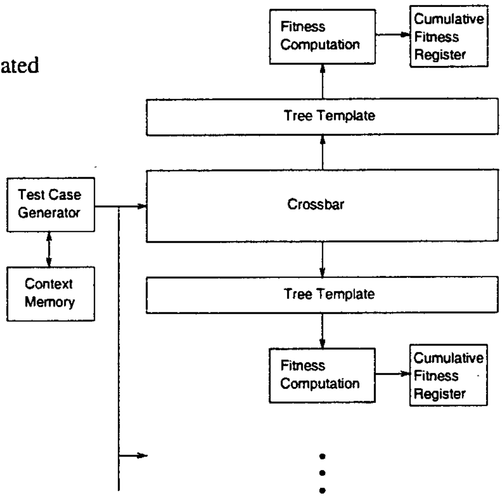


Fig. 5. Datapath for fitness evaluation. All nodes of multiple trees are evaluated in parallel.

5.3 Evolution Phase

Evolution involves modifying some of the programs, letting a portion of the programs die (based on their fitness), and generating new programs to replace the dead ones. In this phase, self-reconfiguration is utilized to modify some of the trees and generate new ones. The modification is achieved through the genetic operations of mutation and crossover while reproduction is used to create new programs. Below we discuss how the ability of self-reconfigurable FPGAs to modify their own configuration is used to implement the genetic operators that manipulate the trees. It should be noted that the evolution phase consumes only 1–5% of the total execution time. Hence the discussion below is qualitative in nature and illustrates how self-reconfiguration elegantly implements the three genetic operations on the FPGA, without any external intervention.

Figure 6 shows the major logic blocks required to perform evolution. This logic operates in bit-serial fashion and is configured on a separate context. The random number generators shown can be efficiently implemented as discussed in [9]. The tree template shown has the same number of nodes as other templates but node contents differ. Each node of the template stores its own offset address. For e.g. the root node stores $\frac{n}{2}$. These can be efficiently stored in each node using $\log_2 n$ flip-flops. The offset—added to a tree base address—is used by the configuration memory interface to access a node of that tree. Each node also has an active bit which when set causes it shift out its address in bit-serial fashion. The logic shown solves in an elegant manner the problem of subtree traversal which is used for all the three operations as described below.

Reproduction The active bit of the root node of the tree template is set. Next the increment signal is applied for k (number of levels in template) clock cycles. In response to the increment signal all active nodes (while remaining active themselves) set the active bits of their child nodes. Thus after k clock cycles, all nodes are active. Next, one bit of the shift register is set (rest are 0). The offset address of the corresponding node is read out and is used to read a node from the source tree and write it into the destination tree template. Next the shift register advances one location and another node gets copied. In this manner, after n shifts, a tree is reproduced.

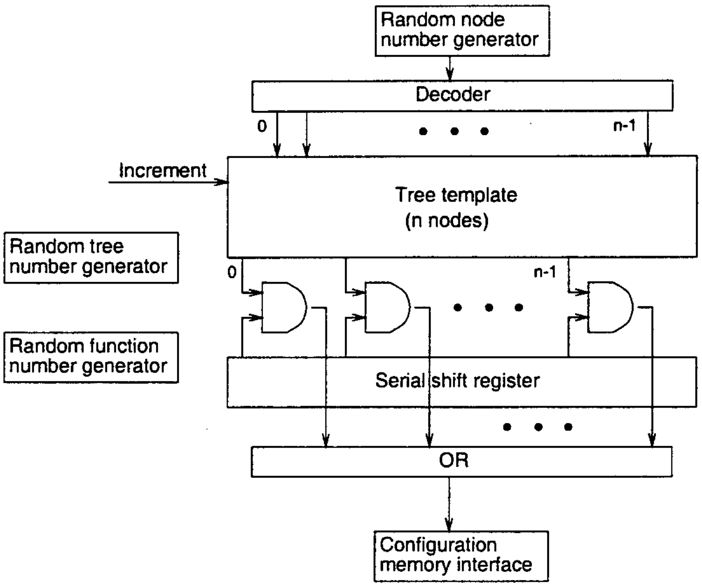

Random node number generator
Decoder
0
n-1
Increment
Tree template (n nodes)
Random tree number generator
Random function number generator
0
n-1
Serial shift register
OR
Configuration memory interface

Fig. 6. Logic for the evolution phase. The tree template shown is used to map the node numbers onto the linear configuration memory address space. Each node stores its own offset address.

Crossover The problem is to swap (randomly chosen) subtrees of two (randomly chosen) parents. The key operation is the subtree traversal which is elegantly done. The output of the random node number generator is used to set the active bit of one node which forms the root of the subtree. Next (as in reproduction), the increment signal is applied for k clocks after which the active bits of all the nodes of the subtree are set. The shift register (with a single bit set) is then shifted n times. On each shift, the address of the corresponding node (if active) is read out. In this manner, after n shifts, the subtree is traversed. The crossover operation requires four such traversals, two for each of the subtrees. In the first two traversals, the subtrees are read into scratchpad configuration memory. In the next two traversals, they are written into each others' original locations.

Mutation Mutation involves replacing a randomly selected subtree with a randomly generated subtree. The subtree selection and traversal are performed as for crossover above. For each node visited during the traversal, its current configuration is replaced by the contents corresponding to the output of the random function number generator.

6 Implementation

We evaluated the performance of our approach in the following manner. We chose two GP problems as benchmarks. Both problems were selected from Koza's *Genetic Programming*[8]. For each we implemented the fitness evaluation logic (as discussed in Section 5.2) onto a conventional FPGA. This implementation was used to obtain the minimum clock cycle time and the maximum number of trees that fit on a single context. Using this information and Equation 1 the time required by our approach to perform fitness evaluation was computed. Next, a software implementation of the two benchmarks was run and the time spent on fitness evaluation measured.

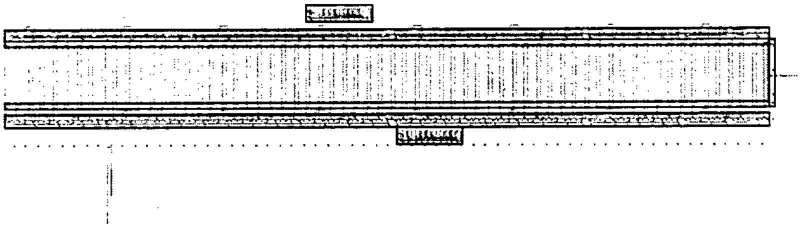


Fig. 7. A section of the layout for the multiplexer problem with 127 node tree templates.

The speedup obtained using our approach was then computed from the fitness evaluation times of both the approaches and Equation 1.

The choice of the two problems was motivated by the nature of their function sets (explained in Section 3). For one of the problems (multiplexer), all the members of its function set were bit-level logic functions. For the other (regression) all function set members were arithmetic functions operating on integers. Clearly, FPGAs would provide a greater speedup over a microprocessor for the former problem compared with the latter. Typically, the function set of a GP problem contains a mix of arithmetic and logic functions. Therefore, performance evaluation of the chosen problems would yield an estimate of the range of speedups that can be obtained over a microprocessor.

The following two GP problems were chosen:

Multiplexer The problem is to evolve a program that exhibits the same behavior as a multiplexer having 8 data inputs and 3 control inputs. The test cases are all 2^{11} possible inputs. The corresponding expected values are the boolean outputs a multiplexer would produce. The function set consists of logic functions `and`, `or` and `not`, and the `if` function which is essentially a multiplexer with 2 data inputs and a control input. The functions are much simpler than needed by many GP applications. But real problems such as evolution of BDDs also employ such simple functions. The terminal set has 11 members—the 8 data inputs and the 3 control inputs.

Regression the problem is to evolve a function that “fits” a number of known (x, y) points. The x coordinates are used as the fitness cases while the corresponding y is the expected value. We use 200 test cases (n_{tests}=200). The function set consists of `add`, `subtract` and `multiply`. The terminal set consists of the input value x and integer constants in the range $[-1, 1)$.

7 Performance Evaluation

The Xilinx XC 6264 was used as the target FPGA. Required logic was specified in structural VHDL and translated to EDIF format using velab. XACT 6000 was used for place, route and configuration file generation. Software implementation of the benchmarks was carried out using the lil-gp kernel [14]. The resulting executable was run on a PC with a 200 MHz Pentium Pro processor and 64 MB RAM. The population size for both approaches was fixed at 100 individuals.

7.1 Multiplexer

Area Requirements Figure 7 shows the layout on a Xilinx XC 6264 (128 × 128 logic cells) of two trees (each having 127 nodes) and the associated (simulated) crossbar for the multiplexor problem—it is similar to Figure 5 (except for the test case generator which appears on the right side). For this problem, the fitness computation logic reduces to an XOR gate and the cumulative fitness register is just a counter controlled by the XOR gate output (these appear on the top and bottom of the trees). To model the worst case delay through an actual crossbar, all inputs to each tree originate from the crossbar row furthest from it. Since two trees (and cross bar) fit in 20 rows, the 128 row FPGA can accommodate 12 127 node trees on it. The layout for the 63 node trees is similar except they occupy half the number of columns—thus twice as many 63 node trees fit on to the FPGA.

From the above mapped logic, the minimum clock cycle time (t_{clk}) for both tree sizes was determined which is shown in Table 1. It should be noted that the major component of $t_{\text{clk}}$is the crossbar—the critical paths through the 127 and 63 node trees were just 14.31 ns and 12.47 ns. Thus an efficient crossbar can provide even further improvements. Also shown are the number of clock cycles required which are computed using Equation 1 for n_{trees}=100 and n_{tcontext}=12 and 24. It should be clear from Figure 7 that all the times in Equation 1 (including t_{node}) are equal to t_{clk}. Finally multiplying by t_{clk}yields the time T_{FE} required to fitness evaluate a single generation of 100 trees using the proposed approach. It should be noted that T_{FE} is for fitness evaluation of all trees on all contexts.

Table 1. Area requirements for the multiplexor problem for tree templates having 127 and 63 nodes. Each context has 128 × 128 logic cells.

Structure	Area (in logic cells)	
	n = 127 nodes	n = 63 nodes
Tree template	127 × 3	63 × 3
Crossbar	127 × 11	63 × 11
Test case generator	1 × 11	1 × 11
Fitness logic	12 × 3	12 × 3
Number of trees per context (n_{tcontext})	12	24

Table 2. Time required to fitness evaluate 100 trees using proposed approach.

	Area (in logic cells)	
	n = 127 nodes	n = 63 nodes
Clock cycle (t_{clk})	48.96 ns	37.08 ns
Clock cycles	18531	10290
Time taken (T_{FE})	907.3 μs	381.6 μs

Table 3. Fitness evaluation times for a generation of 100 individuals.

Approach	T_{FE}	
	n = 127 nodes	n = 63 nodes
Proposed	907.3 μs	381.6 μs
Software	930 ms	440 ms
Speedup	**1025**	**1153**

Time Requirements To obtain T_{FE} for the software implementation, it was executed for a population size of a 100 individuals. This experiment was conducted twice with the maximum nodes per tree restricted to 127 and 63 thus ensuring that the tree size limits are the same as in our approach. Each time, execution was carried out for a 100 generations and the total time spent on fitness evaluation was noted. From this, the average fitness evaluation time per generation was obtained which is shown in Table 3. As can be seen, the proposed approach is almost *three orders of magnitude* faster than a software implementation (for fitness evaluation).

7.2 Regression

Area Requirements Regression requires much greater area compared to the multiplexer problem due to the (bit-serial) multiply operation—each node requires 4 × 16 logic cells. Table 4 shows the area requirements. Note that since the terminal set consists of just one variable (x), in contrast to 11 for the multiplexer, the crossbar reduces to 124 × 1 logic cells. The other terminal set members (integer constants) are implemented by embedding them in the corresponding terminal nodes. Two 31 node trees and the associated circuitry fit into 35 rows of the XC 6264. Thus 6 trees can be accommodated. The fitness computation and accumulation logic consists of bit-serial comparator and adder.

Table 4. Area requirements for the regression problem for a tree template 31 nodes. Each context has 128 × 128 logic cells.

Structure	Area (in logic cells)
	$n = 31$ nodes
Tree template	124 × 16
Crossbar	124 × 1
Test case generator	1 × 16
Fitness logic	20 × 2
Number of trees per context (n_{tcontext})	6

Table 5. Time required to fitness evaluate 100 trees using proposed approach.

	Area (in logic cells)
	$n = 31$ nodes
Clock cycle (t_{clk})	28.86 ns
Clock cycles	115073
Time taken (T_{FE})	3321.0 μs

Table 6. Fitness evaluation times for a generation of 100 individuals.

Approach	T_{FE}
	$n = 31$ nodes
Proposed	3321.0 μs
Software	62.9 ms
Speedup	**19.0**

Time Requirements Operands are 16-bit values and all operations are performed in a bit-serial fashion. Latency t_{node}=33 clock cycles due to the multiply (only 16 MSB used). As can be seen from Table 5, the latency (in number of clock cycles) is higher but the clock cycle time is lower (since the "crossbar" is smaller and remains fixed) compared to the multiplexor. Clock cycles are computed for n_{tests}=200. Table 6 shows that the proposed approach achieves a speedup of 19 (in fitness evaluation) over a software implementation for the regression problem. This is a significant speedup for a single FPGA considering the arithmetic intensive nature of the problem.

8 Conclusion

We have demonstrated dramatic speedups—of upto three orders of magnitude—for fitness evaluation, the GP phase that consumes 95-99% of execution time. This speedup is achieved due to the fast, compact representation of the program trees on the FPGA. The representation enables parallel, pipelined execution of all nodes in parallel and also concurrent execution of multiple trees.

It should be noted that self-reconfiguration is essential for the above speedup. In the absence of self-reconfiguration, the evolution phase would be performed off-chip, and the resulting trees would have to be reconfigured onto the FPGA doing which would consume about 1 ms per context (much more if configuration done over a slow I/O bus). As can be seen from Section 7 our approach fitness evaluates a *several* contexts of trees in less than 1 ms. Since the reconfiguration time is greater than the execution time, the speedups obtained would be greatly reduced.

Self-reconfiguration eliminates external intervention and the associated penalty by allowing the chip to modify its own configuration and thus perform the evolution phase on-chip. We have also shown an elegant technique for performing the evolution phase using self-reconfiguration.

References

1. BANZHAF, W., NORDIN, P., KELLER, R. E., AND FRANCONE, F. D. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, dpunkt.verlag, Jan. 1998.
2. BENNETT III, F. H., KOZA, J. R., HUTCHINGS, J. L., BADE, S. L., KEANE, M. A., AND ANDRE, D. Evolving computer programs using rapidly reconfigurable FPGAs and genetic programming. In *FPGA'98 Sixth International Symposium on Field Programmable Gate Arrays* (Doubletree Hotel, Monterey, California, USA, 22-24 Feb. 1998), J. Cong, Ed.
3. DEHON, A. Multicontext Field-Programmable Gate Arrays. http:/-/HTTP.CS.Berkeley.EDU/ãmd/CS294S97/papers/dpga_cs294.ps.
4. DONLIN, A. Self modifying circuitry - a platform for tractable virtual circuitry. In *Eighth International Workshop on Field Programmable Logic and Applications* (1998).
5. FRENCH, P. C., AND TAYLOR, R. W. A self-reconfiguring processor. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1993), D. A. Buell and K. L. Pocek, Eds., pp. 50–59.
6. JANG, J. W., AND PRASANNA, V. K. A bit model of reconfigurable mesh. In *Reconfigurable Architectures Workshop* (Apr. 1994).
7. JONES, D., AND LEWIS, D. A time-multiplexed FPGA architecture for logic emulation. In *Proceedings of the 1995 IEEE Custom Integrated Circuits Conference* (May 1995), pp. 495–498.
8. KOZA, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.
9. LAVENIER, D., AND SAOUTER, Y. Computing goldbach partitions using pseudo-random bit generator operators on a fpga systolic array. In *Field-Programmable Logic and Applications, Eighth International Workshop, FPL '98* (1998), pp. 316–325.
10. MOTOMURA, M., AIMOTO, Y., SHIBAYAMA, A., YABE, Y., AND YAMASHINA, M. An embedded DRAM-FPGA chip with instantaneous logic reconfiguration. In *1997 Symposium on VLSI Circuits Digest of Technical Papers* (June 1997), pp. 55–56.
11. SCALERA, S. M., AND VÁZQUEZ, J. R. The design and implementation of a context switching FPGA. In *IEEE Symposium on Field-Programmable Custom Computing Machines* (Apr. 1998), pp. 495–498.
12. SIDHU, R. P. S., MEI, A., AND PRASANNA, V. K. String matching on multicontext FPGAs using self-reconfiguration. In *FPGA '99. Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays* (1999), pp. 217–226.
13. TRIMBERGER, S., CARBERRY, D., JOHNSON, A., AND WONG, J. A time-multiplexed FPGA. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines* (Napa, CA, Apr. 1997), J. Arnold and K. L. Pocek, Eds., pp. 22–28.
14. ZONGKER, D., PUNCH, B., AND RAND, B. lil-gp genetic programming system. http://GARAGe.cps.msu.edu/software/lil-gp/lilgp-index.html.

Military and Aerospace Applications of Programmable Devices and Technologies, September 1999

Mapping Applications onto Reconfigurable Architectures using Dynamic Programming (Summary) *

Kiran Bondalapati, George Papavassilopoulos and Viktor K. Prasanna
Department of Electrical Engineering Systems, EEB-200C
University of Southern California
Los Angeles, CA 90089-2562. USA
{kiran,yorgos,prasanna}@ceng.usc.edu

Introduction

Reconfigurable architectures vary from systems which have FPGAs and glue logic attached to a host computer to systems which include configurable logic on the same die as a microprocessor. Automatic compilation of applications onto reconfigurable architectures involves not only configuration generation, but also configuration management. Currently, there is no unified methodology for mapping applications to configurable hardware.

In this paper we describe algorithmic techniques for automatic mapping of applications in a platform independent fashion. We have developed an abstract model of reconfigurable architectures. This parameterized abstract model is general enough to capture a wide range of configurable systems. These include board level systems which have FPGAs as configurable computing logic to systems on a chip which have configurable logic arrays on the same die as the microprocessor.

Configurable logic is very effective in speeding up regular, repetitive computations. Loop constructs in general purpose programs are one such class of computations. In this paper, we address the problem of mapping a loop construct onto configurable architectures. The Hybrid System Architecture Model(HySAM) that we have developed is utilized to define the mapping problems. Efficient techniques based on dynamic programming are used to develop an optimal schedule for important variants of the problem. The problem of utilizing on-chip reconfiguration cache resources is addressed in this paper. The techniques are illustrated by mapping an example FFT loop onto the Berkeley Garp architecture.

Hybrid System Architecture Model(HySAM)

*This work was supported by the DARPA Adaptive Computing Systems Program under contract DABT63-96-C-0049 monitored by Fort Hauchuca.

To realize a formal framework for algorithm development, we developed the Hybrid System Architecture Model of reconfigurable architectures. The *Hybrid System Architecture* is a general architecture consisting of a conventional microprocessor with an additional Configurable Logic Unit(CLU). The architecture consists of a conventional microprocessor, standard memory, configurable logic, configuration memory and data buffers communicating through an interconnection network. Key parameters of the Hybrid System Architecture Model(HySAM) are outlined below.

F : Set of functions $F_1 \ldots F_n$ which can be performed on configurable logic.

C : Set of possible configurations $C_1 \ldots C_m$ of the Configurable Logic Unit.

A_{ij} : Set of attributes for implementation of function F_i using configuration C_j(execution time, precision etc.).

R_{ij} : Reconfiguration cost in changing configuration from C_i to C_j.

G : Set of generators which abstract the composition of configurations to generate more configurations.

B : Bandwidth of the interconnection network(bytes/cycle).

The parameterized HySAM models a wide range of systems from board level architectures to systems on a chip. The values for each of the parameters establish the architecture and also dictate the class of applications which can be effectively mapped onto the architecture. For example, a system on a chip architecture would have potentially faster reconfiguration times than a board level architecture.

Mapping Loop Statements

Scheduling a general sequence of tasks with a set of dependencies to minimize the total execution time is known to be an NP-complete problem. We consider the problem of generating this sequence of configurations for loop constructs which have a sequence of statements to be executed in linear order. There is a linear data or control dependency between the tasks. Most loop constructs, including those which are mapped onto high performance pipelined configurations, fall into such a class.

The total execution time includes the time taken to execute the tasks in the chosen configurations and the time spent in reconfiguring the logic between successive configurations. We have to not only choose configurations which execute the given tasks fast, but also have to reduce the reconfiguration time. It is possible to choose one of many possible configurations for each task execution. Also, the reconfiguration time depends on the choice of configurations that we make.

Problem: Given a sequence of tasks of a loop, T_1 through T_p to be executed in linear order(T_1 $T_2 \ldots T_p$), where $T_i \in F$, for N number of iterations, find an optimal sequence of configurations S (=C_1 $C_2 \ldots C_q$), where $S_i \in C$ (=$\{C_1$,C_2,$\ldots$,$C_m\}$) which minimizes the execution time cost E. E is defined as

$$E = \sum_{i=1}^{q} (t_{S_i} + R_{i-1i})$$

where t_{S_i} is execution time in configuration S_i and R_{i-1i} is reconfiguration cost.

Optimal Solution for Mapping Loops

A simple greedy approach of choosing the best configuration for each task will not work since the reconfiguration costs for later tasks are affected by the choice of configuration for the current task. We outline our dynamic programming based approach below without proofs:

Lemma 1: Given a sequence of tasks $T_1'T_2' \ldots T_p'$, an optimal sequence of configurations for executing these tasks **once** can be computed in $O(pm^2)$ time.

Lemma 1 provides a solution for an optimal sequence of configurations to compute one iteration of the loop statement. But repeating this sequence of configurations is not guaranteed to give an optimal execution for N iterations.

Lemma 2 An optimal configuration sequence can be computed by unrolling the loop only m times.

Theorem 1 The optimal sequence of configurations for N iterations of a loop statement with p tasks, when each task can be executed in one of m possible configurations, can be computed in $O(pm^3)$ time. $\odot$

Theorem 1 is derived from Lemma 1 and Lemma 2 and the complexity of the algorithm is $O(pm^3)$. This approach can also be used when the number of iterations N is not known at compile time and is determined at runtime. The decision to use this sequence of configurations to execute the loop can be taken at runtime from the statically known loop setup and single iteration execution costs and the runtime determined N.

Multiple Contexts and Configuration Caches

The performance achievable on reconfigurable architectures is limited by the costs involved in reconfiguring the logic. Currently, this overhead is very high and discourages the reconfiguration of the logic during the execution of a single application. To address this problem architectures which support configuration caches and multiple contexts on the devices are being developed. We extend the above approach for these devices with the following assumptions regarding the HySAM model:

1. N_c number of configurations can be loaded on to the device at the start of the computation.
2. There is one active context which can be configured from any of the N_c configurations with a cost k_c.
3. The pre-loaded configurations can not be modified during the execution of the complete application. Only the active context can be reconfigured externally.

We define an additional variable X_{ij}, $1 \leq j \leq 2 * m$, which is the set of contexts which are cached for executing tasks T_1 to T_i with T_i being executed using configuration C_j. The E_{ij} and the X_{ij} ($1 \leq i \leq 2 * m$) values are computed using *dynamic programming*. The recursive equations for computing them are given below(δ_{kj} denotes the reconfiguration cost):

$$mink = k \; s.t. \; min[E_{ik} + \delta_{kj}] \;\; 1 \leq k \leq 2 * m$$

```
$if\ (C_j \in X_{ik})$
        $\delta_{kj} = k_c$
$else\ if\ (|X_{kj}| < N_c\ and\ 1 \leq j \leq m)$
        $\delta_{kj} = k_c$
$else$
        $\delta_{kj} = R_{ij}$
```

Given the value of $mink$, the E_{i+1j} and the X_{i+1j} values are computed as follows:

$$\begin{aligned} E_{i+1j} &= t_{i+1j} + E_{i\ mink} + \delta_{mink\ j} \\ X_{i+1j} &= X_{i\ mink} \cup C_j \\ &\quad\; if\ |X_{imink}| < N_c\ \ and\ \ 1 \leq j \leq m) \\ &= X_{i\ mink} \quad otherwise \end{aligned}$$

The minimum execution cost E and the corresponding set of contexts X for executing tasks T_1 to T_p are given by:

$$minj = j \; s.t. \; min[E_{pj}] \;\; 1 \leq j \leq 2 * m$$

$$E = E_{p\ minj}$$

$$X = X_{p\ minj}$$

The required optimal execution cost and the set of contexts can be computed by using *dynamic programming*. ⊙

Illustrative Example

We illustrate the techniques by mapping the loop containing FFT butterfly operations. The butterfly operation consists of one complex multiply, one complex addition and one complex subtraction. First, the loop statements were decomposed into functions which can be executed on the CLU, given the list of functions in Table 1. One complex multiplication consists of four multiplications, one addition and one subtraction. Each complex addition and subtraction consist of two additions and subtractions respectively. The statements in the loop were mapped to multiplications, additions and subtractions which resulted in the task sequence T_m, T_m, T_m, T_m, T_a, T_s, T_a, T_a, T_s, T_s. Here, T_m is the multiplication task mapped to function F_1, T_a is the addition task mapped to function F_2 and T_s is the subtraction task mapped to function F_3.

Function	Operation	Configuration	Configuration	Execution
			Time	Time
F_1	Multiplication(Fast)	C_1	14.4 μs	37.5 ns
	Multiplication(Slow)	C_2	6.4 μs	52.5 ns
F_2	Addition	C_3	1.6 μs	7.5 ns
F_3	Subtraction	C_4	1.6 μs	7.5 ns
F_4	Shift	C_5	3.2 μs	7.5 ns

Figure 1: Representative Model Parameters for Garp Reconfigurable Architecture

The optimal sequence of configurations for this task sequence, using our algorithm, was C_1,C_3,C_4,C_3,C_4 repeated for all the iterations. The most important aspect of the solution is that the multiplier configuration in the solution is actually the slower configuration. The reconfiguration overhead is lower for C_2 and hence the higher execution cost is amortized over all the iterations of the loop. The total execution time is given by $N * 13.055$ μs where N is the number of iterations.

Conclusions

Mapping of applications in an architecture independent fashion can provide a framework for automatic compilation of applications. Loop structures with regular repetitive computations can be speeded-up by using configurable hardware. We developed dynamic programming based approaches to efficiently map tasks in a loop to a sequence of configurations. We illustrated our approach by developing algorithms for some variants of the mapping problem.

Military and Aerospace Applications of Programmable Devices and Technologies, September 1999.

Run-time Mapping of Graph-Problem Instances onto Reconfigurable Hardware*
(Summary)

Andreas Dandalis, Viktor K. Prasanna, and Jean-Luc Gaudiot

University of Southern California
{dandalis, prasanna, gaudiot}@ceng.usc.edu
Tel: +1-213-740-4483, Fax: +1-213-740-4418

The Problem

During the past few years, the rapid advances in fabrication technology has led to the development of programmable devices (e.g., FPGAs) with substantial computational power. As a result, reconfigurable hardware is being used beyond the initial applications of rapid prototyping and emulation into several areas of general purpose computation. Existing evidence suggests that using programmable devices for DoD applications will result in performance capabilities that are 2-3 orders of magnitude better than technologies currently being used [2].

Currently used military platforms are mainly based on ASICs to meet the real-time constraints and computational demands in battlefield environments. Reconfigurable Computing (i.e., computing using programmable hardware) can "outperform" ASICs by exploiting its ability to create hardware at runtime based on input parameters. If the logic remains static for all the instances of the problem, then an ASIC implementation would provide superior time performance. In addition, the essential struggle against time in the battlefield necessitates that the hardware adaptation has to be performed very fast. However, existing mapping techniques require extensive mapping time which is a major bottleneck in the case of any mapping that needs to be performed at runtime based upon the problem instance.

Existing mapping techniques for FPGAs have adopted the ASIC-based design flow and tools that prevent the Reconfigurable Computing paradigm from achieving its full potential: provide the performance benefits of ASICs and the flexibility of microprocessors. For one thing, current design compilation times are too long and preclude any run-time, dynamic modification of the configurations. For another, the characteristics of the application are not utilized, resulting in sub-optimal designs with respect to area and delay performance unless the designs are optimized by hand.

Our Approach to Run-time Mapping

Most of the mapping techniques proposed in the literature ignore the extensive overhead of the CAD tools at runtime. We believe however, that addressing this overhead is the key to fully exploit the Reconfigurable Computing advantages over ASICs and software based approaches.

Our approach to run-time mapping is to handle the mapping problem as an algorithm synthesis problem as opposed to "stuffing logic into a black box." Our key idea is to develop problem-specific configurations off-line to facilitate run-time mapping. These configurations are specific to the problem to be solved and are based on the algorithm that is used to solve the problem. At runtime, a mapping algorithm adapts the hardware to the input problem-instance. Our performance metric includes the time to compute the logic to be mapped, the time to configure the hardware, and the execution time on hardware.

The novelty of our approach is that the CAD tools bottleneck is alleviated from the critical path to the solution. The mapping process is driven by problem-specific configurations that are derived off-line. Thereby, there is no need for a complete redesign for each problem-instance. Equally important, the mapping process is aware of the characteristics of both the problem and the target architecture. Not only does the approach significantly speed up run-time mapping but also produces fast, compact logic reducing execution time as well. Preliminary results indicate that our approach can result in a speedup of at least two orders of magnitude comparing with the state-of-the-art.

A Case Study: Single-Source Shortest Path Problem

In our current efforts, we are focusing on mapping graph-problem instances onto multi-FPGA systems.

*This research was performed as part of the MAARC project (http://maarc.usc.edu). This work is supported by the DARPA Adaptive Computing Systems program under contract no. DABT63-96-C-0049 monitored by Fort Hauchuca.

Problem size # vertices x # edges	Clock rate (MHz)		Execution time (μsec)		Mapping time		Effective Speed-up
	[1]	Our solution	[1]	Our solution	[1] +	Our ++ solution	
16 x 64	1.79	15	8.94	21.42	~ 4 hours	~ 22 msec	6.5×10^6
64 x 256	1.14	15	56.14	79.02	~ 4 hours	~ 82 msec	1.7×10^6
128 x 515	0.78	15	164.10	199.72	~ 8 hours	~ 161 msec	1.8×10^6
256 x 1140	0.34	15	752.94	493.17	~16 hours	~ 319 msec	1.8×10^6

+ a cluster of 10 workstations was used

++ memory-array bandwidth 4MB/sec is assumed as in [1]

Table 1: Performance comparison with the state-of-the-art

Graph problems are the most frequently solved class of optimization problems (e.g., problems of heuristic search, deterministic optimal control problems, or data routing within a computer communication network).

In the state-of-the-art technique for solving graph problems using FPGAs [1], the input graph instance is embedded in the FPGAs by using general purpose CAD tools. A complete redesign is required for a new problem instance and the resulting implementation lacks modularity. Partitioning and place-and-route take several hours while the corresponding execution time on hardware is in the range of *μsec*. However, the mapping time is usually ignored and only the execution time is considered as runtime.

Besides the mapping overhead, the mapping of edges onto the physical wires of a device results in extremely slow clock rate and very high area requirements. The clock rate depends on the longest wire in the layout. As the number of vertices increases, the longest wire length increases rapidly resulting in fast degradation of the clock rate. Also, the area requirements depend on the connectivity of the input graph and increase rapidly for dense graph instances. Therefore, the clock rate and the area requirements cannot be reliably estimated before actually mapping onto hardware.

To illustrate the superiority of our ideas, we briefly describe a solution for the single-source shortest path problem using our approach and compare it against the state-of-the-art (based on CAD tools). Given a weighted, directed graph and a source vertex, the problem is to find a shortest path from the source to every other vertex.

A problem-specific configuration is developed based on the Bellman-Ford algorithm. The configuration corresponds to a general graph with n vertices and e edges and consists of n modules connected in a pipelined fashion. Each module corresponds to a vertex. At runtime, the problem-specific configuration is adapted to the characteristics of the input graph instance. At the module level, the precision of the functional units is adapted to the precision requirements and the number of vertices and edges of the input graph instance. Moreover, at the layout level, the number of the modules mapped onto hardware is determined by the number of vertices in the input instance. Finally, the clock speed is determined by the computational rate of the modules and the available I/O bandwidth. The resulting implementation is a modular design that can be easily adapted to any input instance without the need for complete redesign.

Our solution is asymptotically faster than the state-of-the-art [1]. The mapping time is six orders of magnitude smaller (see Table 1). As a result, the effective speed-up (e.g., considering both the execution and the mapping time) comparing with the solution in [1] is 10^6. Moreover, the clock speed only depends on the data precision of the input graph instance and not on the size and the connectivity of the graph instance as in [1]. Also, the hardware requirements increase as a linear function of the number of vertices. Consequently, the on-chip execution time and the area requirements can be accurately estimated based on the problem-specific configuration.

Conclusions

In this paper we demonstrated a case-study solution that achieves 6 orders of magnitude speedup over the state-of-the art for mapping graph-problem instances onto FPGAs. The novelty of our approach is that the mapping process performs an incremental adaptation of problem-specific configurations to the input problem instance instead of a complete redesign. Not only does the approach significantly speedup run-time mapping but also produces fast, compact logic which reduces the execution time on hardware as well.

Our approach can also be applied to other application domains (e.g., image and signal processing, cryptography) where adaptivity to problem instance is required. We believe that by addressing the run-time mapping problem, Reconfigurable Computing can become an attractive computing paradigm for specific military applications.

References

[1] J. Babb, M. Frank, and A. Agarwal, "Solving graph problems with dynamic computation structures", *SPIE '96: High-Speed Computing, Digital Signal Processing, and Filtering using Reconfigurable Logic*, 1996.

[2] DARPA ACS Program, http://www.darpa.mil/ito/research/acs/background.html

High-Performance Embedded Computing Workshop, September 1999

Managing Dynamic Precision on Reconfigurable Hardware [*] (Summary)

Kiran Bondalapati, Viktor K. Prasanna and Petros Ioannou
Department of Electrical Engineering Systems
University of Southern California
Los Angeles, CA 90089-2562

1 Introduction

In typical VLSI and processor based architectures, the computational units have fixed precision which can not be modified during computation. The precision of operands implemented in such architectures is based on the worst case bounds for the precision of the input values. Some applications also need precision much higher than that available in typical hardware architectures [3]. For such long-precision arithmetic, software algorithms are employed to obtain the desired precision. Long-precision computations typically operate digit by digit, serially. The iterative computations mean that the execution time of an operation increases as the required precision increases. Performing computations using the exact precision required for accurate results can reduce the resources utilized.

One of the significant advantages of reconfigurable hardware is the ability to perform variable precision computations [4]. Reconfigurable hardware contains fine-grained configurable resources which can be utilized to build computing modules of various sizes. For example, it is possible to build a standard 16-bit×16-bit multiplier or a 8-bit×12-bit multiplier using reconfigurable hardware. The 8-bit×12-bit multiplier would consume less area and execute faster than the standard 16-bit×16-bit multiplier. Reconfigurable architectures also support *dynamic precision*, which is the ability of the hardware to change its precision at run-time in response to variant precision demands of the algorithm.

In this paper we outline our framework for managing the dynamic precision variation. We represent the variation in the required precision for an operation by using a *precision variation curve*. The *precision variation curve* quantifies the variation in the required precision for an operation over time. The concept of time can represented by using various measures such as execution time, program counter, loop counter, etc.

In this paper we analyze the variation of precision in loop computations as the iterations of the loops progress. Compile-time and run-time techniques to determine the precision variation curve for a given computation are described. Various algorithmic techniques are developed for optimal mapping of the computations onto reconfigurable hardware [1, 2]. We illustrate the utility of our approach by demonstrating the performance improvement for an example operation.

2 Quantifying the Precision Variation

For iterative computations in which values are accumulated over the execution time of the application, the

[*] This work was supported by the DARPA Adaptive Computing Systems Program under contract DABT63-96-C-0049 monitored by Fort Hauchuca.

precision varies as the iterations progress. We represent this variation in terms of the loop iterations by using the *precision variation curve.*

2.1 Precision Variation Curve

The *precision variation curve* facilitates the representation of the notion of the variation in the precision of the operands and the operation as the execution of the loop progresses. A simple method to represent such a variation is to indicate the precision of the operand for each iteration so that the precision is defined for the complete iteration space. But, the precision usually varies very slowly as the iterations progress. Thus the *precision variation curve* can be represented by specifying the points where the precision of the operands or the operation changes.

Definition: The *precision variation curve* for a given operation or operand in a loop computation can be represented by the sequence (l_i, p_i), $1 \leq i \leq u$. l_i denotes the iteration number at which a change in precision takes place due to the computation. $l_i \leq N$ where N is the total number of iterations. p_i denotes the precision required for performing iterations l_i to $l_{i+1} - 1$ for $1 \leq i < u$ and p_u denotes the precision required for performing iterations l_u to N.

2.2 Compile-time Analysis of Loops

We can theoretically determine the *precision variation curve* for the operations in a given computation. The precision of computed variables in a loop is determined by the precision of the variables before the iteration, the number of iterations and the operations performed on the variable. For each type of arithmetic operation, the maximum possible precision of the result can be expressed using the above values. For example, the precision of an integer variable X (initially 0) after N iterations of a loop which contains the statement $X = X + C$ is bounded by

$$Pr(X) \leq Pr(C) + \lceil \log(N+1) \rceil$$

where $Pr(X)$ denotes the bit size of the variable X. The analysis is not limited to simple expressions, but extends to complex arithmetic expressions in loops. For recursive expressions in loops where the value of the variable X in iteration i is given by X_i, if

$$X_i = c_1 * X_{j_1} + c_2 * X_{j_2} + \ldots + c_k * X_{j_k} = \Sigma_{l=1}^{l=k} c_l * X_{j_l}$$

then the upper bound on the precision of X_i is given by

$$Pr(X_i) \leq (i-1) * \log C + (i-1) * \log k + Pr(X_1)$$

where $C = max[c_1, c_2, \ldots, c_k]$, the maximum of the constant coefficients. The analysis is valid for integer and fixed-point computations and is not necessarily valid for floating point computations. But, the analysis still covers a large class of signal and image processing applications.

2.3 Run-time Analysis

Theoretical analysis of expressions in loops computes the upper bounds on the precision of the variables and computations. This determines the minimum precision required to represent these variables. The estimates using theoretical analysis are conservative and can usually be much higher than the actual precision of the operands. For example, using the above analysis for the Fibonacci series $X_i = X_{i-1} + X_{i-2}$, we obtain $Pr(X_i) = i - 1$ and hence, $Pr(X_{15}) = 14$. But, $X_{15} = 610$ which needs only 10 bits. Even when a tight bound can be computed, the actual precision might be lower than theoretical estimate. This can occur when the data inputs are assumed to have maximum precision, but are actually randomly distributed over the complete input range.

Theoretical analysis can provide significant performance benefits which can be augmented by using profiling based analysis. For computations which do not have a tight bound on the precision and for computations with complex control flow, computing the required precision by using run-time statistics is a viable alternative. The application can be instrumented to measure the precision of the different variables and the knowledge can be utilized by the mapping tool or the compiler to identify the required precision at various program points.

3 Dynamic Precision Management

Given the *precision variation curve* for a loop, we need to determine the mapping of the iterations to a set of configurations which are used to execute the operations in the loop. For each iteration, the precision of the corresponding configuration which executes the iteration should be equal to or greater than the required precision for that iteration. The greedy strategy of reconfiguring the hardware whenever the required precision changes can result in significant reconfiguration overheads. For architectures in which the reconfiguration times are much higher than the execution times, the reconfiguration overhead might be prohibitive. Also, the set of configurations which are available for executing an operation might not encompass all the possible precision values that are required. Some of the operations will have to be executed with more precision than is necessary in the absence of configurations with the exact precision.

Thus, it is necessary to identify an optimal set of configurations which minimizes the overall execution cost, including the reconfiguration cost. We have developed efficient techniques to map application tasks onto available configurations using dynamic programming. Our algorithmic techniques consider the reconfiguration overheads in minimizing the total execution time for a given operation in a loop.

4 An Illustrative Example

We illustrate our approach by mapping the multiplication operation from the example code segment given below. We measured the total execution time for the $MAXQ * SCALE(I)$ computation on Xilinx XC6200 [5] using five different approaches. The first two approaches do not exploit the dynamic precision variation.

```
-----------------------------------
 DO 10 I=1,N
   DO 20 J=1,N
     RSQ(J) = RSQ(J)+XDIFF(I,J)*YDIFF(I,J)
20   IF (MAXQ.LT.RSQ(J)) THEN
       MAXQ = RSQ(J)
   POVERR = POVERR / MAXQ
10 VIRTXY = VIRTXY + MAXQ * SCALE(I)
------------------------------------
```

The execution times including the reconfiguration times are summarized in Table 1. The approaches using *dynamic precision* achieve significantly lower execution times compared to the fixed precision approaches. Our dynamic programming based algorithm(DPMA) executed all the iterations of the loop in the minimum time for the theoretical(DPMA) and run-time *precision variation curves*(DPMA-run). The resultant optimal schedules have up to 30% lower execution cost compared with other approaches.

5 Conclusions and Future Research

Reconfigurable hardware can be utilized to exploit the dynamic precision variation in applications. We have shown how the variable precision in computations can be captured by using the *precision variation curve*. The information obtained from the precision variation is used to develop optimal schedules for dynamic precision management. As illustrated using the example, reconfigurable hardware can provide significant benefits in application performance by using dynamic precision management.

Table 1: Execution times using different approaches

Algorithm	Execution Time (ns)	Reconfiguration Time (ns)	Total (ns)
Standard	655360	20480	675840
Static	532480	17920	550400
Greedy	468010	56320	524330
DPMA	471160	33280	504440
DPMA-run	409600	15360	424960

The reduction of the resources by using dynamic precision can be utilized to achieve higher speed-up by realizing more parallelization and pipelining of the application. The given analysis represents the variation and optimization for a single operation in a loop. The application of such techniques for multiple operations and generic programs in addition to loops is under investigation.

References

[1] K. Bondalapati and V.K. Prasanna. Mapping Loops onto Reconfigurable Architectures. In *8th International Workshop on Field-Programmable Logic and Applications*, September 1998.

[2] K. Bondalapati and V.K. Prasanna. Dynamic Precision Management for Loop Computations on Reconfigurable Architectures. In *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1999.

[3] A.F. Tenca and M.D. Ercegovac. A Variable Long-Precision Arithmetic Unit Design for Reconfigurable Coprocessor Architectures. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.

[4] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable Active Memories: Reconfigurable Systems Come of Age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, March 1996.

[5] Xilinx. XC6200 Field Programmable Gate Arrays, 1996.

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;

/**
 * DRIVE: Dynamically Reconfigurable Systems Interpretive Simulation
 * and Visualization Environment.
 * The main framework class which maintains the different components being
 * shown in the visualizer. Uses Mylistener to invoke actions for the menus.
 * The visualizer maintains many internal classes for different visual
 * components which are not accessible externally. It does not provide any
 * methods since it is the controller entity. The only public methods are
 * for displaying messages in the Log window. Drive can be instantiated by
 * another class and messages can be displayed using these methods.
 *
 * For latest information see <A HREF="http://maarc.usc.edu/">
 *
 * @author Kiran Bondalapati
 * @version 2.0 1999
 * @see Hysam
 * @see Device
 * @see Scheduler
 * @see EventList
 * @see Event
 * @see CLU
 */
public class Drive extends Frame {

  Hysam myHysam;
  Device myDevice;

  CLUWin myCluWin;
  Scheduler mySchedule;

  private int params_loaded;
  private int appl_loaded;
  private int sched_loaded;

  /** The font style */
  static final String fontName = "SansSerif";
  static final int fontStyle = Font.BOLD;
  static final int fontSize = 12;

  static final int driveWidth = 1200;
  static final int driveHeight = 700;

  private int cluWidth = 500;
  private int cluHeight = 420;

  static final int xoffset = 10;
  static final int yoffset = 10;

  /* the time progress bar defaults */
  static final int progressWidth = 400;
  static final int progressHeight = 20;

  Panel tpanel;
  Panel bpanel;
  Panel cpanel;
  Panel ipanel;
  Panel lpanel;

  static final String timeFontName = "SansSerif";
  static final int timeFontStyle = Font.BOLD;
static final int timeFontSize = 12;

Label timeTitle;
Label timeSpace;
Label timeCurrentLabel;
Label timeStartLabel;
Canvas timeProgressWrap;
ProgressBar timeProgress;
Label timeFinishLabel;

Label funcLabel;
Label funcIdLabel;
Label funcNameLabel;
Label confLabel;
Label confIdLabel;
Label confNameLabel;
Label bwLabel;
Label bwValueLabel;

TextArea logText;

MyMenuBar menuBar;
EventHandler eh = new EventHandler();
MessageDialog dialog;
FileDialog fdialog;
DialogHandler dh = new DialogHandler();

public static void main(String args[]){

  Drive tool = new Drive();
}

public Drive() {

  super("DRIVE 2.0");
  this.setTitle("DRIVE 2.0");
  setTitle("DRIVE 2.0");

  myHysam = new Hysam();
  myDevice = new Device();
  myDevice.resetCLU();

  setFont(new Font(fontName, fontStyle, fontSize));

  setupMenuBar();
  setupWidgets();

}

void setupWidgets() {

  GridBagLayout driveLay = new GridBagLayout();
  GridBagConstraints driveConst = new GridBagConstraints();

  setLayout(driveLay);

  tpanel = new Panel();
  bpanel = new Panel();
  cpanel = new Panel();
  ipanel = new Panel();
  lpanel = new Panel();

  //    driveConst.weightx = 1.0;
  //    driveConst.weighty = 0.1;
  driveConst.anchor = GridBagConstraints.NORTHWEST;

  driveConst.gridwidth = GridBagConstraints.REMAINDER;
  driveLay.setConstraints(tpanel, driveConst);
  add(tpanel);


  driveConst.anchor = GridBagConstraints.NORTHWEST;
  driveConst.gridx = 0;
  driveConst.gridy = 2;
  driveConst.gridwidth = GridBagConstraints.RELATIVE;
  //    driveConst.weightx = 0.75;
  //    driveConst.weighty = 1.0;
  driveLay.setConstraints(cpanel, driveConst);
  add(cpanel);

  driveConst.gridx = 1;
  driveConst.gridy = 2;
  //    driveConst.weightx = 1.0;
  //    driveConst.weighty = 1.0;
  driveConst.anchor = GridBagConstraints.NORTH;
  driveConst.gridwidth = GridBagConstraints.REMAINDER;
  driveLay.setConstraints(ipanel, driveConst);
  add(ipanel);

  driveConst.anchor = GridBagConstraints.NORTHWEST;
  driveConst.gridx = 0;
  driveConst.gridy = 3;
  //    driveConst.weightx = 1;
  //    driveConst.weighty = 0.5;
  driveLay.setConstraints(lpanel, driveConst);
  add(lpanel);

  //    setupButtons();
  setupTimeWidgets();
  setupCluWidgets();
  setupInfoWidgets();
  setupLogWidgets();

  setSize(driveWidth,driveHeight);
  addWindowListener(eh);

  pack();
  show();

}

/* Set up the shortcut buttons
void setupButtons() {
  bpanel.setFont(new Font(buttonFontName, buttonFontStyle, buttonFontSize));
  loadParamButton = new Button("Step");
  loadParamButton = new Button("Run");
}
*/

/* Set up the time panel */
void setupTimeWidgets() {

  tpanel.setLayout(new FlowLayout(FlowLayout.LEFT));

  tpanel.setFont(new Font(timeFontName, timeFontStyle, timeFontSize));

  timeTitle = new Label("Time: ");
  timeCurrentLabel = new Label("0.000");
```

```
    timeCurrentLabel.setSize(100,20);
    timeSpace = new Label("    ");
    timeStartLabel = new Label("0.000");
    timeStartLabel.setSize(100,20);
    timeProgress = new ProgressBar(200,0,progressWidth,progressHeight);
    timeFinishLabel = new Label("0.000");
    timeFinishLabel.setSize(100,20);

    tpanel.add(timeTitle);
    tpanel.add(timeCurrentLabel);
    tpanel.add(timeSpace);
    tpanel.add(timeStartLabel);
    tpanel.add(timeProgress);
    tpanel.add(timeFinishLabel);

    tpanel.setVisible(true);
    timeProgress.repaint();
  }

  /* Set up the CLU Windows */
  void setupCluWidgets() {

    cpanel.setLayout(new BorderLayout());

    Label cluLabel = new Label(" CLU");
    cpanel.add("North", cluLabel);

    myCluWin = new CLUWin(myDevice.getCLURows(),myDevice.getCLUCols());

    cluHeight = myDevice.getCLURows() * (CLUWin.cell_size+CLUWin.cell_space) + 2 * CLUWin.
yoffset + 2*CLUWin.cell_size;
    cluWidth = myDevice.getCLUCols() * (CLUWin.cell_size+CLUWin.cell_space) + 2 * CLUWin.x
offset;

    myCluWin.setSize(cluWidth, cluHeight);
    cpanel.setSize(cluHeight+ 2*CLUWin.yoffset, cluWidth+2*CLUWin.xoffset);
    cpanel.add("South", myCluWin);

    cpanel.setVisible(true);

  }

  /* Set up the windows for the information widgets */
  void setupInfoWidgets() {

    GridBagLayout infoLay = new GridBagLayout();
    GridBagConstraints infoC = new GridBagConstraints();

    ipanel.setSize(600,400);
    ipanel.setLayout(infoLay);

    infoC.fill = GridBagConstraints.NONE;
    infoC.anchor = GridBagConstraints.NORTHWEST;

    funcLabel = new Label("Function: ");
    infoLay.setConstraints(funcLabel, infoC);
    ipanel.add(funcLabel);

    infoC.gridx = 1;
    funcIdLabel = new Label("0");
    infoLay.setConstraints(funcIdLabel, infoC);
    ipanel.add(funcIdLabel);

    infoC.gridx = 2;
    infoC.gridwidth = GridBagConstraints.REMAINDER;
    funcNameLabel = new Label("                    ");
    infoLay.setConstraints(funcNameLabel, infoC);
    ipanel.add(funcNameLabel);

    infoC.gridy = 1;

    infoC.gridx = 0;
    infoC.gridwidth = GridBagConstraints.RELATIVE;
    confLabel = new Label("Configuration: ");
    infoLay.setConstraints(confLabel, infoC);
    ipanel.add(confLabel);

    infoC.gridx = 2;
    confIdLabel = new Label("0   ");
    infoLay.setConstraints(confIdLabel, infoC);
    ipanel.add(confIdLabel);

    infoC.gridx = 4;
    infoC.gridwidth = GridBagConstraints.REMAINDER;
    confNameLabel = new Label("Initial           ");
    infoLay.setConstraints(confNameLabel, infoC);
    ipanel.add(confNameLabel);

    infoC.gridy = 2;

    infoC.gridx = 0;
    infoC.gridwidth = GridBagConstraints.RELATIVE;
    bwLabel = new Label("Bandwidth: ");
    infoLay.setConstraints(bwLabel, infoC);
    ipanel.add(bwLabel);

    infoC.gridx = 2;
    infoC.gridwidth = GridBagConstraints.REMAINDER;
    bwValueLabel = new Label("32   ");
    infoLay.setConstraints(bwValueLabel, infoC);
    ipanel.add(bwValueLabel);

  }

  /* Set up the windows for the log widgets */

  void setupLogWidgets() {

    GridBagLayout logLayout = new GridBagLayout();
    GridBagConstraints logC = new GridBagConstraints();

    lpanel.setLayout(logLayout);

    logC.fill = GridBagConstraints.BOTH;

    logC.gridwidth = GridBagConstraints.REMAINDER;
    logC.gridheight = 1;

    Label logTitle = new Label("Log");
    logLayout.setConstraints(logTitle, logC);
    lpanel.add(logTitle);

    logC.gridx = 0;
    logC.gridy = 1;

    logC.weightx = 0.0;
    logText = new TextArea("******* DRIVE 2.0 Log ********\n",7,80,TextArea.SCROLLBARS_BOT
H);
```

```
    logLayout.setConstraints(logText, logC);
    lpanel.add(logText);


  }

  void setupMenuBar(){
    String computemenu[] = {"Compute","Linear","~Precision"};
    Object menuItems[][] = {{"File","Load Parameters","Load Application","Load Configurati
on","Save Configuration","Exit"},
                            {"Edit","Parameters","Application"},
                            {"Schedule","Load Schedule",computemenu},
                            {"Simulate","~Step","~Run","~Reset"},
                            {"Help","About DRIVE"}
    };

    menuBar = new MyMenuBar(menuItems,eh,eh);
    setMenuBar(menuBar);
  }

  /** paint routine which calls CLU and time progress bar paints to make sure
      they draw themselves */
  public void paint(Graphics g) {
    myCluWin.repaint();
    timeProgress.repaint();
  }

  /** Used in the menu bar selections */
  class EventHandler extends WindowAdapter implements ActionListener,
                                                      ItemListener {
    public void actionPerformed(ActionEvent e){

      String selection=e.getActionCommand();

      if ("Load Parameters".equals(selection)) {
        cmd_load_params();
      } else if ("Load Application".equals(selection)) {
        cmd_load_appl();
      } else if ("Exit".equals(selection)){
        System.exit(0);
      } else if ("Linear".equals(selection)) {
        cmd_compute_linear();
      } else if ("Step".equals(selection)) {
        cmd_step_simul();
      } else if ("Run".equals(selection)) {
        cmd_run_simul();
      } else if ("Reset".equals(selection)) {
        cmd_reset_simul();
      } else if ("About DRIVE".equals(selection)) {
        cmd_about();
      }

    }


    public void itemStateChanged(ItemEvent e){
    }

    public void windowClosing(WindowEvent e){
      System.exit(0);
    }

  }
```

```
void cmd_load_params() {

  fdialog = new FileDialog(Drive.this, "Model Parameters File",fdialog.LOAD);

  fdialog.show();

  String filename = fdialog.getFile();

  System.out.print(filename+"\n");

  try {

    StreamTokenizer pstream = new StreamTokenizer(new FileReader(filename));
    pstream.commentChar('#');
    pstream.eolIsSignificant(false);

    if (myHysam.readParams(pstream) == 1) {
      params_loaded = 1;
      sched_loaded = 0;

      logText.append("Paramaters loaded from file: "+filename+"\n");

    }
  } catch (Exception IOException) {
    logText.append("Error: Opening parameters file: "+filename+"\n");
  }

  if ( (params_loaded == 1) && (appl_loaded == 1)) {
    menuBar.getMenu("Schedule").getItem("Linear").setEnabled(true);
  }

}

void cmd_load_appl() {

  int type =1;
  int result;
  int token;

  fdialog = new FileDialog(Drive.this, "Application Tasks File",fdialog.LOAD);

  fdialog.show();

  String filename = fdialog.getFile();

  try {
    StreamTokenizer astream = new StreamTokenizer(new FileReader(filename));

    token = astream.nextToken();
    type = (int)astream.nval;

    result = myHysam.readAppl(type,astream);

    if (result > 0) {
      appl_loaded = 1;
      sched_loaded = 0;

      logText.append("Application loaded from file: "+filename+"\n");

    }
    else {
      errorMessage(1, "Application file: "+filename+" has errors.\n");
    }
  } catch (Exception IOException) {
    errorMessage(1,"Error: Could not open application file: "+filename+"\n");
  }

  /*
  if ((appl_loaded == 1) && (params_loaded == 1)) {
    if (type == 1) {
      menuBar.getMenu("Schedule").getItem("Linear").setEnabled(true);
      menuBar.getMenu("Schedule").getItem("Precision").setEnabled(true);
    }
    else if (type == 2) {
      menuBar.getMenu("Schedule").getItem("Linear").setEnabled(false);
      menuBar.getMenu("Schedule").getItem("Precision").setEnabled(true);
    }
  }
  */

}

/* Computes the linear schedule and gets the value of mySchedule */
void cmd_compute_linear() {

  if ((params_loaded == 1) && (appl_loaded == 1)) {

    mySchedule = myHysam.computeSchedule(1);

    if (mySchedule != null ) {
      sched_loaded = 1;

      logMessage("Computed schedule\n");

      menuBar.getMenu("Simulate").getItem("Step").setEnabled(true);
      menuBar.getMenu("Simulate").getItem("Run").setEnabled(true);
      menuBar.getMenu("Simulate").getItem("Reset").setEnabled(true);
    }
    else {
      errorMessage(1,"Error in computing schedule\n");
    }
  }
  else {
    /* flag error message saying load params and appl first */

    errorMessage(1,"Error: Parameters or Application not loaded\n");
  }
}

/* Reset the simulation */
void cmd_reset_simul() {
  if (sched_loaded == 0) {
    errorMessage(1,"Schedule not computed or loaded\n");
  }
  else {

    setFuncId("0");
    setFuncName("");
    setConfId("0");
    setConfName("Initial");
    mySchedule.reset();
    myCluWin.reset();
    resetTime();
    logMessage("Simulation Reset\n");
  }
}

/* Run the simulation */
```

```
void cmd_run_simul() {
  if (sched_loaded == 0) {
    errorMessage(1,"Schedule not computed or loaded\n");
  }
  else {
    logMessage("Running Simulation\n");
    setFinishTime( (new Float(mySchedule.getFinishTime())).toString() );

    Event ev = mySchedule.getNextEvent();

    while (ev != null) {
      show_one_step(ev);

      try {
        wait(1000000,0);
      } catch (Exception InterruptedException) {
      }

      ev = mySchedule.getNextEvent();
    }
  }
}

/* Step through one event in the simulation */
void cmd_step_simul() {
  if (sched_loaded == 0) {
    errorMessage(1,"Schedule not computed or loaded\n");
  }
  else {

    setFinishTime( (new Float(mySchedule.getFinishTime())).toString() );

    Event ev = mySchedule.getNextEvent();

    if (ev!= null)
      show_one_step(ev);
  }
}

void show_one_step(Event ev) {
  if (ev.getType() == Scheduler.EXECUTE) {

    logMessage("Execute " +ev.getId1()+ " in " + ev.getId2()+"\n");

    int fid = ev.getId1();
    setFuncId( (new Integer(fid)).toString() );
    setFuncName( myHysam.getFuncName(fid) );

    int cid = ev.getId2();
    setConfId( (new Integer(cid)).toString() );
    setConfName( myHysam.getConfName(cid));

    setCurrentTime( (new Float(ev.getStartTime())).toString() );
    setTimeProgress(ev.getStartTime() / mySchedule.getFinishTime());

    ConfigBit[] cbits = myHysam.getConfig(cid);

    if (cbits != null) {
        System.out.print("numbits in conf "+cid+" is "+cbits.length+"\n");

        for(int i=0; i< cbits.length; i++) {
            cbits[i].setState(ConfigBit.ACTIVE);
        }
        myDevice.updateCLUConfig(cbits);
```

```
          myCluWin.updateDisplay(cbits);
      }
      else {
          System.out.print("Not able to access data for conf "+cid+"\n");
          logMessage("Could not find configuration data for C"+cid+"\n");
      }

    } else if (ev.getType() == Scheduler.RECONFIG) {

      logMessage("Reconfig from " +ev.getId1()+ " to " + ev.getId2()+"\n");

      setFuncId("0");
      setFuncName("");
      int cid1 = ev.getId1();
      int cid2 = ev.getId2();
      setConfId( (new Integer(cid1)).toString()
                   + " -> "
                 + (new Integer(cid2)).toString()
                 );
      setConfName( myHysam.getConfName(cid1)
                   + " -> "
                 + myHysam.getConfName(cid2)
                 );

      setCurrentTime( (new Float(ev.getStartTime())).toString() );
      setTimeProgress(ev.getStartTime() / mySchedule.getFinishTime());

      ConfigBit[] cbits = myHysam.getConfig(cid2);

      if (cbits != null) {
          System.out.print("numbits in conf "+cid2+" is "+cbits.length+"\n");

          for(int i=0; i< cbits.length; i++) {
              cbits[i].setState(ConfigBit.RECONFIG);
          }
          myDevice.updateCLUConfig(cbits);
          myCluWin.updateDisplay(cbits);
      }
      else {
          System.out.print("Not able to access data for conf "+cid2+"\n");
          logMessage("Could not find configuration data for C"+cid2+"\n");
      }
    }

    setCurrentTime( (new Float(ev.getFinishTime())).toString() );
    setTimeProgress(ev.getFinishTime() / mySchedule.getFinishTime());

  }

  void cmd_about() {

    String about[] = {" ",
                      "DRIVE: Dynamically Reconfigurable-systems Interpretive-simulation a
nd Visualization Environment",
                      "Version 2.0",
                      "Kiran Bondalapati",
                      "University of Southern California",
                      "Copyright (c) 1999."
    };
    String buttons[] = {"OK"};

    dialog = new MessageDialog(Drive.this,"About DRIVE",false,about,buttons,dh,dh);
    dialog.setLocation(200,200);
    dialog.show();
```

```
}

class DialogHandler extends WindowAdapter implements ActionListener {

  public void windowClosing(WindowEvent e) {
    Drive.this.show();
    dialog.dispose();
  }

  public void actionPerformed(ActionEvent e) {
    Drive.this.show();
    dialog.dispose();
  }
}


/** Sets the label value for Function ID */
void setFuncId(String func) {
  funcIdLabel.setText(func);
}

/** Sets the label value for Function Name */
void setFuncName(String func) {
  funcNameLabel.setText(func);
}

/** Sets the label value for Configuration ID */
void setConfId(String conf) {
  confIdLabel.setText(conf);
}

/** Sets the label value for Configuration Name */
void setConfName(String conf) {
  confNameLabel.setText(conf);
}

/** Sets the label value for current time */
void setCurrentTime(String time) {
  timeCurrentLabel.setText(time);
}

/** Sets the label value for finish time */
void setFinishTime(String time) {
  timeFinishLabel.setText(time);
}

/** Updates the time progress bar */
void setTimeProgress(float percent) {
  timeProgress.setPercent(percent);
}

/** Resets the time texts and bar displays.
 */
void resetTime() {
  setTimeProgress(0);
  if (mySchedule != null) {
    setFinishTime( (new Float(mySchedule.getFinishTime())).toString() );
  }
  else
    setFinishTime("0.000");
  setCurrentTime("0.000");
}

/** Displays error message.
```

```
      The first parameters denotes the severity of the error and can be used
      to filter messages.
      Second parameter is the error message. */
  public void errorMessage(int level, String message) {
    logText.append("Error["+level+"]: "+message);
  }

  /** Displays log message. */
  public void logMessage(String message) {
    logText.append(message);
  }

}
```

```
/*
 * @(#)Application.java
 *
 */

import java.io.*;
import java.util.*;

/**
 * The class Application implements the Application Tasks and supports
 * read/write and other access functions.
 *
 * @author Kiran Bondalapati
 * @see Hysam
 * @see Configuration
 * @see Function
 */
public class Application {

  private int id;         /* Id of the task */
  private int type;       /* type of application tasks */
  private int num_tasks;  /* Number of tasks in the list */
  private int[] taskid;   /* Id number of the task read from input */
  private int[] list;     /* List of function Ids or Iteration numbers */
  private int[] precision;/* precision values for Precision problem */

  /** Reads the application tasks as a linear sequence from the file
      handle. File has next value as the number of tasks.
      Returns 0 on error, #tasks on success */
  public int readLinear(StreamTokenizer lst) {
    int tok;
    try {
      tok = lst.nextToken();
      num_tasks = (int)lst.nval;

      taskid = new int[num_tasks];
      list = new int[num_tasks];

      for(int i=0; i<num_tasks; i++) {
        tok = lst.nextToken();
        taskid[i] = (int)lst.nval;
        tok = lst.nextToken();
        list[i] = (int)lst.nval;
      }

      System.out.print("Read Tasks: "+num_tasks+"\n");

      return num_tasks;

    } catch (Exception IOException) {
      return 0;
    }
  }

  /** Reads the application tasks as precision curve from the file
      handle. File has next value as the number of tasks.
      Returns 0 on error, #tasks on success */
  public int readPrecision(StreamTokenizer pst) {
    int tok;
    try {
      tok = pst.nextToken();
      num_tasks = (int)pst.nval;

      list = new int[num_tasks];
      precision = new int[num_tasks];

      for(int i=0; i<num_tasks; i++) {
        tok = pst.nextToken();
        list[i] = (int)pst.nval;
        tok = pst.nextToken();
        precision[i] = (int)pst.nval;
      }
      return num_tasks;

    } catch (Exception IOException) {
      return 0;
    }
  }

  /** Returns number of tasks in the task list */
  public int getNumTasks() {
    return num_tasks;
  }

  /** Returns the index'th element. Returns 0 on out of range index */
  public int getFuncId(int index) {
    if ((0 <= index) && (index < num_tasks))
      return list[index];
    else
      return 0;
  }

}
```

```java
/*
 * @(#)Attributes.java
 *
 */

import java.io.*;

/**
 * The class Attributes contains the properties when one Function is
 * implemented in one configuration.
 *
 * @author Kiran Bondalapati
 * @see Hysam
 * @see Configuration
 * @see Function
 */
public class Attributes {

  private int numA;
  private Map[] attr;

  /** Constructor */
  public Attributes(int num_attr) {

    numA = num_attr;
    //    attr = new Map[numA];
  }

  public Attributes() {
    numA = 0;
  }

  /** Internal class which stores mapping of attributes for one function
      in one configurations. Attributes is an array of Maps */
  private class Map {
    private int func_id;
    private int conf_id;         /* Id of the function */
    private float extime;
    private int seqinp;
    private int parinp;
    private int[] precision;

    public int readData(StreamTokenizer ast) {

      try {

        int tok = ast.nextToken();
        func_id = (int)ast.nval;
        tok = ast.nextToken();
        conf_id = (int)ast.nval;

        System.out.print("\nReading Attr for func ");
        System.out.print(func_id);
        System.out.print(" config ");
        System.out.print(conf_id);

        tok = ast.nextToken();
        extime = (float)ast.nval;
        tok = ast.nextToken();
        seqinp = (int)ast.nval;
        tok = ast.nextToken();
        parinp = (int)ast.nval;

        precision = new int[parinp];
      for(int i=0; i<parinp; i++) {
        tok = ast.nextToken();
        precision[i] = (int)ast.nval;
      }
      return 1;
    } catch (Exception IOException) {
      return 0;
    }
  }

  /** Returns the Function Id */
  public int getFuncId() {
    return func_id;
  }

  /** Returns the Configuration Id */
  public int getConfId() {
    return conf_id;
  }

  /** Returns the execution time */
  public float getExTime() {
    return extime;
  }
}

/** Initialize the number of attributes to num */
public int setNum(int num) {
  numA = num;
  return numA;
}

/** Returns the number of attributes stored in the matrix */
public int getNum() {
  return numA;
}

/** Reads the attributes data from the file */
public int readData(StreamTokenizer stream) {

  try {

    int tok = stream.nextToken();
    numA = (int)stream.nval;

    attr = new Map[numA];

    for(int i=0; i<numA; i++) {
      attr[i] = new Map();
      attr[i].readData(stream);
    }
    return 1;
  } catch (Exception IOException) {
    return 0;
  }
}

/** Returns the execution time of a given function in a given configuration.
    Returns Hysam.INFINITY if there is no stored value */
public float getExecCost(int fid, int cid) {

  for(int i=0; i<numA; i++) {
    if ((attr[i].getFuncId() == fid) && (attr[i].getConfId() == cid)) {
        return attr[i].getExTime();
      }
    }

    return Hysam.INFINITY;
  }


}
```

```
/*
 * @(#)CLU.java
 *
 */

import java.io.*;

/**
 * The class CLU implements the CLU in the HySAM model.
 * Initialized with row and column sizes. The default size is 32 x 32.
 * Warning: Resizing the CLU results in loss of previous state as the
 * configuration matrix is reallocated and not copied.
 *
 * @author Kiran Bondalapati
 * @see Hysam
 * @see Configuration
 * @see Function
 * @see Attributes
 * @see ConfigBit
 */
public class CLU {

  private int status;
  private int rows;
  private int columns;
  private ConfigBit[][] config;  /* stores the config bits for the CLU */

  /** Constructor which initializes the rows and columns and allocates
      the memory for the cells. */
  public CLU(int r, int c) {
    rows = r;
    columns = c;

    config = new ConfigBit[r][c];

    for(int i=0; i<rows; i++) {
      for(int j=0; j<columns; j++) {
        config[i][j] = new ConfigBit(i,j,0,ConfigBit.INIT);
      }
    }

    System.gc();
  }

  /** Constructor which initializes to the default rows and columns of 32. */
  public CLU () {

    rows = 32;
    columns = 32;

    config = new ConfigBit[32][32];

    for(int i=0; i<rows; i++) {
      for(int j=0; j<columns; j++) {
        config[i][j] = new ConfigBit(i,j,0,ConfigBit.INIT);
      }
    }

  }


  /** Initializes all the configuration bits of the CLU to 0 */
  public void reset() {
    for (int i=0; i<rows; i++)
    for (int j=0; j<columns; j++) {
      config[i][j] = new ConfigBit(i,j,0,ConfigBit.INIT);
    }
}

/** Reads the configuration matrix from a file.
    The first two numbers in the file are the rows and columns.
    The remaining rows*columns numbers are the configuration bit
    values for each cell. They are formatted as per ConfigBit class. */
public int readData(StreamTokenizer bstream) {

  int tok;

  try {
    tok = bstream.nextToken();
    rows = (int)bstream.nval;
    tok = bstream.nextToken();
    columns = (int)bstream.nval;

    config = new ConfigBit[rows][columns];

    for(int i=0; i< rows; i++) {
      for(int j=0; j<columns; j++) {
        config[i][j] = new ConfigBit();
        config[i][j].readData(bstream);
      }
    }
    return 1;
  } catch (Exception IOException) {
    return 0;
  }
}

/** Writes the configuration matrix to a file. (TO BE IMPLEMENTED).
    The first two numbers in the file are the rows and columns.
    The remaining rows*columns numbers are the configuration bit values
    and state for each cell */
public int writeData() {
  try {
    return 1;
  } catch (Exception IOException) {
    return 0;
  }
}

/** Sets the number of rows and columns in CLU */
public void setSize(int row, int col) {
  rows = row;
  columns = col;

  config = new ConfigBit[rows][columns];

}

/** Returns the number of rows in CLU */
public int getRows() {
  return rows;
}

/** Returns the number of columns in CLU */
public int getCols() {
  return columns;
}
```

```
  /** Sets configuration and state data of a CLU cell.
      Returns the previous configuration.
      If the row and column are out of range returns -1. */
  public int setCellConfig(int row, int col, int cfg, int st) {
    if ((row <= rows) && (col <= columns)) {
      int tmp = config[row][col].getBit();
      config[row][col].setValue(row,col,cfg,st);
      return tmp;
    }
    else
      return -1;
  }

  /** Returns configuration data of a CLU cell.
      Returns -1 for out of range cells. */
  public int getCellConfig(int row, int col) {
    if ((row <= rows) && (col <= columns))
      return config[row][col].getBit();
    else
      return -1;
  }

  /** Sets configuration and state data of the complete CLU. */
  public int updateConfig(ConfigBit[] cfg) {

    int num = cfg.length;

    for (int i=0; i<num; i++) {
      config[cfg[i].getRow()][cfg[i].getColumn()].setValue(cfg[i].getRow(),
                                                           cfg[i].getColumn(),
                                                           cfg[i].getBit(),
                                                           cfg[i].getState()
                                                           );
    }

    return 1;
  }

  /** Returns the configuration data of the complete CLU. */
  public ConfigBit[][] getConfig() {

    return config;
  }

  /** Returns state data of a CLU cell.
      Returns -1 for out of range cells. */
  public int getState(int row, int col) {
    if ((row <= rows) && (col <= columns))
      return config[row][col].getState();
    else
      return -1;
  }
}
```

```
/*
 * @(#)CLUWin.java
 *
 */

import java.io.*;
import java.awt.*;

/**
 * The class CLUWin implements the CLU display in DRIVE.
 * Initialized with row and column sizes. The default size is 32 x 32.
 *
 * @author Kiran Bondalapati
 * @see Drive
 * @see Device
 * @see Hysam
 * @see Configuration
 * @see ConfigBit
 */
public class CLUWin extends Canvas {

  private int rows;
  private int columns;

  ConfigBit[][] config;

  /* Some colors for the CLU */

  static final Color cluCellColor = Color.black;
  static final Color cluInitColor = Color.gray;
  static final Color cluActiveColor = Color.red;
  static final Color cluReconfColor = Color.green;

  /* The graphics constants */

  private int width = 500;
  private int height = 420;
  static final int cell_size = 10;
  static final int cell_space = 2;
  static final int xoffset = 10;
  static final int yoffset = 10;

  private Graphics cluGraphics;
  private Image cluImage;

  /** Default constructor which has 32 rows and 32 columns.
      Initializes the cells to ConfigBit.INIT state.
  */
  public CLUWin() {
    rows = 32;
    columns = 32;

    config = new ConfigBit[rows][columns];

    for(int i=0; i<rows; i++) {
      for (int j=0; j<columns; j++) {
        config[i][j] = new ConfigBit(i,j,0,ConfigBit.INIT);
      }
    }

    repaint();

  }
 /** Constructor which initializes the number of rows and columns.
   @param rows integer number of rows
   @param cols integer number of columns
 */
 public CLUWin(int r, int c) {
  rows = r;
  columns = c;

  config = new ConfigBit[rows][columns];

  for(int i=0; i<rows; i++) {
   for (int j=0; j<columns; j++) {
    config[i][j] = new ConfigBit(i,j,0,ConfigBit.INIT);
   }
  }

  repaint();

 }

 public void paint(Graphics g) {

  if (cluGraphics == null) {
   cluImage = createImage(width, height);
   cluGraphics = cluImage.getGraphics();
  }

  cluGraphics.clearRect(0,0,width, height);

  for(int i=0; i<rows; i++) {
   for(int j=0; j<columns; j++) {

    if (config[i][j].getState() == ConfigBit.INIT) {
     cluGraphics.setColor(cluCellColor);
     cluGraphics.fillRect(i*(cell_size+cell_space)+yoffset, j*(cell_size+cell_space)+
xoffset, cell_size, cell_size);

     cluGraphics.setColor(cluInitColor);
     cluGraphics.drawRect(i*(cell_size+cell_space)+yoffset, j*(cell_size+cell_space)+
xoffset, cell_size, cell_size);
    }
    else if (config[i][j].getState() == ConfigBit.ACTIVE) {
     cluGraphics.setColor(cluCellColor);
     cluGraphics.fillRect(i*(cell_size+cell_space)+yoffset, j*(cell_size+cell_space)+
xoffset, cell_size, cell_size);

     cluGraphics.setColor(cluActiveColor);
     cluGraphics.drawRect(i*(cell_size+cell_space)+yoffset, j*(cell_size+cell_space)+
xoffset, cell_size, cell_size);
    }
    else if (config[i][j].getState() == ConfigBit.RECONFIG) {
     cluGraphics.setColor(cluCellColor);
     cluGraphics.fillRect(i*(cell_size+cell_space)+yoffset, j*(cell_size+cell_space)+
xoffset, cell_size, cell_size);

     cluGraphics.setColor(cluReconfColor);
     cluGraphics.drawRect(i*(cell_size+cell_space)+yoffset, j*(cell_size+cell_space)+
xoffset, cell_size, cell_size);
    }
   }
  }

  cluGraphics.setColor(cluInitColor);
  cluGraphics.drawRect(23,403,cell_size, cell_size);
```

```
    cluGraphics.setColor(cluActiveColor);
    cluGraphics.drawRect(120,403,cell_size, cell_size);

    cluGraphics.setColor(cluReconfColor);
    cluGraphics.drawRect(220,403,cell_size, cell_size);

    cluGraphics.setColor(Color.black);
    cluGraphics.drawString("Inactive", 45, 413);
    cluGraphics.drawString("Active", 140, 413);
    cluGraphics.drawString("Reconf", 240, 413);

    g.drawImage(cluImage, 0, 0, null);

  }

  /** Sets the complete CLU display to the matrix of configuration info that
      is passed.
      @param cfg is a matrix of configurations for the full CLU.
  */
  public void setDisplay(ConfigBit[][] cfg) {

    config = cfg;

    repaint();
  }

  /** Resets the display to nothing loaded.
   */
  public void reset() {

      config = new ConfigBit[rows][columns];

      for(int i=0; i<rows; i++) {
          for (int j=0; j<columns; j++) {
              config[i][j] = new ConfigBit(i,j,0,ConfigBit.INIT);
          }
      }

    repaint();
    }

  /** Updates the CLU display with the new configuration information.
      Only the updated info is passed and previous cells retain the old
      configuration.
      @param cfg is an array of ConfigBit stream.
  */
  public void updateDisplay(ConfigBit[] cfg) {

    System.out.print("Updating display with "+cfg.length+" bits data\n");

      for(int i=0; i<cfg.length; i++) {
          config[cfg[i].getRow()][cfg[i].getColumn()].setBit(cfg[i].getBit());
          config[cfg[i].getRow()][cfg[i].getColumn()].setState(cfg[i].getState());
      }

      repaint();
  }

}
```

```
/*
 * @(#)CacheBlock.java
 *
 */

import java.io.*;

/**
 * The class CacheBlock implements one Configuration Cache unit.
 * Stores only meta information such as the configurations. The actual
 * configuration data can be fetched from the model to initialize the
 * CLU for a specific cache unit.
 *
 * @author Kiran Bondalapati
 * @see Drive
 * @see Hysam
 * @see System
 * @see Configuration
 * @see Function
 */
public class CacheBlock {

  private int confid;

  /** Constructor which initializes the CacheBlock with the configuration */
  public CacheBlock(int cid) {
    confid = cid;
  }

  /** Constructor which initializes to the default NULL conf */
  public CacheBlock () {
    confid = 0;
  }

}
```

```java
/*
 * @(#)ConfigBit.java
 *
 */

import java.io.*;

/**
 * The class ConfigBit implements the configuration bitstream of the CLU.
 * It encapsulates the configuration of one unit of the CLU which is one
 * cell on device. Can be used to store and transform hardware bitstreams.
 *
 * @author Kiran Bondalapati
 * @version 2.0 1999
 * @see Hysam
 * @see Configuration
 */
public class ConfigBit {

  static final int INIT = 0;     /* Configuration in Initial state. */
  static final int ACTIVE = 1;   /* Currently in Active state */
  static final int RECONFIG = 2;   /* Currently being reconfigured */

  private int row;
  private int column;
  private int bit;
  private int state;

  /** Default constructor. */
  public ConfigBit() {
  }

  /** Constructor to initialize the values for the bit. */
  public ConfigBit(int r, int c, int b, int s) {
    row = r;
    column = c;
    bit = b;
    state =s;
  }

  /** Sets the value of the bit. */
  public void setValue(int r, int c, int b, int s) {
    row = r;
    column = c;
    bit = b;
    state =s;
  }

  /** Reads the private variables of a ConfigBit.
   *  Returns 0 on error, 1 on success */
  public int readData(StreamTokenizer cst) {

    try {
      int tok = cst.nextToken();
      row = (int)cst.nval;
      tok = cst.nextToken();
      column = (int)cst.nval;
      tok = cst.nextToken();
      bit = (int)cst.nval;
      tok = cst.nextToken();
      state = (int)cst.nval;
      return 1;
    } catch (Exception IOException) {
      return 0;
    }
  }

  /** Returns row of bit */
  public int getRow() {
    return row;
  }

  /** Returns column of bit */
  public int getColumn() {
    return column;
  }

  /** Returns value of bit */
  public int getBit() {
    return bit;
  }

  /** Returns state of bit */
  public int getState() {
    return state;
  }

  /** Sets the value of the bit. The value returned is the previous value.
      @param b is the bit value.
      @returns old value of bit. */
  public int setBit(int b) {
    int temp = bit;
    bit = b;
    return temp;
  }

  /** Sets the state of bit */
  public int setState(int s) {
    int temp = state;
    state = s;
    return temp;
  }

}
```

```
/*
 * @(#)Configuration.java
 *
 */

import java.io.*;

/**
 * The class Configuration implements the Configuration in HySAM model.
 *
 * @author Kiran Bondalapati
 * @see Hysam
 * @see ConfigBit
 * @see Function
 * @see Attributes
 * @see Reconfiguration
 */
public class Configuration {

  private int id;       /* Id of the function */
  private String name;  /* Name for storage/display purpose */
  private int numbits;  /* Number of cells for which configuration data is
                           stored in the configuration file */
  private ConfigBit[] cbits; /* The bits which store the information */

  public Configuration() {
    id =0;
    name ="";
    numbits = 0;
  }

  public Configuration(int cid, String cname, int nbits, ConfigBit[] config) {

    id = cid;
    name = cname;
    numbits = nbits;

    if (numbits > 0) {
      cbits = new ConfigBit[numbits];

      for(int i=0; i<numbits; i++) {
        cbits[i] = new ConfigBit(config[i].getRow(), config[i].getColumn(), config[i].getB
it(), config[i].getState());

      }
    }

  }

  /** Reads the private variables of a Configuration from file.
      Opens the input file if specified and reads the configuration data into
      cbits (of type ConfigBits).
      Returns 0 on error, 1 on success.
      Returns 2 on problem reading config data. numbits is reset in this case*/
  public int readData(StreamTokenizer fstream) {

    try {
      int token = fstream.nextToken();
      id = (int)fstream.nval;
      token = fstream.nextToken();
      name = (String)fstream.sval;
      token = fstream.nextToken();
      numbits = (int)fstream.nval;
      if (numbits > 0) {

        /* make sure this directory character doesnt break filename */

        fstream.wordChars('/','/');

        token = fstream.nextToken();
        String cfilename = (String)fstream.sval;

          try {
            StreamTokenizer cstream = new StreamTokenizer(new FileReader(cfilename));

            cstream.commentChar('#');
            cstream.eolIsSignificant(false);

            System.out.print("Reading "+numbits+" bits configuration info from: "+cfilenam
e+"\n");

            /* The file also has row and column numbers. Discard them ?? */
            token = cstream.nextToken();
            token = cstream.nextToken();

            /* The number in file is assumed to be more accurate */

            token = cstream.nextToken();
            numbits = (int)cstream.nval;

            System.out.print("Reading "+numbits+" bits configuration info from: "+cfilenam
e+"\n");

            cbits = new ConfigBit[numbits];

            for (int i=0; i < numbits; i++) {
              cbits[i] = new ConfigBit();
              cbits[i].readData(cstream);
            }
          } catch (Exception IOException) {

            numbits = 0;
            return 2;
          }

      }
      return 1;

    } catch (Exception IOException) {
      return 0;
    }
  }

  /** Returns name of configuration */
  public String getName() {
    return name;
  }

  /** Returns Id of configuration */
  public int getId() {
    return id;
  }

  /** Returns the number of bits of config data available */
  public int getNumbits() {
    return numbits;
  }
  /** Returns the ConfigBits data which stores the configuration */
  public ConfigBit[] getConfig() {

    System.out.print("Constructing configuration data for "+name+" C"+id+" with "+numbits+
" bits\n");

    ConfigBit[] temp = new ConfigBit[numbits];
    for(int i=0; i< numbits; i++) {
      temp[i] = new ConfigBit(cbits[i].getRow(),cbits[i].getColumn(),
                             cbits[i].getBit(), cbits[i].getState()
                             );
    }
    return temp;
  }


}
```

```
import java.io.*;

/**
 * Device.java
 *
 * The system component of the framework. This maintains the various
 * components of the system and their state. These components consists of the
 * CLU, the Cache, etc. To be extended to the interconnection network,
 * memory etc. in future upgrades and versions.
 *
 * @author Kiran Bondalapati
 * @version 2.0 1999
 * @see Function
 * @see Configuration
 * @see Attributes
 * @see Reconfiguration
 * @see Scheduler
 * @see EventList
 * @see Event
 */
public class Device {

  private CacheBlock[] cCache;
  private int cacheSize;

  private CLU myCLU;

  /** Constructor */
  public  Device() {
    myCLU = new CLU();
    int cacheSize = 0;
  }

  /** Reads the parameters??? Currently not used */
  public int readParams(StreamTokenizer pstream) {
    return 1;
  }

  /** Initializes all the configuration bits of the CLU to 0 */
  public void resetCLU() {
    myCLU.reset();
  }

  /** Reads the CLU configuration matrix from a file.
      The first two numbers in the file are the rows and columns.
      The remaining rows*columns numbers are the configuration bit
      values for each cell. They are formatted as per ConfigBit class. */
  public int readCLUData(StreamTokenizer bstream) {

    return myCLU.readData(bstream);
  }

  /** Writes the CLU configuration matrix to a file. (TO BE IMPLEMENTED).
      The first two numbers in the file are the rows and columns.
      The remaining rows*columns numbers are the configuration bit values
      and state for each cell */
  public int writeCLUData() {
    return myCLU.writeData();
  }

  /** Sets the number of rows and columns in CLU */
  public void setCLUSize(int row, int col) {
    myCLU = new CLU(row,col);
  }

  /** Returns the number of rows in CLU */
  public int getCLURows() {
    return myCLU.getRows();
  }

  /** Returns the number of columns in CLU */
  public int getCLUCols() {
    return myCLU.getCols();
  }

  /** Sets configuration and state data of a CLU cell.
      Returns the previous configuration.
      If the row and column are out of range returns -1. */
  public int setCLUCellConfig(int row, int col, int cfg, int st) {
    return myCLU.setCellConfig(row,col,cfg,st);
  }



  /** Returns configuration data of a CLU cell.
      Returns -1 for out of range cells. */
  public int getCLUCellConfig(int row, int col) {
    return myCLU.getCellConfig(row,col);
  }

  /** Sets configuration and state data of the CLU.
      The input is an array ConfigBit[] and the number of data points. */
  public int updateCLUConfig(ConfigBit[] cfg) {
    return myCLU.updateConfig(cfg);
  }


  /** Returns configuration data of the complete CLU. */
  public ConfigBit[][] getCLUConfig() {
    return myCLU.getConfig();
  }

  /** Returns state data of a CLU cell.
      Returns -1 for out of range cells. */
  public int getCLUCellState(int row, int col) {
    return myCLU.getState(row,col);
  }
}
```

```
/**
 * The class Event is used to as a placeholder for events.
 * WARNING: Internal class only. Do not extend!
 * It has to be synchronized with the EventList in all versions!!
 * It is mainly used to transfer events between modules.
 * The methods are also duplicates of EventList without the wrappers.
 *
 * @author Kiran Bondalapati
 * @see Hysam
 * @see Configuration
 * @see Function
 * @see Scheduler
 * @see EventList
 */
public class Event {

  private int type;
  private float start_time;
  private float fin_time;       /* The finish time of the event */

  private int id_one;         /* The semantics of the ids depend on type */
  private int id_two;

  /** Constructor that initializes the Event */
  public Event() {
  }

  /** Constructor that initializes the Event */
  public Event(int t, float stime, float ftime, int id1, int id2) {
    type = t;
    start_time = stime;
    fin_time = ftime;
    id_one = id1;
    id_two = id2;
  }

  /** Gets the type of the event */
  public int getType() {
    return type;
  }

  /** Gets the start time of the event */
  public float getStartTime() {
    return start_time;
  }

  /** Gets the Finish Time of the event */
  public float getFinishTime() {
    return fin_time;
  }

  /** Gets the first id of the event */
  public int getId1() {
    return id_one;
  }

  /** Gets the second id of the event */
  public int getId2() {
    return id_two;
  }
}
```

```
/*
 * @(#)EventList.java
 *
 */

import java.util.*;

/**
 * The class EventList is used to maintain a vector of events. Dynamically
 * adapts size using java.lang.Vector class. Is mainly used by the scheduler
 * to maintain the eventlist.
 *
 * @author Kiran Bondalapati
 * @see Hysam
 * @see Configuration
 * @see Function
 * @see Scheduler
 * @see Event
 */
public class EventList {

  private int num_events;  /* number of events in the list */
  private Vector type;     /* type of the event currently EXECUTE = 1
                              and RECONFIG = 2 */
  private Vector start_time;    /* The starting time of the event */
  private Vector fin_time;      /* The finish time of the event */

  private Vector id_one;        /* The semantics of the ids depend on type */
  private Vector id_two;

  /** Constructor that initializes the EventList */
  public EventList() {
    num_events = 0;

    type = new Vector();
    start_time = new Vector();
    fin_time = new Vector();
    id_one = new Vector();
    id_two = new Vector();

    System.out.print("Constructer called\n");

  }

  /** Adds an event at the end to the EventList. The semantics of id1 and id2
      parameters are based on the event type */
  public int addEvent(int event_type, int id1, int id2, float s_time,
                      float f_time) {

    type.insertElementAt(new Integer(event_type), num_events);
    id_one.insertElementAt(new Integer(id1), num_events);
    id_two.insertElementAt(new Integer(id2), num_events);
    start_time.insertElementAt(new Float(s_time), num_events);
    fin_time.insertElementAt(new Float(f_time), num_events);

    num_events++;

    return 1;
  }

  /** Gets the type of the index'th event */
  public int getType(int ind) {
    return ((Integer) type.elementAt(ind)).intValue();
  }
  /** Gets the start time of the index'th event */
  public float getStartTime(int ind) {
    return ((Float)start_time.elementAt(ind)).floatValue();
  }

  /** Gets the Finish Time of the index'th event */
  public float getFinishTime(int ind) {
    return ((Float)fin_time.elementAt(ind)).floatValue();
  }

  /** Gets the first id of the index'th event */
  public int getId1(int ind) {
    return ((Integer)id_one.elementAt(ind)).intValue();
  }

  /** Gets the second id of the index'th event */
  public int getId2(int ind) {
    return ((Integer)id_two.elementAt(ind)).intValue();
  }


  /** Constructs and returns an Event node */
  public Event getEvent(int ind) {
    if (ind > num_events) {
      return null;
    }
    else {
      return (new Event(getType(ind), getStartTime(ind), getFinishTime(ind), getId1(ind),
getId2(ind)));
    }
  }


}
```

```
/*
 * @(#)Function.java
 *
 */

import java.io.*;

/**
 * The class Function implements the Function in HySAM model.
 *
 * @author Kiran Bondalapati
 * @see Hysam
 * @see Configuration
 * @see Attributes
 */
public class Function {

  private int id;         /* Id of the function */
  private String name;  /* Name for storage/display purpose */


  /** Constructor intializes the data of the Function */
  public Function() {
    id = 0;
    name = "";
  }

  /** Reads the private variables of a Function from DataInput argument.
   *  Returns 0 on error, 1 on success */
  public int readData(StreamTokenizer fstream) {

    System.out.print("Reading a fn data\n");

    try {

      int token = fstream.nextToken();
      id = (int) fstream.nval;

      System.out.print("id = ");
      System.out.print(id);
      System.out.print("\n");

      token = fstream.nextToken();
      name = (String) fstream.sval;

      System.out.print(" name = ");
      System.out.print(name);
      System.out.print("\n");

      return 1;
    } catch (Exception IOException) {

      System.out.print("Read Error in Function\n");
      return 0;
    }
  }

  /** Returns name of function */
  public String getName() {
    return name;
  }

  /** Returns Id of function */
  public int getId() {
```

```
/** HySAM : Hybrid System Architecture Model */

import java.io.*;

/**
 * Hysam : Hybrid System Architecture Model.
 *
 * The main class which describes and implements the model. Contains the
 * various components of the hybrid system architecture.
 * The components are the CPU, CLU, Configuration Cache, Interconnection
 * Network.
 * The components of the model are described by the Functions, Configurations,
 * Attributes, Reconfiguration, etc.
 *
 * @author Kiran Bondalapati
 * @version 2.0 1999
 * @see Function
 * @see Configuration
 * @see Attributes
 * @see Reconfiguration
 * @see Scheduler
 * @see EventList
 * @see Event
 */
public class Hysam {

  static final int INFINITY = 1000000000; /* Some large number */
  private int numRows;
  private int numCols;
  private int numF;
  private int numC;
  private int numA;
  private int numT;
  private int numR;

  private Function[] F;
  private Configuration[] C;
  private Attributes A;
  private Application T;
  private Reconfiguration R;


  /** Constructor */
  public void Hysam() {
  }

  /** Reads the model parameters by calling the readData functions of each
      of the components. */
  public int readParams(StreamTokenizer pstream) {

    int res = 0;
    int i, j;

    System.out.print("Reading Data\n");

    /* force garbage collection just to make sure we have max memory */
    System.gc();

    try {
      int token = pstream.nextToken();

      numF = (int)pstream.nval;

      System.out.print("Read functions = ");
  System.out.print(numF);
  System.out.print("\n");


  F = new Function[numF];

  for (i=0; i<numF; i++) {

    F[i] = new Function();
    System.out.print("Reading function ");
    System.out.print(i);
    System.out.print("\n");

    res = (F[i]).readData(pstream);

    System.out.print("res =");
    System.out.print(res);
    System.out.print("\n");
  }

  System.out.print("Finished\n");

  token = pstream.nextToken();

  numC = (int)pstream.nval;
  numC = numC + 1;              /** C0 is a dummy configuration **/
  C = new Configuration[numC];

  /* Set up some initial configuration pattern here in the config
     and pass it to the C0 configuration */

  ConfigBit[] cbits = new ConfigBit[numRows*numCols];

  for (i=0; i<numRows; i++) {
    for (j=0; j<numCols; j++) {
      cbits[i] = new ConfigBit(i,j,0, ConfigBit.INIT);
    }
  }

  C[0] = new Configuration(0, "Initial", numRows*numCols, cbits);

  for (i=1; i<numC; i++) {
    C[i] = new Configuration();
    res = C[i].readData(pstream);
  }

  System.out.print("Read configurations =");
  System.out.print(numC);
  System.out.print("\n");

  A = new Attributes();
  res = A.readData(pstream);

  System.out.print("Read Attributes =");
  System.out.print(A.getNum());
  System.out.print("\n");

  R = new Reconfiguration();
  res = R.readData(pstream);

  return res;
} catch (Exception IOException) {
  return 0;
}
```

```
}

/** Reads the application data from a file. The type of data read is based
    on the type parameter. Currently 1 Linear, 2 Precision */
public int readAppl(int type, StreamTokenizer astream) {

  int res = 0;

  if (type == 1) {
    T = new Application();
    res = T.readLinear(astream);
  }

  return res;
}

/** Computes the schedule using various algorithms based on the type
    of application input */
public Scheduler computeSchedule(int type) {
  Scheduler S = new Scheduler();

  if (type == 1) {
    int res = S.Linear(numF, F, numC, C, A, R, T);
    if (res > 0)
      return S;
    else
      return null;
  } else if (type == 2) {
  }

  return null;
}


/** Gets the name of a function.
    @param functionID
    @returns functionName or "X" if Id not found
*/
public String getFuncName(int fid) {
  for (int i=0; i<numF; i++) {
    if (F[i].getId() == fid) {
      return F[i].getName();
    }
  }
  return "X";
}

/** Gets the name of a configuration.
    @param configuration ID
    @returns configurationName or "Y" if Id not found
*/
public String getConfName(int cid) {
  for (int i=0; i<numC; i++) {
    if (C[i].getId() == cid) {
      return C[i].getName();
    }
  }
  return "Y";
}

/** Gets the configuration data for a configuration.
    @param configuration ID
    @returns configuration data as ConfigBit[]
  */
  public ConfigBit[] getConfig(int cid) {
    for (int i=0; i<numC; i++) {
      if (C[i].getId() == cid) {
        System.out.print("Found configuration "+cid+"\n");
        return C[i].getConfig();
      }
    }
    System.out.print("Did not find configuration "+cid+"\n");
    return null;
  }


}
```

```
import java.awt.*;
import java.awt.event.*;

public class MessageDialog extends Dialog {

  public MessageDialog(Frame parent,String title,boolean modal,String text[],
                       String buttons[], WindowListener wh, ActionListener bh) {
    super(parent,title,modal);

    int textLines = text.length;
    int numButtons = buttons.length;
    Panel textPanel = new Panel();
    Panel buttonPanel = new Panel();
    textPanel.setLayout(new GridLayout(textLines,1));

    for(int i=0;i<textLines;++i) textPanel.add(new Label(text[i]));

    for(int i=0;i<numButtons;++i){
      Button b = new Button(buttons[i]);
      b.addActionListener(bh);
      buttonPanel.add(b);
    }
    add("North",textPanel);
    add("South",buttonPanel);
    setBackground(Color.lightGray);
    setForeground(Color.black);
    pack();
    addWindowListener(wh);
  }
}
```

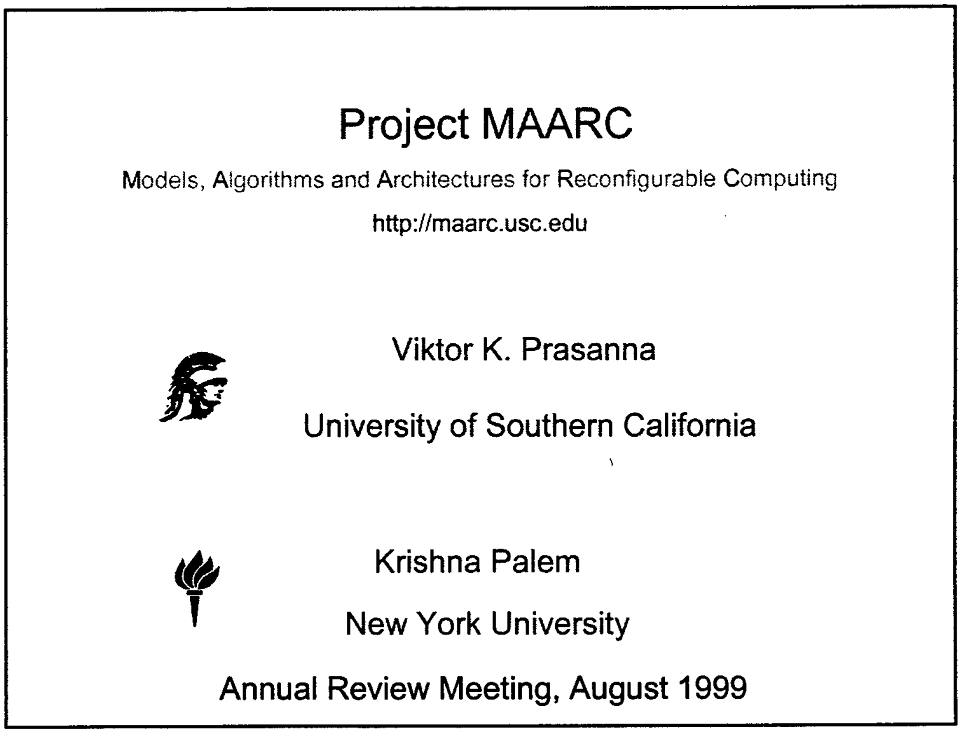
```
import java.awt.*;
import java.awt.event.*;

public class MyMenu extends Menu {

  public MyMenu(Object labels[],ActionListener al,ItemListener il) {

    super((String)labels[0]);
    String menuName = (String) labels[0];
    char firstMenuChar = menuName.charAt(0);

    if(firstMenuChar == '~' || firstMenuChar =='!'){
      setLabel(menuName.substring(1));
      if(firstMenuChar == '~') setEnabled(false);
    }

    for(int i=1;i<labels.length;++i) {
      if(labels[i] instanceof String){
        if("-".equals(labels[i])) addSeparator();
        else{
          String label = (String)labels[i];
          char firstChar = label.charAt(0);
          switch(firstChar){
          case '+':
            CheckboxMenuItem checkboxItem = new CheckboxMenuItem(label.substring(1));
            checkboxItem.setState(true);
            add(checkboxItem);
            checkboxItem.addItemListener(il);
            break;
          case '#':
            checkboxItem = new CheckboxMenuItem(label.substring(1));
            checkboxItem.setState(true);
            checkboxItem.setEnabled(false);
            add(checkboxItem);
            checkboxItem.addItemListener(il);
            break;
          case '-':
            checkboxItem = new CheckboxMenuItem(label.substring(1));
            checkboxItem.setState(false);
            add(checkboxItem);
            checkboxItem.addItemListener(il);
            break;
          case '=':
            checkboxItem = new CheckboxMenuItem(label.substring(1));
            checkboxItem.setState(false);
            checkboxItem.setEnabled(false);
            add(checkboxItem);
            checkboxItem.addItemListener(il);
            break;
          case '~':
            MenuItem menuItem = new MenuItem(label.substring(1));
            menuItem.setEnabled(false);
            add(menuItem);
            menuItem.addActionListener(al);
            break;
          case '!':
            menuItem = new MenuItem(label.substring(1));
            add(menuItem);
            menuItem.addActionListener(al);
            break;
          default:
            menuItem = new MenuItem(label);
            add(menuItem);
            menuItem.addActionListener(al);
          }
        }
      }else{
        add(new MyMenu((Object[])labels[i],al,il));
      }
  }
  }
  public MenuItem getItem(String menuItem) {
    int numItems = getItemCount();
    for(int i=0;i<numItems;++i)
      if(menuItem.equals(getItem(i).getLabel())) return getItem(i);
    return null;
  }
}
```

```
import java.awt.*;
import java.awt.event.*;

public class MyMenuBar extends MenuBar {
  public MyMenuBar(Object labels[][],ActionListener al, ItemListener il) {
    super();
    for(int i=0;i<labels.length;++i)
      add(new MyMenu(labels[i],al,il));
  }
  public MyMenu getMenu(String menuName) {
    int numMenus = getMenuCount();
    for(int i=0;i<numMenus;++i)
      if(menuName.equals(getMenu(i).getLabel())) return((MyMenu)getMenu(i));
    return null;
  }
}
```

```
import java.awt.*;

/**
 * ProgressBar: A canvas widget to display the progress of some task.
 * It is similar to the Swing progressbar widget but is much simpler currently.
 *
 * Can be extended to display percent value as String.
 * @author Kiran Bondalapati
 */
public class ProgressBar extends Canvas {

  int x, y, width, height;
  float percent;

  static final int borderWidth = 2 ;

  Graphics graphics;
  Image image;

  Color barColor = Color.blue;
  Color textColor = Color.white;
  Color backColor = new Color(50,50,50);
  Color borderColor = new Color(200,10,10);

  /** Default constructor.
      Initializes width and height to 100 and 20 and percent to 0 */
  public ProgressBar() {
    x =0;
    y = 0;
    width = 100;
    height = 20;
    setSize(100,20);
    percent = 0;

    repaint();
  }

  /** Parameterized Constructor.
      Sets the values for the bar */
  public ProgressBar(int xpos, int ypos, int w, int h) {
    x = xpos;
    y = ypos;
    width = w;
    height = h;
    setSize(width,height);
    percent = 0;

    repaint();
  }


  /**
   * Paint the ProgressBar bar.
   */
  public void paint(Graphics g) {
    if (graphics == null) {
      image = createImage(width, height);
      graphics = image.getGraphics();
    }

    int left = (int)((float)(width) * percent);
    int right = width;

    //  System.out.print("Time "+percent+"% left "+left+" right "+right+"\n");
    graphics.setColor(borderColor);
    graphics.drawRect(0, 0, width, height);

    if (left > 0) {
        graphics.setColor(barColor);
        graphics.fillRect(0, borderWidth,left, height -borderWidth);
    }

    if (right > 0) {
        graphics.setColor(backColor);
        graphics.fillRect(left, borderWidth, right, height - borderWidth);
    }

    graphics.setColor(textColor);
    graphics.drawString((new Integer((int)(percent*100))).toString()+"%", width/2 - 20, he
ight-3);

    g.drawImage(image, 0, 0, null);

  }

  /** Set the various colors for the bar */
  public void setColors(Color bar, Color back, Color border) {
    barColor = bar;
    backColor = back;
    borderColor = border;
    repaint();
  }

  /** Sets the percent value of the bar and redraws it. */
  public void setPercent(float per) {
    percent = per;
    repaint();
  }

}
```

```
/*
 * @(#)Reconfiguration.java
 *
 */

import java.io.*;

/**
 * The class Reconfiguration contains the cost of chaging configurations
 *
 * @author Kiran Bondalapati
 * @see Hysam
 * @see Configuration
 * @see Function
 */
public class Reconfiguration {

  private int numR;                 /* the number of data pairs stored */
  private int[] from_cid;           /* source configuration id */
  private int[] to_cid;             /* target configuration id */
  private float[] cost;             /* reconfiguration cost */
  private int partial;              /* not used currently */

  public int readData(StreamTokenizer rst) {

    int tok;

    try {
      tok = rst.nextToken();
      numR = (int)rst.nval;

      from_cid = new int[numR];
      to_cid = new int[numR];
      cost = new float[numR];

      for(int i=0; i<numR; i++) {
        tok = rst.nextToken();
        from_cid[i] = (int)rst.nval;
        tok = rst.nextToken();
        to_cid[i] = (int)rst.nval;
        tok = rst.nextToken();
        cost[i] = (float)rst.nval;
      }
      return 1;

    } catch (Exception IOException) {
      return 0;
    }
  }

  /** Returns the Source configuration Id at [ind] */
  public int getFromId(int ind) {
    return from_cid[ind];
  }

  /** Returns the target Configuration Id at [ind] */
  public int getToId(int ind) {
    return to_cid[ind];
  }

  /** Returns the cost of reconfiguration between two configurations.
      If the pair does not exist then Hysam.INFINITY is returned */
  public float getReconfCost(int frmid, int toid) {
    for(int i=0; i<numR; i++) {
      if ((from_cid[i] == frmid) && (to_cid[i] == toid)) {
        return cost[i];
      }
    }

    return Hysam.INFINITY;
  }


}
```

```java
/*
 * @(#)Scheduler.java
 *
 */

import java.io.*;

/**
 * Scheduler
 *
 * This class incorporates the scheduling components of the DRIVE framework.
 * Various scheduling algorithm routines are available in this class.
 * This class interacts dynamically with the other classes by providing
 * mechanisms to query current event and compute the next scheduled event.
 * These mechanisms facilitate dynamic scheduling algorithms.
 *
 * @author Kiran Bondalapati
 * @version 2.0 1999
 * @see Hysam
 * @see ConfigBit
 * @see Function
 * @see Attributes
 * @see Reconfiguration
 */
public class Scheduler {

  static final int EXECUTE  = 1;
  static final int RECONFIG = 2;

  private int computed = 0;
  private int numEvents;
  private int currEvent = -1;

  private EventList list;

  private float[][] matrix;

  public void Scheduler() {

    computed = 0;
    numEvents = 0;
    currEvent = -1;

    list = new EventList();

  }

  /** Resets the schedule to the beginning of the event list.
      If schedule has not been computed returns 0.
      Returns 1 on success. */
  public int reset() {
    if (computed == 1) {
      currEvent = -1;
      return 1;
    }
    else
      return 0;
  }

  /** Returns the next event in the schedule */
  public Event getNextEvent() {

    if (computed == 1) {
      currEvent++;
      if (currEvent == numEvents) {
        currEvent = -1;
        return null;
      } else {
        return list.getEvent(currEvent);
      }
    }

    return null;

  }

  /** Returns the current event in the schedule */
  public Event getCurrentEvent() {
    if (computed == 1) {
      if ((currEvent > -1) && (currEvent < numEvents)) {
        return list.getEvent(currEvent);
      } else {
        return null;
      }
    }
    return null;
  }

  /** Returns the finish time of schedule.
      If the schedule is a static schedule then the finishing time of the
      schedule can be extracted for display purposes. */
  public float getFinishTime() {
    if (computed == 1) {
      return list.getFinishTime(numEvents -1);
    }
    return 0;
  }

  /** Computes the schedule for a linear dependent list of tasks.
      Uses dynamic programming to compute the matrix of execution timings. */
  public int Linear(int numF, Function[] F, int numC, Configuration[] C, Attributes A, Rec
onfiguration R, Application T) {

    int i,j, k;
    int numT = T.getNumTasks();

    matrix = new float[numT][numC];

    int[][] Sol = new int[numT][numC];

    System.out.print("\nInitializing\n");

    /* First initialize everything to INFINITY */
    for(i=0; i<numT; i++) {
      for(j=0; j<numC; j++) {
        matrix[i][j] = Hysam.INFINITY;
      }
    }

    System.out.print("First Step\n");

    for(j=0; j<numC; j++) {

      matrix[0][j] = A.getExecCost(T.getFuncId(0), C[j].getId()) +
        R.getReconfCost(0,C[j].getId());

      Sol[0][j] = 0;
      System.out.print("Cost ["+j+"] is "+matrix[0][j]+"\n");
    }

    for(i=1; i<numT; i++) {

      System.out.print("Step "+i+"\n");

      for(j=0; j<numC; j++) {

        int mink = 0;
        for(k=0; k<numC; k++) {

          if (matrix[i-1][k] + R.getReconfCost(C[k].getId(),C[j].getId()) < matrix[i-1][mi
nk] + R.getReconfCost(C[mink].getId(),C[j].getId())) {
            mink = k;
          }
        }

        Sol[i][j] = mink;
        matrix[i][j] = A.getExecCost(T.getFuncId(i), C[j].getId()) + matrix[i-1][mink] + R
.getReconfCost(C[mink].getId(),C[j].getId());

        System.out.print("Cost ["+i+" "+j+"] is "+matrix[i][j]+"\n");

      }
    }

    int minj = 1;

    for(j=0; j<numC; j++) {

      System.out.print("Cost of ["+j+"] is "+matrix[numT-1][j]+"\n");
      if (matrix[numT-1][j] < matrix[numT-1][minj] )
        minj = j;
    }

    int cnum = minj;

    System.out.print("Minimum cost found ending in conf ");
    System.out.print(C[minj].getId()+" cost "+matrix[numT-1][minj]+"\n");

    /* reverse the Sol matrix first to obtain an ordered list */

    int prev;
    int next = -1;

    for(i=numT-1; i> -1; i--) {
      prev = Sol[i][cnum];
      Sol[i][cnum] = next;
      next = cnum;
      cnum = prev;
    }

    cnum = next;    /* the first configuration */

    list = new EventList();

    float s_time, f_time;

    f_time = 0;
    int curr = C[0].getId();
    int to_cid;
    for(i=0; i<numT; i++) {

      to_cid = C[cnum].getId();

      s_time = f_time;
      f_time = s_time + R.getReconfCost(curr, to_cid);

      list.addEvent(RECONFIG, curr, to_cid, s_time, f_time);
      numEvents++;

      System.out.print("Reconf "+curr+" "+to_cid+" Time "+s_time+" to "+f_time+"\n");

      s_time = f_time;
      f_time = s_time + A.getExecCost(T.getFuncId(i), C[cnum].getId());

      list.addEvent(EXECUTE, T.getFuncId(i), to_cid, s_time, f_time);
      numEvents++;

      System.out.print("Execute "+T.getFuncId(i)+" "+to_cid+" Time "+s_time+" to "+f_time+
"\n");

      curr = to_cid;
      cnum = Sol[i][cnum];

    }

    computed = 1;

    return 1;

  }

}
```



Project MAARC

Models, Algorithms and Architectures for Reconfigurable Computing

http://maarc.usc.edu

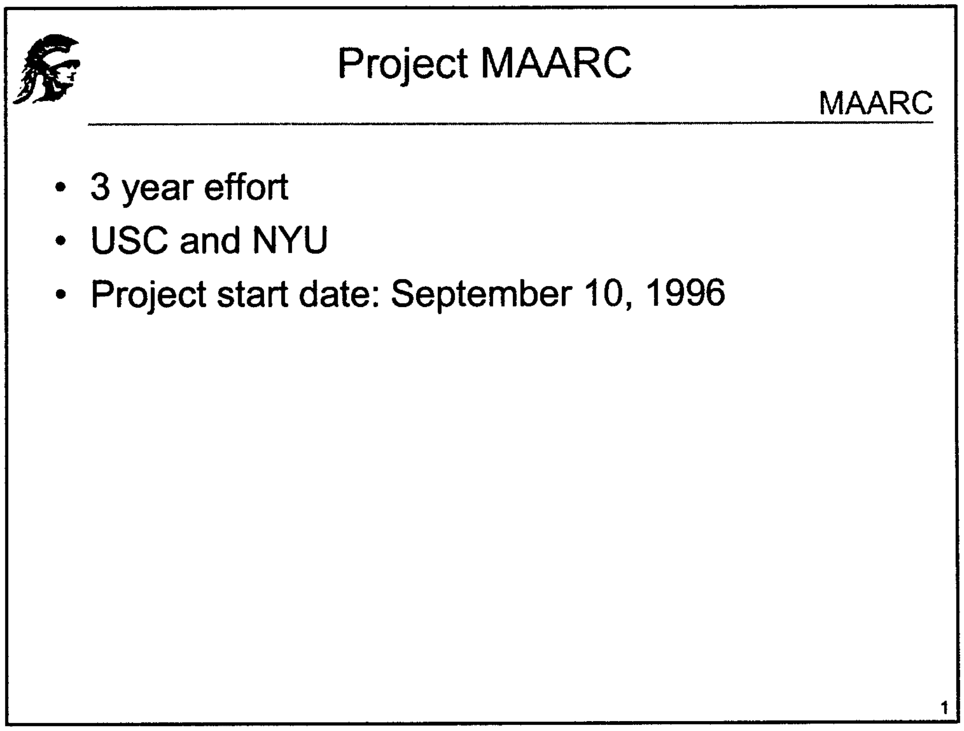
Viktor K. Prasanna

University of Southern California

Krishna Palem

New York University

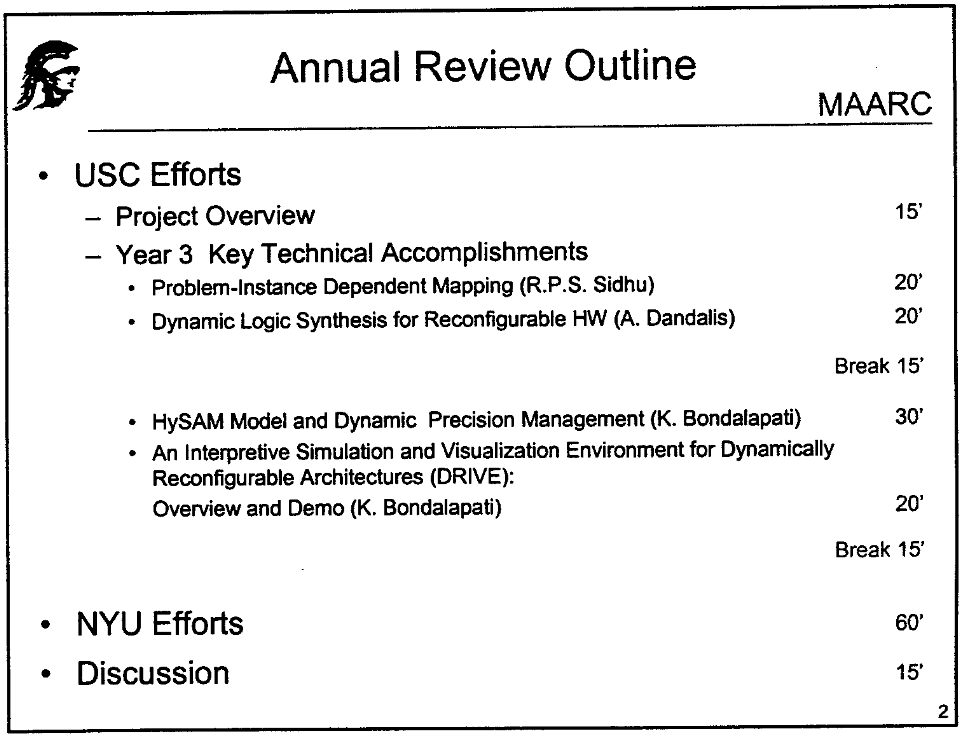
Annual Review Meeting, August 1999



Project MAARC

MAARC

- 3 year effort
- USC and NYU
- Project start date: September 10, 1996



Annual Review Outline

MAARC

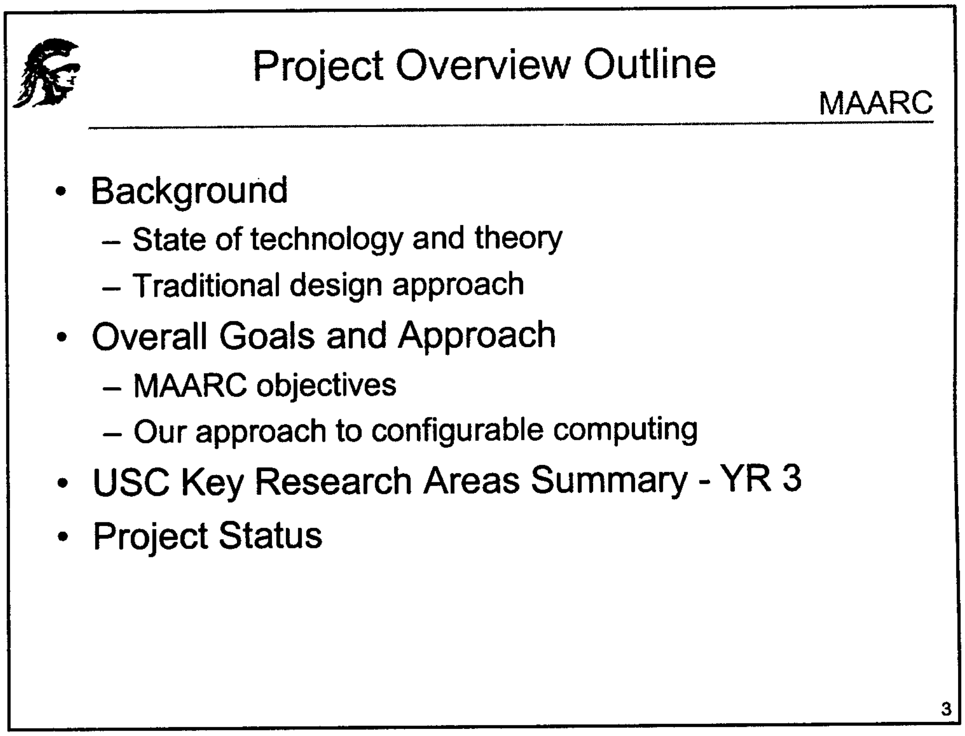
- USC Efforts
  - Project Overview 15'
  - Year 3 Key Technical Accomplishments
    - Problem-Instance Dependent Mapping (R.P.S. Sidhu) 20'
    - Dynamic Logic Synthesis for Reconfigurable HW (A. Dandalis) 20'

Break 15'

    - HySAM Model and Dynamic Precision Management (K. Bondalapati) 30'
    - An Interpretive Simulation and Visualization Environment for Dynamically Reconfigurable Architectures (DRIVE): Overview and Demo (K. Bondalapati) 20'

Break 15'

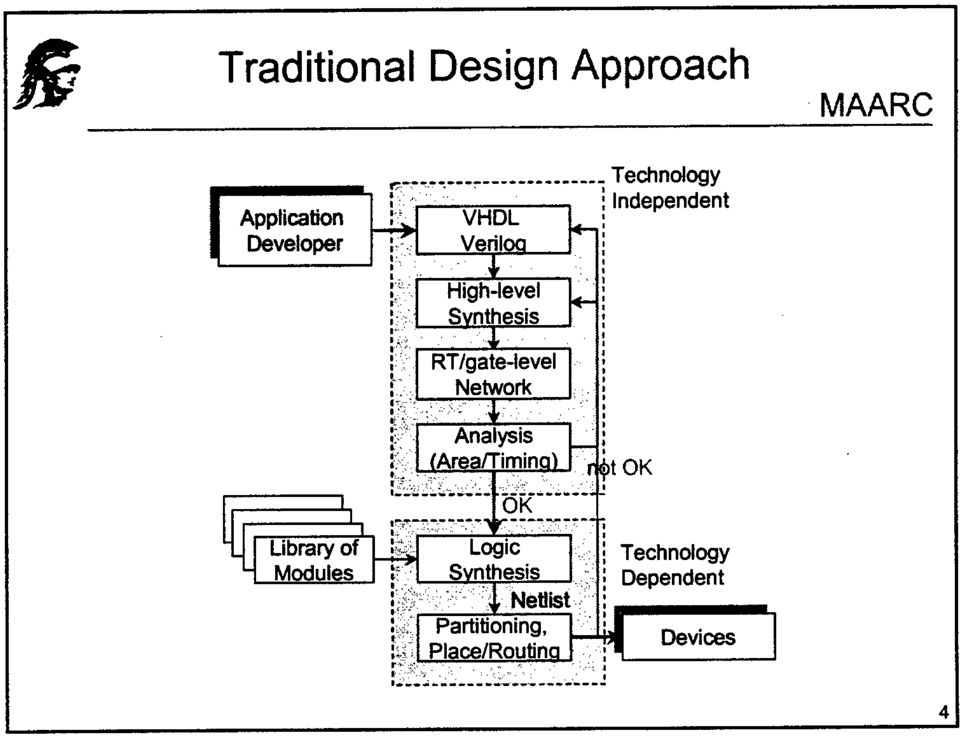
- NYU Efforts 60'
- Discussion 15'



Project Overview Outline

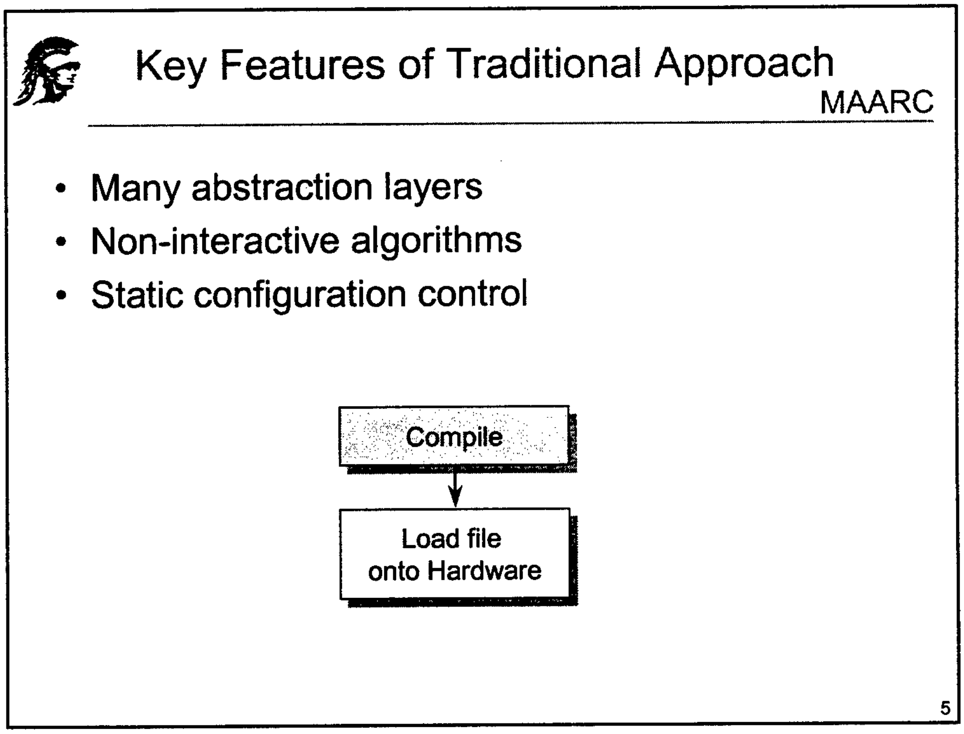
MAARC

- Background
  - State of technology and theory
  - Traditional design approach
- Overall Goals and Approach
  - MAARC objectives
  - Our approach to configurable computing
- USC Key Research Areas Summary - YR 3
- Project Status



Traditional Design Approach

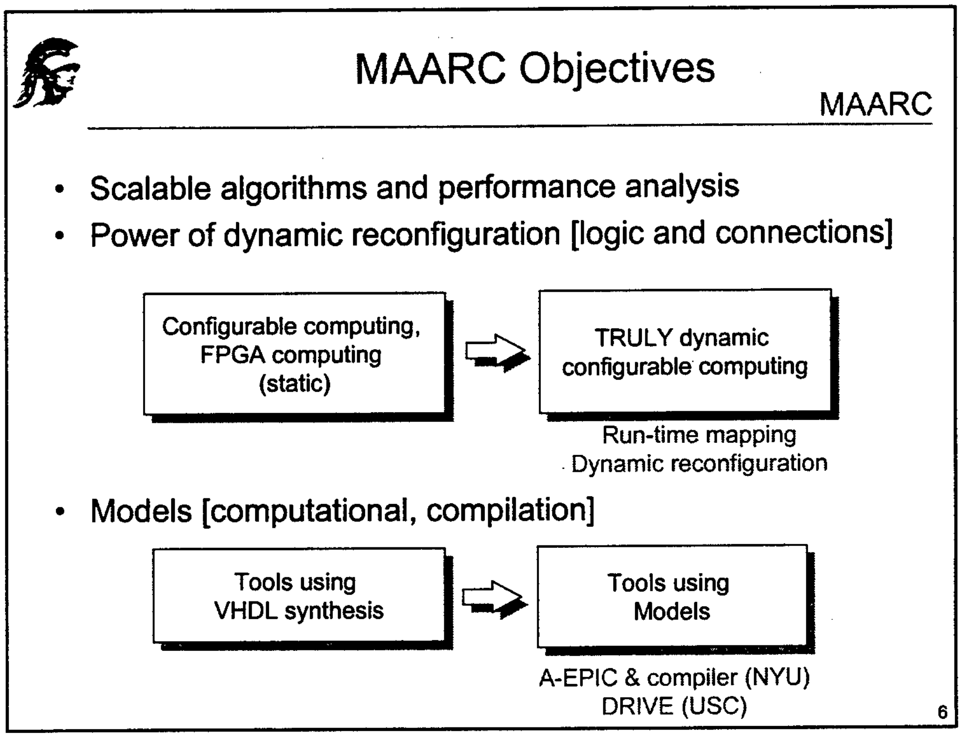
MAARC



Key Features of Traditional Approach

MAARC

- Many abstraction layers
- Non-interactive algorithms
- Static configuration control



MAARC Objectives

MAARC

- Scalable algorithms and performance analysis
- Power of dynamic reconfiguration [logic and connections]

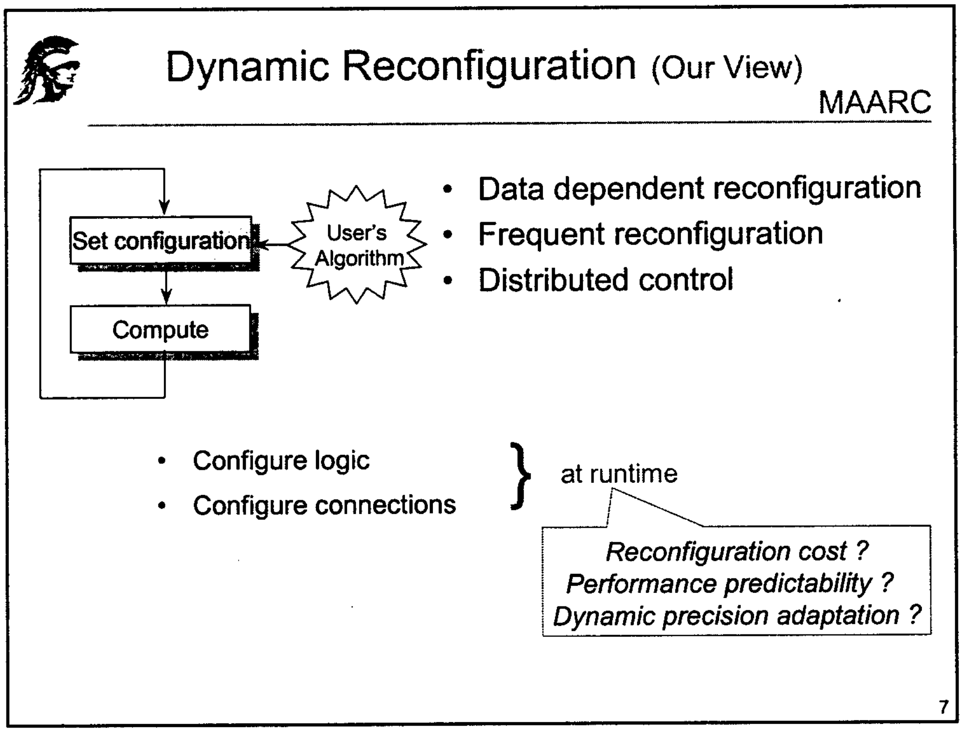
Configurable computing, FPGA computing (static) → TRULY dynamic configurable computing

Run-time mapping
Dynamic reconfiguration

- Models [computational, compilation]

Tools using VHDL synthesis → Tools using Models

A-EPIC & compiler (NYU)
DRIVE (USC)



Dynamic Reconfiguration (Our View)

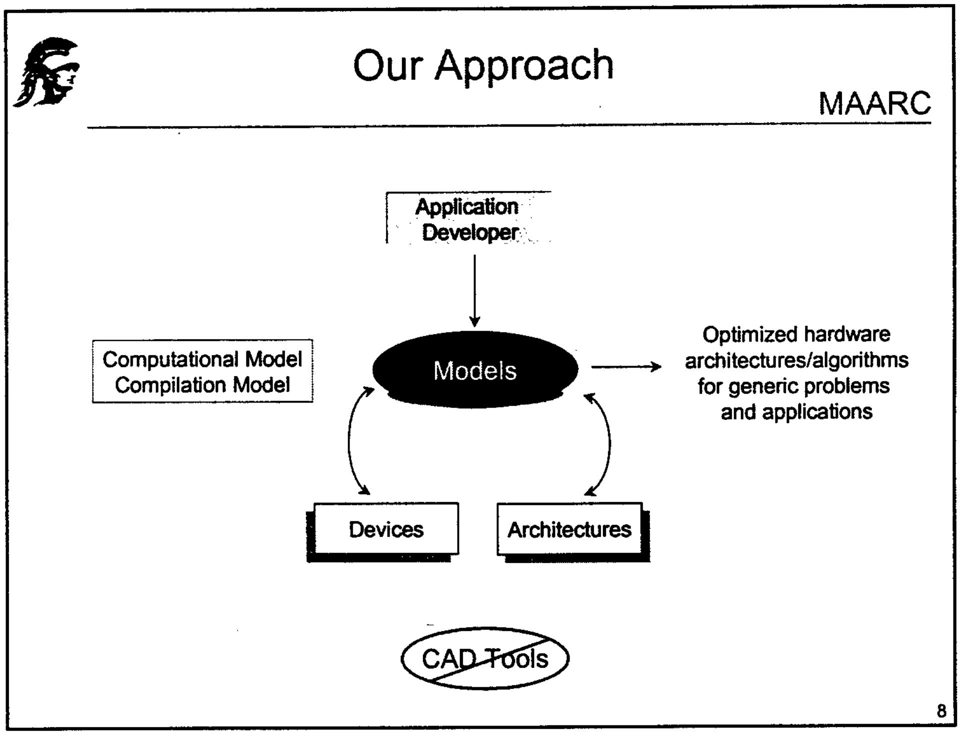
MAARC

- Data dependent reconfiguration
- Frequent reconfiguration
- Distributed control

- Configure logic
- Configure connections

} at runtime

Reconfiguration cost ?
Performance predictability ?
Dynamic precision adaptation ?



Our Approach

MAARC

Application Developer

Computational Model
Compilation Model

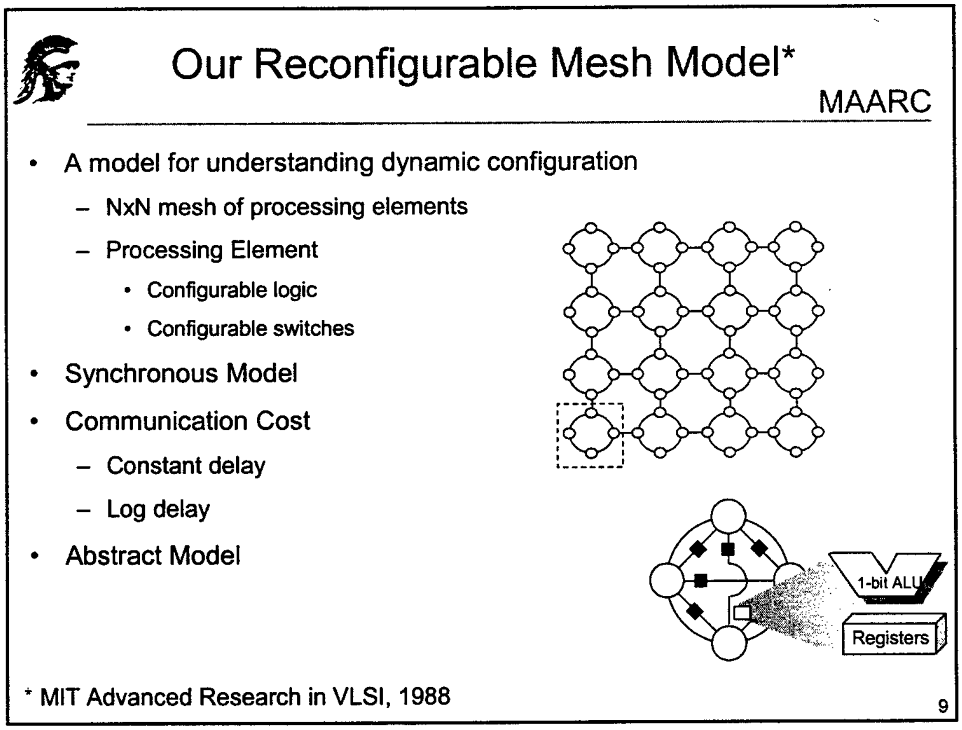
Models

Optimized hardware architectures/algorithms for generic problems and applications

Devices

Architectures

~~CAD Tools~~



Our Reconfigurable Mesh Model*

MAARC

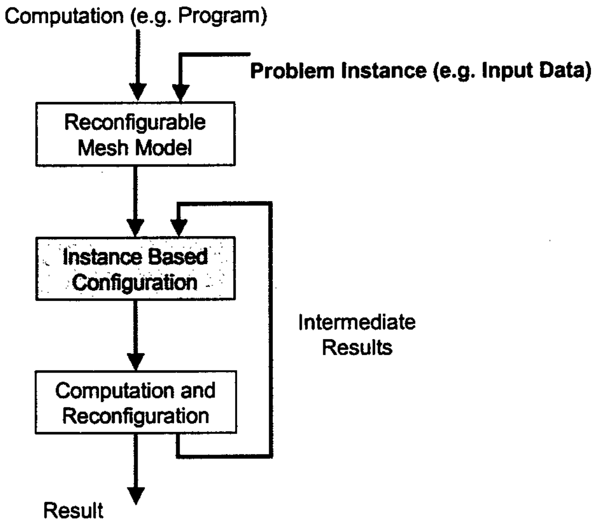
- A model for understanding dynamic configuration
  - NxN mesh of processing elements
  - Processing Element
    - Configurable logic
    - Configurable switches
- Synchronous Model
- Communication Cost
  - Constant delay
  - Log delay
- Abstract Model

1-bit ALU
Registers

* MIT Advanced Research in VLSI, 1988

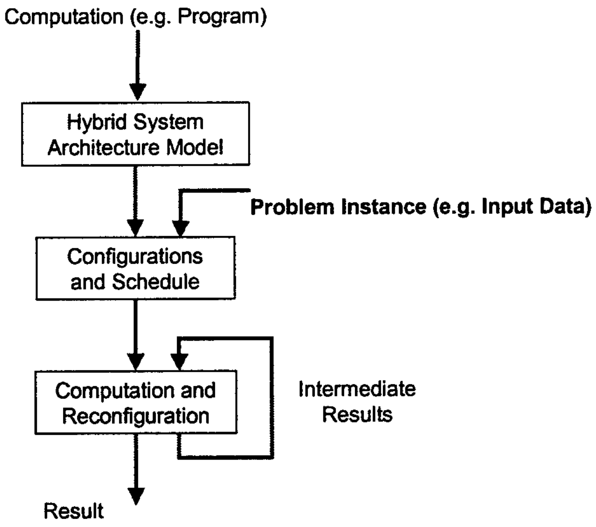
Reconfigurable Meshes: Dynamic Reconfiguration

MAARC


Computation (e.g. Program)
Problem Instance (e.g. Input Data)
Reconfigurable Mesh Model
Instance Based Configuration
Intermediate Results
Computation and Reconfiguration
Result

Our HySAM Model: Scheduled Reconfiguration

MAARC


Computation (e.g. Program)
Hybrid System Architecture Model
Problem Instance (e.g. Input Data)
Configurations and Schedule
Computation and Reconfiguration
Intermediate Results
Result

Advantages of Our Approach

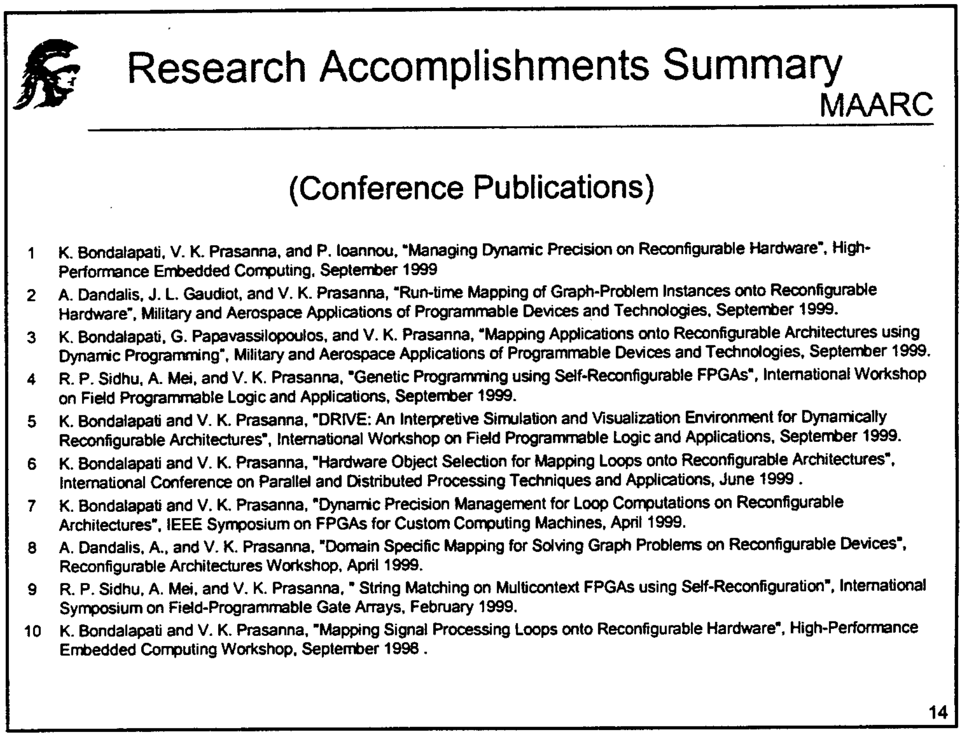
MAARC

- Algorithmic design methodology
- Application developer “sees” the device and architectural features in the algorithm design phase
- Runtime interaction between algorithm and hardware
- Better exploitation of dynamic reconfiguration
- Scalable algorithm development

USC Team Members

MAARC

- Faculty
  - Viktor Prasanna (PI)
- Students
  - Kiran Bondalapati
  - Seonil Choi
  - Andreas Dandalis
  - Reetinder Sidhu



Research Accomplishments Summary

MAARC

(Conference Publications)

1 K. Bondalapati, V. K. Prasanna, and P. Ioannou, "Managing Dynamic Precision on Reconfigurable Hardware", High-Performance Embedded Computing, September 1999

2 A. Dandalis, J. L. Gaudiot, and V. K. Prasanna, "Run-time Mapping of Graph-Problem Instances onto Reconfigurable Hardware", Military and Aerospace Applications of Programmable Devices and Technologies, September 1999.

3 K. Bondalapati, G. Papavassilopoulos, and V. K. Prasanna, "Mapping Applications onto Reconfigurable Architectures using Dynamic Programming", Military and Aerospace Applications of Programmable Devices and Technologies, September 1999.

4 R. P. Sidhu, A. Mei, and V. K. Prasanna, "Genetic Programming using Self-Reconfigurable FPGAs", International Workshop on Field Programmable Logic and Applications, September 1999.

5 K. Bondalapati and V. K. Prasanna, "DRIVE: An Interpretive Simulation and Visualization Environment for Dynamically Reconfigurable Architectures", International Workshop on Field Programmable Logic and Applications, September 1999.

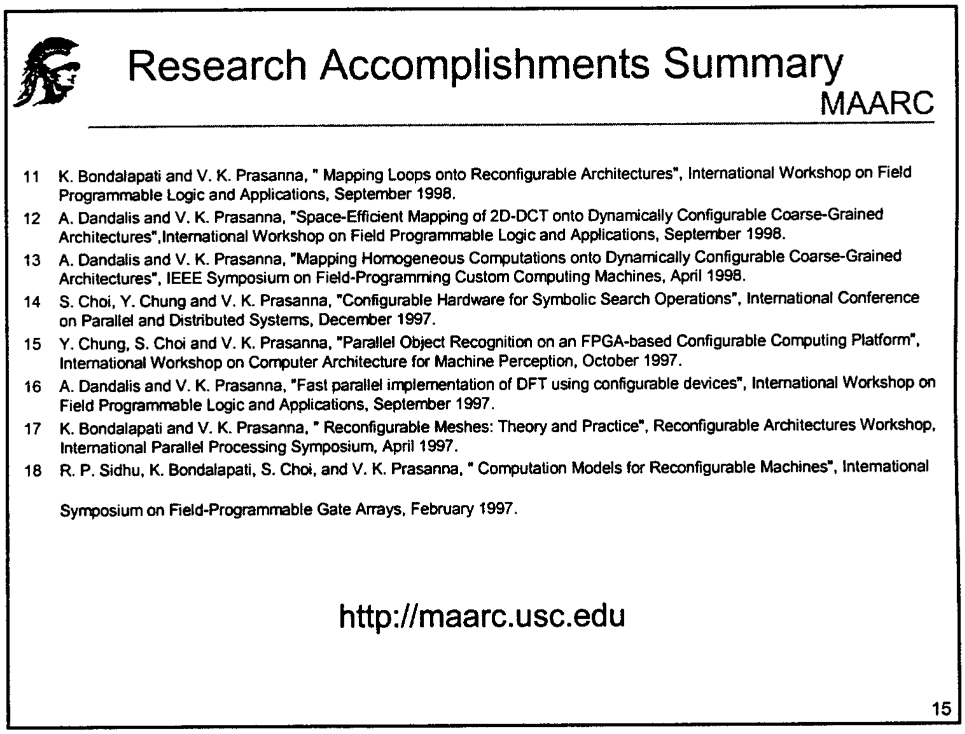
6 K. Bondalapati and V. K. Prasanna, "Hardware Object Selection for Mapping Loops onto Reconfigurable Architectures", International Conference on Parallel and Distributed Processing Techniques and Applications, June 1999 .

7 K. Bondalapati and V. K. Prasanna, "Dynamic Precision Management for Loop Computations on Reconfigurable Architectures", IEEE Symposium on FPGAs for Custom Computing Machines, April 1999.

8 A. Dandalis, A., and V. K. Prasanna, "Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices", Reconfigurable Architectures Workshop, April 1999.

9 R. P. Sidhu, A. Mei, and V. K. Prasanna, " String Matching on Multicontext FPGAs using Self-Reconfiguration", International Symposium on Field-Programmable Gate Arrays, February 1999.

10 K. Bondalapati and V. K. Prasanna, "Mapping Signal Processing Loops onto Reconfigurable Hardware", High-Performance Embedded Computing Workshop, September 1998 .



Research Accomplishments Summary

MAARC

11 K. Bondalapati and V. K. Prasanna, " Mapping Loops onto Reconfigurable Architectures", International Workshop on Field Programmable Logic and Applications, September 1998.

12 A. Dandalis and V. K. Prasanna, "Space-Efficient Mapping of 2D-DCT onto Dynamically Configurable Coarse-Grained Architectures",International Workshop on Field Programmable Logic and Applications, September 1998.

13 A. Dandalis and V. K. Prasanna, "Mapping Homogeneous Computations onto Dynamically Configurable Coarse-Grained Architectures", IEEE Symposium on Field-Programming Custom Computing Machines, April 1998.

14 S. Choi, Y. Chung and V. K. Prasanna, "Configurable Hardware for Symbolic Search Operations", International Conference on Parallel and Distributed Systems, December 1997.

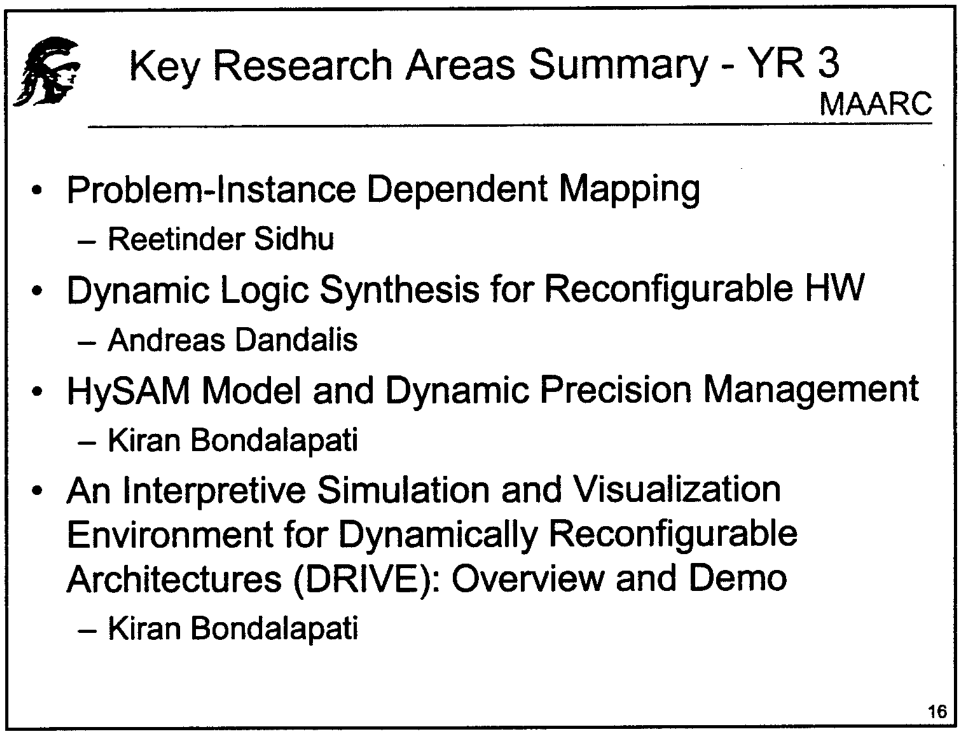
15 Y. Chung, S. Choi and V. K. Prasanna, "Parallel Object Recognition on an FPGA-based Configurable Computing Platform", International Workshop on Computer Architecture for Machine Perception, October 1997.

16 A. Dandalis and V. K. Prasanna, "Fast parallel implementation of DFT using configurable devices", International Workshop on Field Programmable Logic and Applications, September 1997.

17 K. Bondalapati and V. K. Prasanna, " Reconfigurable Meshes: Theory and Practice", Reconfigurable Architectures Workshop, International Parallel Processing Symposium, April 1997.

18 R. P. Sidhu, K. Bondalapati, S. Choi, and V. K. Prasanna, " Computation Models for Reconfigurable Machines", International Symposium on Field-Programmable Gate Arrays, February 1997.

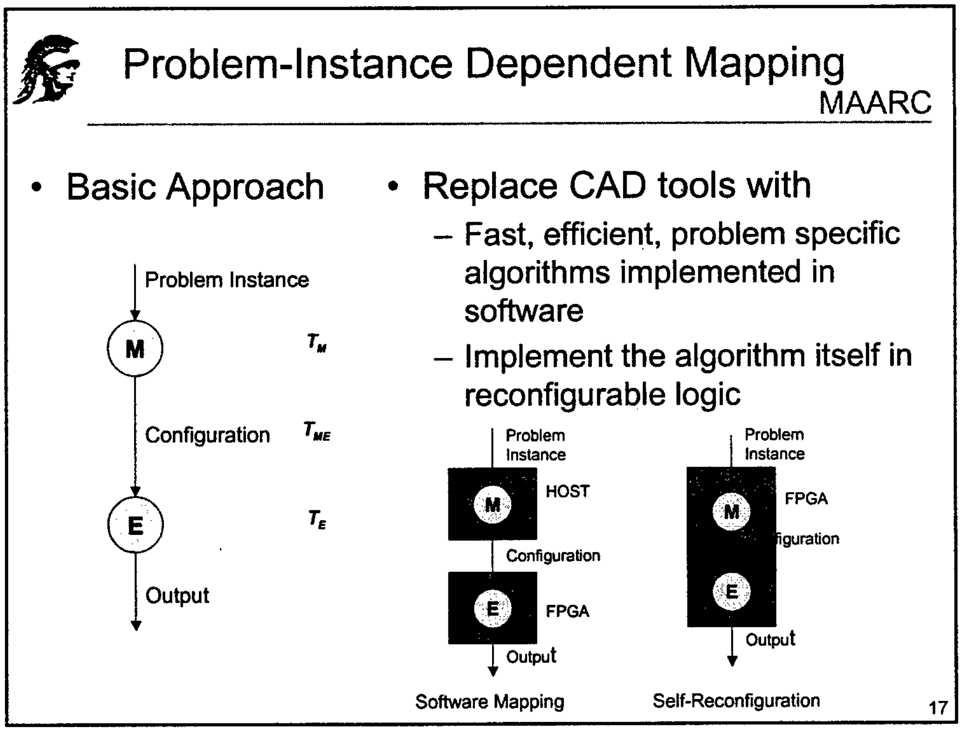
http://maarc.usc.edu



Key Research Areas Summary - YR 3

MAARC

- Problem-Instance Dependent Mapping
  - Reetinder Sidhu
- Dynamic Logic Synthesis for Reconfigurable HW
  - Andreas Dandalis
- HySAM Model and Dynamic Precision Management
  - Kiran Bondalapati
- An Interpretive Simulation and Visualization Environment for Dynamically Reconfigurable Architectures (DRIVE): Overview and Demo
  - Kiran Bondalapati



Problem-Instance Dependent Mapping

MAARC

- Basic Approach
- Replace CAD tools with
  - Fast, efficient, problem specific algorithms implemented in software
  - Implement the algorithm itself in reconfigurable logic

Software Mapping

Self-Reconfiguration

Problem-Instance Dependent Mapping

MAARC

- 3 orders of magnitude speedup in mapping time
- Order of magnitude speedup in overall execution time compared to a microprocessor implementation
- Successful validation of feasibility of our approach
  - Dynamic String Matching

Approach	$T_M + T_{ME} + T_E$	**Speedup**
Proposed	1.8 ms	**1.0**
CAD tool mapping	76.0 s	**$\approx 10^5$**
Software mapping	21.8 ms	**12.1**
Sun Ultra 1	30.0 ms	**16.6**

Overall Execution Time

Text size *n*=10^4

Pattern size *m*=8

Dynamic Logic Synthesis for Reconfigurable Hardware

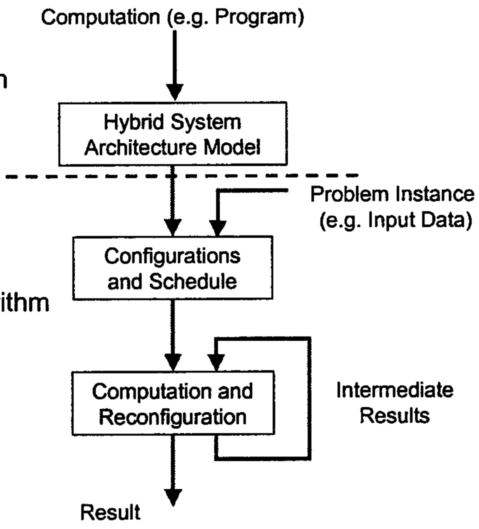
MAARC

- Application developer "sees" the device and architectural features in the algorithm design phase

  Off-line process

- Runtime interaction between algorithm and hardware

  Run-time mapping


Computation (e.g. Program)
Hybrid System Architecture Model
Problem Instance (e.g. Input Data)
Configurations and Schedule
Computation and Reconfiguration
Intermediate Results
Result

Dynamic Logic Synthesis for Reconfigurable Hardware

MAARC

- Case study: graph problems
- 6 orders of magnitude speedup in overall execution time
  - compared with the state-of-the-art (MIT DCS)
- Order of magnitude speedup in overall execution time
  - compared with uniprocessor implementation

Dynamic Precision Management

MAARC

- Dynamic precision
  - Modify precision on the fly
  - Match implementation to algorithm requirements
  - Reconfigurable architectures can support dynamic precision
- Lower precision requires less resources
  - Logic area
  - Execution time
  - Power consumption
- Run-time precision management
  - dynamic modification
  - algorithmic optimization

Example: Mapping onto XC6200

MAARC

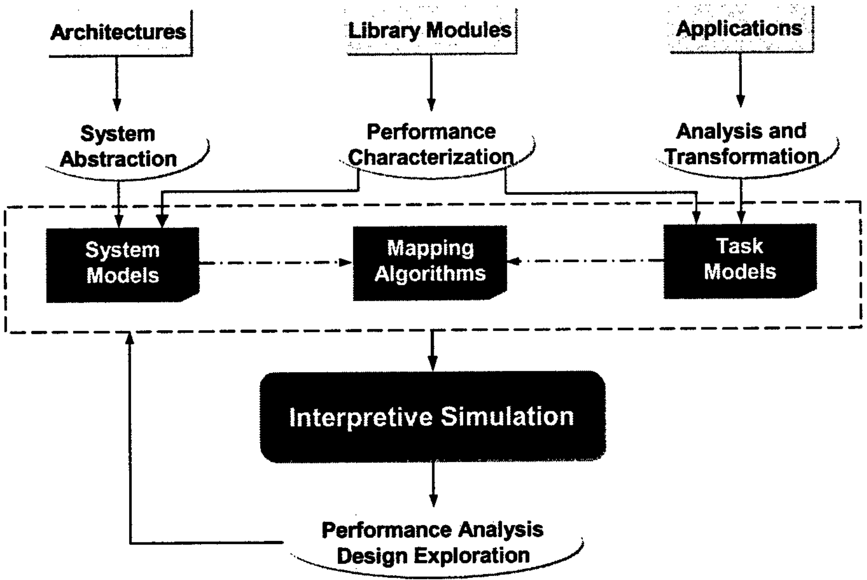
Mapping a multiply operation in a loop computation

Algorithm	Execution Time (*ns*)	Reconfig. Time (*ns*)	Total Time (*ns*)
Raw	655360	20480	675840
Static	532480	17920	550400
Greedy	468010	56320	524330
DPMA	471160	33280	504440
DPMA-run	409600	15360	424960

More than 30% improvement for one multiply operation

Interpretive Simulation Framework

MAARC



Project Status

MAARC

- Models
  - Reconfigurable Mesh, HySAM, RCSP
- Scalable Algorithms
  - Dynamic Precision Management, FFT, 2D-DCT, Geometric Hashing, Symbolic Search, String Matching, Genetic Programming, Graph Problems
- Compiler Optimization Techniques (NYU)
  - Adaptive-EPIC Architectures
- DRIVE Software
  - Interpretive Simulation and Visualization Environment for Dynamically Reconfigurable Architectures

Completion by September 9

Annual Review Outline

MAARC

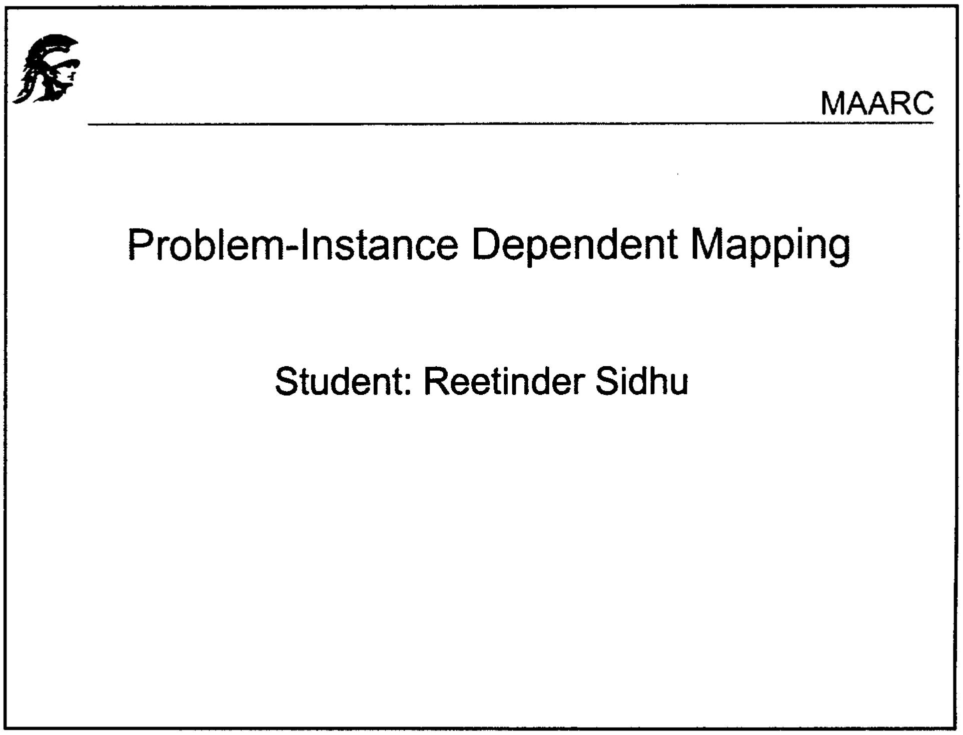
- USC Efforts
  - Project Overview 15'
  - Year 3 Key Technical Accomplishments
    - Problem-Instance Dependent Mapping (R.P.S. Sidhu) 20'
    - Dynamic Logic Synthesis for Reconfigurable HW (A. Dandalis) 20'

Break 15'

    - HySAM Model and Dynamic Precision Management (K. Bondalapati) 30'
    - An Interpretive Simulation and Visualization Environment for Dynamically Reconfigurable Architectures (DRIVE): Overview and Demo (K. Bondalapati) 20'

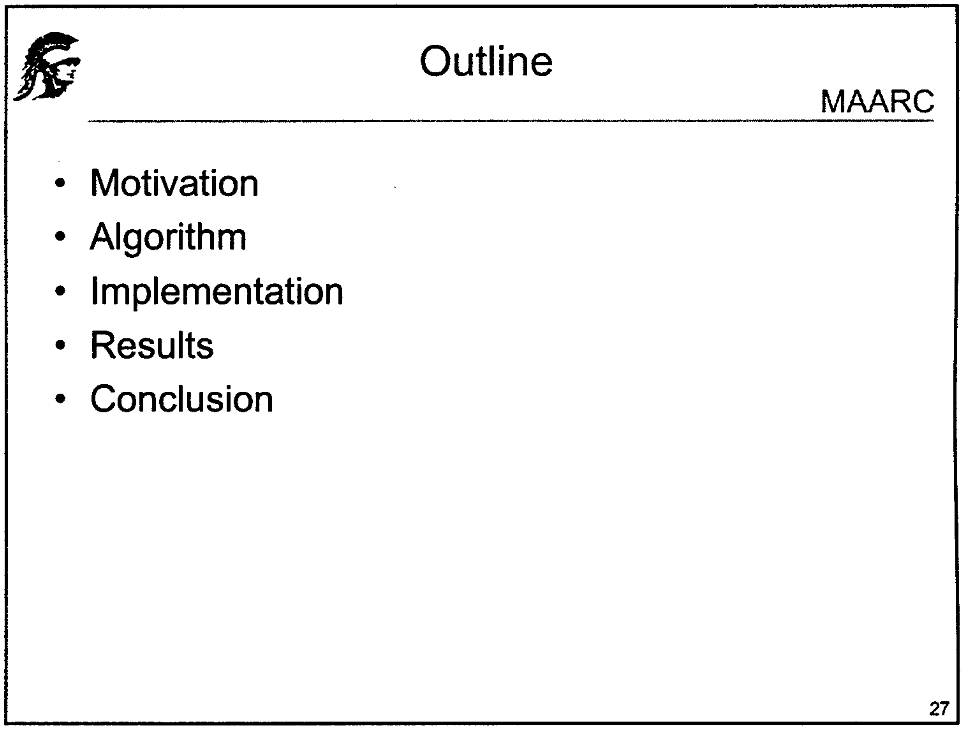
Break 15'

- NYU Efforts 60'
- Discussion 15'



Problem-Instance Dependent Mapping

Student: Reetinder Sidhu



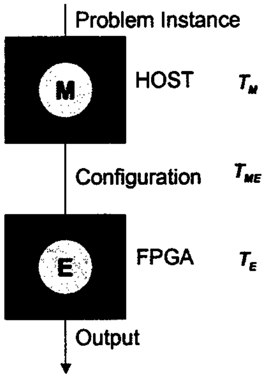
Outline

- Motivation
- Algorithm
- Implementation
- Results
- Conclusion

Problem-Instance Dependent Logic

MAARC

- FPGAs can outperform ASICs only if logic mapped onto them is optimized for each problem instance



Problem	T_M	T_{ME}	T_E
Satisfiability	2904 s	1-10 s	566 s
Shortest path	14400 s	1-10 s	100 us
Text filtering	0.16 s	3 s	50 ms

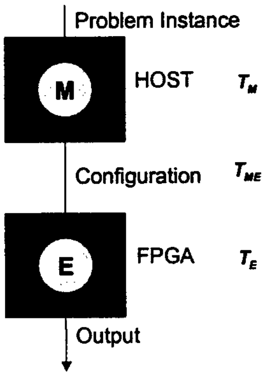
No CAD tools at runtime

MAARC

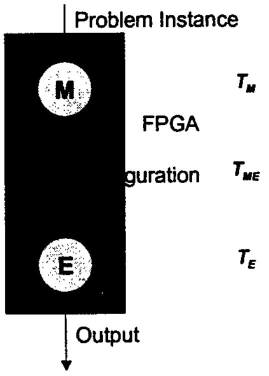
- CAD tools can be used to offline (compile time) to generate optimized logic
- No CAD tools at runtime

Replace CAD tools with what?

- Fast, efficient, problem specific algorithms implemented in software



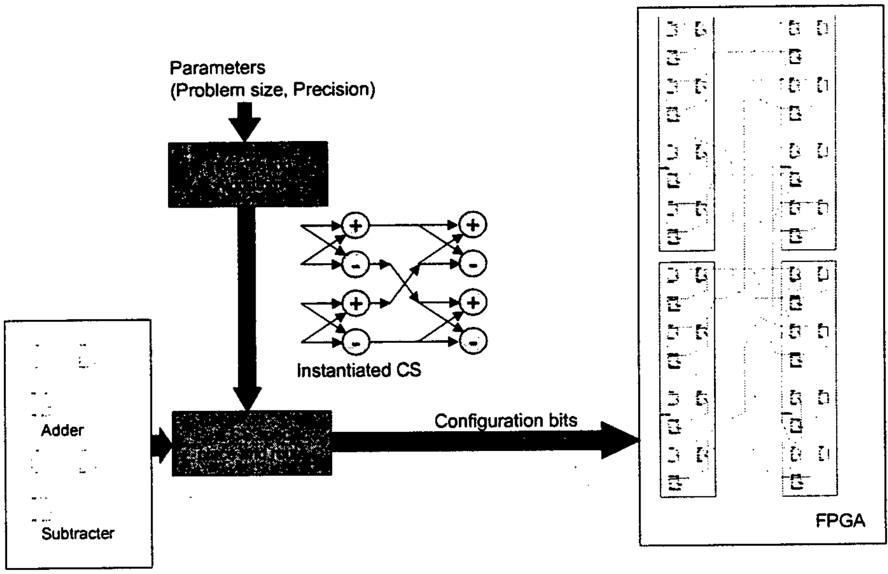
- Implement the algorithm itself in reconfigurable logic



Outline

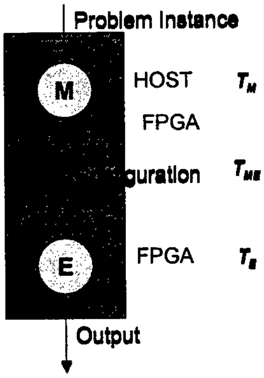
- Motivation
- Algorithm
- Implementation
- Results
- Conclusion

Proposed Approach: Parameterized Computation Structures (PCS)



KMP Algorithm

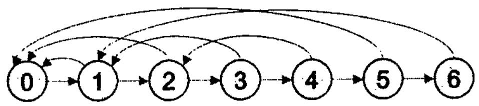
- Problem: Find all occurrences of text *T* (length *n*) in pattern *P* (length *m*)
- KMP (Knuth, Morris, Pratt) algorithm searches in *O(m+n)* time
  - Phase I: Construct FSM by looking at the pattern in O(m) time
  - Phase II: Search text using FSM in O(n) time



KMP Algorithm

MAARC

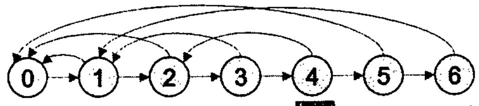
- Phase I (FSM construction)



Pattern a b a b c a *match*

PatternPatternPattern b b b b a a

- Phase I (Text search)



Pattern	a	b	a	b		a	*match*
Text	a	b	a	b			

Outline

MAARC

- Motivation
- Algorithm
- Implementation
- Results
- Conclusion

Implementation Approach

MAARC

- Clock cycle level analysis of proposed multicontext FPGA implementation to obtain I

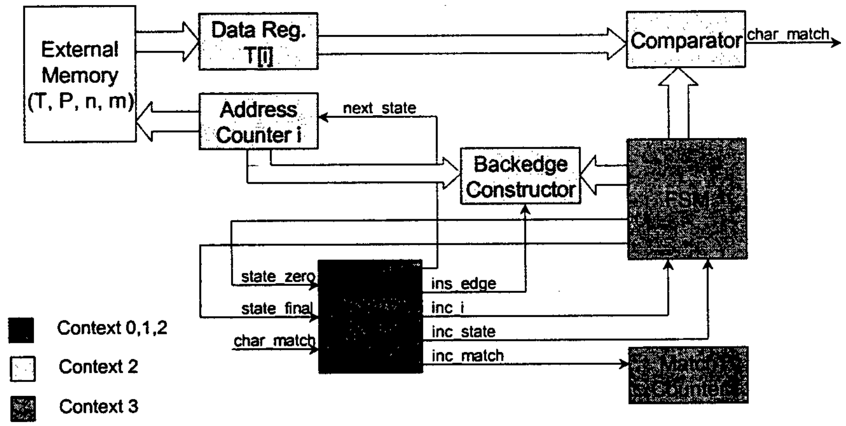
T_M, T_{ME} and T_E in terms of

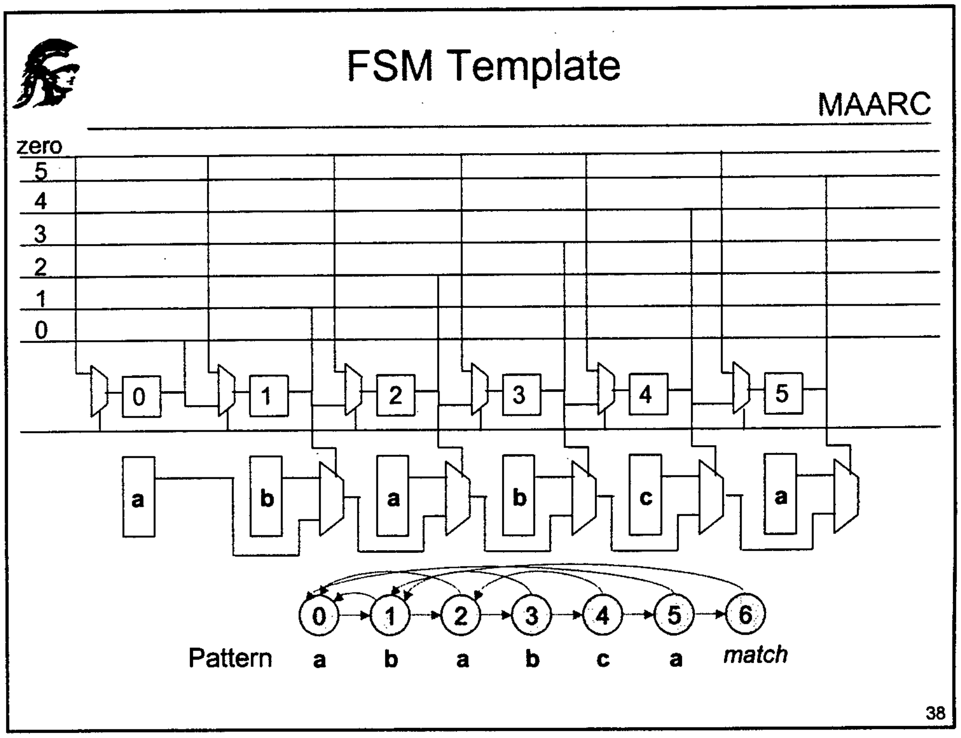
t_{clk}	Clock cycle time
t_{cm}	Config. memory access time
t_{em}	External memory access time
m	Pattern length
n	Text length

- Implemented most logic on a Xilinx XC 6216 to obtain above parameters

Implementation

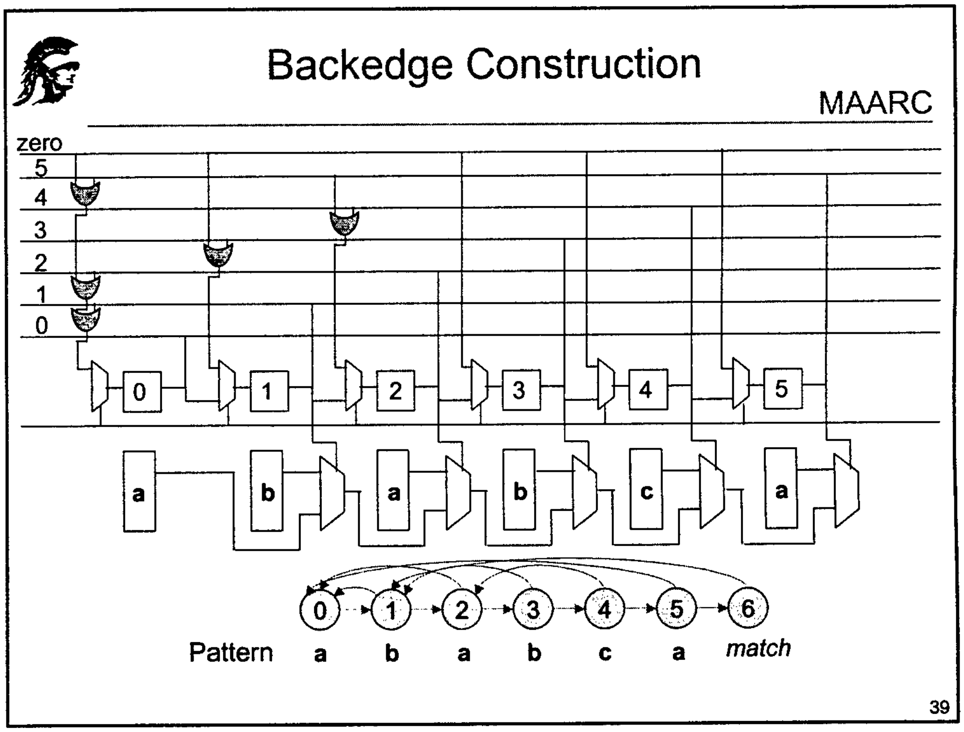
MAARC





FSM Template

MAARC

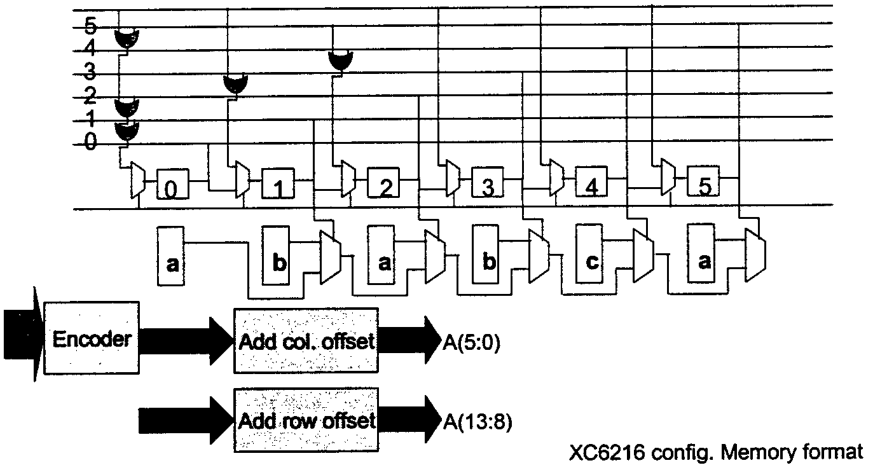


Backedge Construction

MAARC

OR-gate Insertion Logic

MAARC



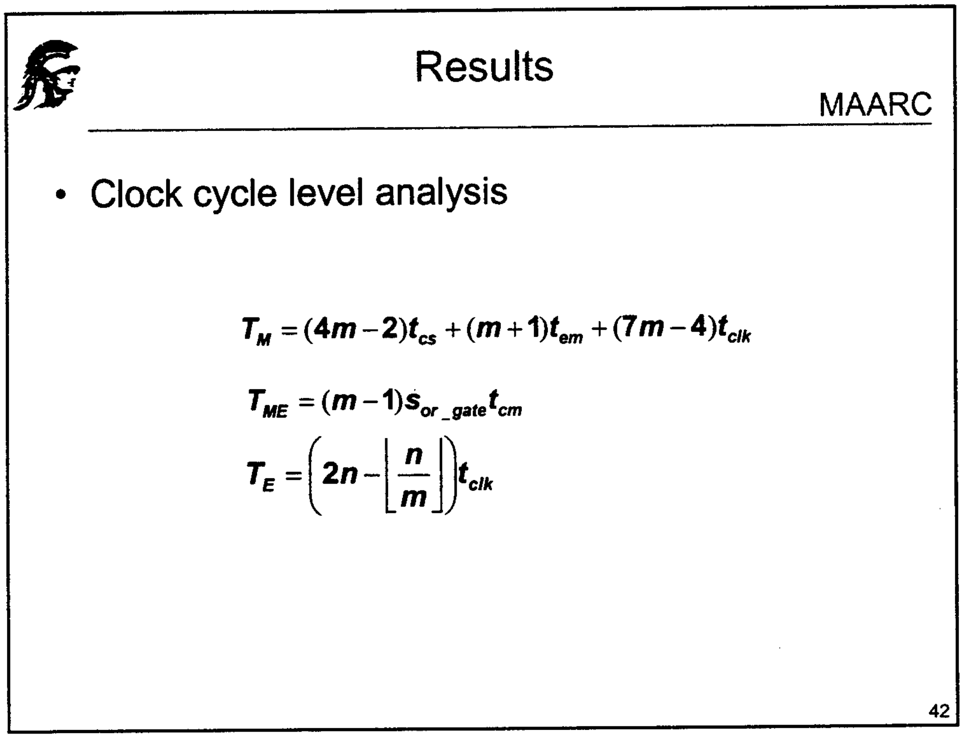
XC6216 config. Memory format

OR-gate address generation logic

Outline

MAARC

- Motivation
- Algorithm
- Implementation
- Results
- Conclusion



Results

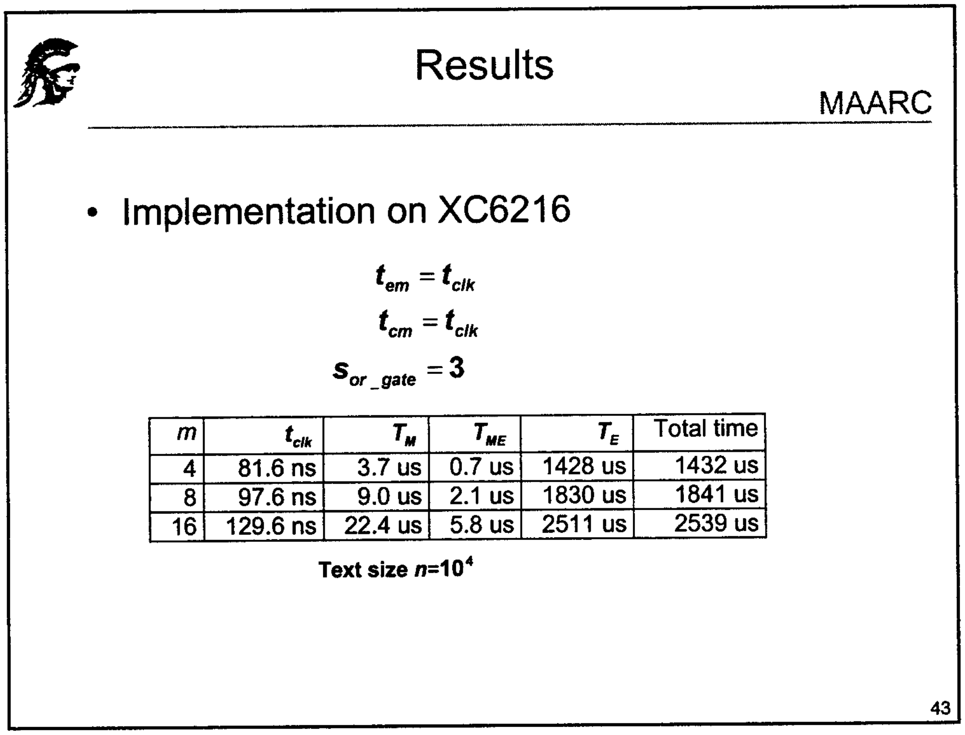
MAARC

- Clock cycle level analysis

$$T_M = (4m-2)t_{cs} + (m+1)t_{em} + (7m-4)t_{clk}$$

$$T_{ME} = (m-1)s_{or\_gate}t_{cm}$$

$$T_E = \left(2n - \left\lfloor \frac{n}{m} \right\rfloor\right)t_{clk}$$



Results

MAARC

- Implementation on XC6216

$$t_{em} = t_{clk}$$

$$t_{cm} = t_{clk}$$

$$s_{or\_gate} = 3$$

m	t_{clk}	T_M	T_{ME}	T_E	Total time
4	81.6 ns	3.7 us	0.7 us	1428 us	1432 us
8	97.6 ns	9.0 us	2.1 us	1830 us	1841 us
16	129.6 ns	22.4 us	5.8 us	2511 us	2539 us

Text size n=10^4

Performance Comparison (Mapping Time)

MAARC

- CAD tool mapping
  - Place and route using XACT 6000 for each pattern
- Software mapping
  - KMP phase I in software
- Proposed approach

Approach	T_M	T_{ME}	$T_M + T_{ME}$	**Speedup**
Multicontext FPGA	9.0 us	2.1 us	11.1 us	**1.0**
CAD tool mapping	76 s	1 ms	76 s	**~6x10^6**
Software mapping	20 ms	1 ms	21 ms	**1892**

Pattern size *m*=8

Performance Comparison (Total Time)

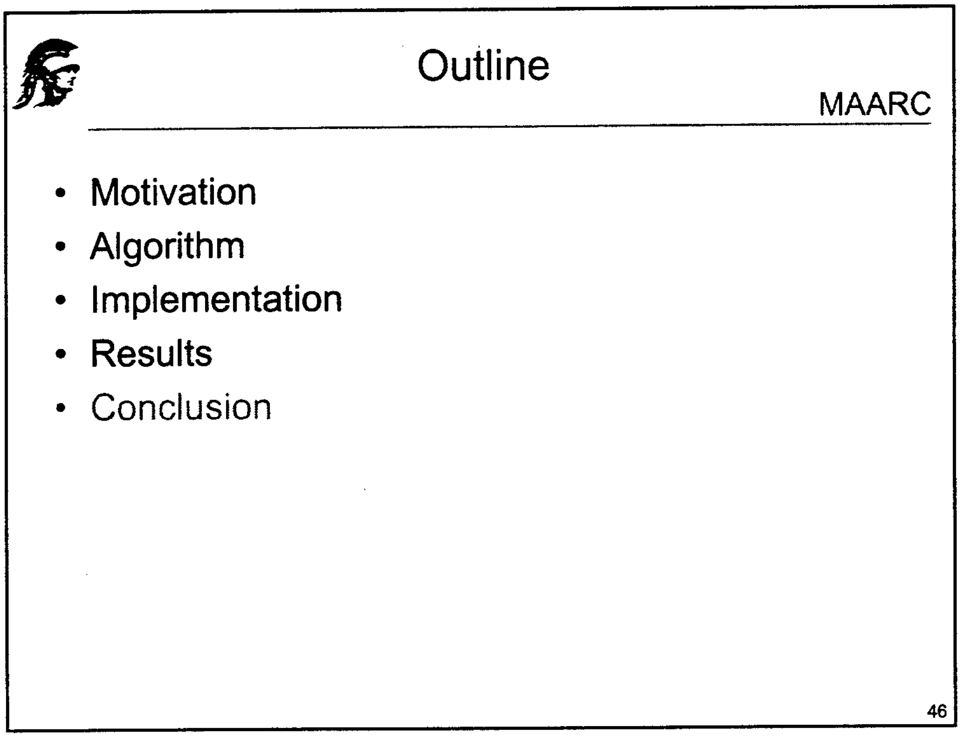
MAARC

- CAD tool mapping
- Software mapping
- Proposed approach
- Sun Ultra I Model 140
  - C implementation of KMP algorithm

Approach	$T_M + T_{ME} + T_E$	**Speedup**
Proposed	1.8 ms	**1.0**
CAD tool mapping	76.0 s	$\approx 10^5$
Software mapping	21.8 ms	**12.1**
Sun Ultra 1	30.0 ms	**16.6**

Text size *n*=10^4

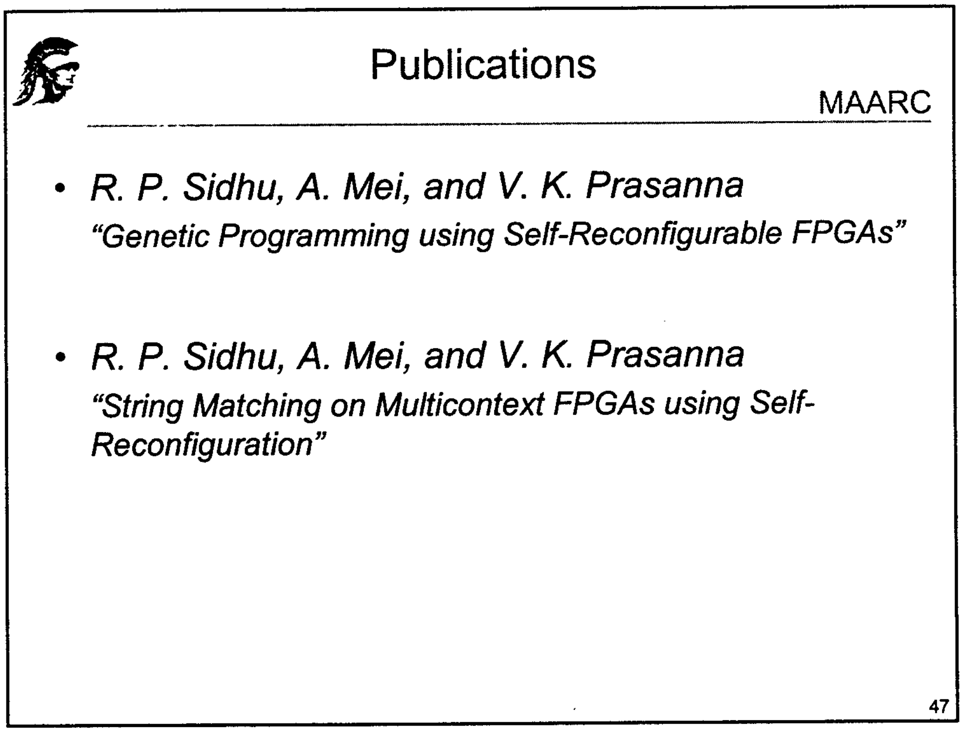
Pattern size *m*=8



Outline

MAARC

- Motivation
- Algorithm
- Implementation
- Results
- Conclusion



Publications

MAARC

- *R. P. Sidhu, A. Mei, and V. K. Prasanna*
  "Genetic Programming using Self-Reconfigurable FPGAs"
- *R. P. Sidhu, A. Mei, and V. K. Prasanna*
  "String Matching on Multicontext FPGAs using Self-Reconfiguration"

Conclusion

- High mapping and reconfiguration times
- 3 orders of magnitude speedup in mapping time
- Order of magnitude speedup in overall execution time compared to a microprocessor implementation
- Successful validation of approach feasibility

Dynamic Logic Synthesis for Reconfigurable Hardware

Andreas Dandalis

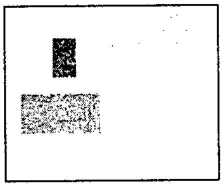
Outline

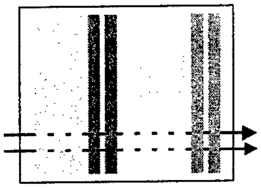
MAARC

- Introduction
  - Dynamic Logic Synthesis for Reconfigurable Hardware
- Accomplishments (Year III)
  - Mapping Graph Problems
  - Example
  - Summary
- Conclusions

Computation Structures on FPGAs?

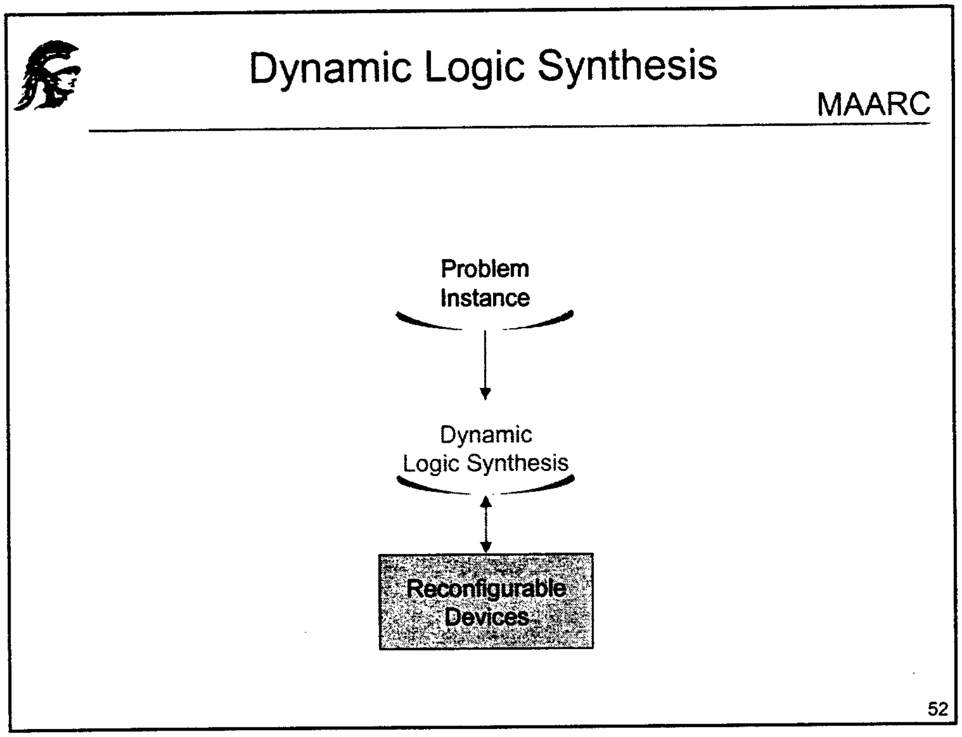
MAARC





- Library-based modules configurations
- Problem
  - optimization across module boundaries
  - reconfiguration cost

- Library-based array configurations
  - well specified boundaries
  - less reconfiguration cost ?
- Problem
  - PEs computational power ?
  - space vs performance ?



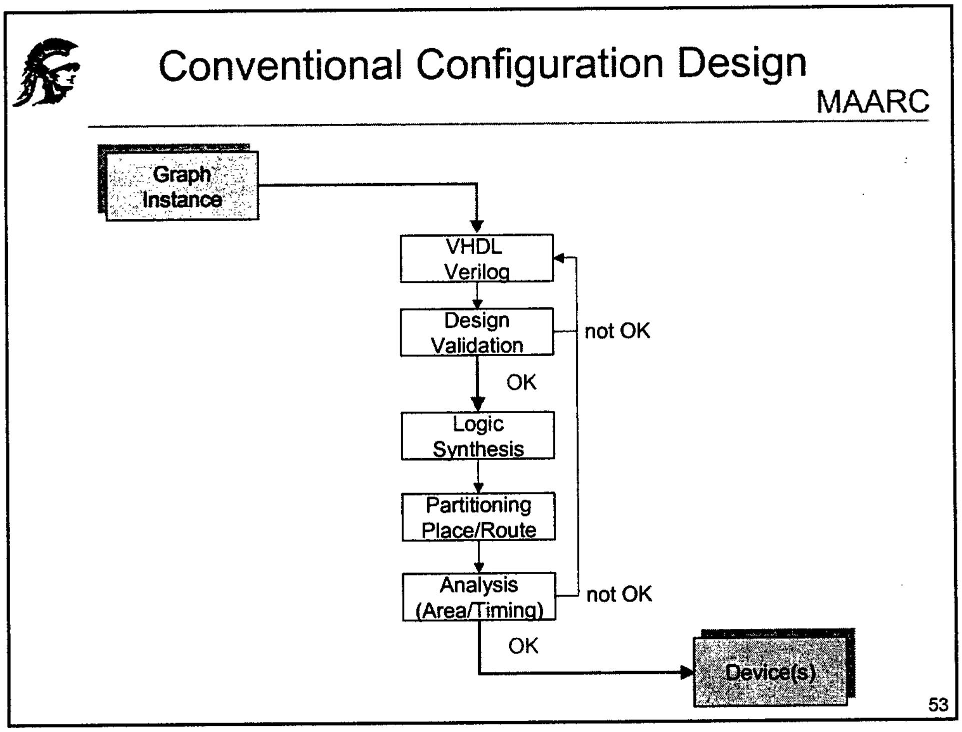
Dynamic Logic Synthesis

MAARC

Problem Instance

Dynamic Logic Synthesis

Reconfigurable Devices



Conventional Configuration Design

MAARC

Graph Instance

VHDL Verilog

Design Validation

not OK

OK

Logic Synthesis

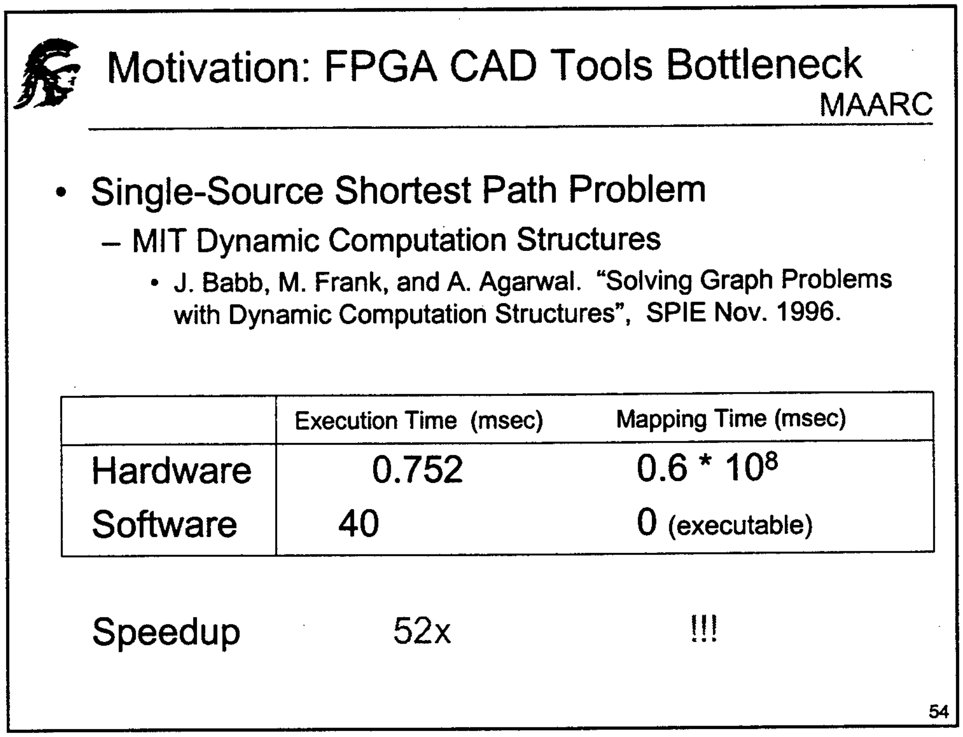
Partitioning Place/Route

Analysis (Area/Timing)

not OK

OK

Device(s)



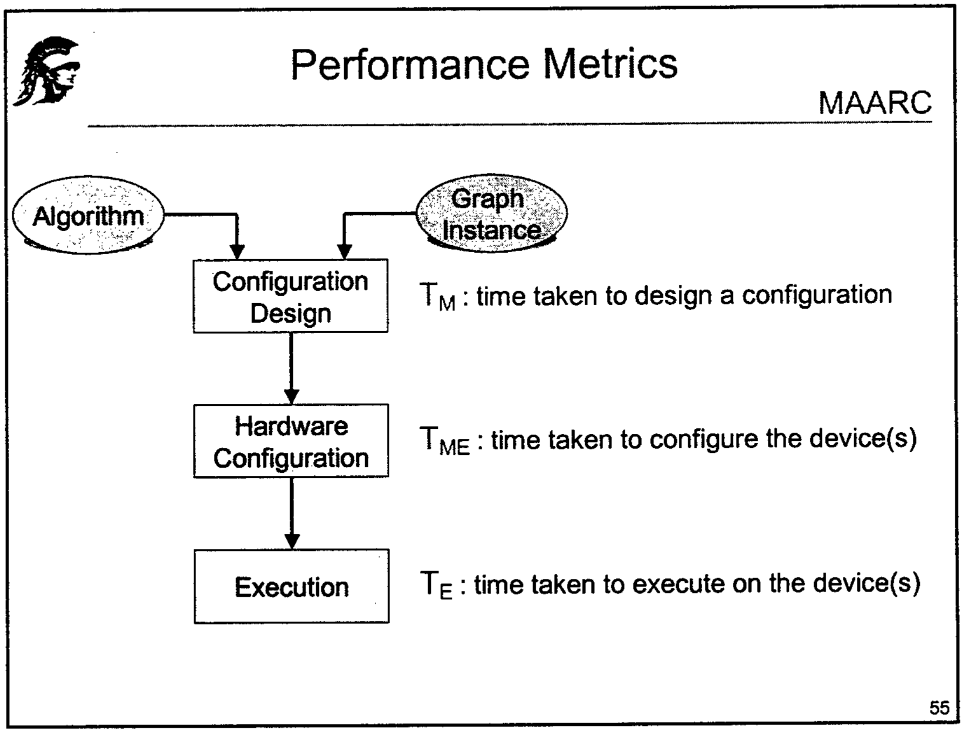
Motivation: FPGA CAD Tools Bottleneck

MAARC

- Single-Source Shortest Path Problem
  - MIT Dynamic Computation Structures
    - J. Babb, M. Frank, and A. Agarwal. "Solving Graph Problems with Dynamic Computation Structures", SPIE Nov. 1996.

	Execution Time (msec)	Mapping Time (msec)
Hardware	0.752	$0.6 * 10^8$
Software	40	0 (executable)

Speedup	52x	!!!



Performance Metrics

MAARC

Algorithm

Graph Instance

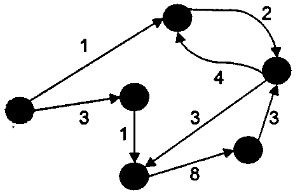
Configuration Design — T_M : time taken to design a configuration

Hardware Configuration — T_{ME} : time taken to configure the device(s)

Execution — T_E : time taken to execute on the device(s)

An Example: Single-Source Shortest Path Problem

MAARC

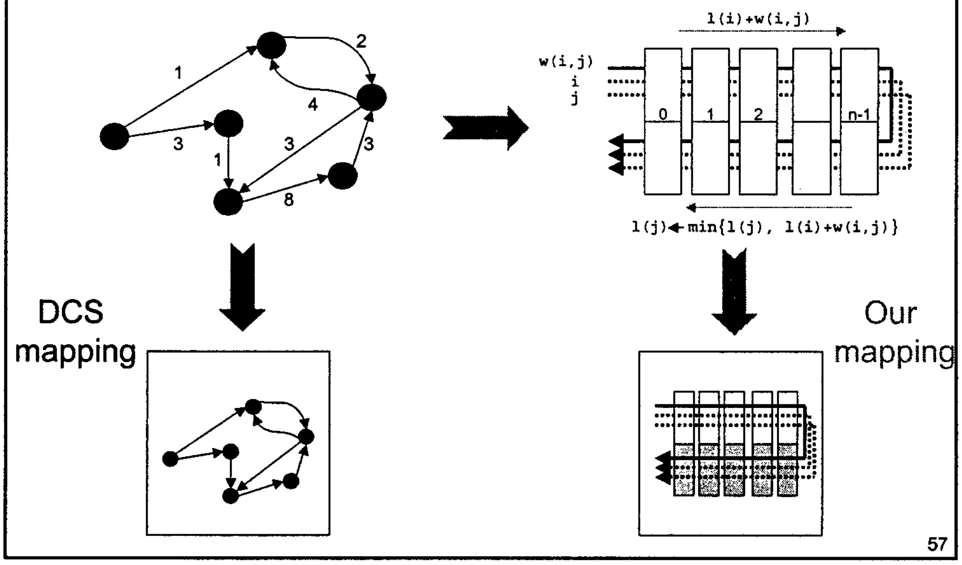


"Bellman-Ford" Loop

```
FOR k=1..n-1
    DO FOR each edge (i,j)
        DO l(j)← min{l(j), l(i)+w(i,j)}
```

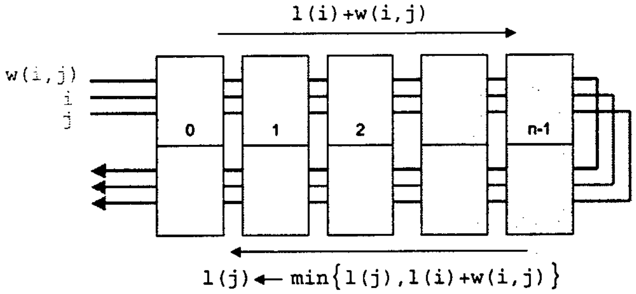
An Example: Single-Source Shortest Path Problem

MAARC



Skeleton

MAARC

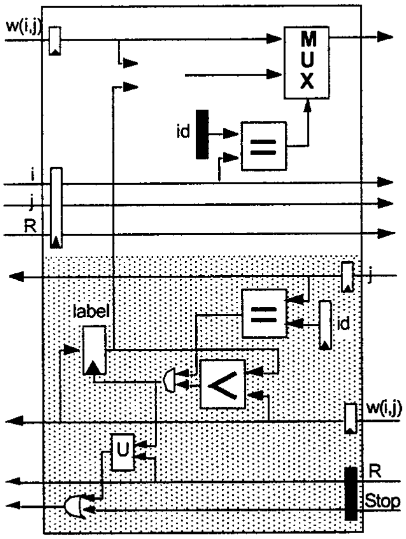


- Parameters
  - # of vertices *n*
  - data precision *p*
  - I/O bandwidth available *B*

Parameterized modules for XC6xxx

MAARC

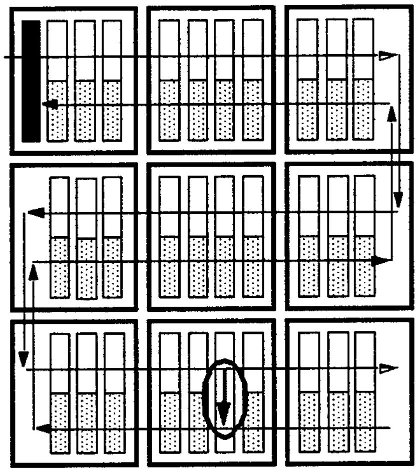
- Width
  - (p+⌈log n⌉)
- Height
  - 4p+2⌈log n⌉+10
- Clock rate
  - 15 MHz
    - p=16, log n =16
  - 25 MHz
    - p=8, log n=16



Placement and Routing for XC6xxx

MAARC

- Placement
  - pre-defined
- Routing
  - nearest neighbors



Run-time Adaptation

MAARC

- Module-level
  - data precision
  - width of { i, j } buses
- Layout-level
  - # of modules
  - placement

Implementation

- Platform
  - Xilinx XC 6200 based
  - VCC HOT Works PCI board
- Tools
  - Velab VHDL compiler
  - XACT6000 (place-and-route)
- Area
  - the same as estimated
- Clock rate
  - 14 MHz (p=16, log n=16)
  - 23 MHz (p= 8, log n=16)

Comparison with MIT DCS*

Problem Size	Clock Rate MHz		T_E µsec		$T_M + T_{ME}$		$T_E + T_M + T_{ME}$
	DCS	Our	DCS	Our	DCS	Our	Speedup
16 x 64	1.79	14	8.94	22.95	~ 4 h	~ 22 msec	6.5×10^6
64 x 256	1.14	14	56.14	84.66	~ 4 h	~ 82 msec	1.7×10^6
128 x 515	0.78	14	164.10	213.98	~ 8 h	~ 161 msec	1.8×10^6
256 x 1140	0.34	14	752.94	528.39	~ 16 h	~ 319 msec	1.8×10^6

* "Solving graph problems with dynamic computation structures" J. Babb et al., SPIE, Nov. 1996

Qualitative Comparison

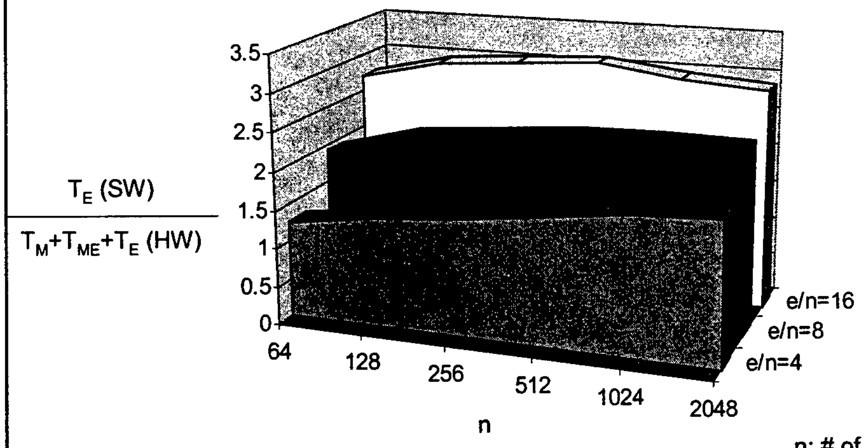
MAARC

- Our Mapping
  - module-based
    - regular layout
  - incremental designs
  - area/timing estimates
    - determined by the computation structure
  - # of iterations
    - height of the shortest path tree
  - negative cycle detection

- DCS Mapping
  - cell & wire-based
    - irregular layout
  - complete redesign
  - area/timing estimates
    - determined by "tools"
  - # of iterations
    - # of vertices
  - correct only for non-negative cycle graphs

Comparison with SW Implementation*

MAARC



n: # of vertices
e: # of edges

* SUN ULTRA1 64MB/143MHz

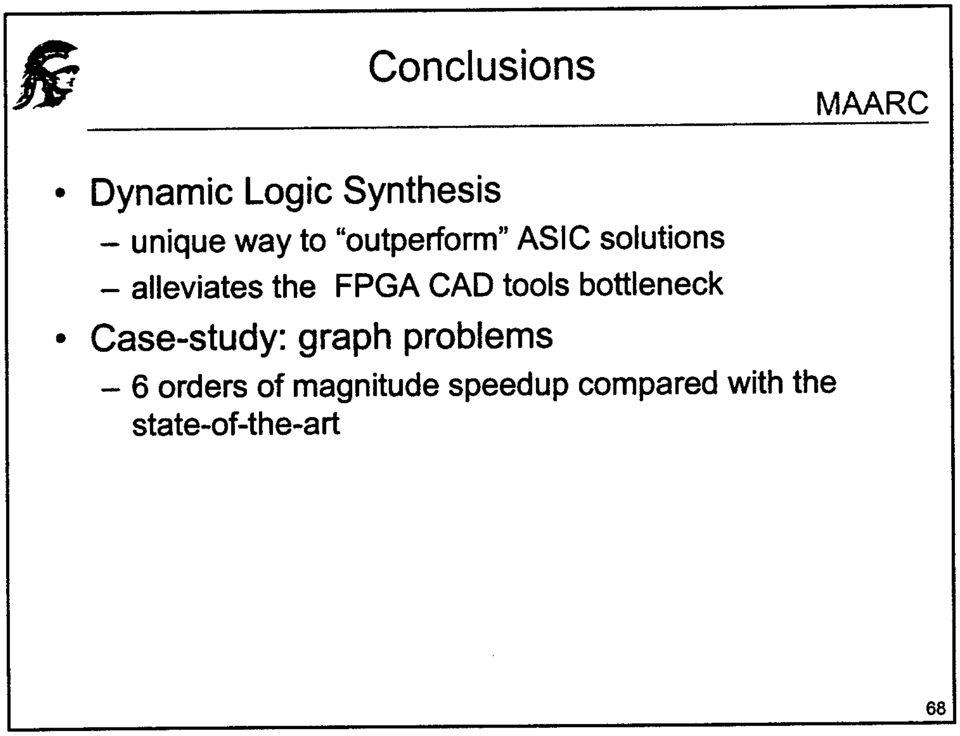
Implementation on VIRTEX

MAARC

Summary

MAARC

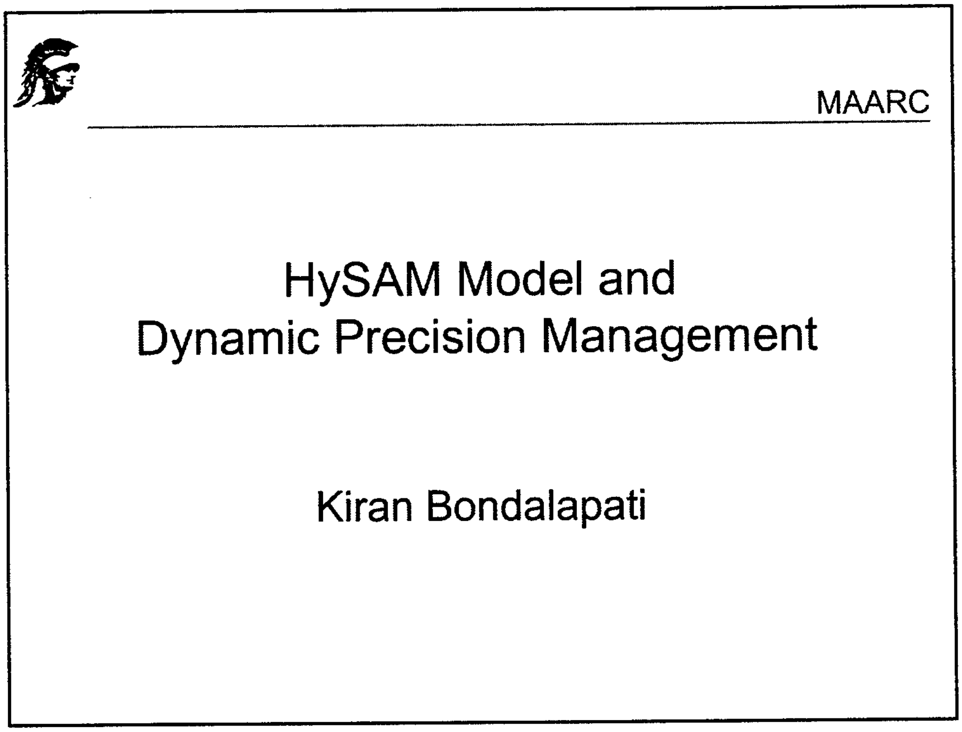
- *A. Dandalis, J. L. Gaudiot, and V. K. Prasanna*
  "Run-time Mapping of Graph-Problem Instances onto Reconfigurable Hardware"
- *A. Dandalis, A. Mei, and V. K. Prasanna*
  "Domain Specific Mapping for Solving Graph Problems on Reconfigurable Devices"
- *A. Dandalis*
  "Dynamic Logic Synthesis for Reconfigurable Hardware"



Conclusions

MAARC

- Dynamic Logic Synthesis
  - – unique way to “outperform” ASIC solutions
  - – alleviates the FPGA CAD tools bottleneck
- Case-study: graph problems
  - – 6 orders of magnitude speedup compared with the state-of-the-art



MAARC

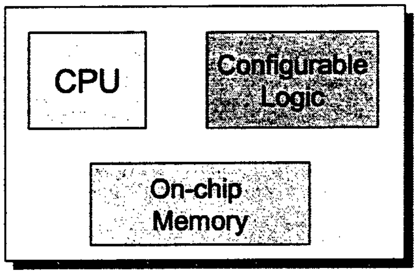
HySAM Model and Dynamic Precision Management

Kiran Bondalapati

Hybrid Architectures

MAARC

Hybrid Architectures

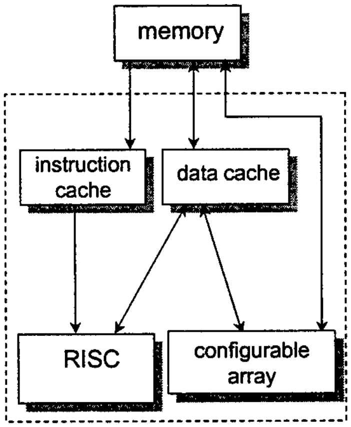


- Feasible architectures with the availability of nearly billion transistors on a chip
- Availability of on-chip configuration and data storage memory
- Potential for fast and dynamic reconfiguration

BRASS - Garp

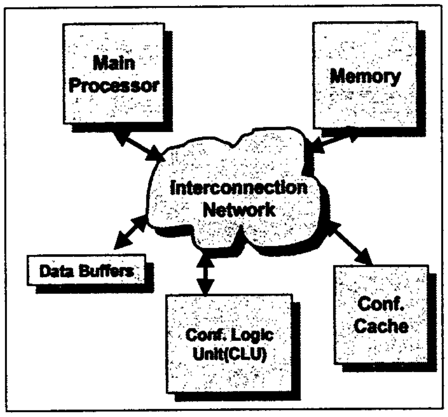
MAARC

- Reconfigurable array unit with a RISC processor
- Gate array of 32x24 logic blocks
- Partial configuration of gate array in row increments
- Configuration cache for fast reconfiguration
- 4 cycles on-chip and 12 cycle off-chip reconfiguration time



Hybrid System Architecture Model

MAARC



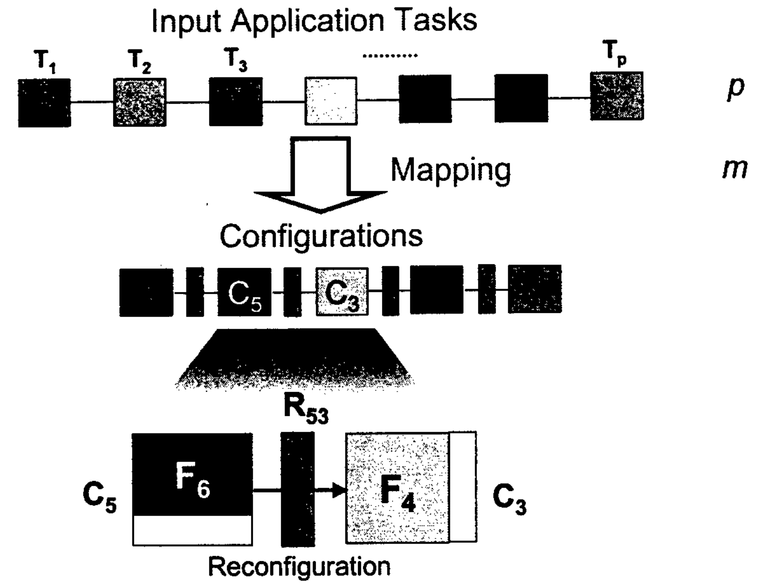
- Parameterized model
- Architecture independent algorithm development
- Algorithmic analysis of mapping techniques

CLU Functions and Configurations

MAARC

- Functions (F)
  - *Computational units (e.g. Add, Multiply, Select)*
  - *Library Modules*
- A Function can be executed by different Configurations
- Configurations (C)
  - *Area, Configuration time, Execution time, Precision, Power consumption, I/O requirement*
- t_{ij} - execution time for function F_i in configuration C_j
- R_{ij} - reconfiguration cost from C_i to C_j
  - *depends on both C_i and C_j*
  - *partial reconfiguration*
  - *reconfiguration cost matrix*

Tasks and Configurations



Input Application Tasks

Reconfiguration

Example : Garp Architecture Parameters

Function	Operation	Configuration	Conf. Time R_{0j}	Exec. Time t_{ij}

F_1	Multiplication(Fast)
	Multiplication(Slow)
		C_1
		C_2
			14.4 us
			6.4 us
				37.5 ns
				52.5 ns

F_2	Addition	C_3	1.6 us	7.5 ns
F_3	Subtraction	C_4	1.6 us	7.5 ns
F_4	Shift	C_5	3.2 us	7.5 ns

Example : XC 6200 Multipier Configurations

MAARC

For the multiplier function F_i
Precision is bit-sizes of the two inputs to multiplier

Configuration C_j	Precision $Pr(C_j)$	Conf. Time R_{0j}	Exec. Time t_{ij}
C_1	8 x 8	5120 ns	140 ns
C_2	8 x 16	10240 ns	250 ns
C_3	8 x 20	12800 ns	300 ns
C_4	8 x 24	15360 ns	400 ns
C_5	8 x 28	17920 ns	520 ns
C_6	8 x 32	20480 ns	640 ns

Outline

MAARC

- Introduction
- HySAM Model
- Variable Precision Computations
- Dynamic Precision Management
- Example Results

Precision Variation in Loop Computations

MAARC

Ex:
```
DO 10 I=1,N
   DO 20 J=1,N
     RSQ(J) = RSQ(J)+XDIFF(I,J)*YDIFF(I,J)
20   IF (MAXQ.LT.RSQ(J)) THEN
        MAXQ = RSQ(J)
10 VIRTXY = VIRTXY + MAXQ * SCALE(I)
```

- 8-bit inputs XDIFF(I,J) and YDIFF(I,J)
- MAXQ operand and * operation
  - precision changes with iterations of `I`
  - lower than maximum possible precision (for most iterations)

Variable Precision Computations

MAARC

- Precision requirement is lower than implemented
- Match implementation to algorithm requirements
- Less resources
  - Logic area
  - Execution time
  - Power consumption
- Run-time precision management
  - dynamic modification

Dynamic Precision Management

MAARC

- Precision variation analysis
  - Loop computations
  - Theoretical analysis
  - Run-time analysis
  - *precision variation curve*
- Utilize variable precision library
- Dynamic Precision Management Algorithm
- Optimal configuration sequence

Precision Variation in Loop Computations

MAARC

Ex:

```
      DO 10 I=1,N
         DO 20 J=1,N
           RSQ(J) = RSQ(J)+XDIFF(I,J)*YDIFF(I,J)
 20        IF (MAXQ.LT.RSQ(J)) THEN
              MAXQ = RSQ(J)
 10      VIRTXY = VIRTXY + MAXQ * SCALE(I)
```

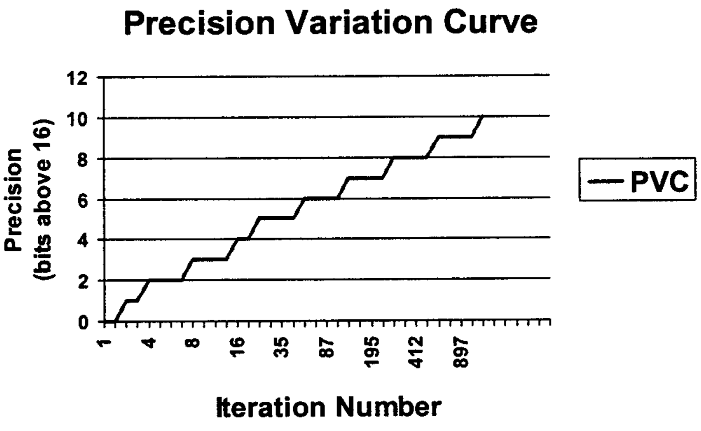
- MAXQ operand and * operation
  - precision changes with iterations of `I`
  - lower than maximum possible precision (for most iterations)

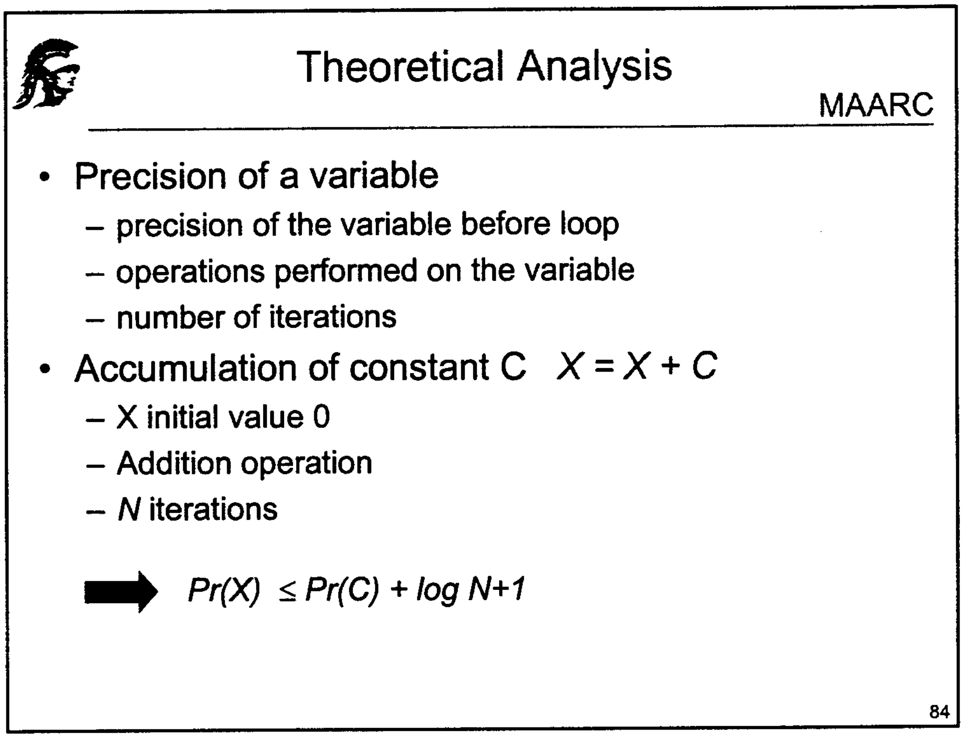
Does not change every iteration

Precision Variation Curve(PVC) for Loops

- *Precision Variation Curve*
  - Change in required precision of an operand or an operation over the given iteration space
- *PVC* Points
  - Iterations in which precision changes
  - Subset of iteration space of loop
- Definition
  - $<L_i,P_i>$ $1 \le i \le u+1$, $L_{u+1} = N$
  - P_i is the minimum precision required to execute iterations $L_i \ldots L_{i+1}-1$
  - N = number of iterations

Precision Variation Curve



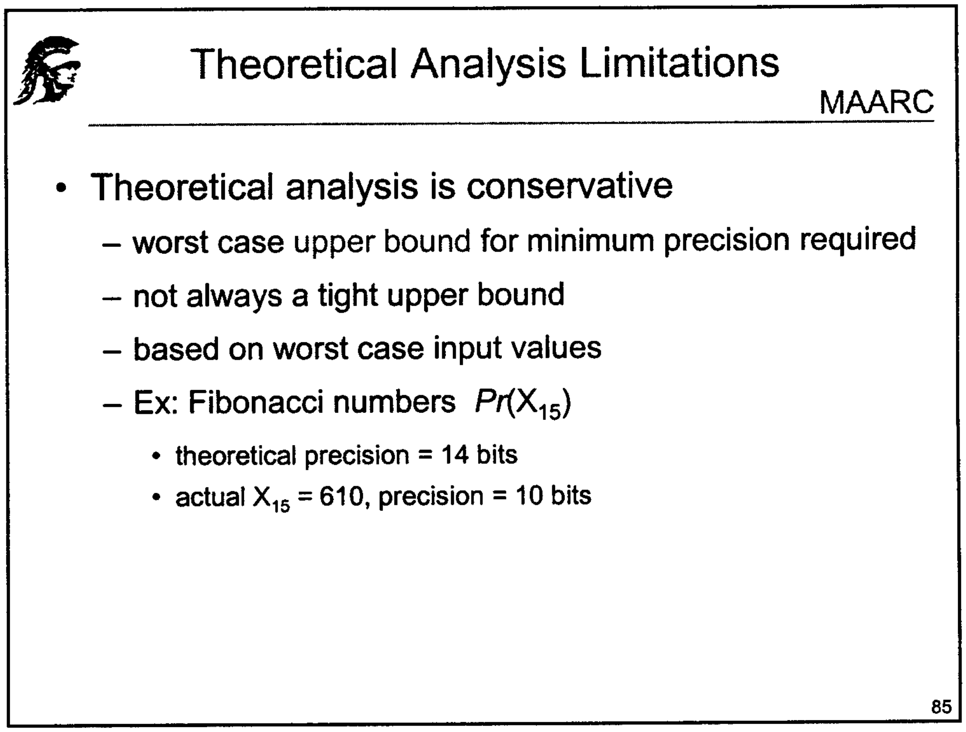


Theoretical Analysis

MAARC

- Precision of a variable
  - precision of the variable before loop
  - operations performed on the variable
  - number of iterations
- Accumulation of constant C $X = X + C$
  - X initial value 0
  - Addition operation
  - N iterations

➡ $Pr(X) \le Pr(C) + \log N+1$



Theoretical Analysis Limitations

MAARC

- Theoretical analysis is conservative
  - worst case upper bound for minimum precision required
  - not always a tight upper bound
  - based on worst case input values
  - Ex: Fibonacci numbers $Pr(X_{15})$
    - theoretical precision = 14 bits
    - actual X_{15} = 610, precision = 10 bits

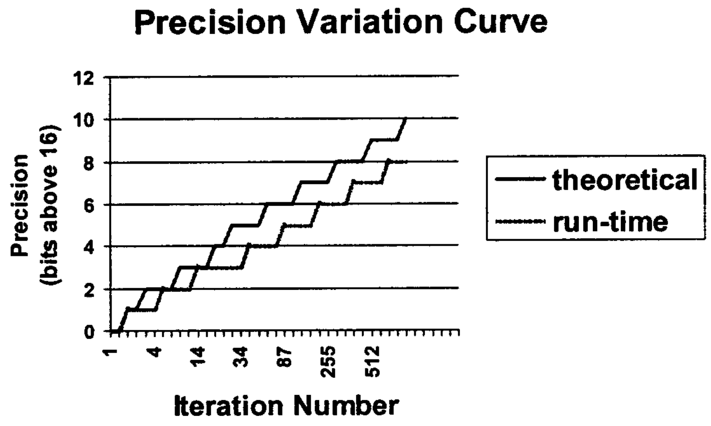
Run-time Analysis

MAARC

- Run-time profiling
  - Precision analysis
  - Instrumented code
  - Simulations with typical data sets
  - Measure precision in all iterations
  - Estimate required precision
- Execution
  - Use estimated precision values
  - run-time verification
    - low cost precision check
    - check not in critical path

Precision Variation Curve

MAARC



Precision Management Problem

MAARC

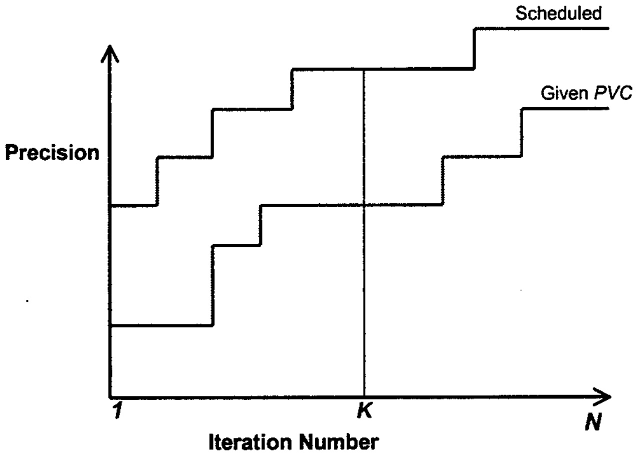
- Given
  - *PVC* for a given operation in the loop
- Find
  - A valid optimal schedule which minimizes total execution time
- Valid schedule
  - satisfies the precision requirements of the computation
- Total execution time
  - execution time + reconfiguration time

Precision Management Problem

MAARC

Valid schedule for a given *PVC*

- For every iteration ***K* (*1* ≤ *K* ≤ *N*)** the precision of scheduled configuration is less than the required precision given by the ***PVC***



Assumptions

- Higher precision requires more resources
  - execution time
  - logic area
- Monotonic variation in precision
  - several image processing and signal processing applications
  - split non-monotonic *PVC* into monotonic subsequences
- Optimal solution for the given PVC
  - near optimal if actual precision variation is different

Dynamic Precision Management Algorithm (DPMA)

Lemma: The reconfiguration points are a subset of the *PVC* points

Greedy algorithm

- best configuration for each *PVC* interval
- sub-optimal schedule

DPMA algorithm

- Dynamic programming based
- Explores non-optimal configurations
  - for some iterations
  - reduces reconfiguration overhead
- $O(um^2)$ complexity
  - u = # of *PVC* points, m = # of configurations

Example :
XC 6200 Multipier Configurations

MAARC

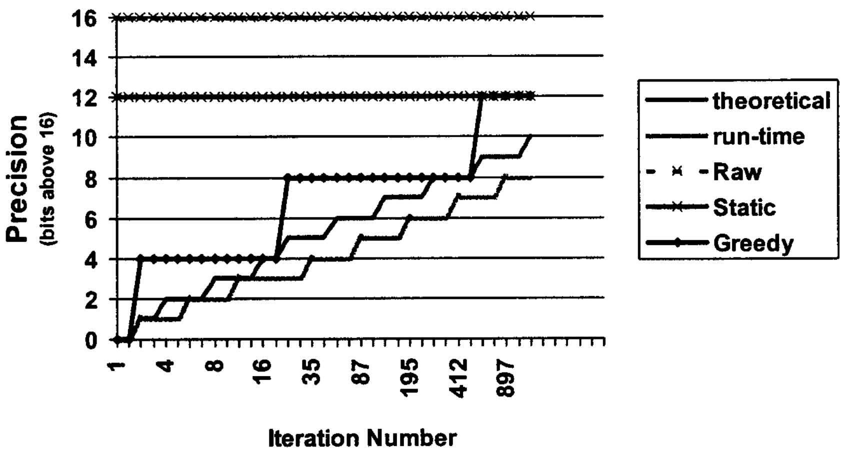
For the multiplier function F_i
Precision is bit-sizes of the two inputs to multiplier

Configuration C_j	Precision $Pr(C_j)$	Conf. Time R_{0j}	Exec. Time t_{ij}
C_1	8 x 8	5120 ns	140 ns
C_2	8 x 16	10240 ns	250 ns
C_3	8 x 20	12800 ns	300 ns
C_4	8 x 24	15360 ns	400 ns
C_5	8 x 28	17920 ns	520 ns
C_6	8 x 32	20480 ns	640 ns

Results: Ad-hoc approaches

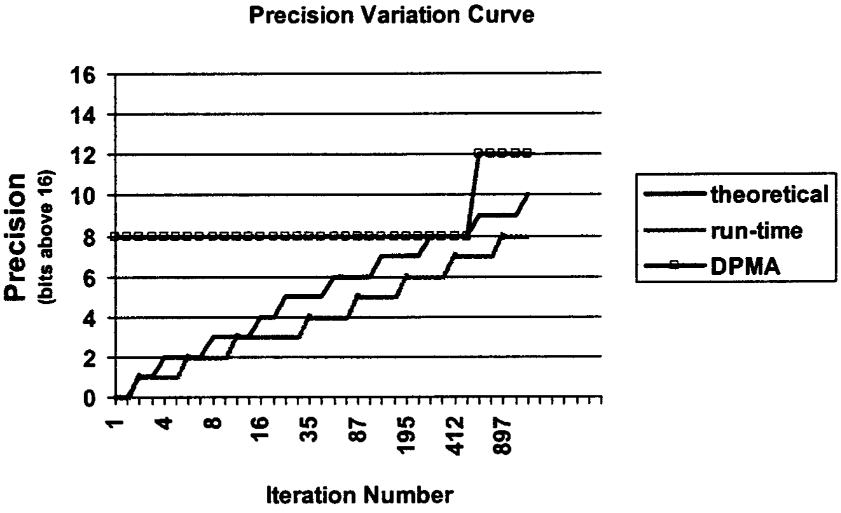
MAARC

Precision Variation Curve



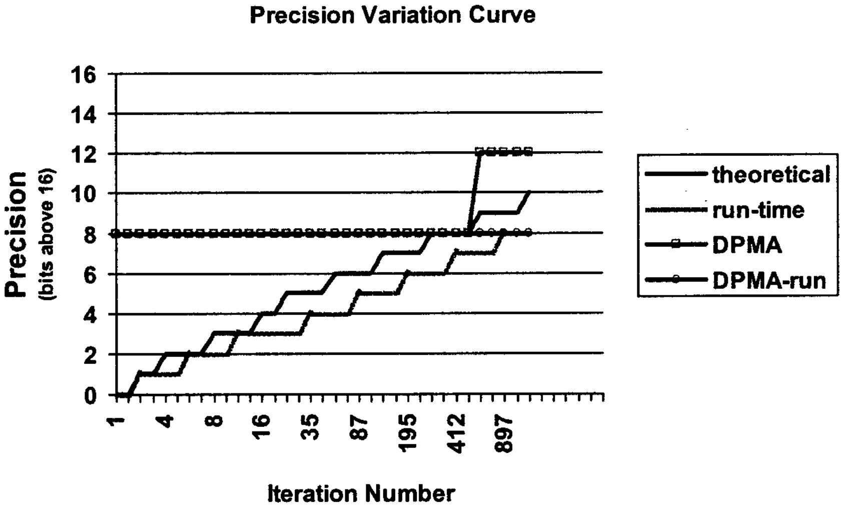
Results: DPMA Approach

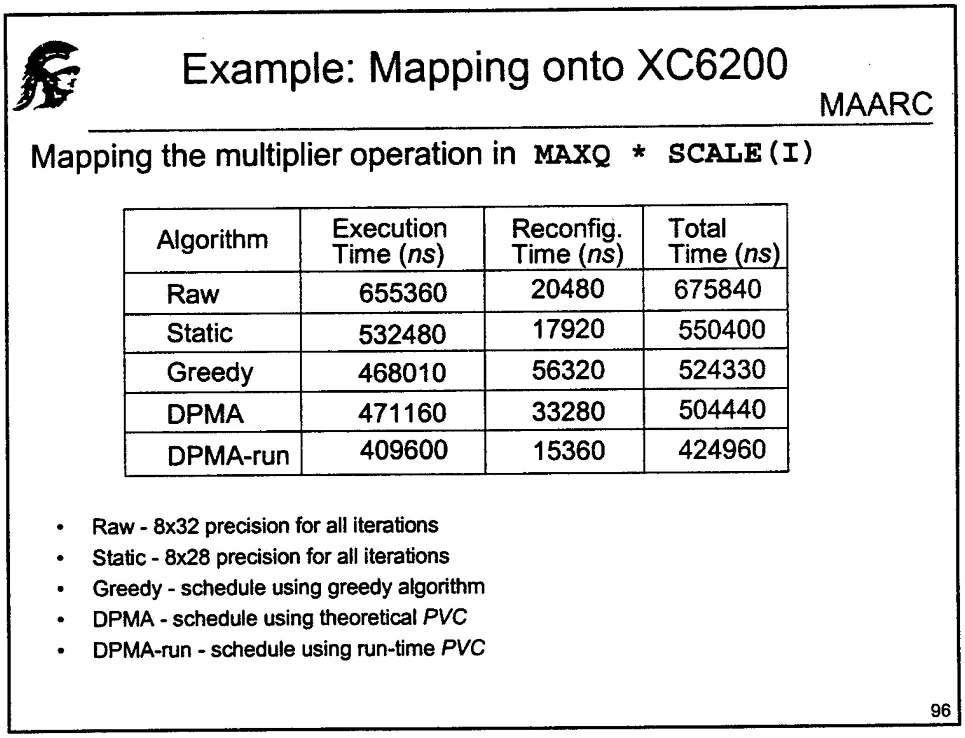
MAARC



Results: DPMA Run-time Approach

MAARC





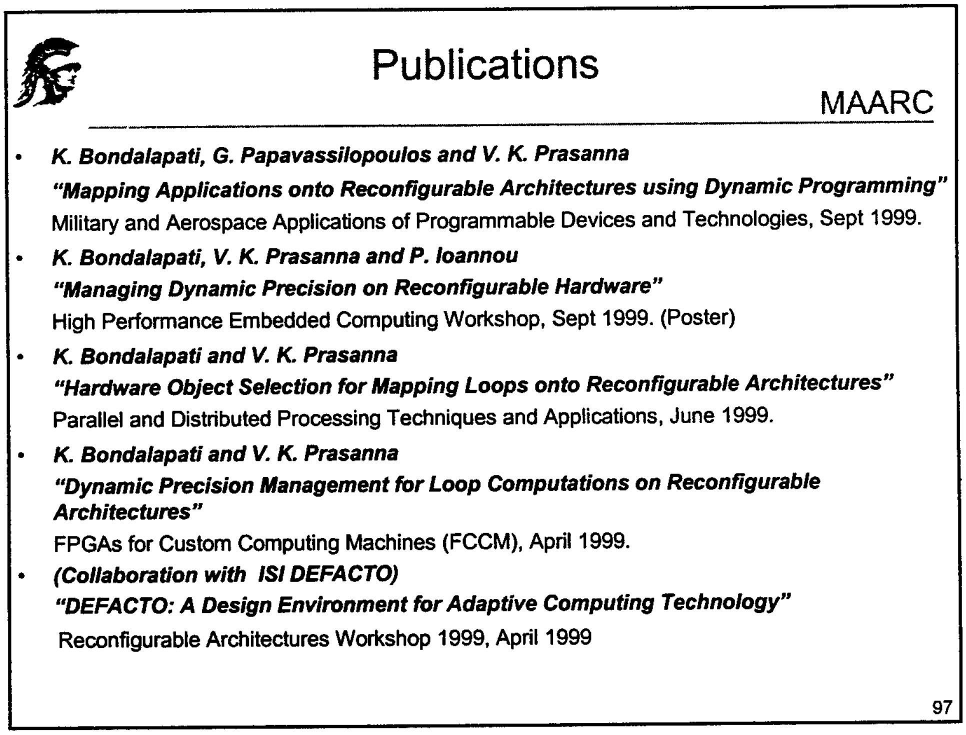
Example: Mapping onto XC6200

MAARC

Mapping the multiplier operation in `MAXQ * SCALE(I)`

Algorithm	Execution Time (*ns*)	Reconfig. Time (*ns*)	Total Time (*ns*)
Raw	655360	20480	675840
Static	532480	17920	550400
Greedy	468010	56320	524330
DPMA	471160	33280	504440
DPMA-run	409600	15360	424960

- Raw - 8x32 precision for all iterations
- Static - 8x28 precision for all iterations
- Greedy - schedule using greedy algorithm
- DPMA - schedule using theoretical *PVC*
- DPMA-run - schedule using run-time *PVC*



Publications

MAARC

- ***K. Bondalapati, G. Papavassilopoulos and V. K. Prasanna***
  "Mapping Applications onto Reconfigurable Architectures using Dynamic Programming"
  Military and Aerospace Applications of Programmable Devices and Technologies, Sept 1999.
- ***K. Bondalapati, V. K. Prasanna and P. Ioannou***
  "Managing Dynamic Precision on Reconfigurable Hardware"
  High Performance Embedded Computing Workshop, Sept 1999. (Poster)
- ***K. Bondalapati and V. K. Prasanna***
  "Hardware Object Selection for Mapping Loops onto Reconfigurable Architectures"
  Parallel and Distributed Processing Techniques and Applications, June 1999.
- ***K. Bondalapati and V. K. Prasanna***
  "Dynamic Precision Management for Loop Computations on Reconfigurable Architectures"
  FPGAs for Custom Computing Machines (FCCM), April 1999.
- ***(Collaboration with ISI DEFACTO)***
  "DEFACTO: A Design Environment for Adaptive Computing Technology"
  Reconfigurable Architectures Workshop 1999, April 1999

Dynamic Precision Management

- Precision variation in loop computation
- Run-time adaptation of configurable hardware
- Efficient dynamic precision management algorithm
- Potential for speeding-up large class of applications

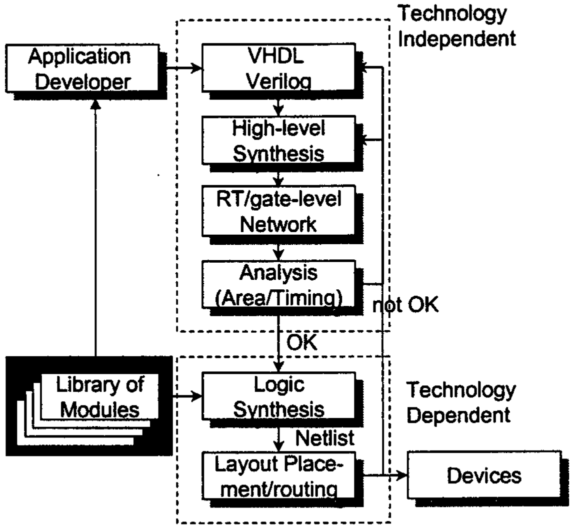
DRIVE

An Interpretive Simulation and Visualization Environment for Dynamically Reconfigurable Systems

Kiran Bondalapati

Traditional Design Approach

MAARC



Technology Independent

Application Developer

VHDL Verilog

High-level Synthesis

RT/gate-level Network

Analysis (Area/Timing)

not OK

OK

Library of Modules

Logic Synthesis

Netlist

Layout Placement/routing

Technology Dependent

Devices

Simulation Tools

MAARC

- Performance Analysis
  - execution time, memory access, power, ...
- Algorithmic Analysis
  - various mapping and scheduling algorithms
- Architectural Exploration
  - device and architectural alternatives

EDA Simulation Tools

MAARC

- Simulation of VHDL designs
  - high level behavioral simulation
  - verifies correctness
  - does not provide performance characteristics
- Simulation of netlist/placed and routed design
  - low level timing simulation
  - fixed to specific implementation on specific device
  - needs final design for each alternative device/algorithm

➡ Application developer needs to understand low level device and architecture details

DRIVE Goals

MAARC

- High level performance analysis
  - based on module level performance characterization
- Architecture abstraction
  - insulate application developer from hardware intricacies
- Algorithm analysis
  - extensible tools to study various algorithmic techniques
- Architecture exploration
  - parameterized architectural model for exploration

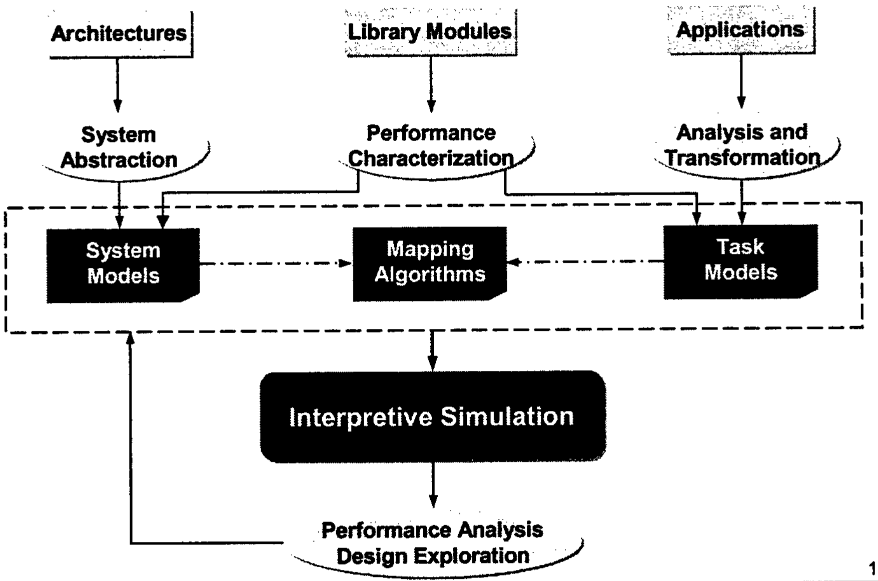
Related Work

MAARC

- Dynamic Circuit Switching, Lysaght et. al.
  - Integrate VHDL modules for dynamic reconfiguration by using multiplexers for inputs and outputs
- CHASTE, Brebner et. al.
  - Low level simulation tool for a specific FPGA(XC6200)
- JHDL, BOOM, JBits etc.
  - Languages and libraries for CAD with simulation embedded into the framework

Interpretive Simulation Framework

MAARC



Architectures

Library Modules

Applications

System Abstraction

Performance Characterization

Analysis and Transformation

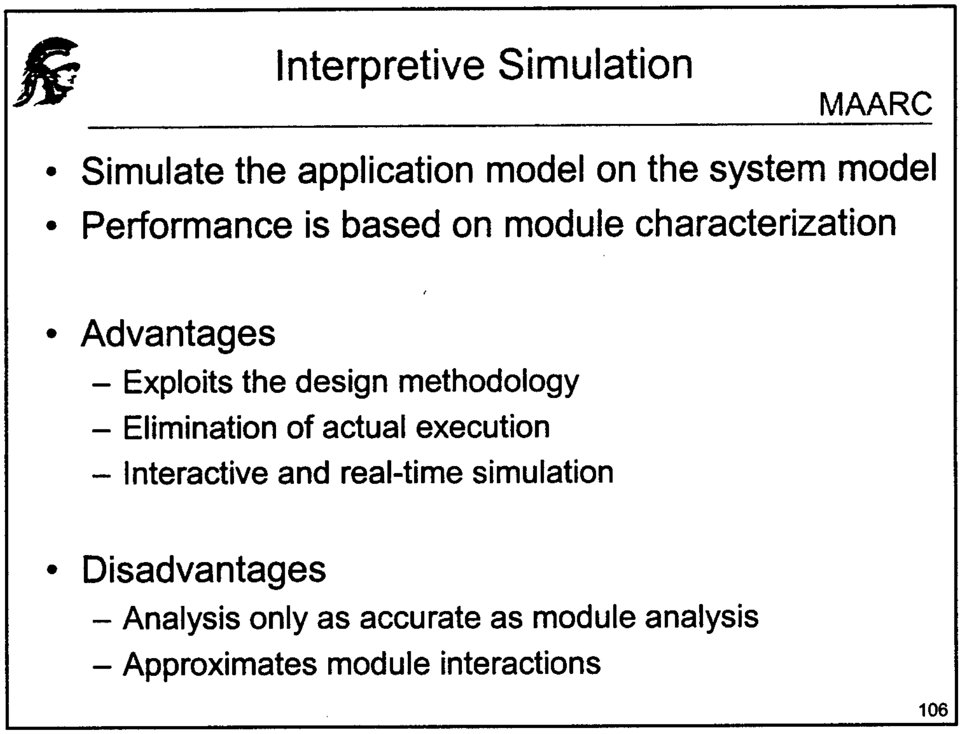
System Models

Mapping Algorithms

Task Models

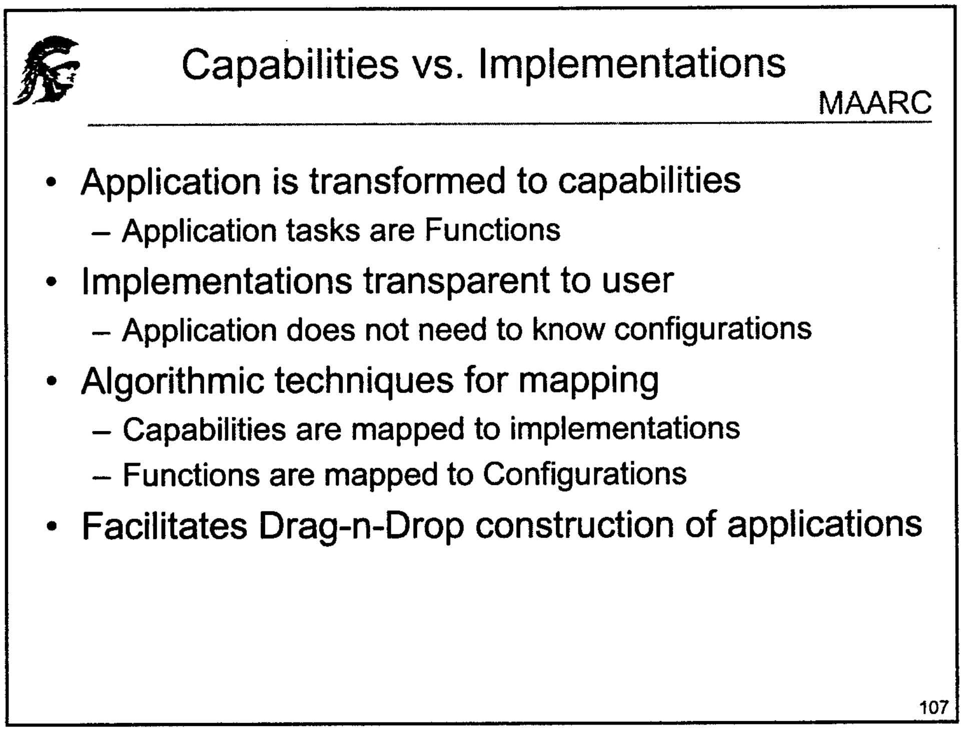
Interpretive Simulation

Performance Analysis Design Exploration



Interpretive Simulation

- Simulate the application model on the system model
- Performance is based on module characterization
- Advantages
  - Exploits the design methodology
  - Elimination of actual execution
  - Interactive and real-time simulation
- Disadvantages
  - Analysis only as accurate as module analysis
  - Approximates module interactions

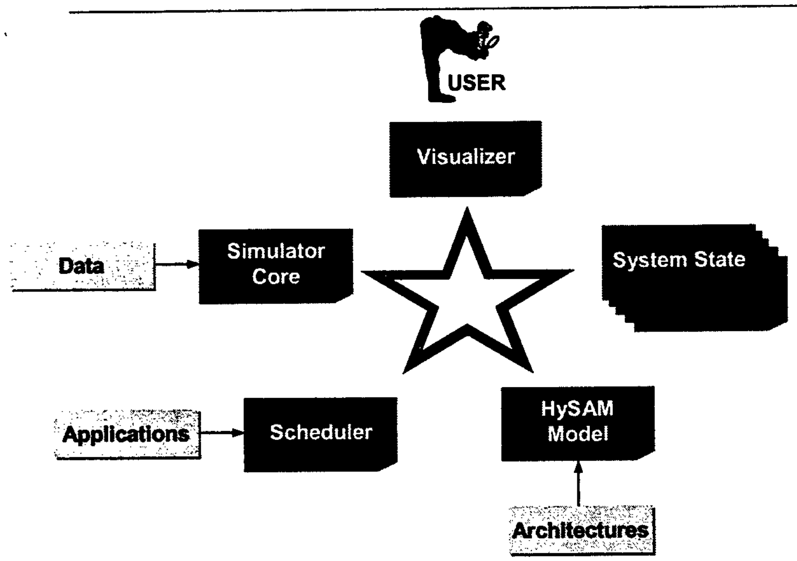


Capabilities vs. Implementations

- Application is transformed to capabilities
  - Application tasks are Functions
- Implementations transparent to user
  - Application does not need to know configurations
- Algorithmic techniques for mapping
  - Capabilities are mapped to implementations
  - Functions are mapped to Configurations
- Facilitates Drag-n-Drop construction of applications

Drive Components

MAARC



Simulator Core

MAARC

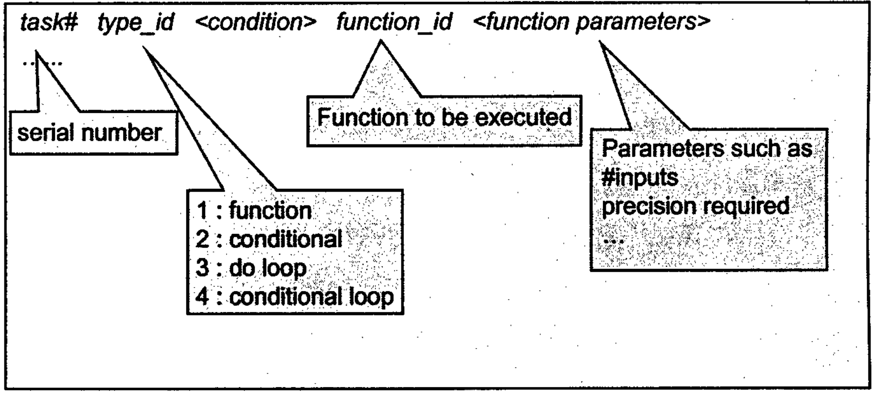
- Execution of functions
  - by dynamic loading of Java classes
  - Java class specified in input for each configuration
- Uniform interface to dynamic Java classes
  - data input and output as Strings
  - internal data type conversion
- Storage of intermediate results
  - for data dependent scheduling
- Easy integration of libraries
  - BOOM, JHDL, etc. Java classes can be utilized

Scheduler Component

- Event based scheduler
  - execution, reconfiguration, memory events
- Dynamic scheduling
  - events accessed dynamically from the scheduler component
- Schedules the simulator core operations
- Scheduling algorithms
  - implements current algorithms
  - easy extensibility to add new algorithms

Application Input Format

- Task Specification



task# type_id <condition> function_id <function parameters>

....

- Dependency Specification

System State Component

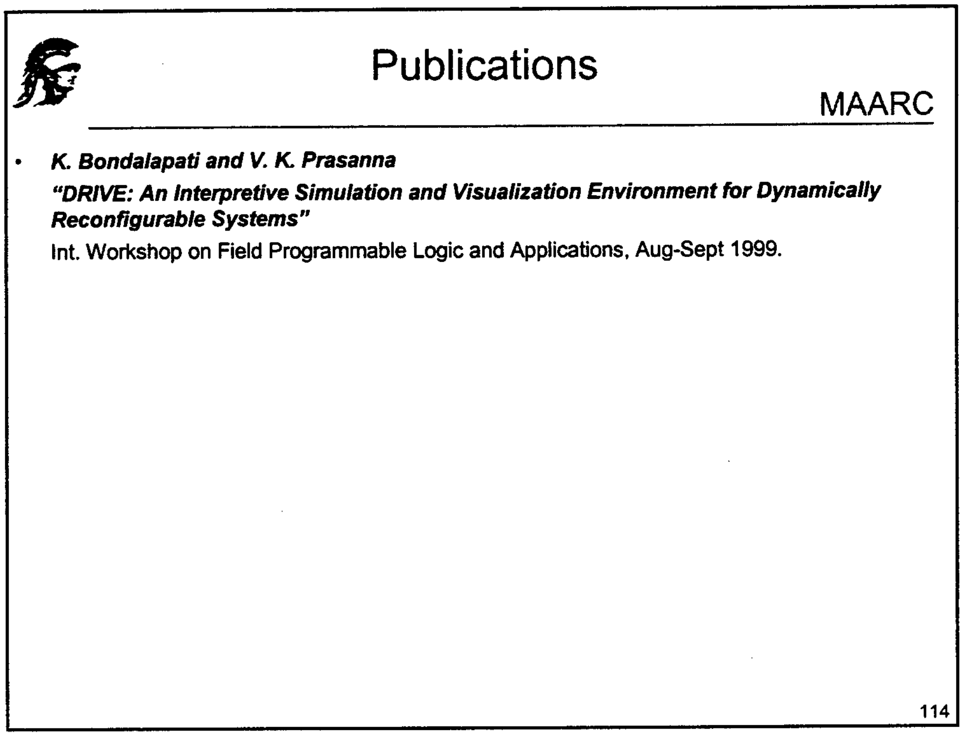
MAARC

- Status of various components of the system
- CLU configuration information
- Configuration cache status
- Memory access information

Visualizer

MAARC

- Human Computer Interface to the simulator
- Java based GUI and components
- Independent of other components
- Performance analysis data



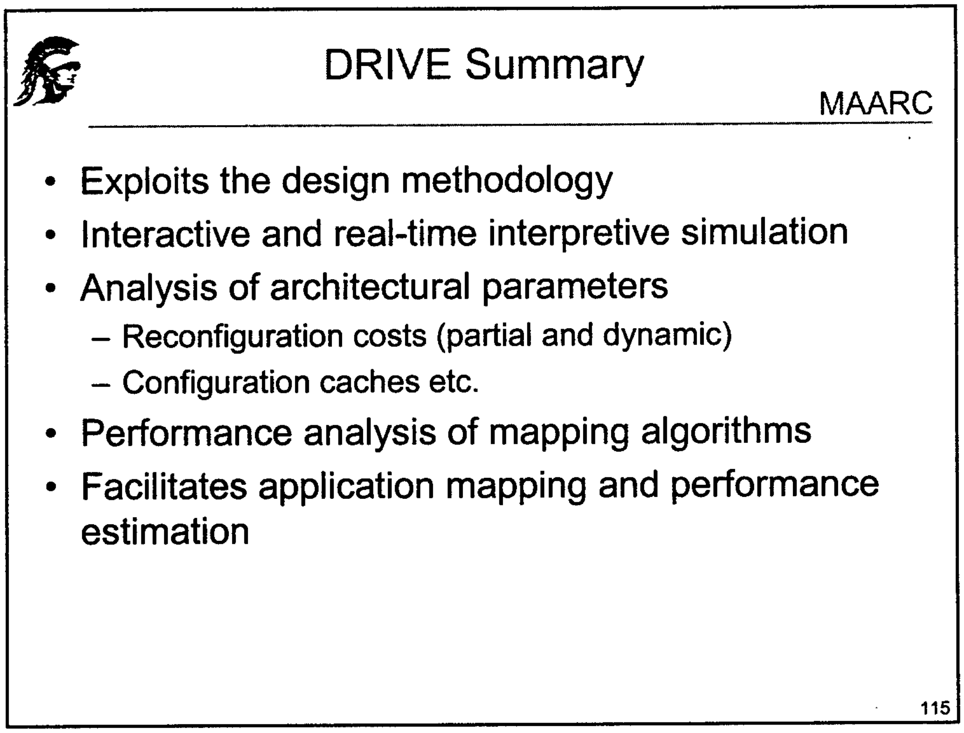
Publications

MAARC

- ***K. Bondalapati and V. K. Prasanna***

  "DRIVE: An Interpretive Simulation and Visualization Environment for Dynamically Reconfigurable Systems"

  Int. Workshop on Field Programmable Logic and Applications, Aug-Sept 1999.



DRIVE Summary

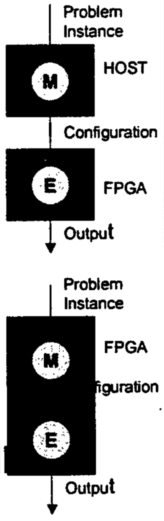
MAARC

- Exploits the design methodology
- Interactive and real-time interpretive simulation
- Analysis of architectural parameters
  - Reconfiguration costs (partial and dynamic)
  - Configuration caches etc.
- Performance analysis of mapping algorithms
- Facilitates application mapping and performance estimation

Sidhu's Contribution

MAARC

- Higher performance through reconfigurable computing than custom VLSI
- Problem instance dependent mapping
- String matching (KMP) algorithm
  - Order of magnitude speedup in overall execution time
  - **3** orders of magnitude speedup in mapping time (software)
  - **6** orders of magnitude speedup in mapping time (self-reconfigurable device)
- **1 to 3** orders of magnitude speedup in overall execution time for Genetic Programming



Impact

MAARC

- Sanders
- JPL
- Encouraging feedback from the community

Andreas' Contribution

MAARC

- Mapping based on algorithmic design
  - new performance metric
    - scalability & partitionability
  - FFT
    - 2-8 times faster than the "Fastest FFT in the West" (1997)
  - Matrix Operations
    - 50% memory savings compared to the state-of-the-art
- Run-time mapping
  - new performance metric
    - mapping time is critical
    - predictable performance is essential
  - case study: graph problems
    - 10^6 speed-up compared to the state-of-the-art

Impact

MAARC

- Advance state-of-the-art
  - RAW (MIT)
  - RAPID (Univ. of Washington)
- Provide evidence to the community about the necessity of
  - scalable and partitioned solutions
  - new performance metric for run-time mapping
- Preliminary development of efficient techniques for run-time mapping
  - expected speed-up: 2-6 orders of magnitude compared to the state-of-the-art

Kiran's Contribution

MAARC

- HySAM: Hybrid System Architecture Model of reconfigurable architectures
- Mapping of application loops onto configurable architectures
- Dynamic precision management to exploit run-time reconfiguration
- DRIVE: Module based interpretive simulation framework

Part II

NYU Efforts

1 Summary of Accomplishments

This effort was focused towards developing a model consistent with the constraints of adaptive hardware on the one hand, and the need to compile and optimize applications developed to execute them on the other. The primary technical goals for the NYU portion of the subcontract were:

- Develop a model that can serve as a target for the compiler.
- Innovate the framework of an optimizing compiler to target the adaptive processor model.
- Achieve fast compilation times.
- Develop and validate instruction scheduling optimizations as a proof-of-concept that the compilation framework can be used in the context of the paticular model developed as part of this effort.
- Develop language support techniques that can serve as a basis for interactive specificatoin of partititioning and mapping.

1.1 Model for Compilation onto Adaptive Systems

The initial proposal for a Reduced Configuration Space Processor developed and presented in 1997 served as the basis for the final model which is referred to as the Adaptive EPIC or A-EPIC architecture. The A-EPIC architecture is parametric and achieves the stated goals in the following sense. It provides an abstract representation of adaptive logic that can used as a basis for compilation. Experimental validation (sketched below) has demonstranted the feasibility of using the model to help achieve speedups for challenge applications. A novel feature of an A-EPIC class processor is the ability to use features in the EPIC core notably speculation to help prefetch configurations, and thus reduce configuration switching times with the intent of supporting dynamic configurability. Another interesting feature is an adaptive configuration cache also intended to help with dynamic switching times.

1.2 Compilation Framework

The framework for compilation utilizes interactive partitioning and mapping techniques, that the programmer is intended to specify to the compiler front-end. In this context, the application is divided into a portion that is meant to be executed on the EPIC core (part of the A-EPIC) which is typically the control skeleton part of the computation. The compute intensive kernels—identified via profiling, for example, the IDCT kernel in the context of MPEG2—are executed on the adaptive part of the processor. Scheduling and related compiler optimizations will help optimize the issuing of such "adaptive instructions".

1.3 Compilation Techniques

Instruction scheduling algorithms have been developed, and when tractable, proven to be optimum within the context of the A-EPIC framework. A language-independent notation TimeC has been developed for specifying time-constraints in applications. The goal of this notation is to specify time constraints in a base language such as C or C++ using TimeC. The structure of this notation in terms of its language independent aspects is applicable to be applicable in the partitioning and mapping contexts as well. Speedups for challenge applications in the range of 5-35 have been obtained within the context of the A-EPIC model and the compilation framework innovated here.

Final Review

Krishna V. Palem
ReaCT-ILP Laboratory
http://react-ilp.cs.nyu.edu
New York University

This work was supported by DARPA Contract No. DABT63-96-C-0049 performed under the USC MAARC project.

Project Overview

A Summary of Accomplishments

- Architectural models for compilation
- Compiler optimizations
- Application studies
- Compiler infrastructure

NYU

Our Goals

- Define a processor model that serves as a target for an optimizing compiler, that
  - takes advantage of the technology growth curve
  - leverages the advantages of adaptive logic
- Architect an optimizing compiler framework for targeting the defined architecture that
  - demonstrates performance improvements for challenge applications
  - achieves the performance goals in reasonable compilation time

Architectural Models

NYU

Architectural Models for ACS Compilation

- Developed compiler models for EPIC processors extended with reconfigurable logic
  - EPIC processors : Merced, McKinley
- Initial design of suitable extensions to current machine description framework to target adaptive EPIC processors

RCSP : A Reduced Configuration Space Processor and its Programming Environment,
K.V.Palem, S. Talla, HPEC'97, Lincoln Labs, MIT, Sept., 1997

Adaptive Explicitly Parallel Instruction Computing,
K.V.Palem, S.Talla, P.Devaney, Australasian Computer Architecture Conference, January 1999.

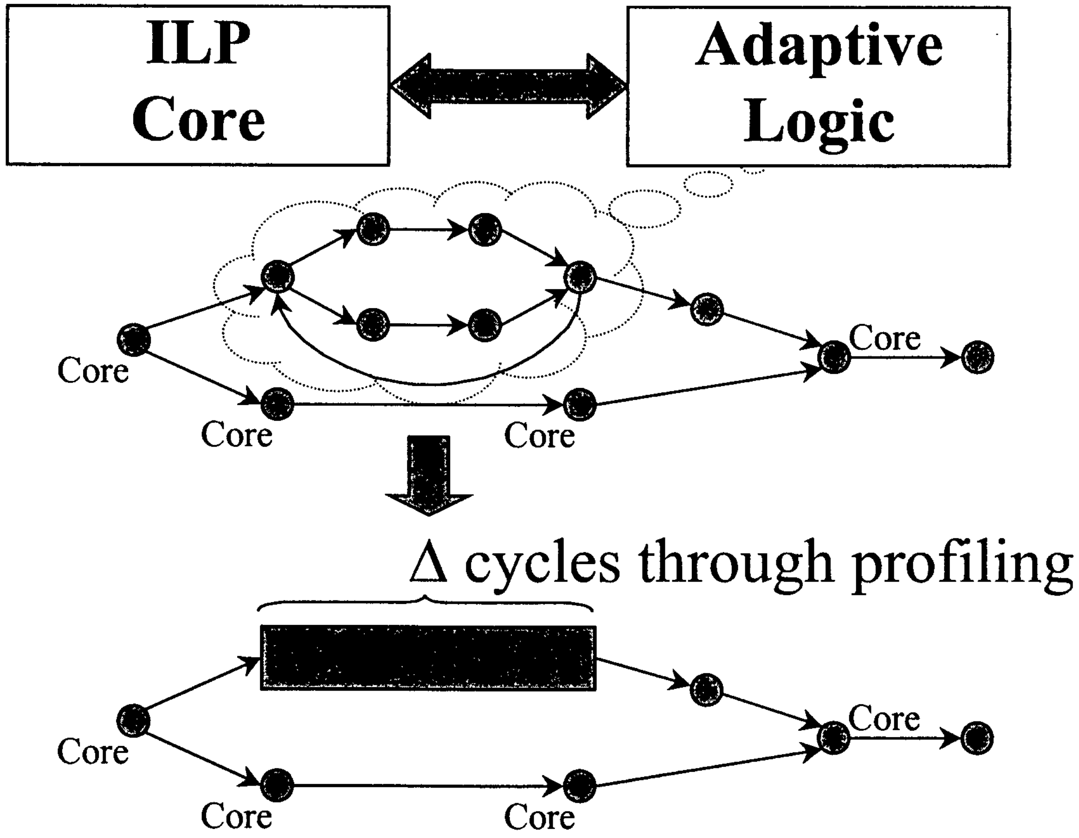
Language Support And Compiler Optimizations

Scheduling and Time-C

- Developed fast polynomial time algorithms for instruction scheduling
  - in the presence of long latency instructions
  - under user specified time constraints

- Developed Time-C for specifying time
  - translate source constraint specifications to constraints on intermediate graphs
  - developed algorithms for scheduling time critical instructions
    - of relevance to developing support for automatic partitioning and mapping

Instruction Scheduling Optimizations for Configurable Hardware



ILP Core

Adaptive Logic

Core

Core

Core

Core

Δ cycles through profiling

Core

Core

Core

Core

A.Leung, K.Palem , A.Pneuli, S. Talla "A Fast Algorithm for Scheduling Time-constrained Instructions on AEPIC" (preliminary version).

Time-C and Time_Tract

- Language independent syntax for “light weight upgrade”
- Can be used with C, C++…
- Embedded applications involving real-time constraints

Time-C : A Time Constraints Language For ILP Compilation
K. V. Palem, A. Leung, A. Pnueli, PACT’98

Compilation challenges

- Code partitioning
- Mapping partitions to reconfigurable logic
- √ Reconfiguration time
- √ Configuration scheduling and resource allocation
- √ Configuration selection
- Intermediate representations

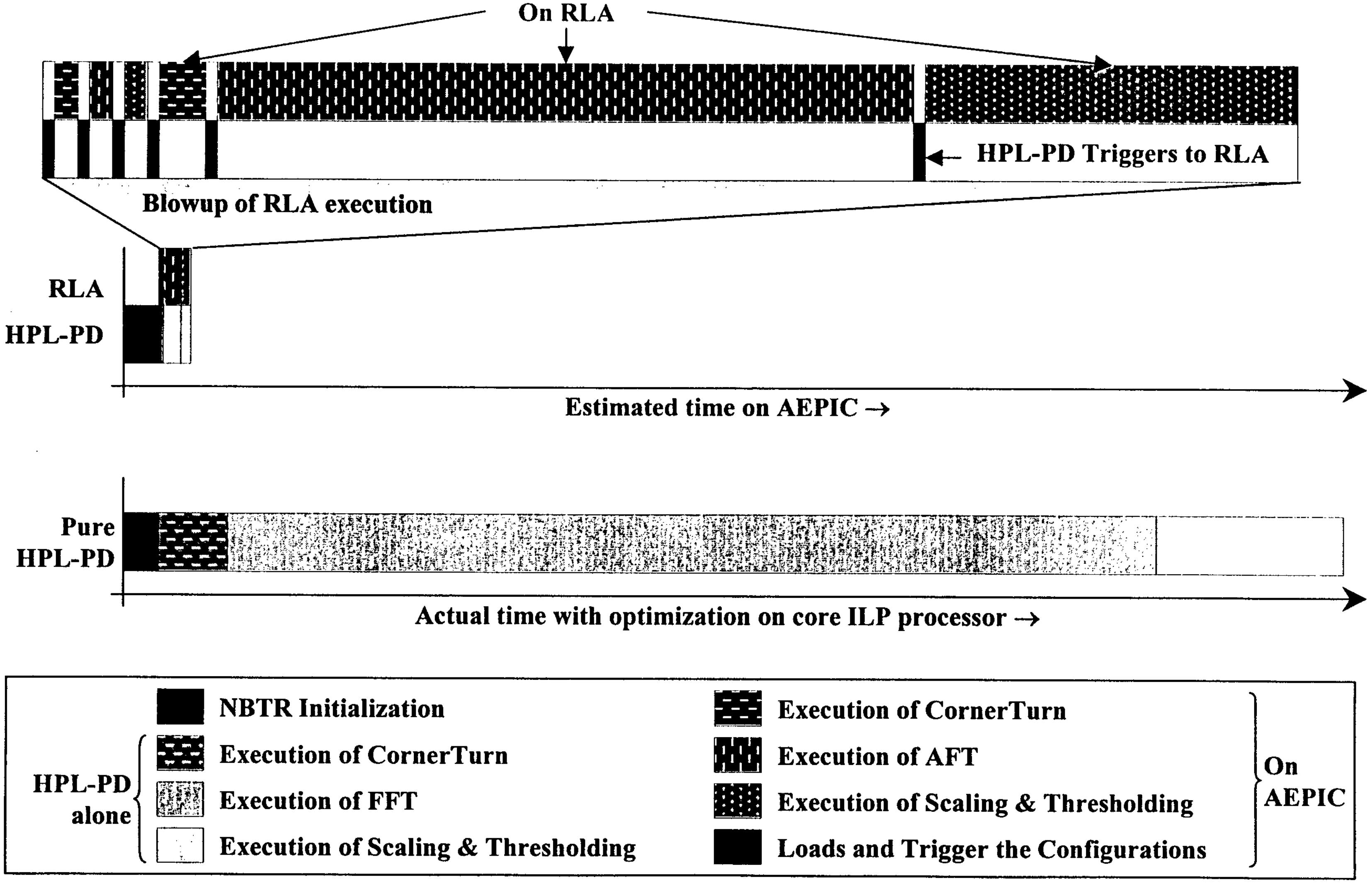
Application Studies

Application Performance Studies

- Several applications from signal, image and media processing domains mapped to proposed adaptive EPIC target

- Applications considered from following benchmarks
  - Honeywell ACS stressmarks
  - MediaBench from UCLA (Bill-Mangione Smith)
  - RAW Benchmarks from MIT (Anant Agarwal)
  - Others from Spec95, public domain

NBTR : Execution Profile



Sample Performance Studies

Application	9-issue EPIC	Optimized compilation for 9-issue EPIC	A-EPIC (SCDA)	A-EPIC Speedup
MPEG2 decoder	n/a	439486198	80686602	5.4
IDEA Encryption (one round)	118	118	18	6.4
32-tap FIR	31533	13491	384	35
NBTR (input sampled 10 times)	22529978	13731573	532800	25.8
IDCT	12127	6633	544	12.2

- Unlimited reconfigurable resource
- Access to reconfigurable array through memory on a 64 bit separate memory bus, array supports same PE architecture as XC6200
- Assumed reconfigurable array clock to be same as core clock

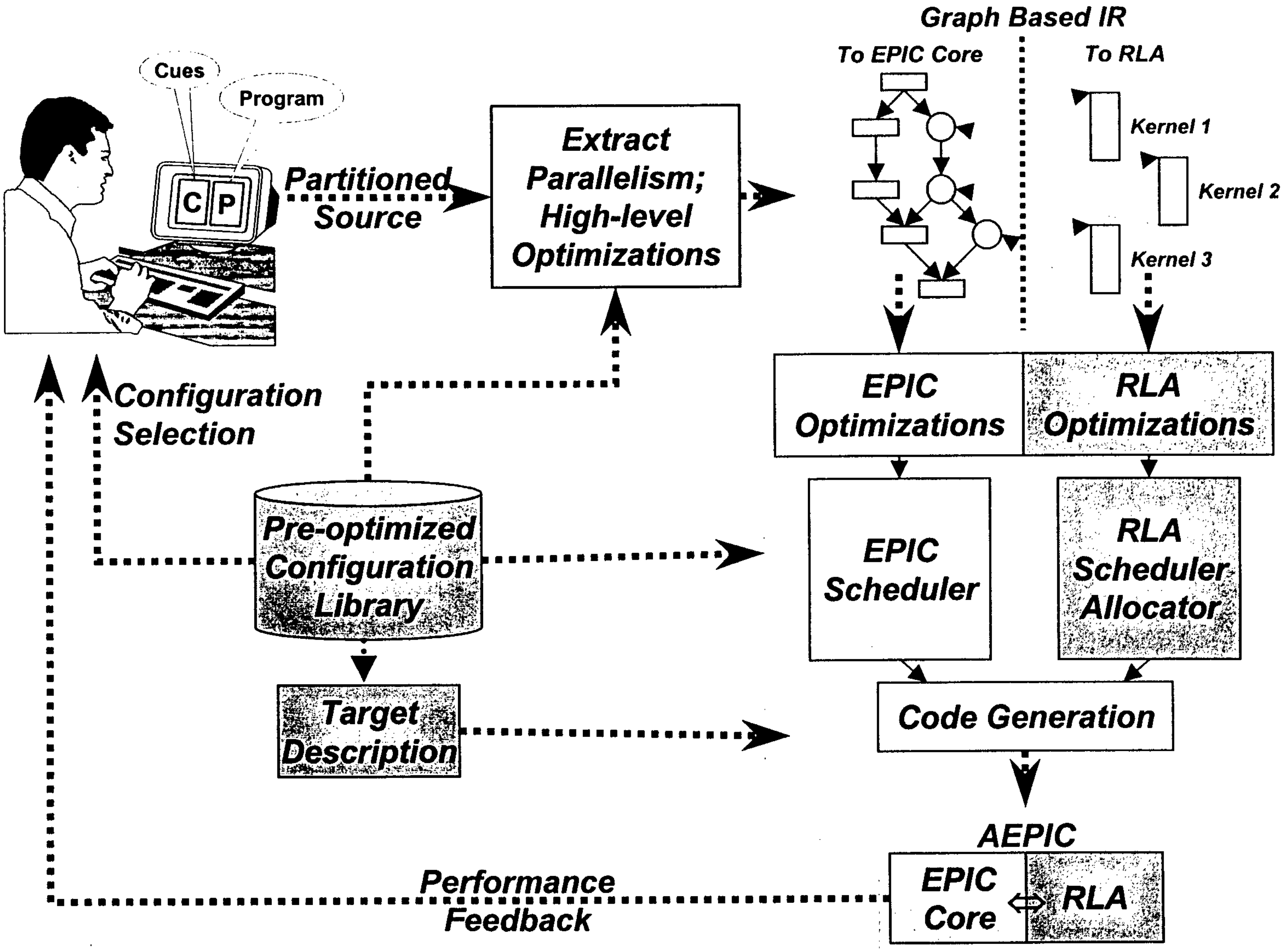
Adaptive Explicitly Parallel Instruction Computing,
K.V.Palem, P. Devaney, S. Talla, Australasian Computer Architecture Conference, January 1999.

Compiler Infrastructure

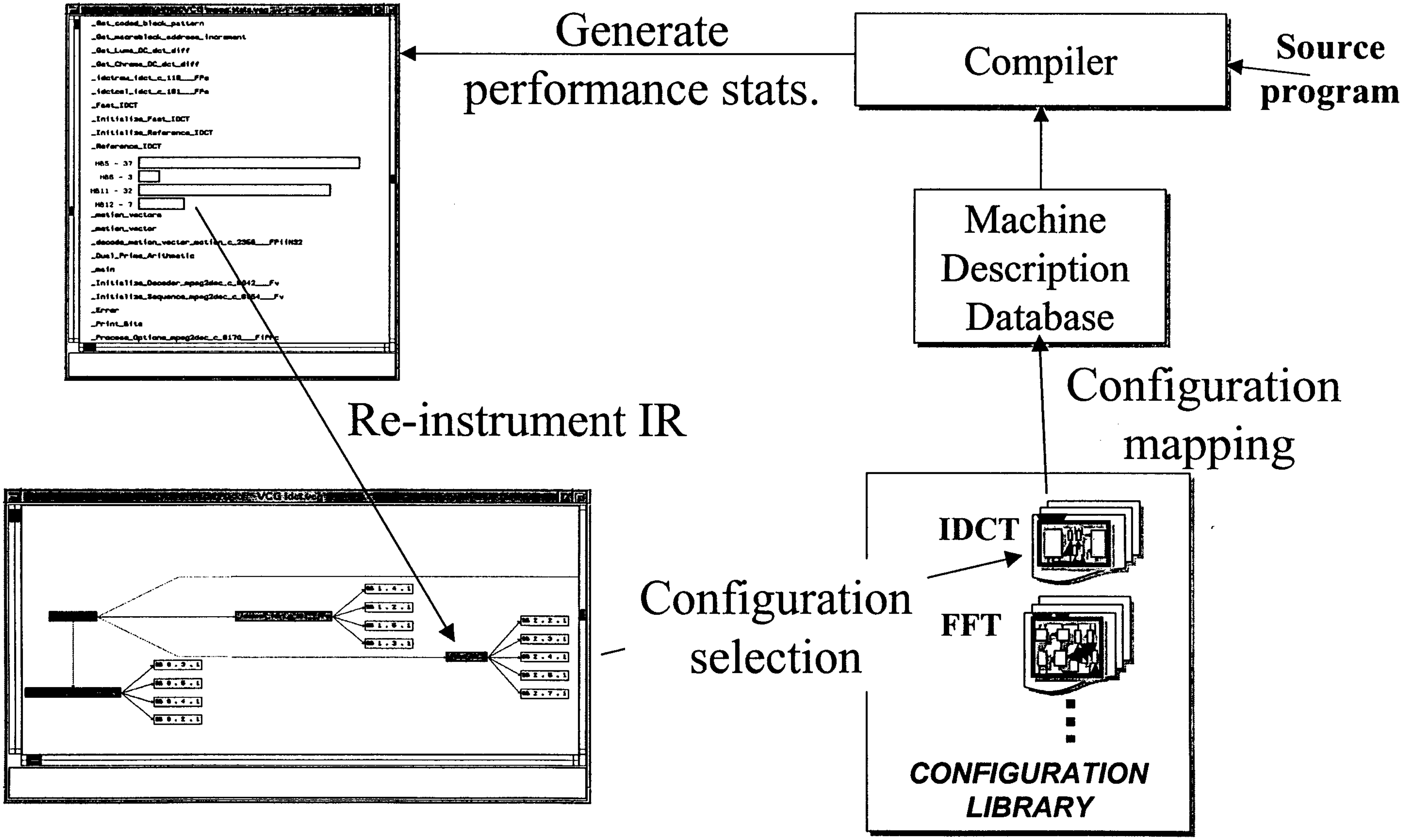
Infrastructure Goals

- Faster and easier compilation
  - extend current compilation frameworks
- Tackle configuration loads
  - dynamic reconfiguration overheads
- Contain partitioning and mapping
  - a focus for the USC MAARC project
- Explore a variety of reconfigurable device parameters

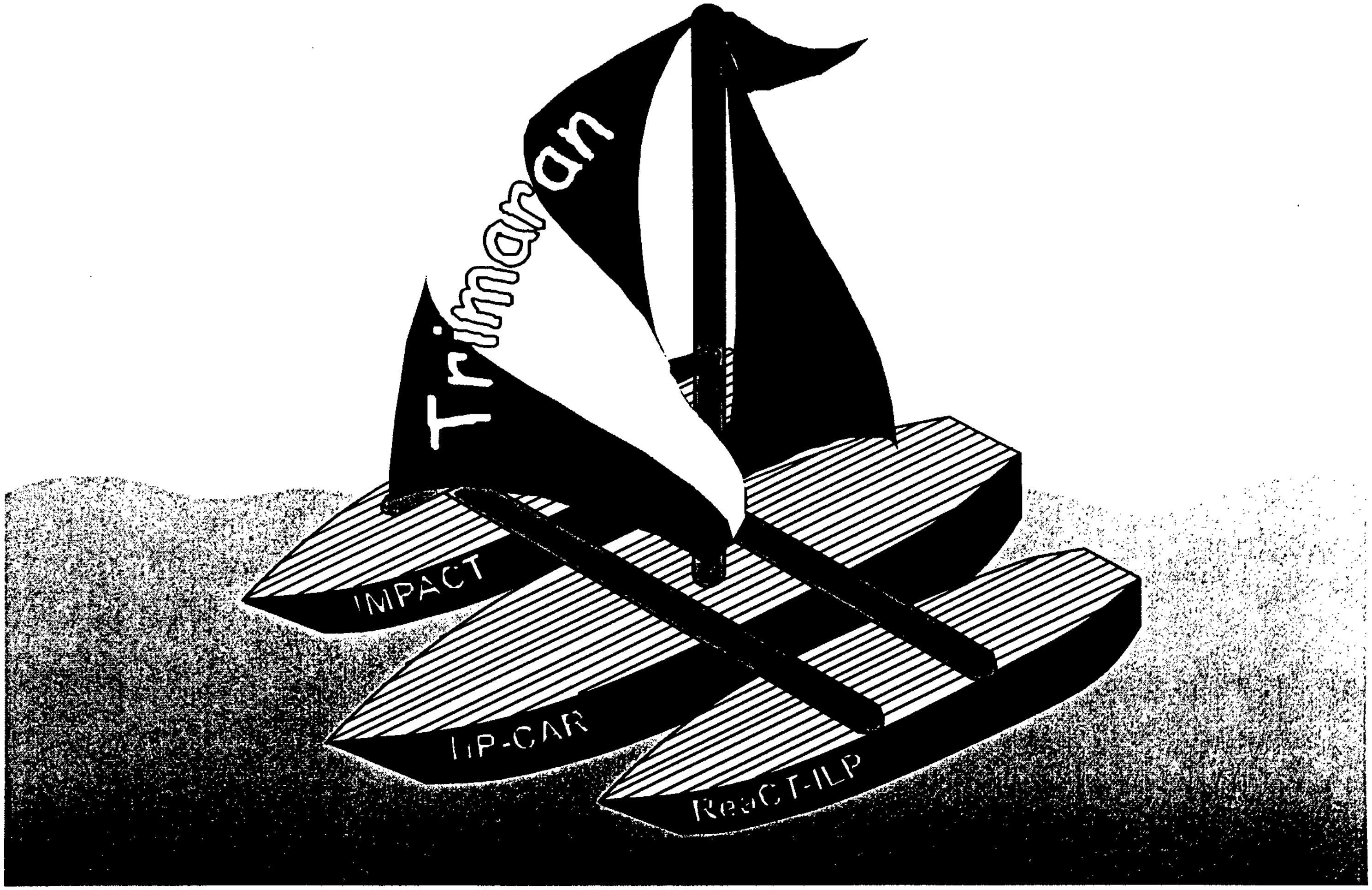
Compilation framework for AEPIC's



Compilation Methodology Using Trimaran



Trimaran Infrastructure for Research in Instruction-level Parallelism



Publications

- RCSP : A Reduced Configuration Space Processor and its Programming Environment, K.V.Palem, S. Talla, HPEC'97, Lincoln Labs, MIT, Sept., 1997 (poster presentation).
- TimeC: A Time Specification Language for ILP Processor Compilation, A. Leung, K. V. Palem, A. Pnueli, 5th Annual Australasian Conference on Real-time Systems, Sept., 1998.
- A Fast Algorithm for Scheduling Time Constrained Instructions on Processors with ILP, A. Leung, K. V. Palem, A. Pnueli, International Conference on Parallel Architectures and Compilation Techniques, Oct., 1998.
- Reconfigurable Computing: High Performance Embedded Computing = ILP + Reconfigurable : A Novel Architecture and its Compiler, K. V. Palem, S. Talla, Adelaide, Australia, Sept., 1998 (invited talk)
- Adaptive Explicitly Parallel Instruction Computing, K.V.Palem, P. Devaney, S. Talla, Australasian Computer Architecture Conference, January 1999.

Work In Progress

- Instruction Scheduling for Adaptive EPIC Processors, A. Leung, K. V. Palem, A. Pnueli, S. Talla.
- Adaptive EPIC Processors, Architectural Specification 1.0 and Validation using Trimaran, A. Leung, H. Kim, R. Rabbah, S. Talla.
- Machine Description Framework for AEPIC Processors, K. V. Palem, S. Talla.

A Summary of Accomplishments

- Adaptive Explicitly Parallel Instruction Computing (AEPIC)
  - architectural models for compilation
- Compiler optimizations
  - fast polynomial time instruction scheduling
  - scheduling under time constraints
- Compiler infrastructure
  - enhance MDES to incorporate AEPIC features
  - incorporate configuration scheduling and allocation
- Benchmark studies

Architectures

NYU

Motivation

- Demands of Embedded Computing
  - Faster, cheaper processors
  - Shorter times to market
- Poor scalability of superscalars
  - complex control units
- EPIC / VLIW
  - Simpler architectures
  - Known compilation technology
- FPGA / Reconfigurable logic
  - Fine grained parallelism
  - Explicit control over micro-architectural features
  - Fast static communication

Technology and Application Trends

- Feature size and the effect on signal delay
- Cost of verification and test of new designs
- The new media/embedded shift
- Chip density

NYU

The Impact Of Technology Trends

- As Wire Delays Become Significant, focus on architectures that
  - do not involve long distance communication
  - distribute control and data processing logic
- Impact of rising verification and test costs
  - keep the architecture simple and regular
  - move complex decision making logic from processor to higher level tools (compiler)

The Impact Of Technology Trends (contd.)

- Lots of hardware parallelism available
  - can accommodate approx. 50 pentiums on one die in 6 years

However,

- Conventional architectures and compilation
  - cannot expose enough parallelism in applications
  - even the “superb” model yields an ILP < 10 on average
- Need for new architectures and compilation techniques!

NYU

Application Trends Summary

- Real-time processing
- Packed 8-, 16-, and 32-bit integer data
- Continuous data streams
- Fine grain parallelism
- Long integer arithmetic, table look-ups
- Common kernels (small code size)
- Low temporal reuse
- High spatial locality

The Impact of Media Application Trends

- Simple regular architectures are desirable
  - scope for lots of MIMD processing
  - those that are tuned for media kernels
  - need newer caching technology
    - requirements of predictability, high throughput
    - low temporal reuse, high spatial reuse

What Is The Response Elsewhere To All This?

Architecture Research Approaches

- Past (conventional) approaches
  - better instruction fetch/issue
  - improved instruction processing
  - better prediction (branches, aliases)
  - statically scheduled variants of VLIW's
- Novel (different) approaches
  - Reconfigurable processors
  - IRAM and variants
  - Simultaneous multi-threading
  - On-chip multi-processing

State Of Compilation Technology

- Compilation for past variants
  - well known technology
  - drawback: bottleneck of “conventional compilation”
- Compilation for radically different architectures
  - no known efficient and automatic compilation technique
  - possibility for breaking through the standard compilation bottleneck

NYU

Two Noteworthy Directions

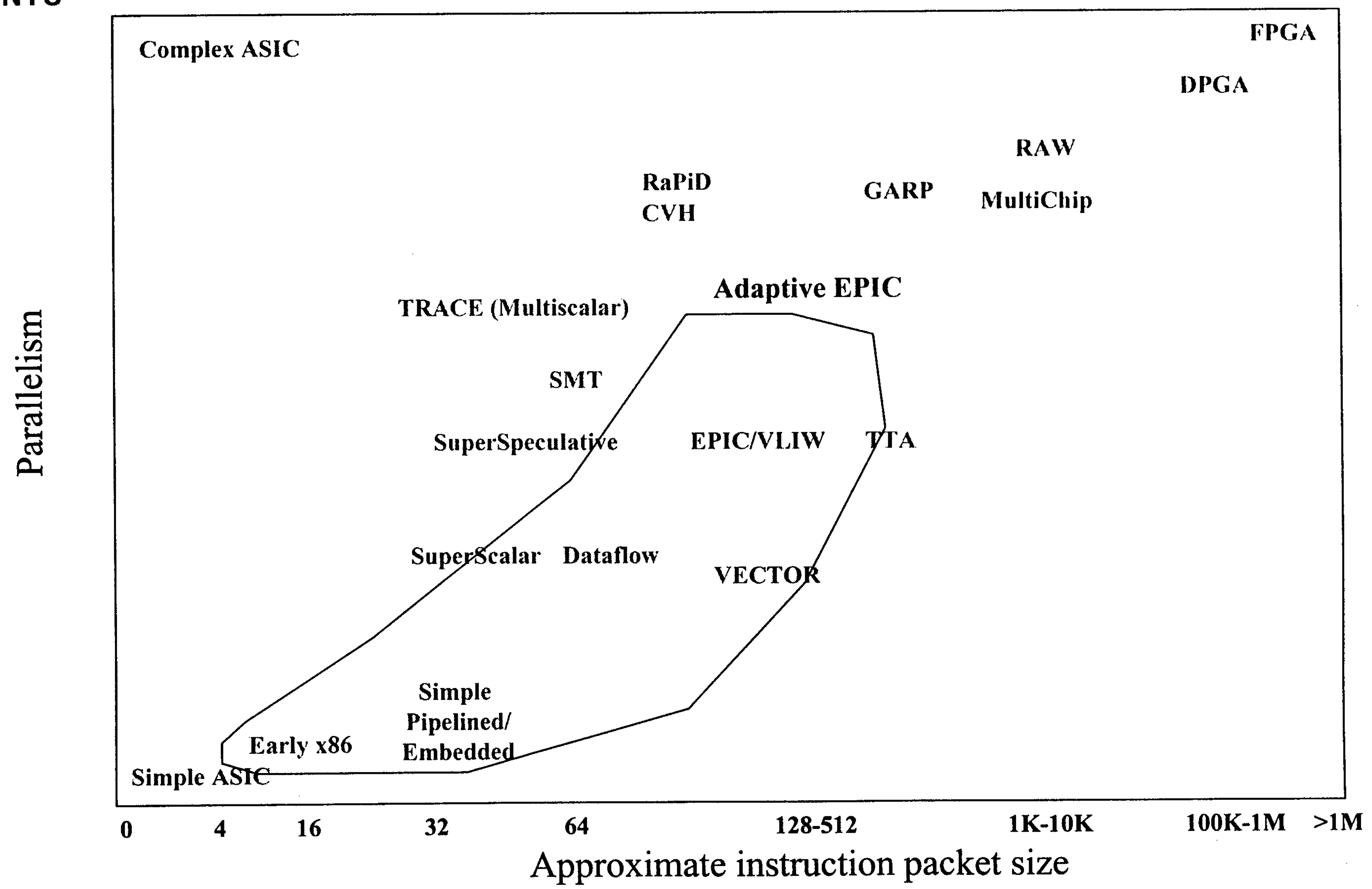
- Reconfigurable Processors
  - let compiler handle everything
  - no commitment to a particular architecture
  - compiler generates architecture and code for it
- Explicitly Controlled Architectures
  - simplify architectures as much as possible
  - architectural template is a known, conventional one
  - compiler handles a lot of processor's decision making
    - explicitly control issue, scheduling, allocation
  - Explicitly Parallel Instruction Computing (EPIC)
    - subset of explicitly controlled architectures

NYU

What are the hurdles?

- Poor compilation times
  - lack of correspondence between standard IR and final configurations
  - place and route inherently complex
- Additional runtime overheads
  - large configuration size implies high reconfiguration costs
  - this also implies context switches are very costly
- Lack of convenient abstract models, language support
  - models for algorithm development (e.g. RMESH, USC-MAARC)
  - models for compiler targets (ReaCT-ILP)
  - language support for hardware structural information
    - but not as complex as HDL's

What can be efficiently compiled for today?



NYU

What can we infer?

- Design/verification costs
  - Simple, regular architectures
- Signal delays
  - Shorter connections ; local interactions
- Media and embedded processing
  - High throughput, highly compute intensive processing, many integer types, customization, temporal predictability
- Not enough ILP through standard compilation
  - Customized/special purpose compilation?
  - Adaptive architectures?

⇩

Adaptive Explicitly Parallel Instruction Computing?

Current Effort

NYU

Adaptive Explicitly Parallel Instruction Computing Architectures

Our Goals

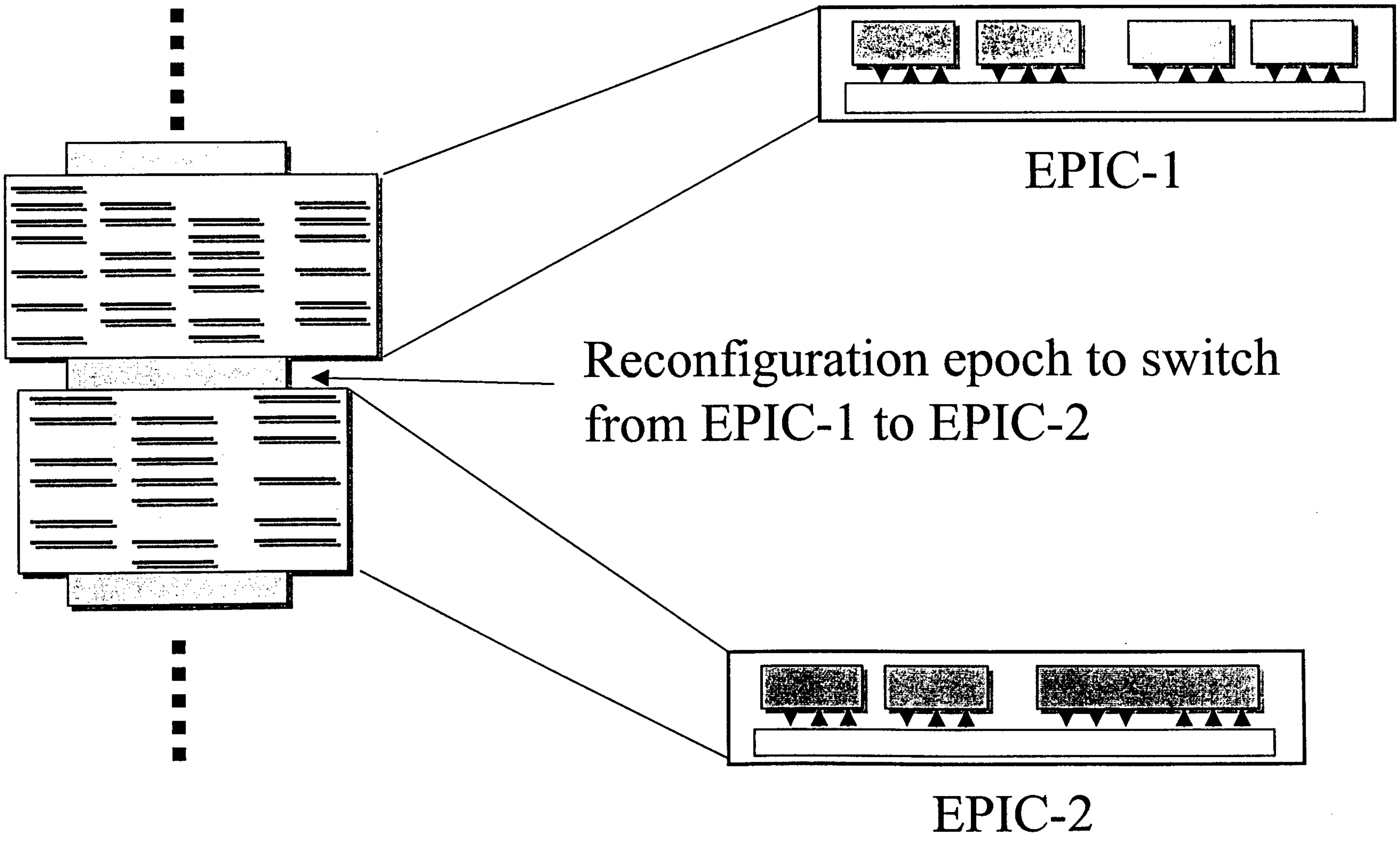
Adaptive EPIC Architectures

Combine advantages of EPIC’s and reconfigurable logic

- Fast automatic compilation
- Explore a variety of reconfigurable processor architectures
  - parameterize hardware space starting with HPL-PD

Adaptive EPIC execution model

Record of execution



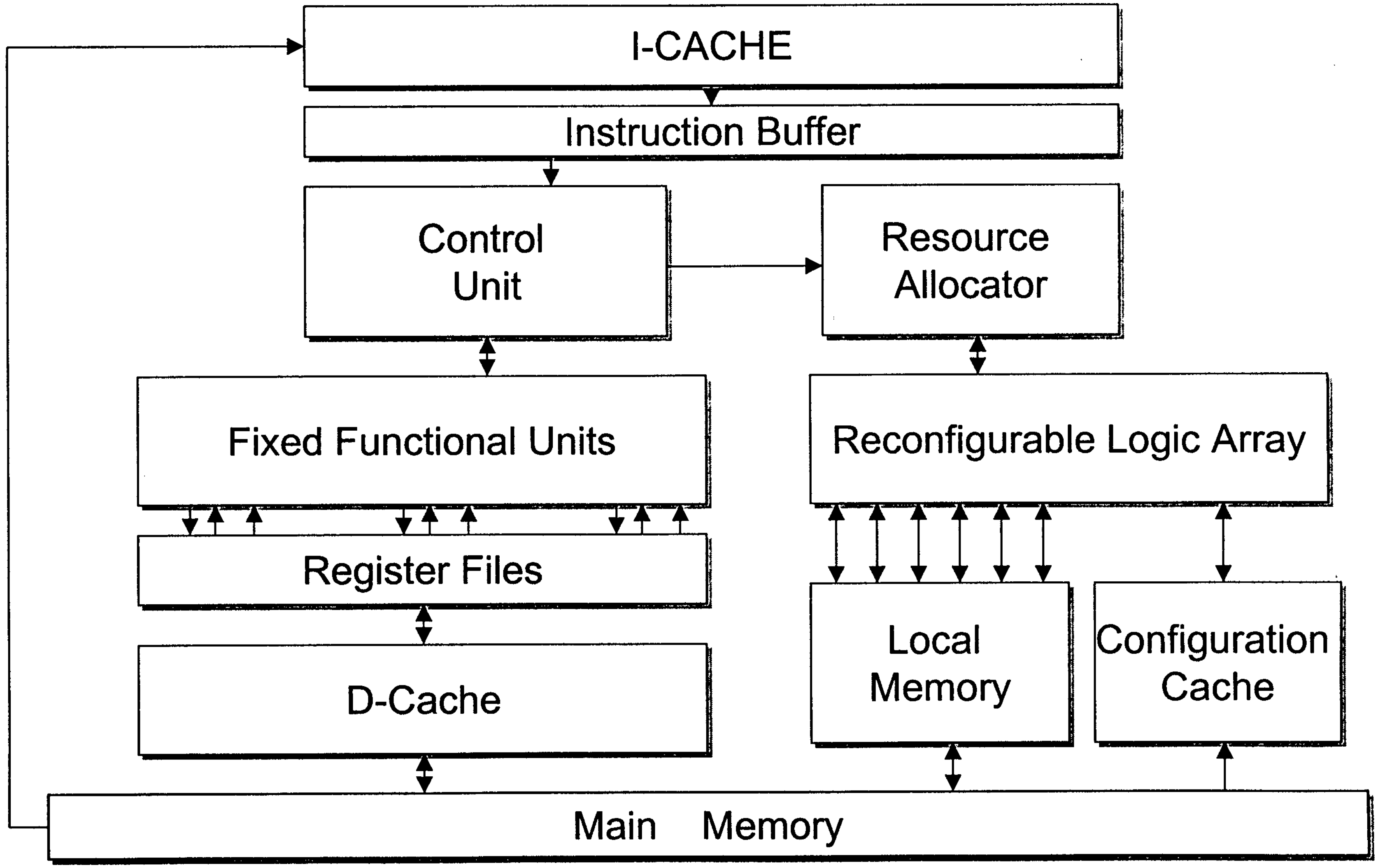
EPIC-1

Reconfiguration epoch to switch from EPIC-1 to EPIC-2

EPIC-2

NYU

Our Adaptive EPIC Architecture

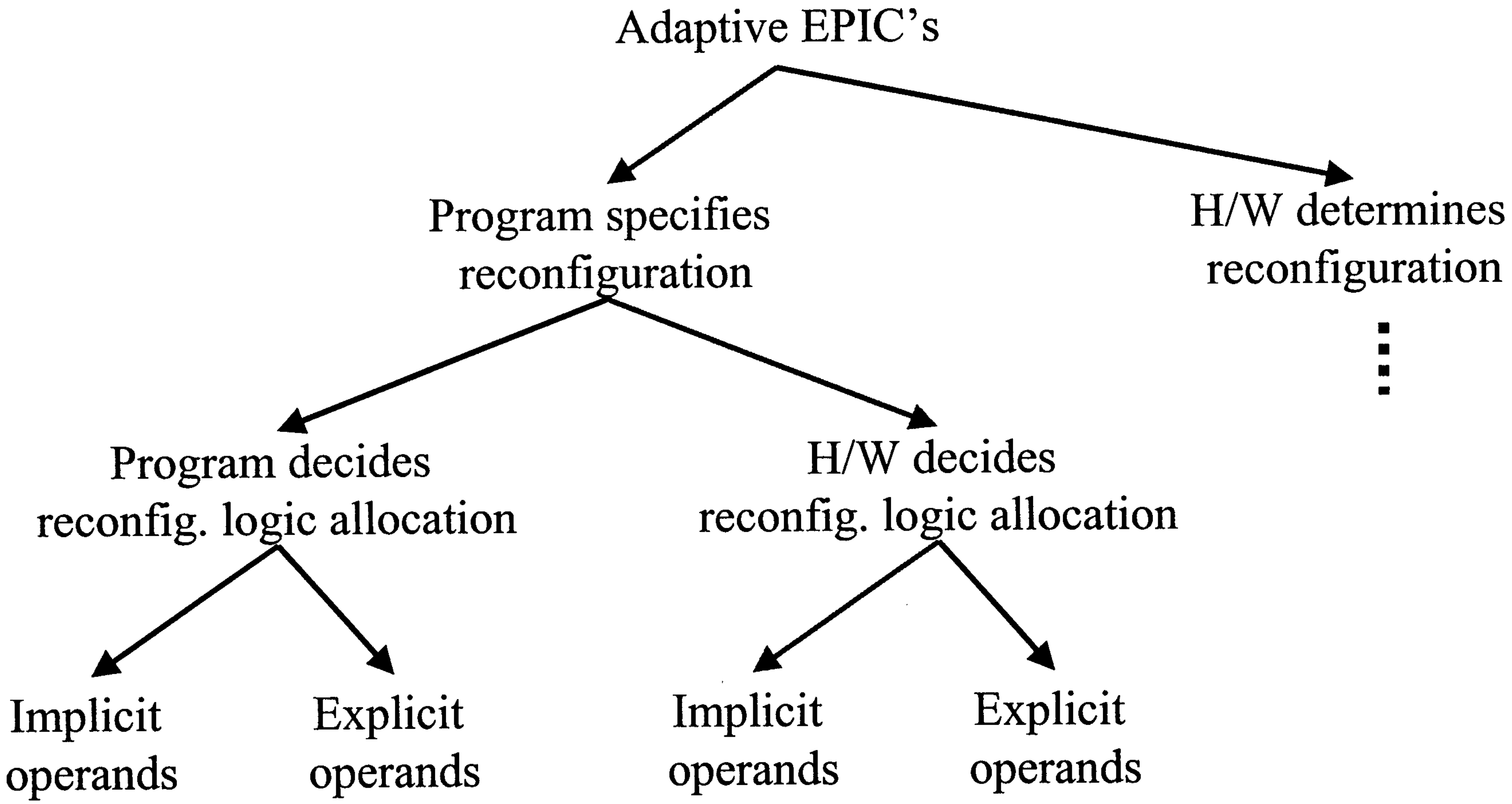


AEPIC Issues

- The issue is of deciding the partitioning of tasks between the compiler and the processor

- The tasks of specific interest and unique to AEPIC's
  - When to reconfigure?
  - Which configurations to keep and which to evict?
  - Where in the reconfigurable logic array should a configuration be instantiated?
  - How are the input operands and results communicated to and from configured functional units?

A Taxonomy of AEPIC Architectures



Adaptive EPIC's

Program specifies reconfiguration

H/W determines reconfiguration

Program decides reconfig. logic allocation

H/W decides reconfig. logic allocation

Implicit operands

Explicit operands

Implicit operands

Explicit operands

The Adaptive EPIC Space of Interest

- Statically scheduled
  - program specifies when and what to reconfigure
- decoupled configuration fetch and execute
- Dynamically allocated
  - processor decides where to allocate configured (custom) functional unit in the reconfigurable logic array
- Custom instructions have implicit operands
  - operands to custom instructions are not part of the custom instruction packet
  - operands pre-loaded using operand load instructions
- Explicitly controlled configuration caches

AEPIC
Rationale For Important Design Decisions

NYU

Statically Scheduled AEPIC's

- Statically scheduled
  - program specifies when and what to reconfigure

Why?

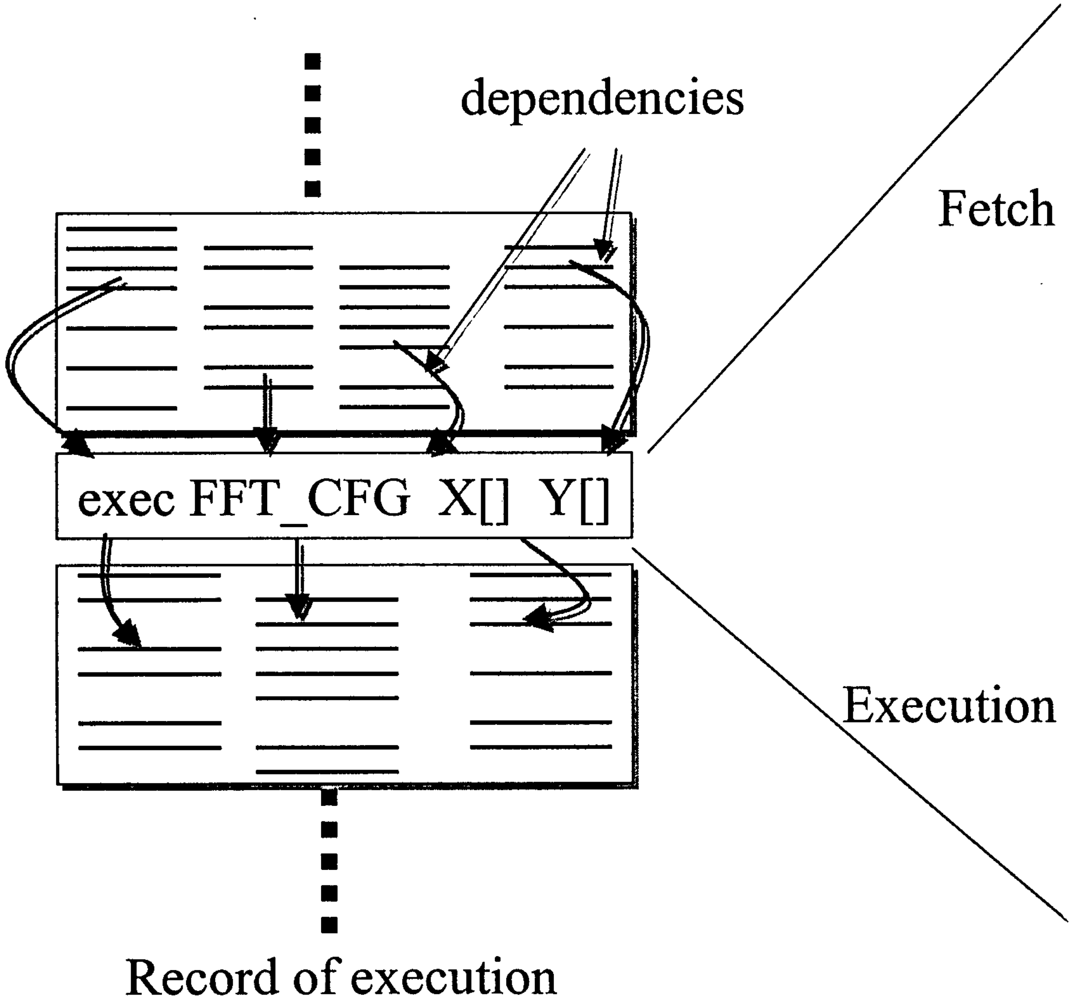
- Processors are good at local optimizations
  - since they can only see a small window of instructions
- Configurations are too big
  - not much room for code movement in local region
- Compilers have the global picture
  - can schedule configuration loading better
  - reuse idle instruction slots far away from local region

Decoupling Fetch And Execute

- decoupled configuration fetch and execute
  - for conventional instructions, fetch time and execute time are of the same order of magnitude
  - for custom instructions (described by configurations) instruction fetch time can be orders of magnitude larger than the instruction execute time
    - hence need to focus on designs that enable fetch time reduction

- This means separate instructions for fetching configurations and executing instructions on configured units

Custom Instruction Execution : Steps

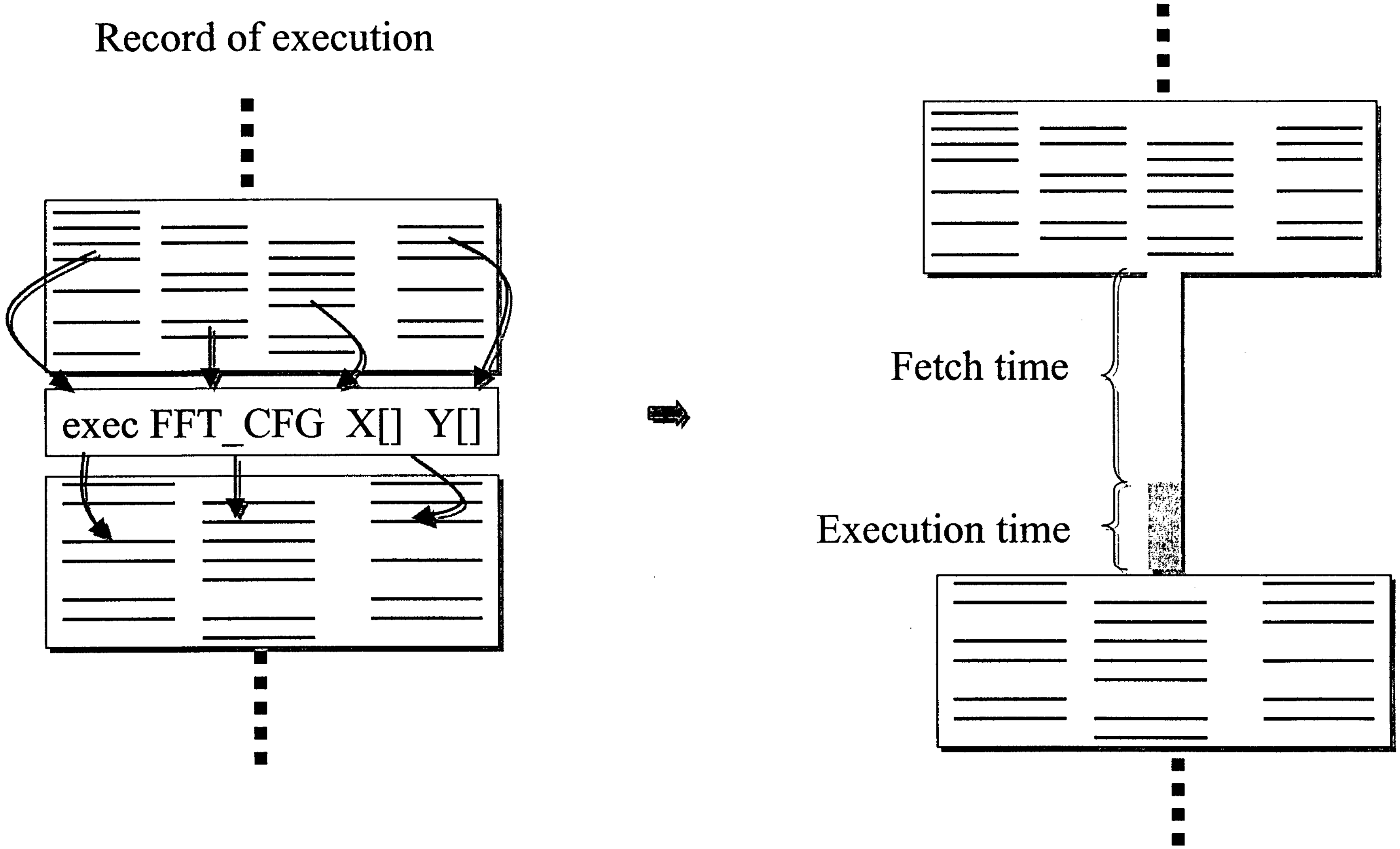


Fetch:
- Allocate space for FFT configuration
- Allocate space for i/o operands
- Load FFT configuration

Execution:
- Load input operand values
- Trigger FFT execution
- Save output operands

If Handled Conventionally

Record of execution

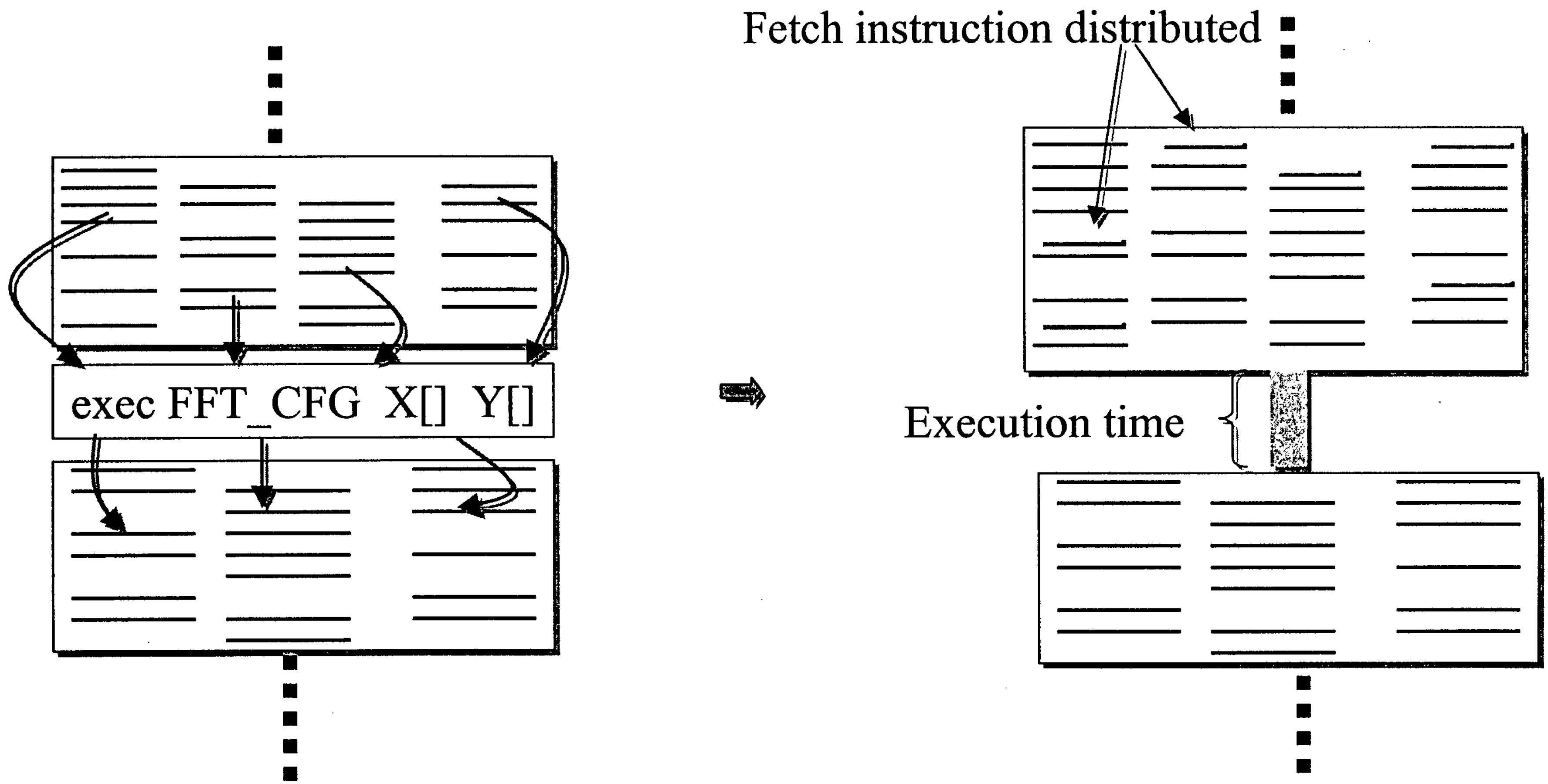


Custom Instruction Execution

However, note that fetch activities are not dependent on any preceding instructions. So one can perform fetch activities as early as possible.

- Fetch
  - Allocate space for FFT configuration
  - Allocate space for i/o operands
  - Load FFT configuration
- Execution
  - Load input operand values
  - Trigger FFT execution
  - Save output operands

Masking Fetch Latency



Record of execution

exec FFT_CFG X[] Y[]

Fetch instruction distributed

Execution time

Configuration Fetch And Execute

- Fetch phase is not dependent on previous instructions
  - can be performed earlier safely
  - if speculated, it may incur extra cost if branch is not taken
  - reduces effective execution time of custom instruction
- Hence, decouple configuration fetching and configuration execution
  - cfg_ld, cfg_lds instructions for loading configurations
  - cfg_exec for triggering configuration execution

Dynamic Configuration Resource Allocation

- Allocation of reconfigurable logic for configured functional units
  - if done by the compiler
    - it gets too machine specific
    - instruction bits consumed for resource (de-)allocation instructions
  - if done by the processor
    - makes hardware more complex, could impact cycle time
    - no impact on compiler or opcode space
- Our choice : dynamically allocated AEPIC’s
  - processor decides where to allocate configured (custom) functional unit in the reconfigurable logic array

Formats For Custom Instructions

- Custom instructions have implicit operands
  - operands to custom instructions are not part of the custom instruction packet
  - operands pre-loaded using operand load instructions

Instruction Formats

```
FFT_OP        X[] , Y[] , SIZE

VECTOR_OP     A[], B[], C[]

CUSTOM_OPA    X1, X2, X3, Y1, Y2

CUSTOM_OPB    X[], Y, Z[]
```

- Custom instructions can have arbitrary input/output formats
- We would like a uniform format for all custom instructions

Implicit Operands For Custom Instructions

- Custom instructions are not a pre-determined set
  - not feasible to allow all possible instruction formats
  - complex instructions means complex instruction decoder
  - also impacts instruction fetch bandwidth
  - why not supply input operands earlier?

- Hence, decouple custom instruction execution from data supply
  - load input operands as early as possible
  - input operands implicitly specified
    - custom unit knows which part of processor state stores its operands

Uniform Instruction Formats

```
                                         ⎧ CFG_INP   X[1]
                                         ⎪ ...
                                         ⎪ CFG_INP   X[N]
                                         ⎪ CFG_INP   SIZE
FFT_OP    X[] , Y[] , SIZE        ⇨     ⎨ ...
                                         ⎪ CFG_EXEC   FFT_OP
                                         ⎪ ...
                                         ⎪ CFG_OUTP  Y[1]
                                         ⎪ ...
                                         ⎩ CFG_OUTP  Y[N]
```

```
                                         ⎧ CFG_INP  X1
                                         ⎪ CFG_INP  X2
                                         ⎪ CFG_INP  X3
                                         ⎪ ...
CUSTOM_OPA  X1, X2, X3, Y1, Y2    ⇨     ⎨ CFG_EXEC  CUSTOM_OPA
                                         ⎪ ...
                                         ⎪ CFG_OUTP  Y1
                                         ⎩ CFG_OUTP  Y2
```

AEPIC
Architecture Specification

Architectural Specification

- Fixed component of processor core
  - based on HPL-PD EPIC architecture
  - incorporates architectural support for predication, control and data speculation, explicitly controlled caches, software pipelining, efficient boolean reduction and several others
- Adaptive extension
  - reconfigurable logic array
  - configuration cache
  - local memory for operands of custom instructions
  - configuration register file
  - resource manager for managing reconfigurable logic resource among configured functional units

NYU

Architectural Specification

- New instructions for
  - allocating resources and loading configurations
  - supplying input operands
  - storing computed results from custom units
  - triggering/suspending execution of instructions on custom configured units
- Definitions of custom functional units and their opcodes not part of the architecture spec!
  - custom functional units not known at manufacture time
- Parameterized architecture
  - customize in each machine description
  - customizable resources are reconfigurable resources, local memories and CRF sizes
  - instruction set not customizable

NYU

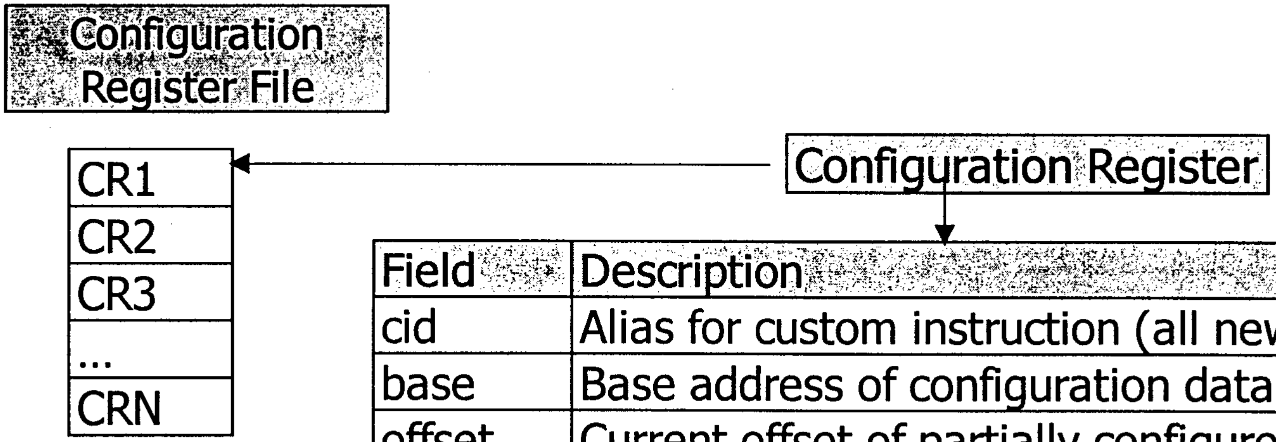
HPL-PD Instruction Set Extensions

instruction type	description
cfg_ld	load configuration data
cfg_lds	load configuration data speculatively
cfg_ldpf	prefetch into configuration cache
cfg_exec	trigger custom instruction
cfg_stick	sticky configured functional unit (will not be evicted)
cfg_kill	delete configured functional unit and reclaim resources
cfg_inp	load input argument
cfg_outp	save output argument
cfg_susp	suspend execution on custom unit
cfg_alloc	allocate configuration id, a configuration register

- Variants for different data types
- Most instructions take configuration register as operand

Architectural Specification

- Architecturally visible state : Configuration Register File (CRF)
- Used by the new instructions for manipulating configurations
- CRF stores information about instantiated configurations
- Additional instructions to copy/update configuration registers



Field	Description
cid	Alias for custom instruction (all new instructions use it)
base	Base address of configuration data in process address space
offset	Current offset of partially configured configuration
size	Configuration size
nio	Number of input operands
niom	Mask for input operand types
noo	Number of output operands
noom	Mask for output operand types

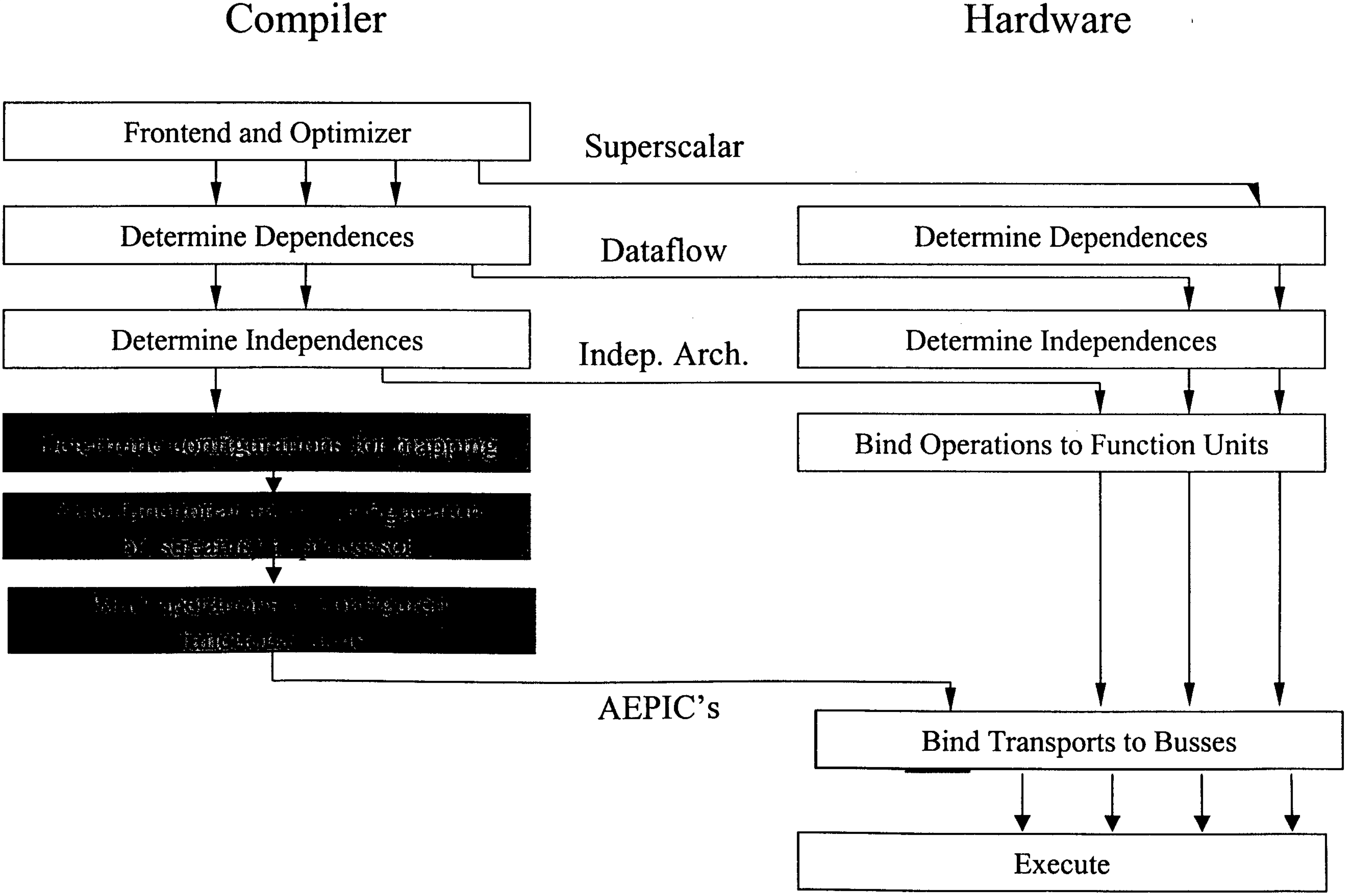
NYU

HPL-PD Instruction Set Extensions

- Example
  - cfg_alloc cr[2], FFT // load FFT configuration, constant FFT is
  - // address of FFT configuration in addr. sp.
  - cfg_lds cr[2] // speculatively load the next word of
  - // FFT configuration (base,offset in cr[2]

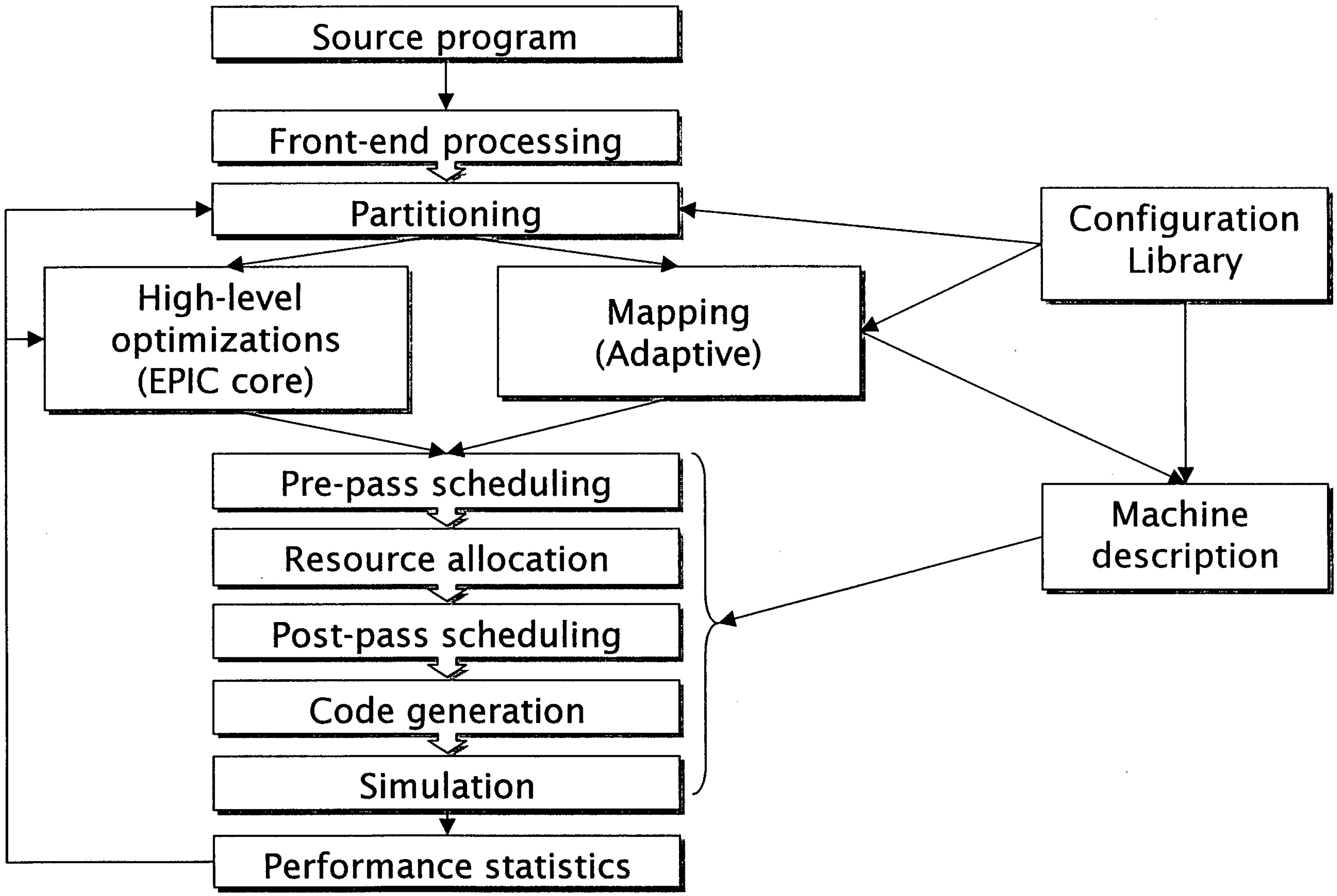
- Exact formats, latency and resource usage of new instructions given in the AEPIC Architecture Specification 1.0
- Non-architecturally visible state, specific architecture of reconfigurable logic not part of architecture spec.

Compiler vs. Processor


Compiler
Hardware
Frontend and Optimizer
Superscalar
Determine Dependences
Dataflow
Determine Dependences
Determine Independences
Indep. Arch.
Determine Independences
Generate configurations for mapping
Bind Operations to Function Units
[illegible] configuration by adaptive processor
Bind operations to configured functional units
AEPIC's
Bind Transports to Busses
Execute

Compiler Infrastructure

Compiler Modules



Source program

Front-end processing

Partitioning

High-level optimizations (EPIC core)

Mapping (Adaptive)

Configuration Library

Machine description

Pre-pass scheduling

Resource allocation

Post-pass scheduling

Code generation

Simulation

Performance statistics

NYU

Compilation challenges

- Code partitioning
- Mapping partitions to reconfigurable logic

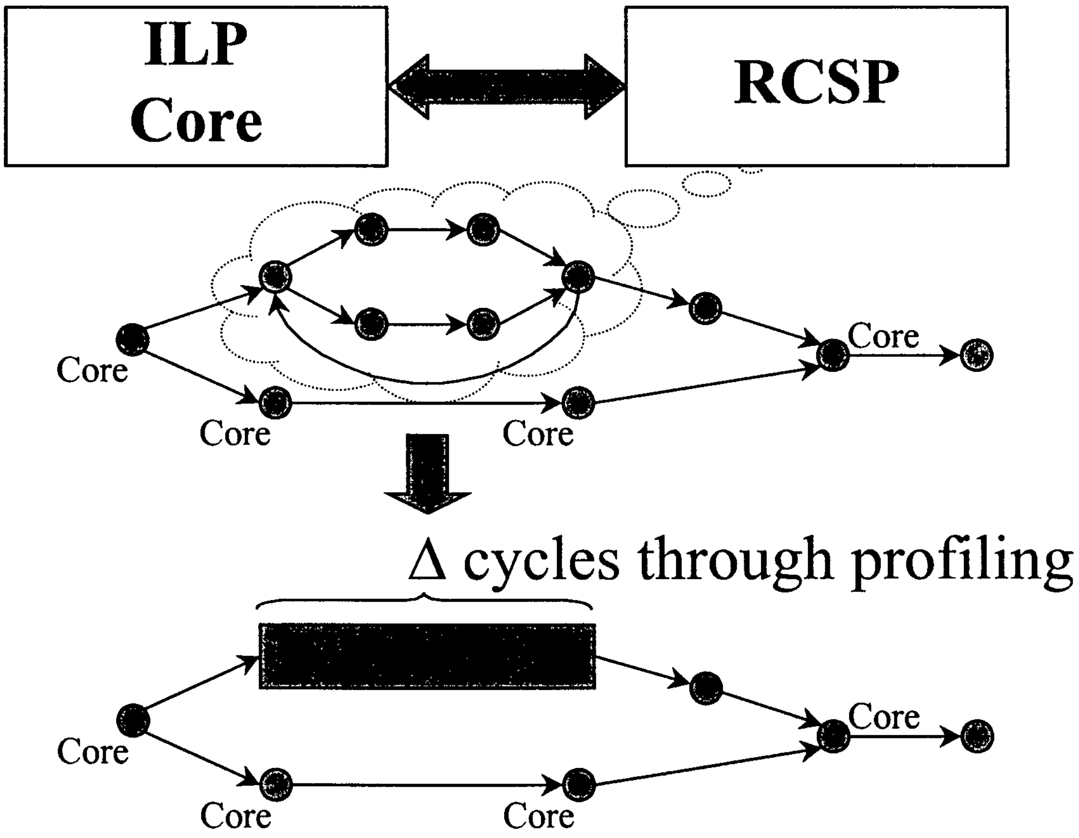
√ Reconfiguration time

√ Configuration scheduling and resource allocation

√ Configuration selection

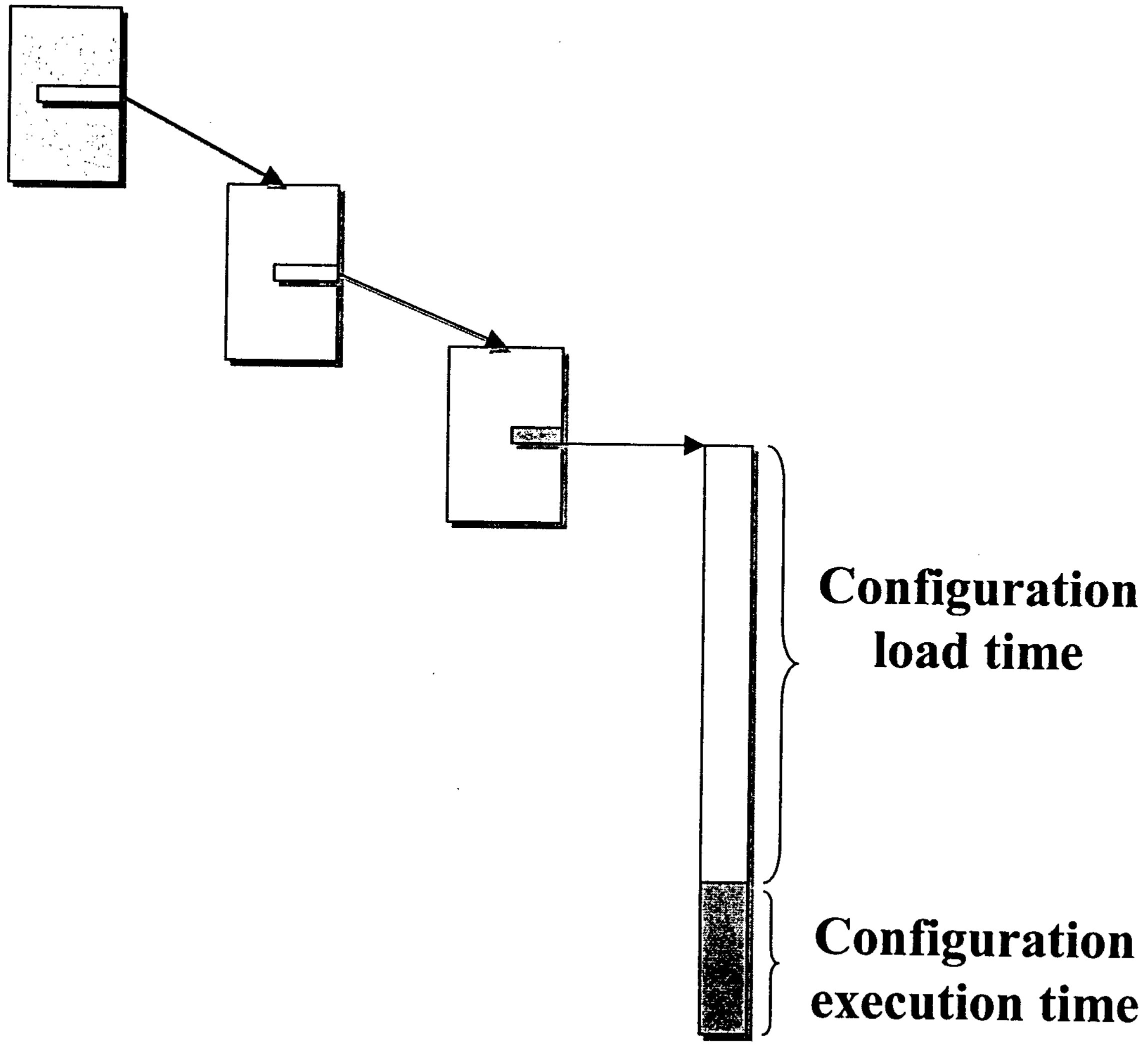
- Intermediate representations

Configurable Scheduling Optimizations

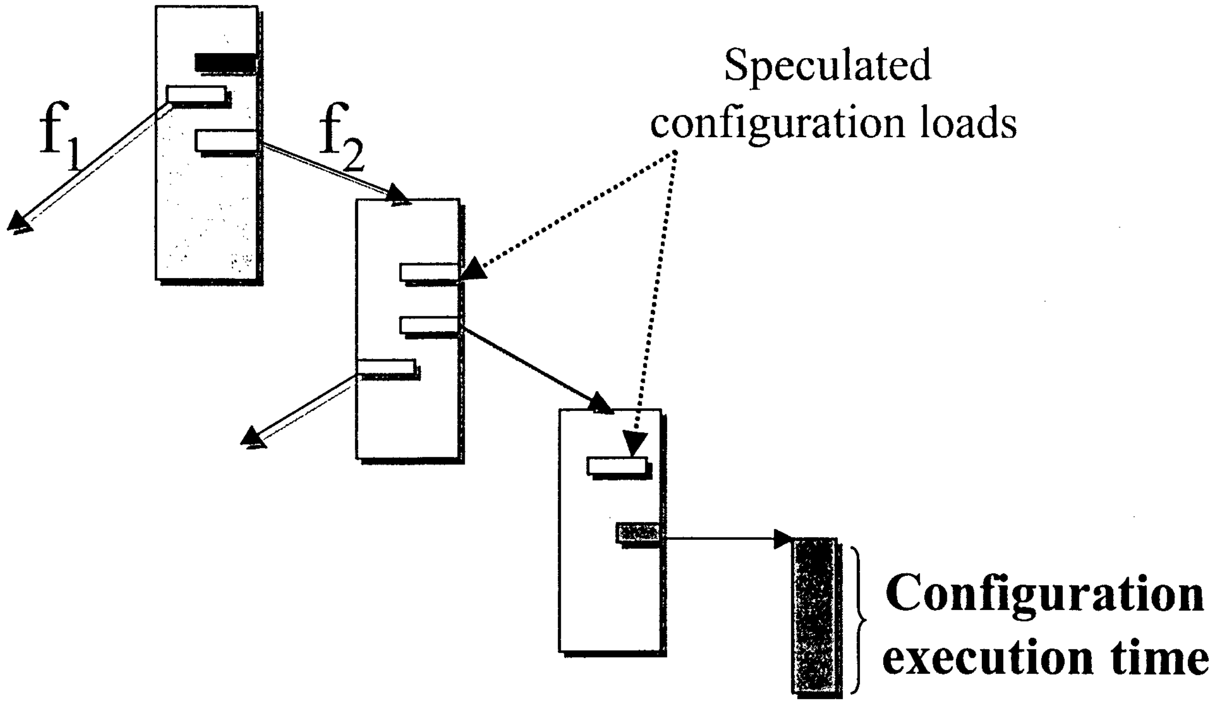


A.Leung, K.Palem , A.Pneuli, S. Talla“A Fast Algorithm for Scheduling Time-constrained Instructions on RCSP”.

The Problem With Long Reconfiguration Times



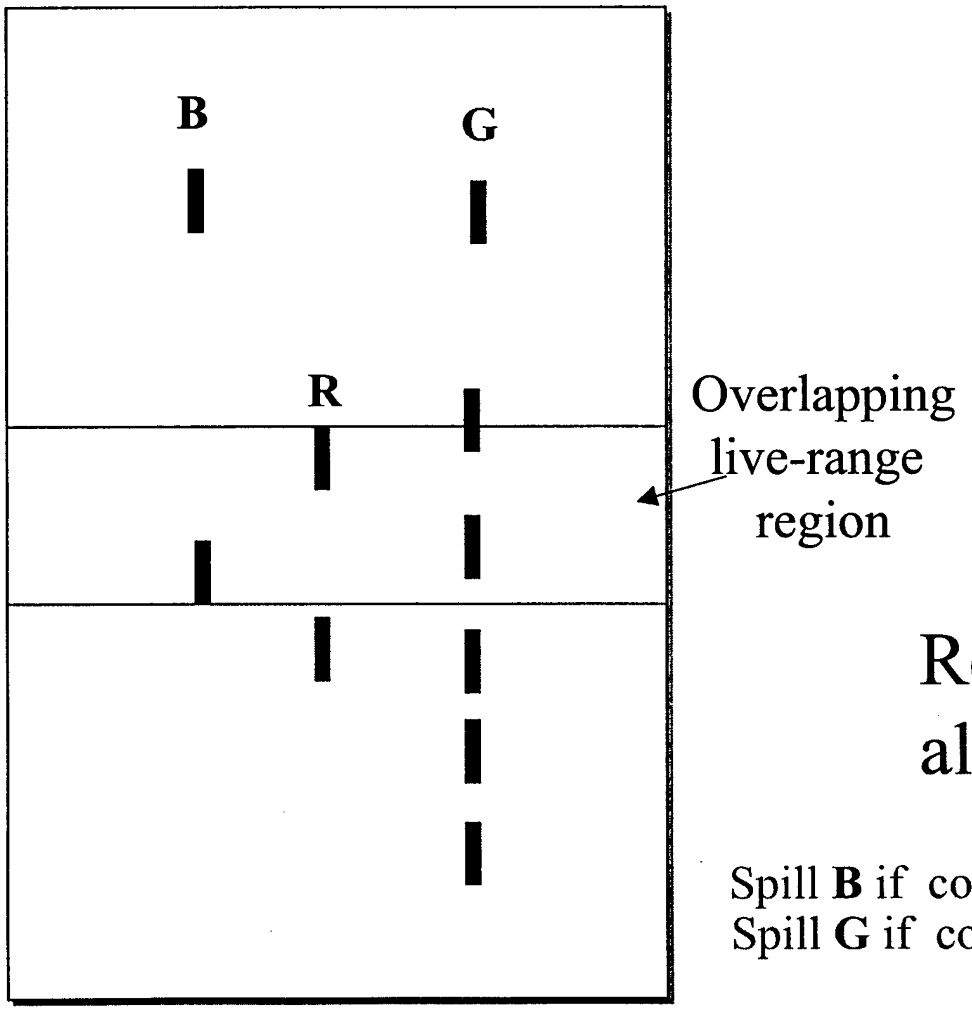
Speculating Configuration Loads



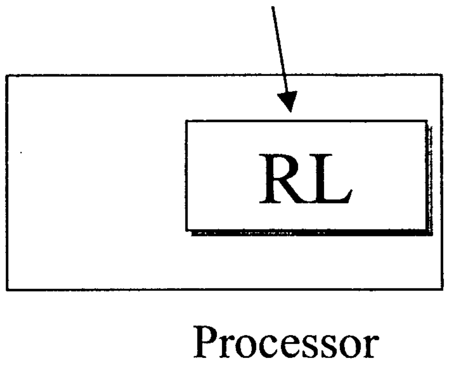
- Mask reconfiguration times!
- Need to know when and where to speculate
- If f1>>f2 do not speculate to “red” empty load slot

Resource allocation for configurations

Record of execution



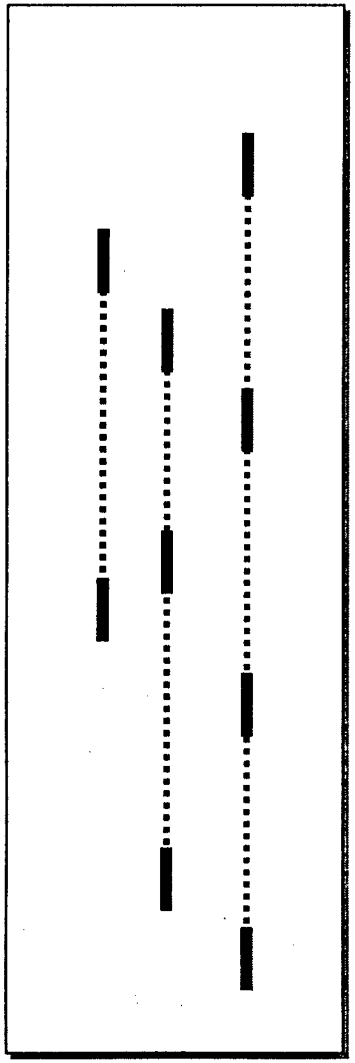
Reconfigurable logic can accommodate only two configurations simultaneously



Reduced to register allocation framework

Spill **B** if configuration size of **B** is same size as **G**
Spill **G** if configuration size of **B** is much larger

Managing Reconfigurable Resource (contd.)



- Configuration Live Ranges
- Directly relates to register allocation problem
- New parameters
  - No need to spill (no self-modifying mappings)
  - Load costs different for different configurations
  - prioritize based on configuration sizes and usage

Spill **B** if configuration size of **B** is same size as **G**
Spill **G** if configuration size of **B** is much larger

Machine Description Enhancements for AEPIC's

Machine Description Requirements

- Retargetability : No built in assumptions about machine in the compiler!
- Easy to understand/modify both by compiler writer and processor architect.
- Language must allow specification of a wide variety of architectures.
- Support automatic generation of tools like assemblers, simulators, verifiers, cost-modelers, etc.
- Support efficient query interfaces to external (compiler) modules.

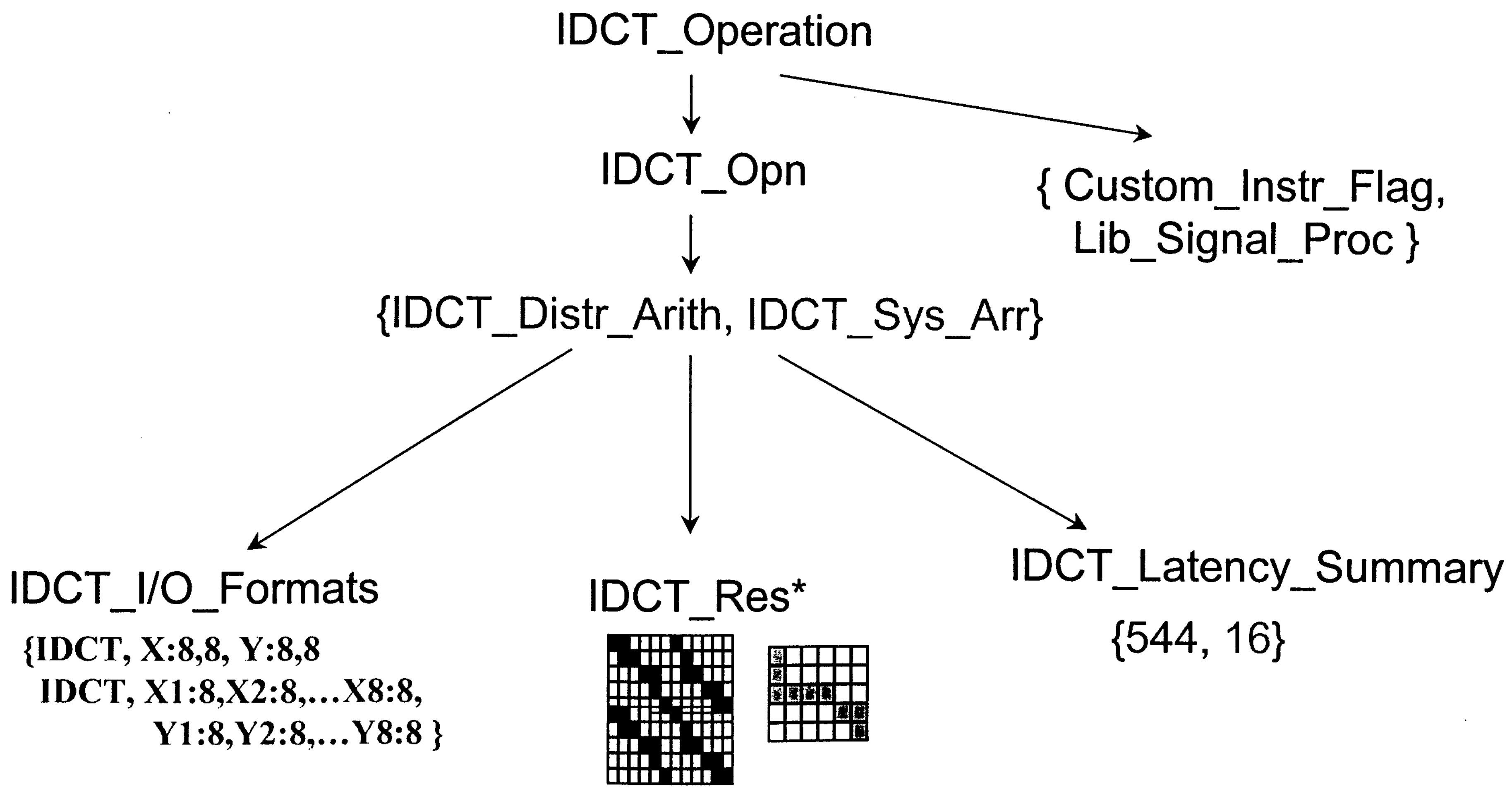
NYU

Requirements for Adaptive EPIC's

- In addition to what is required of a machine description for an EPIC processor, it should provide
  - an ability to describe reconfigurable resource
  - an external interface to the reconfigurable logic resource

- The machine description mechanism should provide a dynamically (during compile time) changing ISA to the compiler

Proposed Extensions (contd.)



IDCT_Operation

IDCT_Opn

{ Custom_Instr_Flag, Lib_Signal_Proc }

{IDCT_Distr_Arith, IDCT_Sys_Arr}

IDCT_I/O_Formats

{IDCT, X:8,8, Y:8,8
IDCT, X1:8,X2:8,...X8:8,
Y1:8,Y2:8,...Y8:8 }

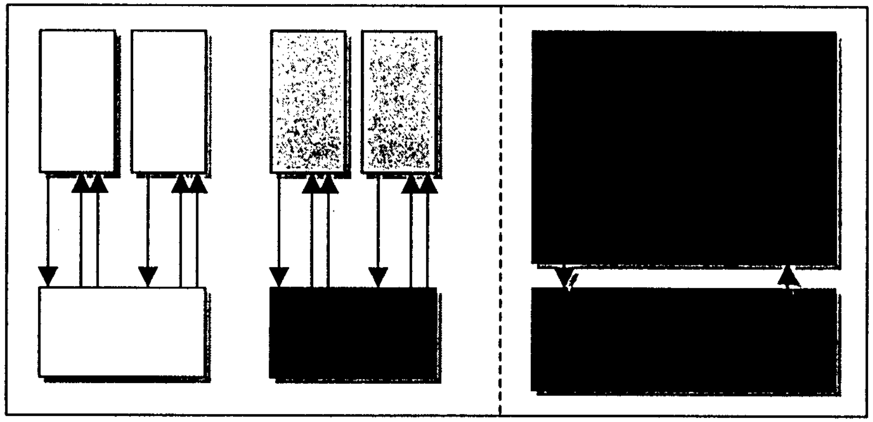
IDCT_Res*

IDCT_Latency_Summary

{544, 16}

* from configuration template files

AEPIC target description



```
GRP_SIZE  64
FPR_SIZE  64
INT_UNITS 2
FLOAT_UNITS 2
RLA_PARAMS  64 128 2
C_CACHE  128 32 1
```

```
r1 = L.W.C2.V1 r2
r1 = ADD r1, r2
ST.C1 r2, r3
CFG_LD  cr2, ADDR
CFG_INP  cr1, r1
CFG_EXEC  cr4
```

MDES Extensions for Adaptive EPIC's

- Custom instructions and base instructions will be handled similarly
- Unlike base instructions, custom instruction resource usages will be pointers to configuration-files

Summary

AEPIC's : A Summary

- AEPIC's
  - at the right granularity for automated compilation
  - leverage wealth of ILP experience
  - simple architectures
- Adaptive computing specific compiler optimizations
  - automatic partitioning and mapping : extremely important but poorly understood - currently library based
  - other problems such as configuration scheduling and allocation can be related to well known optimization problems
- Machine descriptions for retargetability
  - necessary for architectural exploration
  - easy migration path for compiler (and other tools)

Future Directions

NYU

Compiler Framework For AEPIC's

- AEPIC specific compiler optimizations
  - incorporate and validate several configuration scheduling/allocation optimizations
  - tackle partitioning and mapping (semi-automatic)
  - optimization modules need to use profile feedback
  - optimization modules to be parameterized by MDES
- Machine description framework
  - extend MDES with AEPIC architectural features
  - hooks to configuration library
- Simulation, emulation and performance monitoring
  - gather statistics specific to execution on adaptive extension
  - simulation environment adapts to MDES variations

Architectures

- Adaptive targets with DSP/RISC cores
  - use DSP or RISC cores instead of EPIC cores in AEPIC
  - compare cost/performance with respect to EPIC cores
- Extend AEPIC model to scalable AEPIC's
  - just as extra RAM is added to improve memory performance add extra reconfigurable logic and raise processor performance
  - needs innovations in processor-memory interconnect
- Explore adaptivity in other areas of architecture (not just customization of functional units)
  - adaptive cache control logic, malleable caches
    - goes beyond explicit control of caches as in HPL-PD

Emerging Constraints

- Power sensitive adaptation
  - so far, most adaptation geared towards performance gains
  - can we develop techniques to adapt architecture to consume optimal (low) power for a given application?
    - need metrics for instruction combinations that are power friendly
    - information useful for power sensitive scheduling
    - need architectural enhancements to enable power sensitive adaptation
- Other constraints : timing, size and weight
- Compiler-architecture co-design
  - fine tune architecture for specific application domains
  - tune for performance/power
  - potential for high impact in embedded systems

Compiler Infrastructure

- Trimaran infrastructure and architectural model ideal
  - easy to generate MDES variations rapidly
  - retargetability built into the compiler infrastructure
- Compiler - architecture co-design (with Georgia Tech.)
  - VHDL/Verilog structural descriptions for Trimaran targets
    - for area, power estimates
    - path to place and route tools for targeting Xilinx, Altera, ...
  - domain specific customization
- Target commercial architectures
  - e.g., code generators for IA-64
- Extensions to intermediate representations (IR)
  - to handle timing constraints
  - annotations for power/area sensitive optimizations

Summary

NYU

Technical Achievements

- Developed compiler models for EPIC processors extended with reconfigurable logic
- Initial design of suitable extensions to current machine description framework to target AEPIC processors
- Efficient algorithms for instruction scheduling
  - in the presence of long latency instructions
  - under user specified time constraints
- Developed Time-C for specifying time
  - translate source (time) constraint specifications to constraints on intermediate graphs
  - developed algorithms for scheduling time critical instructions
- Released compiler infrastructure for EPIC processors
  - conducted performance tests for simplified AEPIC model

NYU

Impact to the Community

- Model for targeting a modern optimizing compiler
  - vendor neutral parametric processor
  - proof of performance improvements
  - fast compilation
- Dynamic configuration
  - configuration caches
  - compiler optimizations e.g. speculation
- Notation for expressing constraints
  - timing
  - partitioning and mapping (to be done)

NYU

Contact Information

ReaCT-ILP
New York University

719 Broadway,
New York, NY 10003
react-ilp.cs.nyu.edu

NYU

Final Review

Krishna V. Palem
ReaCT-ILP Laboratory
http://react-ilp.cs.nyu.edu
New York University

This work was supported by DARPA Contract No. DABT63-96-C-0049
performed under the USC MAARC project.

Project Overview

NYU

A Summary of Accomplishments

- Architectural models for compilation
- Compiler optimizations
- Application studies
- Compiler infrastructure

Our Goals

- Define a processor model that serves as a target for an optimizing compiler, that
  - takes advantage of the technology growth curve
  - leverages the advantages of adaptive logic
- Architect an optimizing compiler framework for targeting the defined architecture that
  - demonstrates performance improvements for challenge applications
  - achieves the performance goals in reasonable compilation time

Architectural Models

Architectural Models for ACS Compilation

- Developed compiler models for EPIC processors extended with reconfigurable logic
  - EPIC processors : Merced, McKinley
- Initial design of suitable extensions to current machine description framework to target adaptive EPIC processors

RCSP : A Reduced Configuration Space Processor and its Programming Environment,
K.V.Palem, S. Talla, HPEC'97, Lincoln Labs, MIT, Sept., 1997

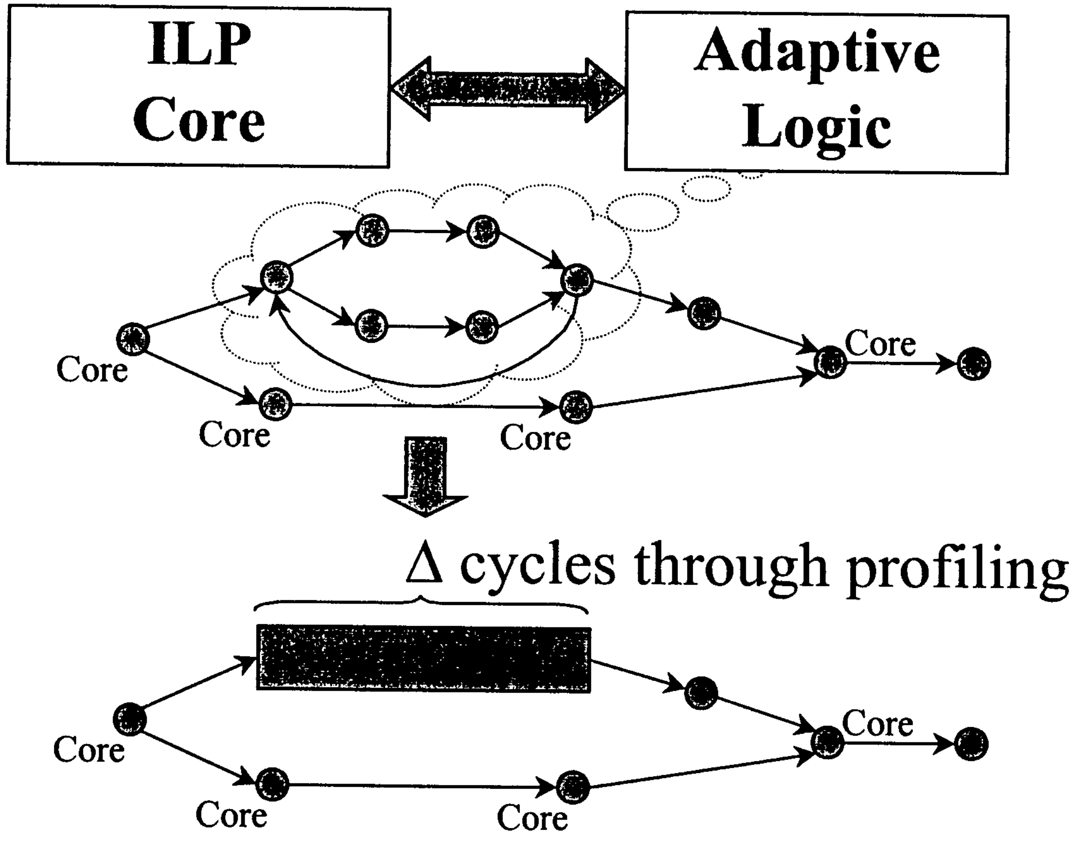
Adaptive Explicitly Parallel Instruction Computing,
K.V.Palem, S.Talla, P.Devaney, Australasian Computer Architecture Conference, January 1999.

Language Support
And
Compiler Optimizations

Scheduling and Time-C

- Developed fast polynomial time algorithms for instruction scheduling
  - in the presence of long latency instructions
  - under user specified time constraints
- Developed Time-C for specifying time
  - translate source constraint specifications to constraints on intermediate graphs
  - developed algorithms for scheduling time critical instructions
    - of relevance to developing support for automatic partitioning and mapping

Instruction Scheduling Optimizations for Configurable Hardware



A.Leung, K.Palem , A.Pneuli, S. Talla “A Fast Algorithm for Scheduling Time-constrained Instructions on AEPIC” (preliminary version).

Time-C and Time_Tract

- Language independent syntax for “light weight upgrade”
- Can be used with C, C++...
- Embedded applications involving real-time constraints

Time-C : A Time Constraints Language For ILP Compilation
K. V. Palem, A. Leung, A. Pnueli, PACT’98

Compilation challenges

- Code partitioning
- Mapping partitions to reconfigurable logic

√ Reconfiguration time

√ Configuration scheduling and resource allocation

√ Configuration selection

- Intermediate representations

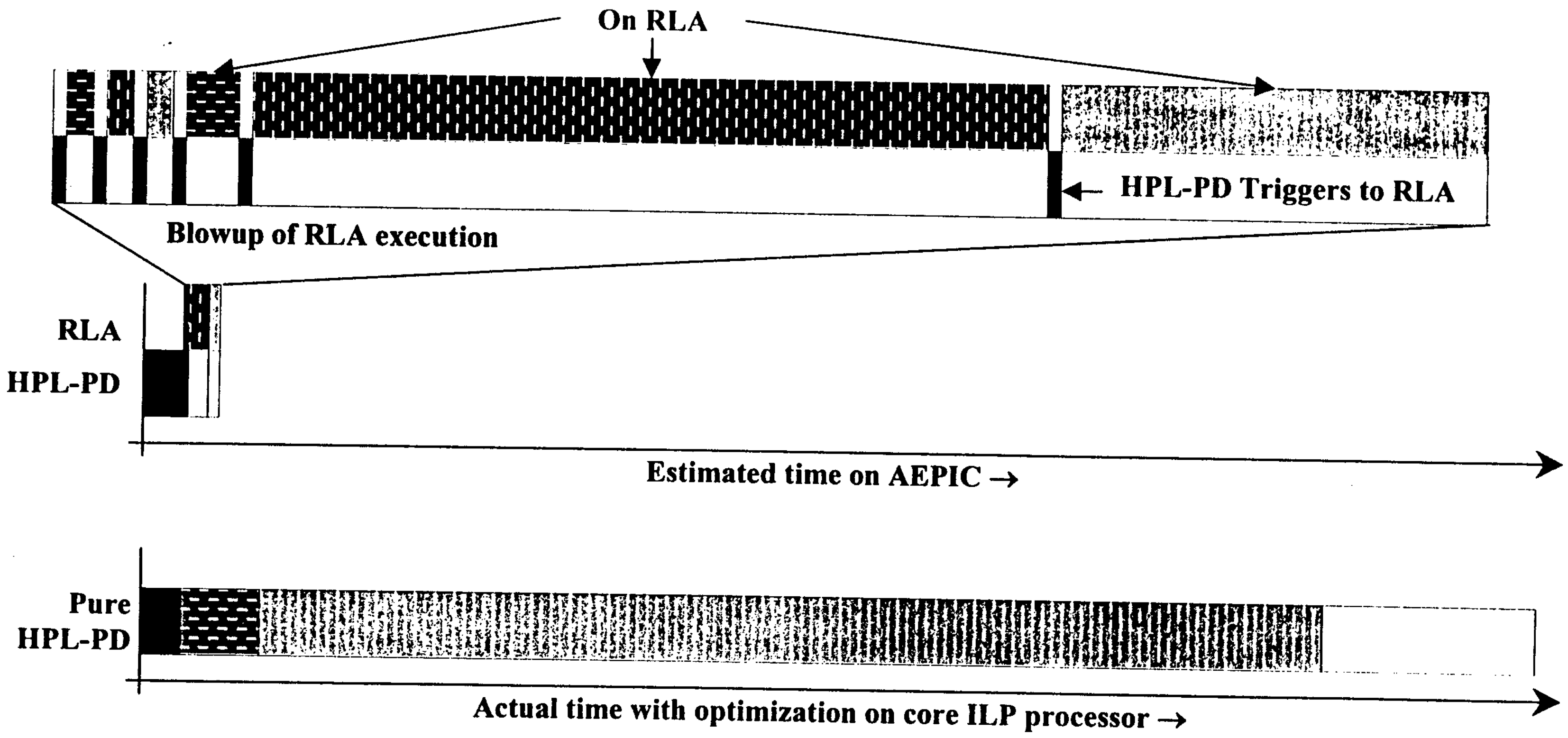
Application Studies

Application Performance Studies

- Several applications from signal, image and media processing domains mapped to proposed adaptive EPIC target

- Applications considered from following benchmarks
  - Honeywell ACS stressmarks
  - MediaBench from UCLA (Bill-Mangione Smith)
  - RAW Benchmarks from MIT (Anant Agarwal)
  - Others from Spec95, public domain

NBTR : Execution Profile



■ NBTR Initialization

HPL-PD alone:
- ▨ Execution of CornerTurn
- ▥ Execution of FFT
- □ Execution of Scaling & Thresholding

On AEPIC:
- ▨ Execution of CornerTurn
- ▦ Execution of AFT
- ▤ Execution of Scaling & Thresholding
- ■ Loads and Trigger the Configurations

Sample Performance Studies

Application	9-issue EPIC	Optimized compilation for 9-issue EPIC	A-EPIC (SCDA)	A-EPIC Speedup
MPEG2 decoder	n/a	439486198	80686602	5.4
IDEA Encryption (one round)	118	118	18	6.4
32-tap FIR	31533	13491	384	35
NBTR (input sampled 10 times)	22529978	13731573	532800	25.8
IDCT	12127	6633	544	12.2

- Unlimited reconfigurable resource
- Access to reconfigurable array through memory on a 64 bit separate memory bus, array supports same PE architecture as XC6200
- Assumed reconfigurable array clock to be same as core clock

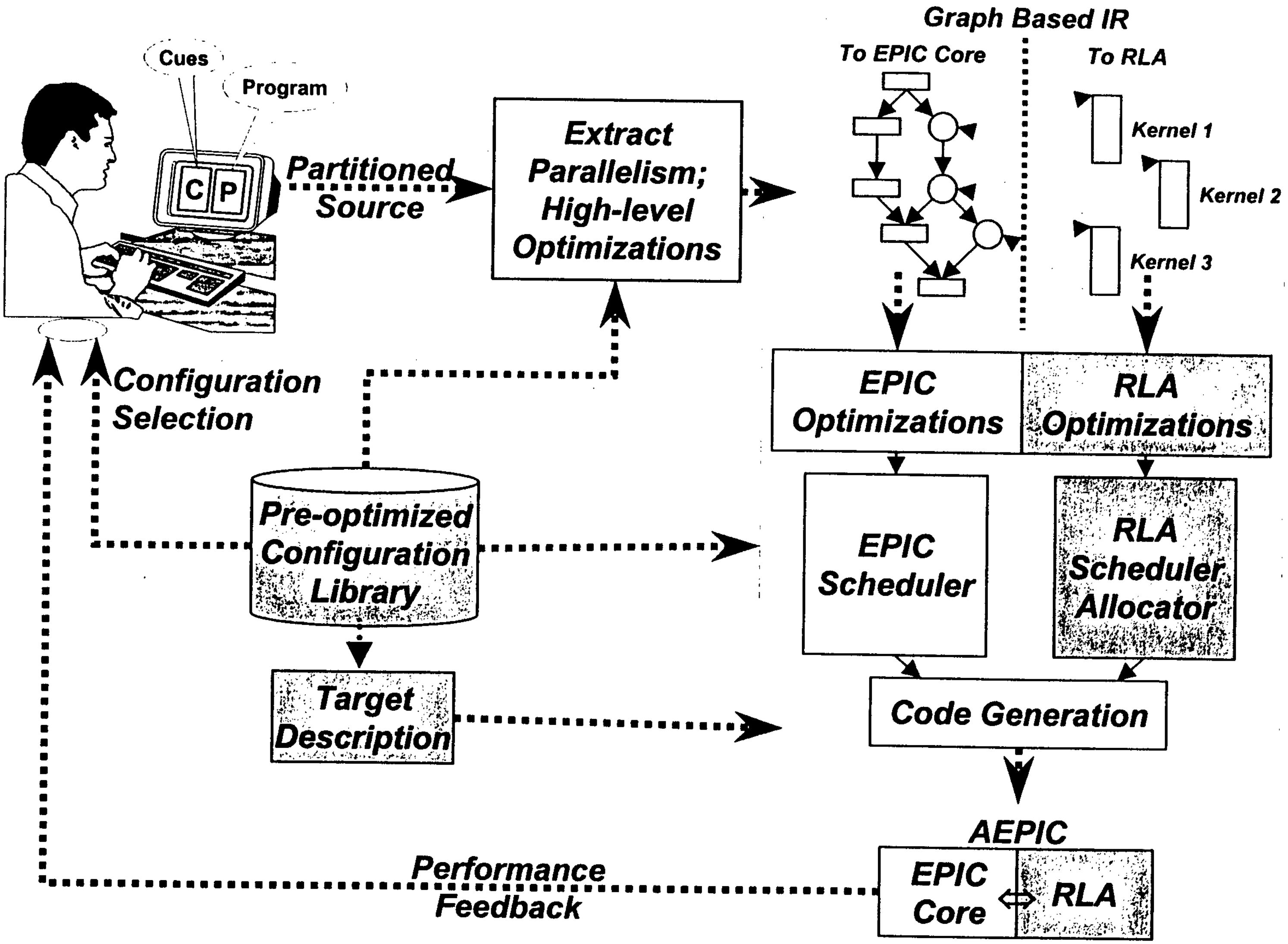
Adaptive Explicitly Parallel Instruction Computing,
K.V.Palem, P. Devaney, S. Talla, Australasian Computer Architecture Conference, January 1999.

Compiler Infrastructure

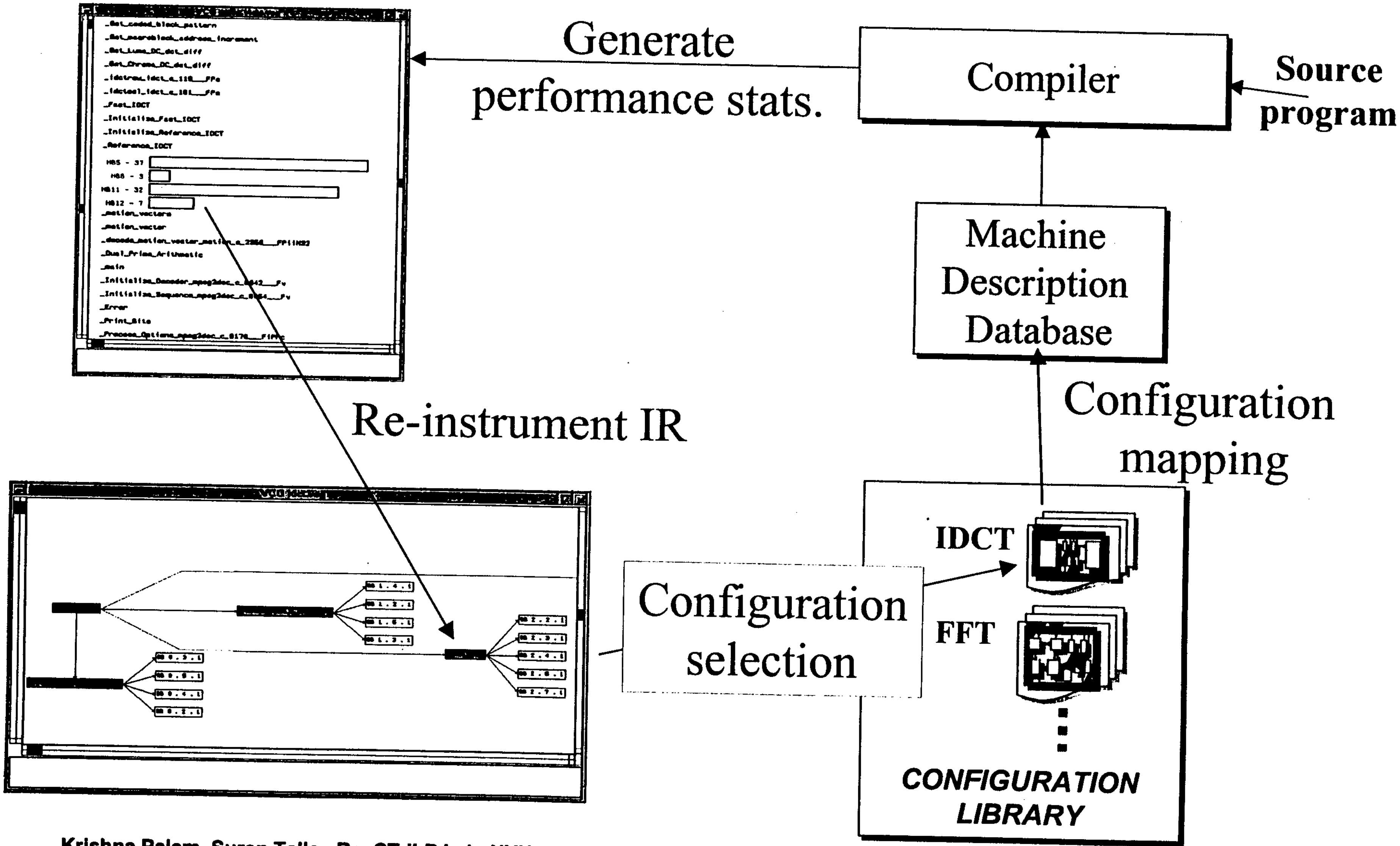
Infrastructure Goals

- Faster and easier compilation
  - extend current compilation frameworks
- Tackle configuration loads
  - dynamic reconfiguration overheads
- Contain partitioning and mapping
  - a focus for the USC MAARC project
- Explore a variety of reconfigurable device parameters

Compilation framework for AEPIC's

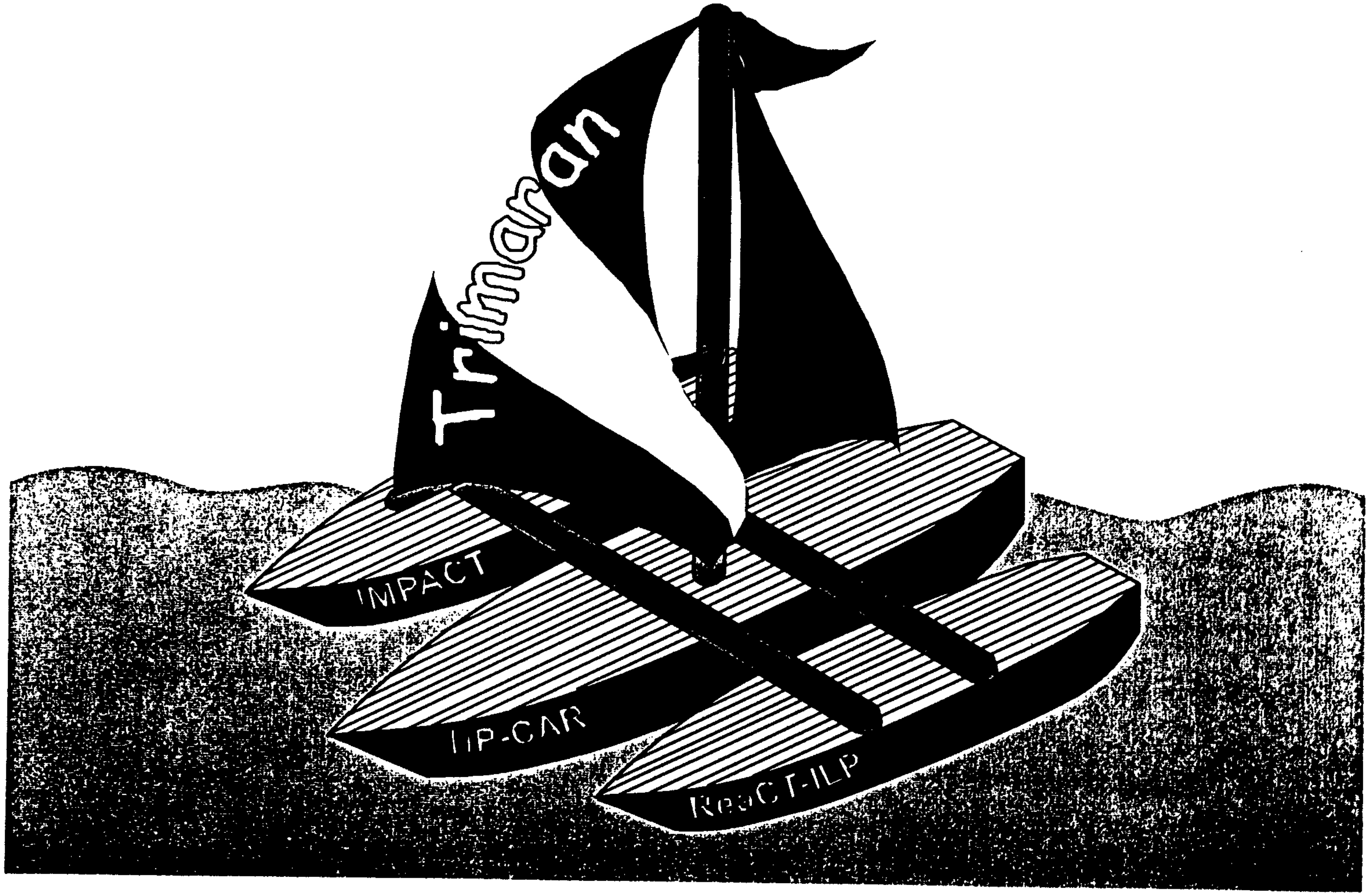


Compilation Methodology Using Trimaran



```
_Get_coded_block_pattern
_Get_macroblock_address_increment
_Get_Luma_DC_dct_diff
_Get_Chroma_DC_dct_diff
_idctrow_idct_c_118__FPs
_idctcol_idct_c_181__FPs
_Fast_IDCT
_Initialize_Fast_IDCT
_Initialize_Reference_IDCT
_Reference_IDCT
 MB5 - 37
  MB6 - 3
MB11 - 32
 MB12 - 7
_motion_vectors
_motion_vector
_decode_motion_vector_motion_c_2364__FPiiN23
_Dual_Prime_Arithmetic
_main
_Initialize_Decoder_mpeg2dec_c_0642__Fv
_Initialize_Sequence_mpeg2dec_c_0664__Fv
_Error
_Print_Bits
_Process_Options_mpeg2dec_c_0178__FiPPc
```

Trimaran Infrastructure for Research in Instruction-level Parallelism



Publications

- RCSP : A Reduced Configuration Space Processor and its Programming Environment, K.V.Palem, S. Talla, HPEC'97, Lincoln Labs, MIT, Sept., 1997 (poster presentation).
- TimeC: A Time Specification Language for ILP Processor Compilation, A. Leung, K. V. Palem, A. Pnueli, 5th Annual Australasian Conference on Real-time Systems, Sept., 1998.
- A Fast Algorithm for Scheduling Time Constrained Instructions on Processors with ILP, A. Leung, K. V. Palem, A. Pnueli, International Conference on Parallel Architectures and Compilation Techniques, Oct., 1998.
- Reconfigurable Computing: High Performance Embedded Computing = ILP + Reconfigurable : A Novel Architecture and its Compiler, K. V. Palem, S. Talla, Adelaide, Australia, Sept., 1998 (invited talk)
- Adaptive Explicitly Parallel Instruction Computing, K.V.Palem, P. Devaney, S. Talla, Australasian Computer Architecture Conference, January 1999.

NYU

Work In Progress

- Instruction Scheduling for Adaptive EPIC Processors, A. Leung, K. V. Palem, A. Pnueli, S. Talla.
- Adaptive EPIC Processors, Architectural Specification 1.0 and Validation using Trimaran, A. Leung, H. Kim, R. Rabbah, S. Talla.
- Machine Description Framework for AEPIC Processors, K. V. Palem, S. Talla.

Year - 3
Review

NYU

A Summary of Accomplishments

- Adaptive Explicitly Parallel Instruction Computing (AEPIC)
  - architectural models for compilation
- Compiler optimizations
  - fast polynomial time instruction scheduling
  - scheduling under time constraints
- Compiler infrastructure
  - enhance MDES to incorporate AEPIC features
  - incorporate configuration scheduling and allocation
- Benchmark studies

Architectures

NYU

Motivation

- Demands of Embedded Computing
  - Faster, cheaper processors
  - Shorter times to market
- Poor scalability of superscalars
  - complex control units
- EPIC / VLIW
  - Simpler architectures
  - Known compilation technology
- FPGA / Reconfigurable logic
  - Fine grained parallelism
  - Explicit control over micro-architectural features
  - Fast static communication

Technology and Application Trends

- Feature size and the effect on signal delay
- Cost of verification and test of new designs
- The new media/embedded shift
- Chip density

NYU

The Impact Of Technology Trends

- As Wire Delays Become Significant, focus on architectures that
  - do not involve long distance communication
  - distribute control and data processing logic

- Impact of rising verification and test costs
  - keep the architecture simple and regular
  - move complex decision making logic from processor to higher level tools (compiler)

NYU

The Impact Of Technology Trends (contd.)

- Lots of hardware parallelism available
  - can accommodate approx. 50 pentiums on one die in 6 years

However,

- Conventional architectures and compilation
  - cannot expose enough parallelism in applications
  - even the “superb” model yields an ILP < 10 on average

- Need for new architectures and compilation techniques!

Application Trends Summary

- Real-time processing
- Packed 8-, 16-, and 32-bit integer data
- Continuous data streams
- Fine grain parallelism
- Long integer arithmetic, table look-ups
- Common kernels (small code size)
- Low temporal reuse
- High spatial locality

The Impact of Media Application Trends

- Simple regular architectures are desirable
  - scope for lots of MIMD processing
  - those that are tuned for media kernels
  - need newer caching technology
    - requirements of predictability, high throughput
    - low temporal reuse, high spatial reuse

What Is The Response Elsewhere To All This?

Architecture Research Approaches

- Past (conventional) approaches
  - better instruction fetch/issue
  - improved instruction processing
  - better prediction (branches, aliases)
  - statically scheduled variants of VLIW's
- Novel (different) approaches
  - Reconfigurable processors
  - IRAM and variants
  - Simultaneous multi-threading
  - On-chip multi-processing

NYU

State Of Compilation Technology

- Compilation for past variants
  - well known technology
  - drawback: bottleneck of “conventional compilation”
- Compilation for radically different architectures
  - no known efficient and automatic compilation technique
  - possibility for breaking through the standard compilation bottleneck

Two Noteworthy Directions

- Reconfigurable Processors
  - let compiler handle everything
  - no commitment to a particular architecture
  - compiler generates architecture and code for it

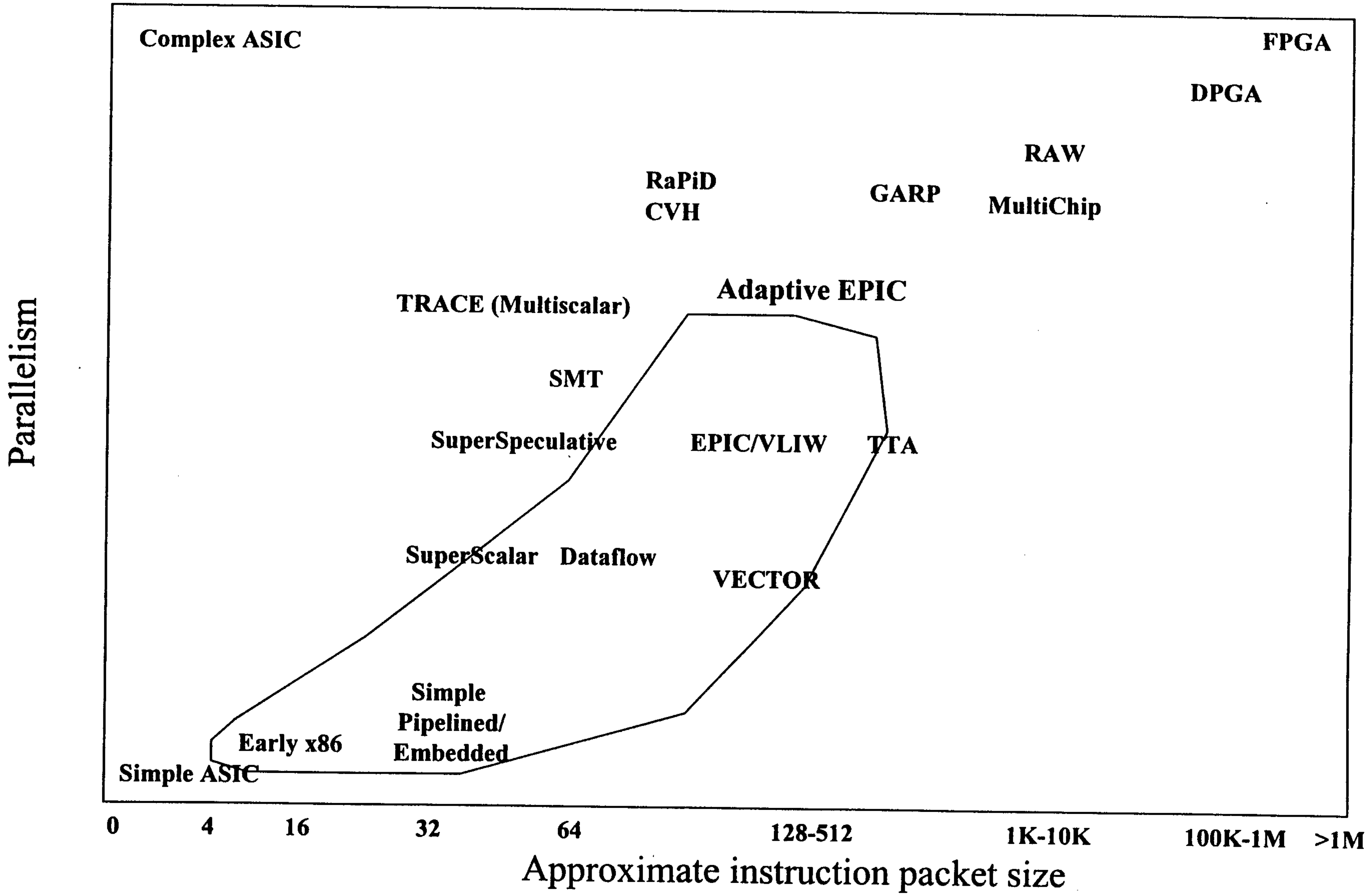
- Explicitly Controlled Architectures
  - simplify architectures as much as possible
  - architectural template is a known, conventional one
  - compiler handles a lot of processor's decision making
    - explicitly control issue, scheduling, allocation
  - Explicitly Parallel Instruction Computing (EPIC)
    - subset of explicitly controlled architectures

NYU

What are the hurdles?

- Poor compilation times
  - lack of correspondence between standard IR and final configurations
  - place and route inherently complex
- Additional runtime overheads
  - large configuration size implies high reconfiguration costs
  - this also implies context switches are very costly
- Lack of convenient abstract models, language support
  - models for algorithm development (e.g. RMESH, USC-MAARC)
  - models for compiler targets (ReaCT-ILP)
  - language support for hardware structural information
    - but not as complex as HDL's

What can be efficiently compiled for today?



What can we infer?

- Design/verification costs
  - Simple, regular architectures
- Signal delays
  - Shorter connections ; local interactions
- Media and embedded processing
  - High throughput, highly compute intensive processing, many integer types, customization, temporal predictability
- Not enough ILP through standard compilation
  - Customized/special purpose compilation?
  - Adaptive architectures?

⇩

Adaptive Explicitly Parallel Instruction Computing?

NYU

Current Effort

Adaptive Explicitly Parallel Instruction Computing Architectures

NYU

Our Goals

Adaptive EPIC Architectures

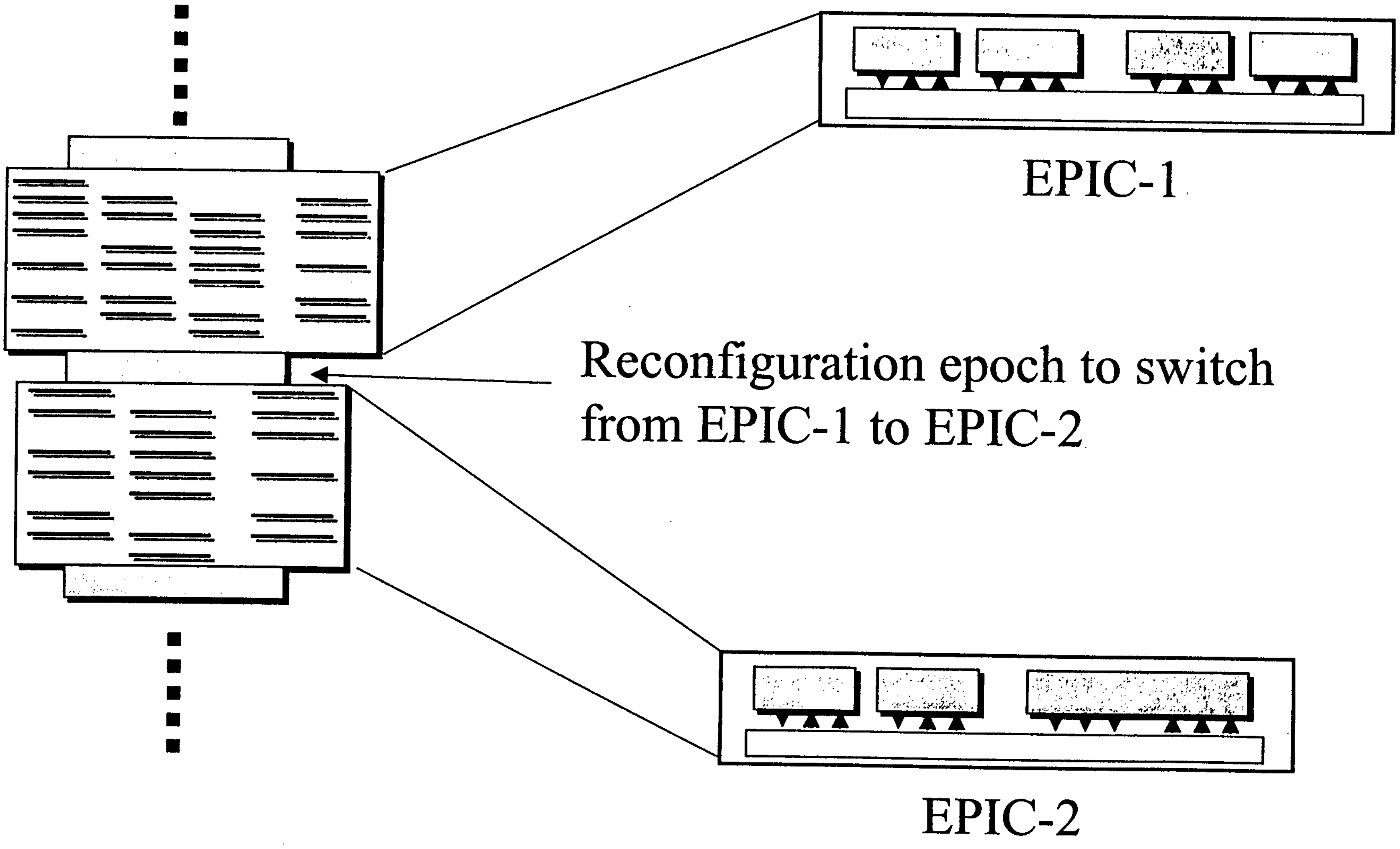
Combine advantages of EPIC's and reconfigurable logic

- Fast automatic compilation
- Explore a variety of reconfigurable processor architectures
  - parameterize hardware space starting with HPL-PD

NYU

Adaptive EPIC execution model

Record of execution

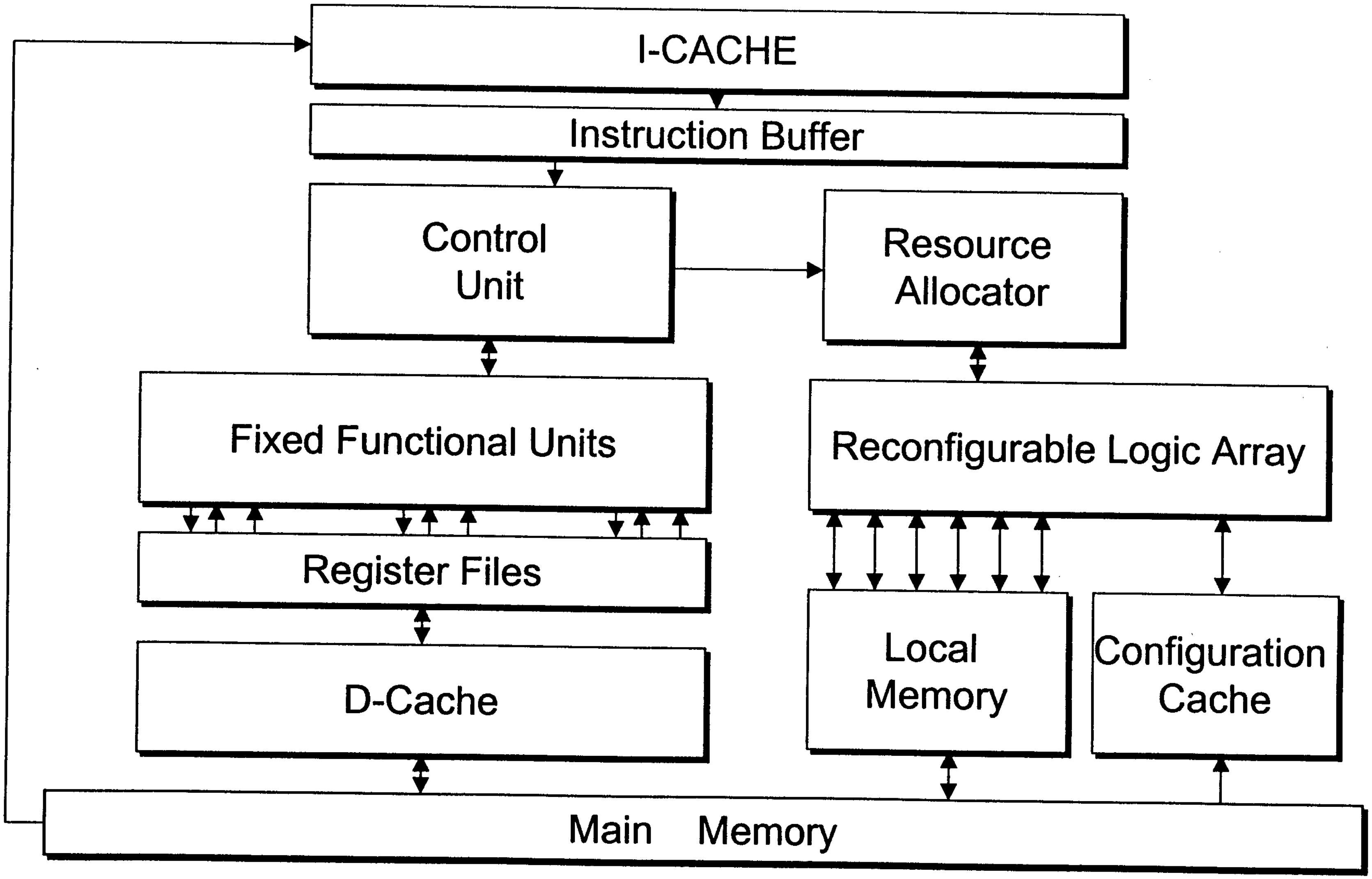


EPIC-1

Reconfiguration epoch to switch from EPIC-1 to EPIC-2

EPIC-2

Our Adaptive EPIC Architecture

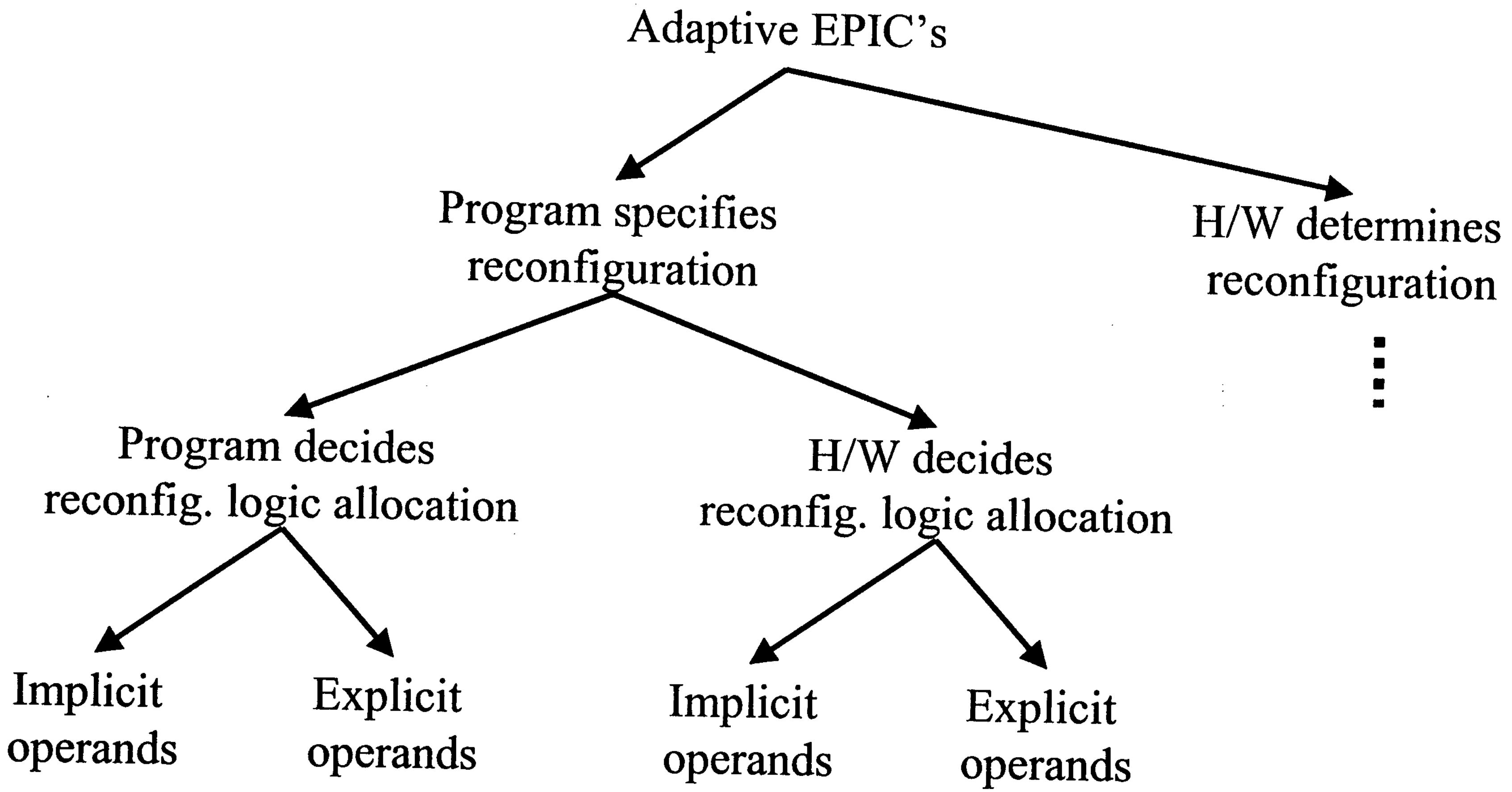


AEPIC Issues

- The issue is of deciding the partitioning of tasks between the compiler and the processor
- The tasks of specific interest and unique to AEPIC's
  - When to reconfigure?
  - Which configurations to keep and which to evict?
  - Where in the reconfigurable logic array should a configuration be instantiated?
  - How are the input operands and results communicated to and from configured functional units?

NYU

A Taxonomy of AEPIC Architectures


Adaptive EPIC's

Program specifies reconfiguration

H/W determines reconfiguration

Program decides reconfig. logic allocation

H/W decides reconfig. logic allocation

Implicit operands

Explicit operands

Implicit operands

Explicit operands

NYU

The Adaptive EPIC Space of Interest

- Statically scheduled
  - program specifies when and what to reconfigure
- decoupled configuration fetch and execute
- Dynamically allocated
  - processor decides where to allocate configured (custom) functional unit in the reconfigurable logic array
- Custom instructions have implicit operands
  - operands to custom instructions are not part of the custom instruction packet
  - operands pre-loaded using operand load instructions
- Explicitly controlled configuration caches

AEPIC
Rationale For Important Design Decisions

Statically Scheduled AEPIC's

- Statically scheduled
  - program specifies when and what to reconfigure

Why?

- Processors are good at local optimizations
  - since they can only see a small window of instructions
- Configurations are too big
  - not much room for code movement in local region
- Compilers have the global picture
  - can schedule configuration loading better
  - reuse idle instruction slots far away from local region

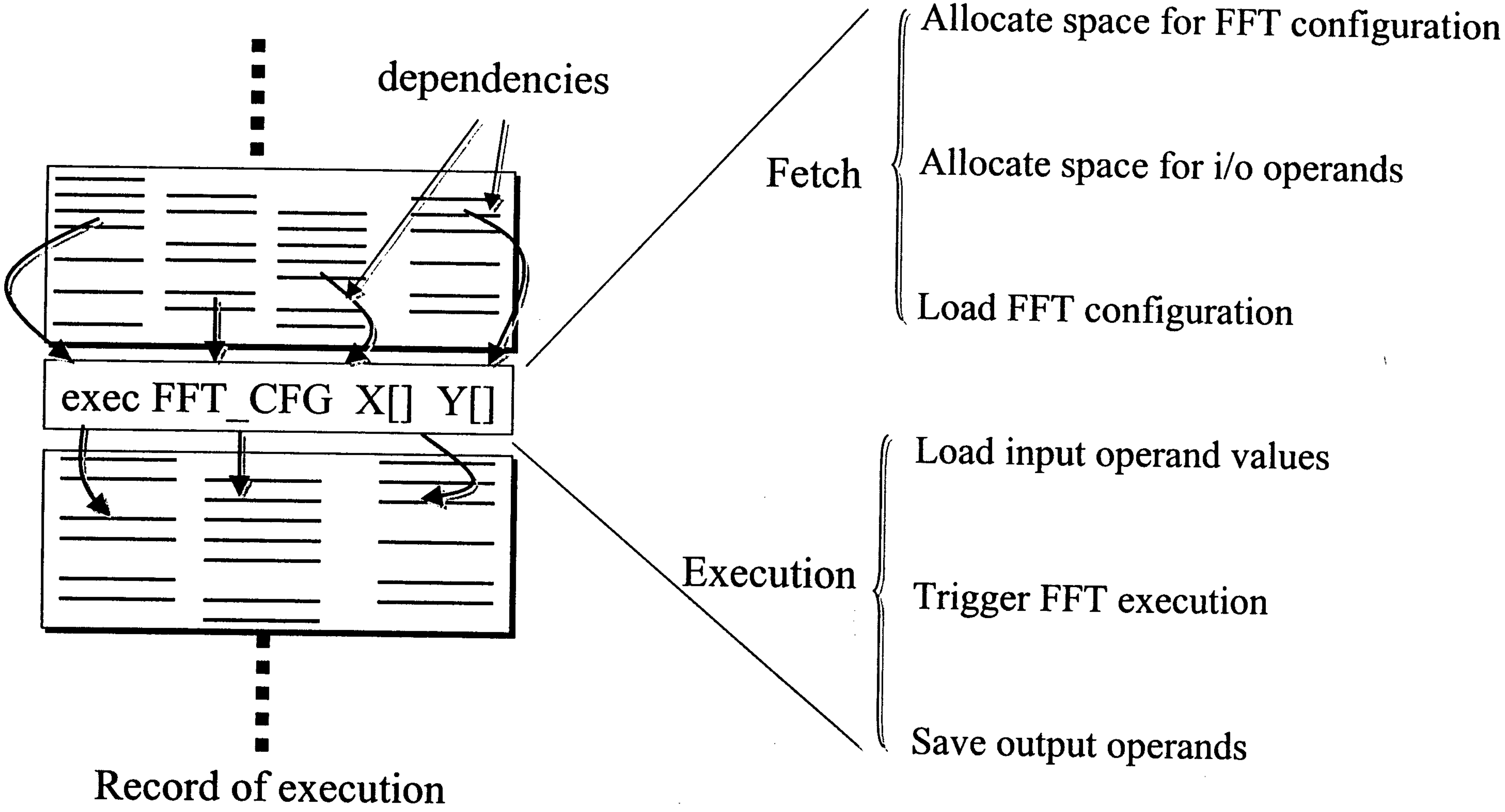
NYU

Decoupling Fetch And Execute

- decoupled configuration fetch and execute
  - for conventional instructions, fetch time and execute time are of the same order of magnitude
  - for custom instructions (described by configurations) instruction fetch time can be orders of magnitude larger than the instruction execute time
    - hence need to focus on designs that enable fetch time reduction

- This means separate instructions for fetching configurations and executing instructions on configured units

Custom Instruction Execution : Steps

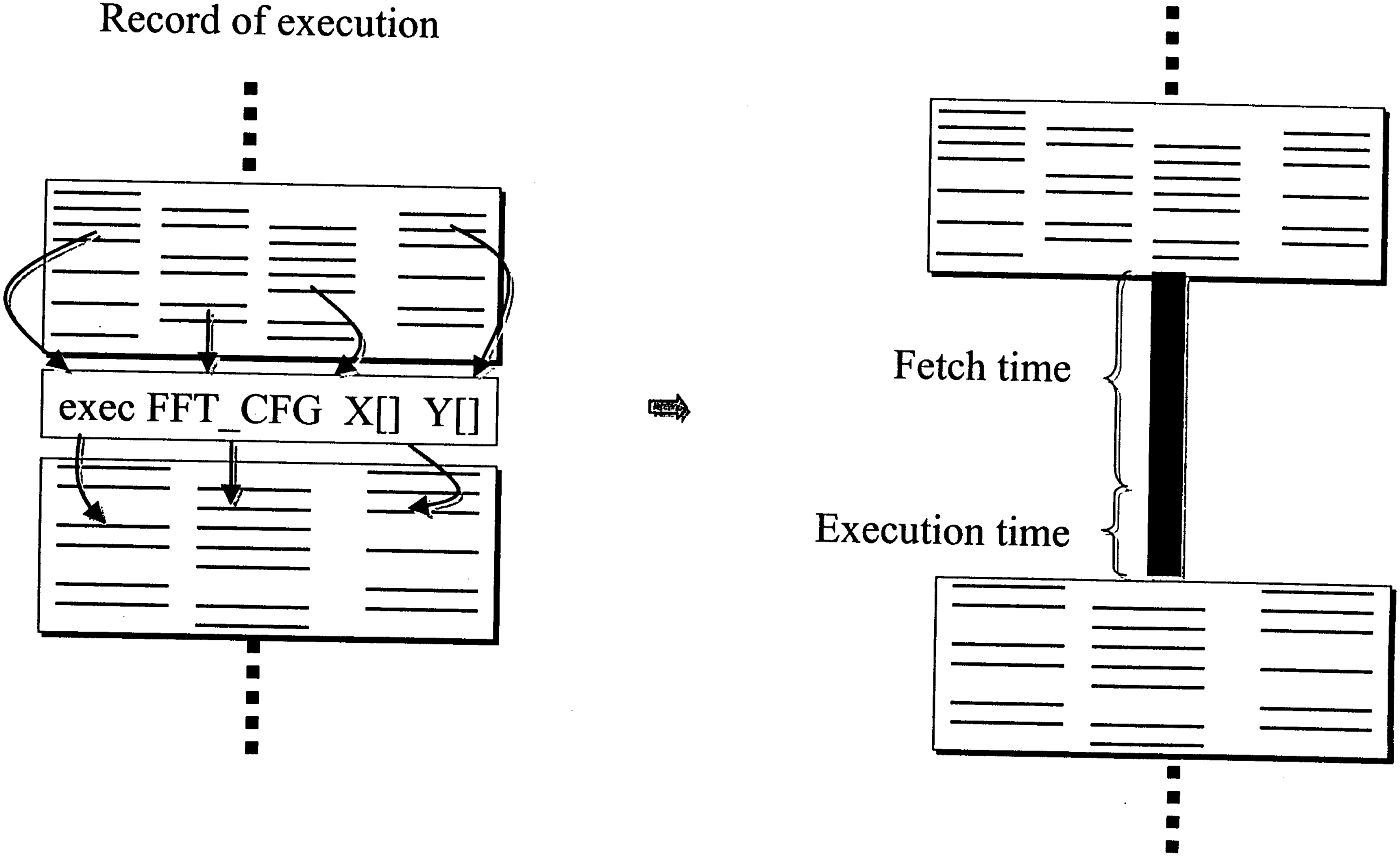


Fetch
- Allocate space for FFT configuration
- Allocate space for i/o operands
- Load FFT configuration

Execution
- Load input operand values
- Trigger FFT execution
- Save output operands

If Handled Conventionally

Record of execution



exec FFT_CFG X[] Y[]

Fetch time

Execution time

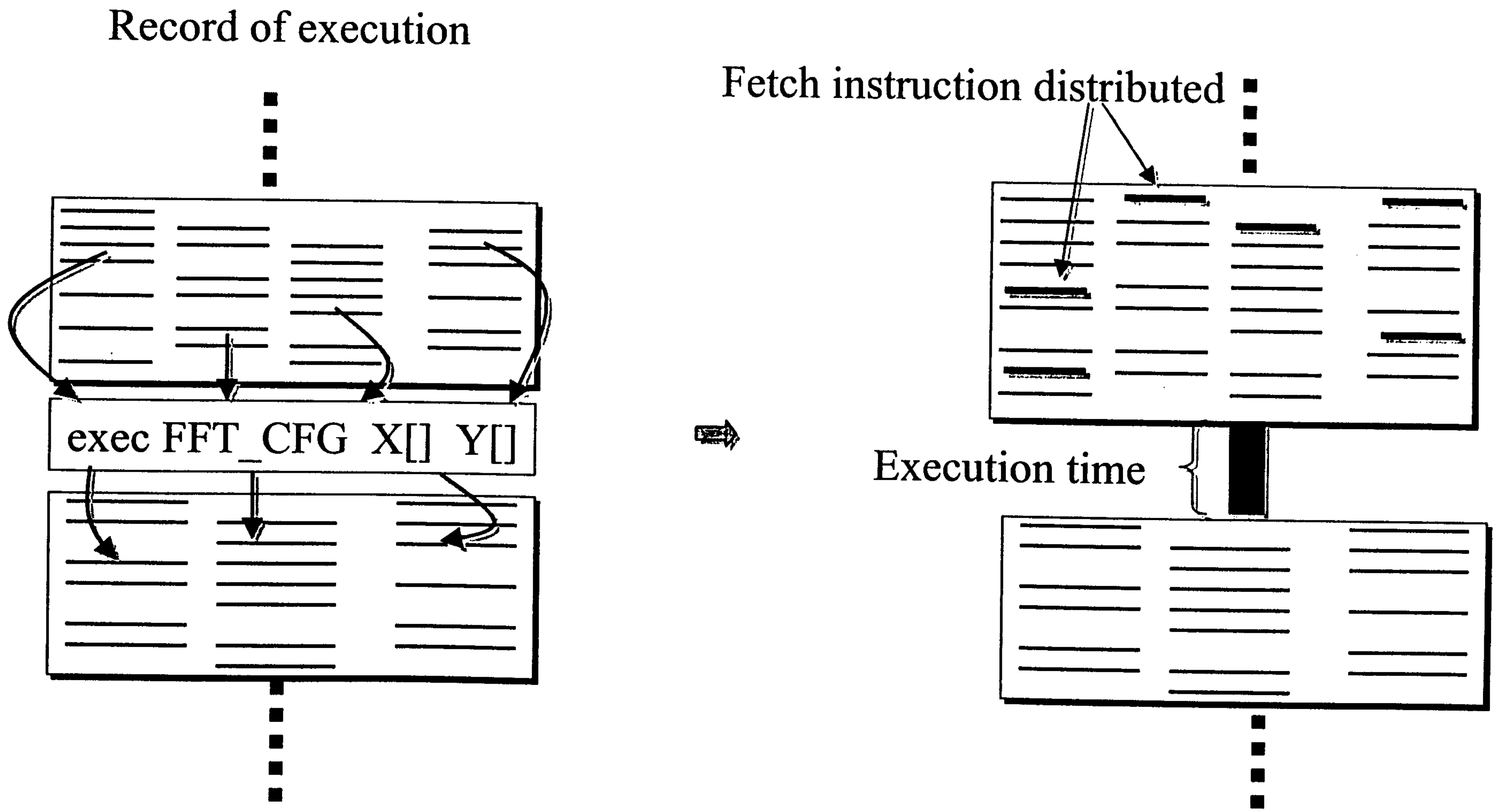
Custom Instruction Execution

However, note that fetch activities are not dependent on any preceding instructions. So one can perform fetch activities as early as possible.

Fetch
- Allocate space for FFT configuration
- Allocate space for i/o operands
- Load FFT configuration

Execution
- Load input operand values
- Trigger FFT execution
- Save output operands

Masking Fetch Latency



Record of execution

exec FFT_CFG X[] Y[]

Fetch instruction distributed

Execution time

NYU

Configuration Fetch And Execute

- Fetch phase is not dependent on previous instructions
  - can be performed earlier safely
  - if speculated, it may incur extra cost if branch is not taken
  - reduces effective execution time of custom instruction
- Hence, decouple configuration fetching and configuration execution
  - cfg_ld, cfg_lds instructions for loading configurations
  - cfg_exec for triggering configuration execution

Dynamic Configuration Resource Allocation

- Allocation of reconfigurable logic for configured functional units
  - if done by the compiler
    - it gets too machine specific
    - instruction bits consumed for resource (de-)allocation instructions
  - if done by the processor
    - makes hardware more complex, could impact cycle time
    - no impact on compiler or opcode space
- Our choice : dynamically allocated AEPIC's
  - processor decides where to allocate configured (custom) functional unit in the reconfigurable logic array

NYU

Formats For Custom Instructions

- Custom instructions have implicit operands
  - operands to custom instructions are not part of the custom instruction packet
  - operands pre-loaded using operand load instructions

Instruction Formats

```
FFT_OP        X[] , Y[] , SIZE
VECTOR_OP     A[], B[], C[]
CUSTOM_OPA    X1, X2, X3, Y1, Y2
CUSTOM_OPB    X[], Y, Z[]
```

- Custom instructions can have arbitrary input/output formats
- We would like a uniform format for all custom instructions

NYU

Implicit Operands For Custom Instructions

- Custom instructions are not a pre-determined set
  - not feasible to allow all possible instruction formats
  - complex instructions means complex instruction decoder
  - also impacts instruction fetch bandwidth
  - why not supply input operands earlier?

- Hence, decouple custom instruction execution from data supply
  - load input operands as early as possible
  - input operands implicitly specified
    - custom unit knows which part of processor state stores its operands

Uniform Instruction Formats

```
FFT_OP    X[] , Y[] , SIZE     ⇨    CFG_INP   X[1]
                                    . . .
                                    CFG_INP   X[N]
                                    CFG_INP   SIZE
                                    . . .
                                    CFG_EXEC  FFT_OP
                                    . . .
                                    CFG_OUTP  Y[1]
                                    . . .
                                    CFG_OUTP  Y[N]
```

```
CUSTOM_OPA  X1, X2, X3, Y1, Y2     ⇨    CFG_INP  X1
                                        CFG_INP  X2
                                        CFG_INP  X3
                                        . . .
                                        CFG_EXEC  CUSTOM_OPA
                                        . . .
                                        CFG_OUTP  Y1
                                        CFG_OUTP  Y2
```

AEPIC
Architecture Specification

NYU

Architectural Specification

- Fixed component of processor core
  - based on HPL-PD EPIC architecture
  - incorporates architectural support for predication, control and data speculation, explicitly controlled caches, software pipelining, efficient boolean reduction and several others
- Adaptive extension
  - reconfigurable logic array
  - configuration cache
  - local memory for operands of custom instructions
  - configuration register file
  - resource manager for managing reconfigurable logic resource among configured functional units

Architectural Specification

- New instructions for
  - allocating resources and loading configurations
  - supplying input operands
  - storing computed results from custom units
  - triggering/suspending execution of instructions on custom configured units
- Definitions of custom functional units and their opcodes not part of the architecture spec!
  - custom functional units not known at manufacture time
- Parameterized architecture
  - customize in each machine description
  - customizable resources are reconfigurable resources, local memories and CRF sizes
  - instruction set not customizable

NYU

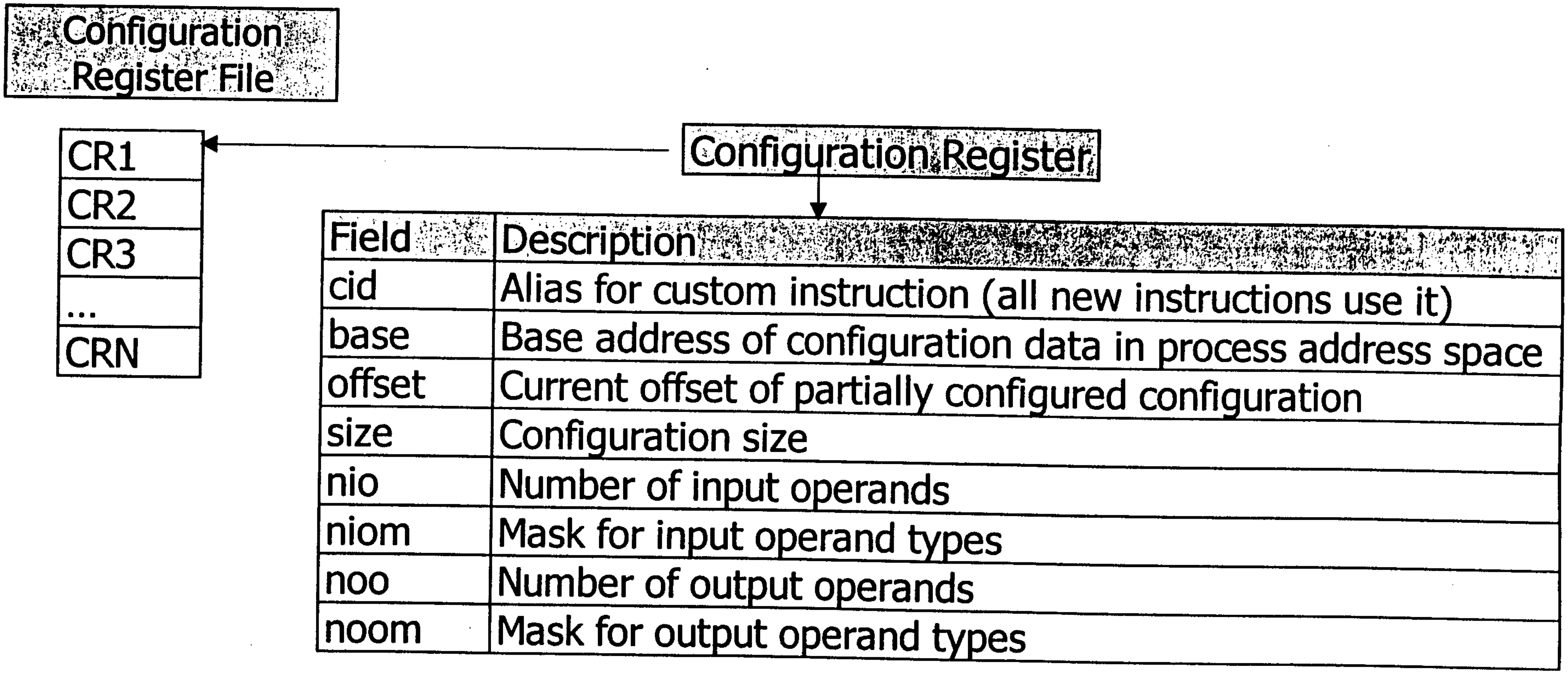
HPL-PD Instruction Set Extensions

instruction type	description
cfg_ld	load configuration data
cfg_lds	load configuration data speculatively
cfg_ldpf	prefetch into configuration cache
cfg_exec	trigger custom instruction
cfg_stick	sticky configured functional unit (will not be evicted)
cfg_kill	delete configured functional unit and reclaim resources
cfg_inp	load input argument
cfg_outp	save output argument
cfg_susp	suspend execution on custom unit
cfg_alloc	allocate configuration id, a configuration register

- Variants for different data types
- Most instructions take configuration register as operand

Architectural Specification

- Architecturally visible state : Configuration Register File (CRF)
- Used by the new instructions for manipulating configurations
- CRF stores information about instantiated configurations
- Additional instructions to copy/update configuration registers



Configuration Register File

CR1
CR2
CR3
...
CRN

Configuration Register

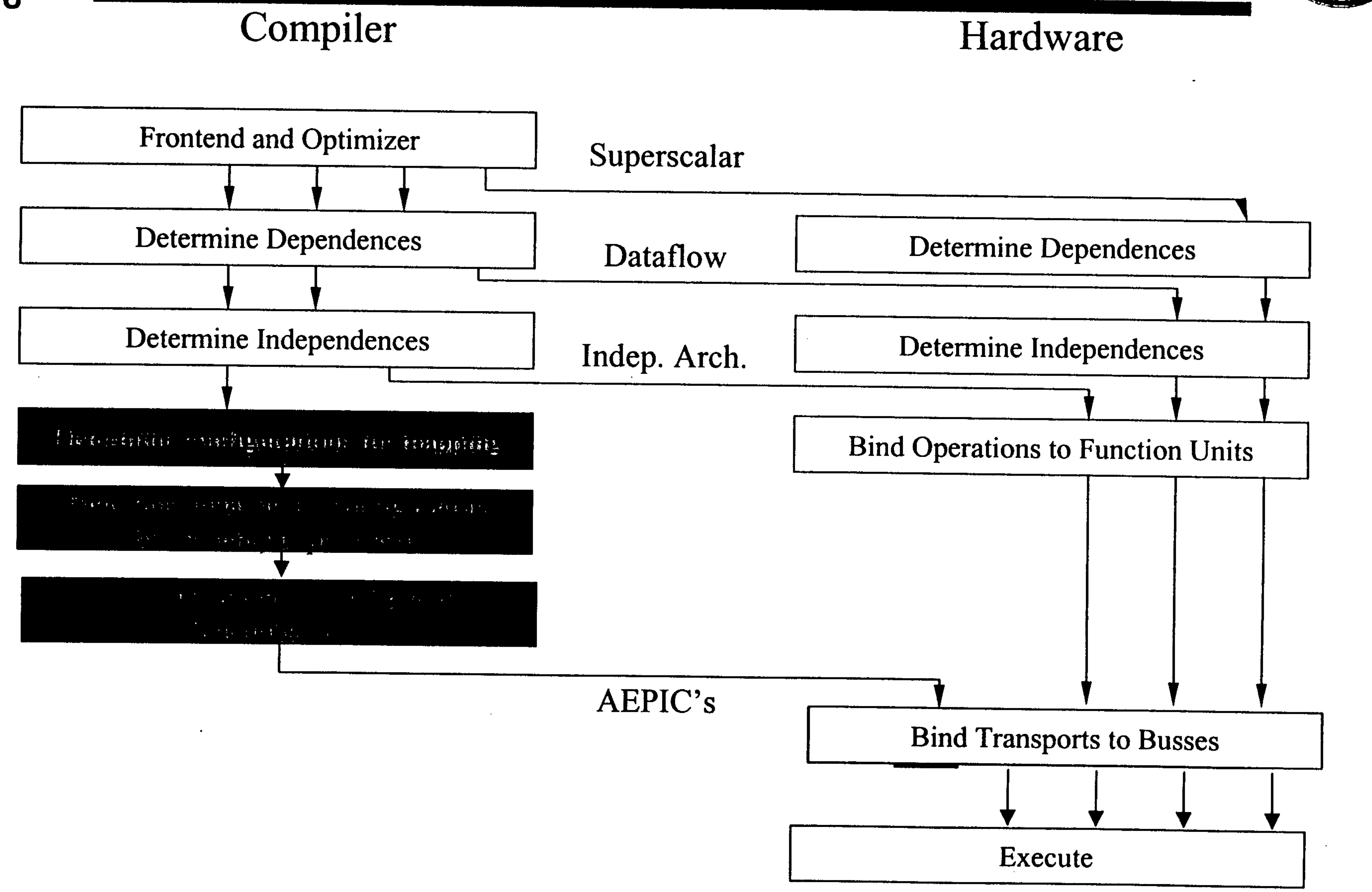
Field	Description
cid	Alias for custom instruction (all new instructions use it)
base	Base address of configuration data in process address space
offset	Current offset of partially configured configuration
size	Configuration size
nio	Number of input operands
niom	Mask for input operand types
noo	Number of output operands
noom	Mask for output operand types

HPL-PD Instruction Set Extensions

- Example
  - cfg_alloc cr[2], FFT // load FFT configuration, constant FFT is
  - // address of FFT configuration in addr. sp.
  - cfg_lds cr[2] // speculatively load the next word of
  - // FFT configuration (base,offset in cr[2]

- Exact formats, latency and resource usage of new instructions given in the AEPIC Architecture Specification 1.0
- Non-architecturally visible state, specific architecture of reconfigurable logic not part of architecture spec.

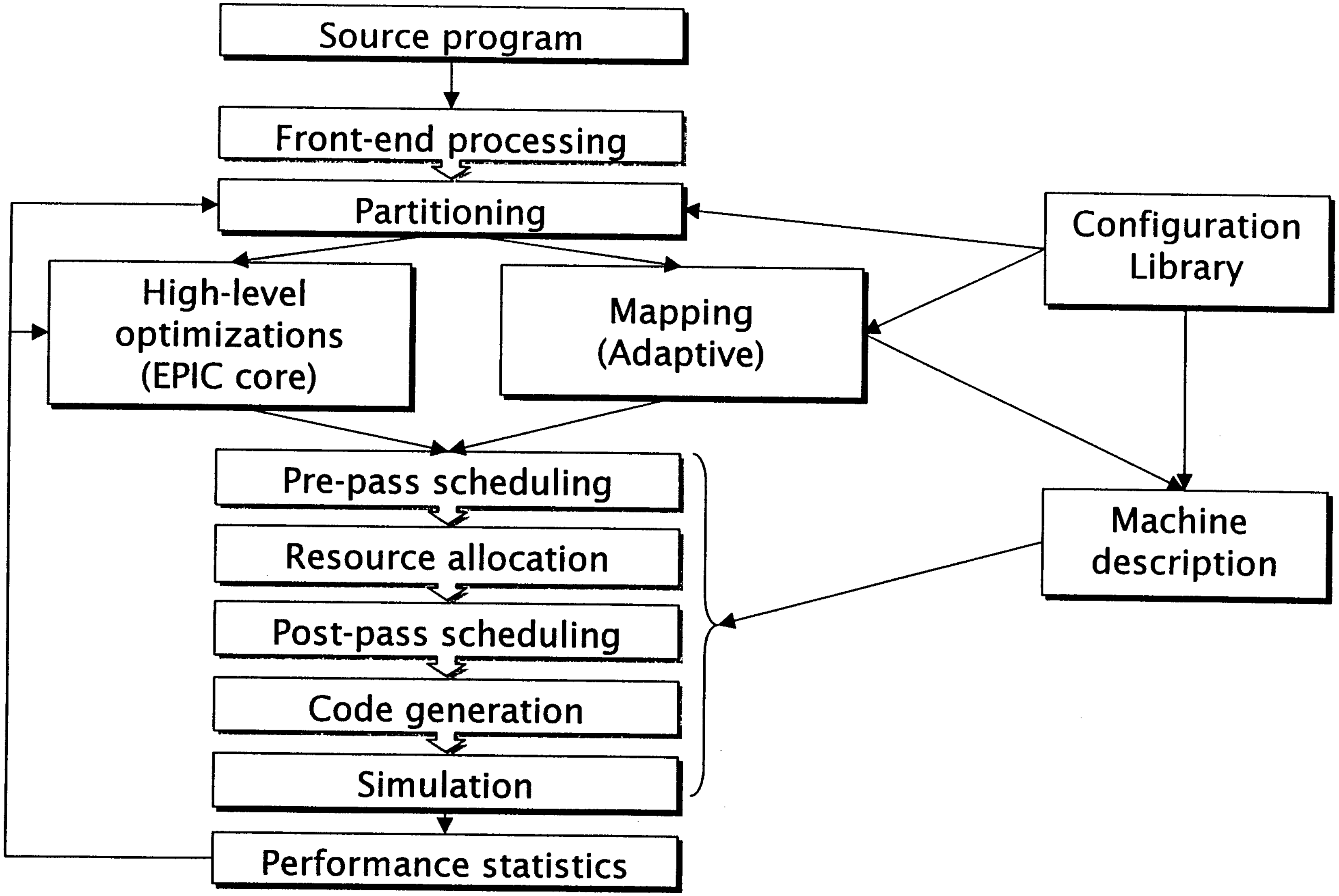
Compiler vs. Processor



Compiler Infrastructure

NYU

Compiler Modules


Source program

Front-end processing

Partitioning

High-level optimizations (EPIC core)

Mapping (Adaptive)

Configuration Library

Machine description

Pre-pass scheduling

Resource allocation

Post-pass scheduling

Code generation

Simulation

Performance statistics

Compilation challenges

- Code partitioning
- Mapping partitions to reconfigurable logic

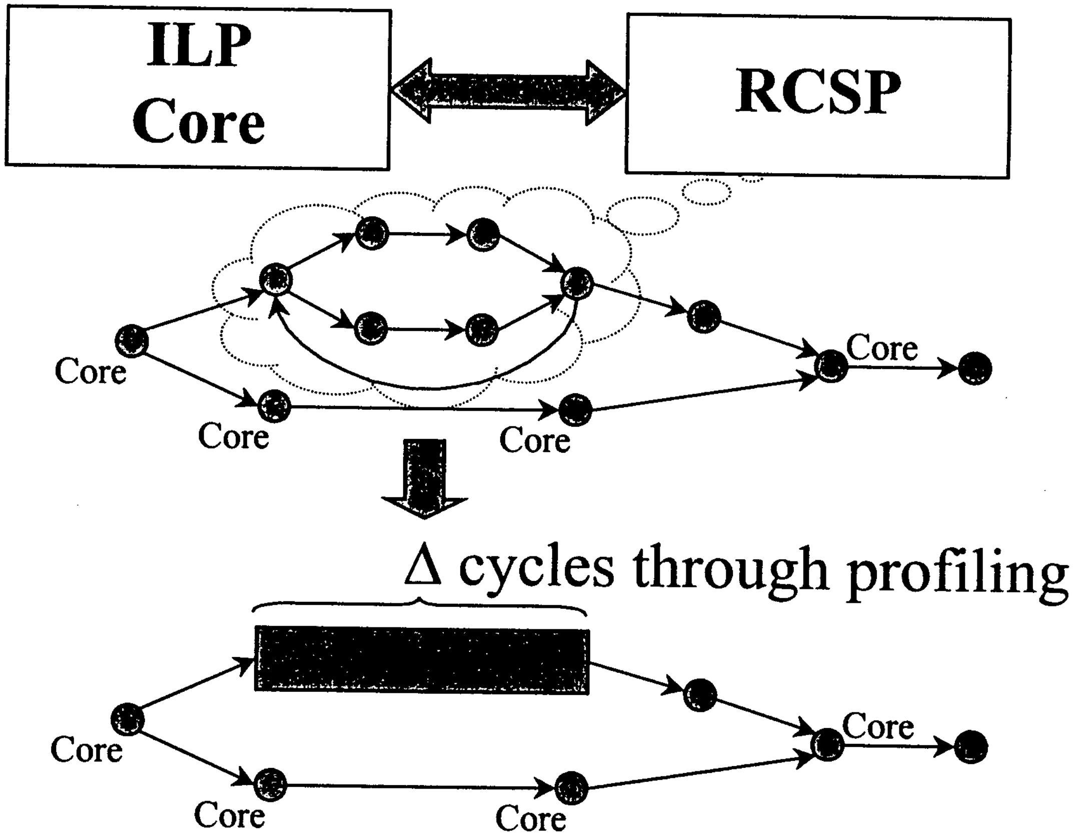
√ Reconfiguration time

√ Configuration scheduling and resource allocation

√ Configuration selection

- Intermediate representations

Configurable Scheduling Optimizations



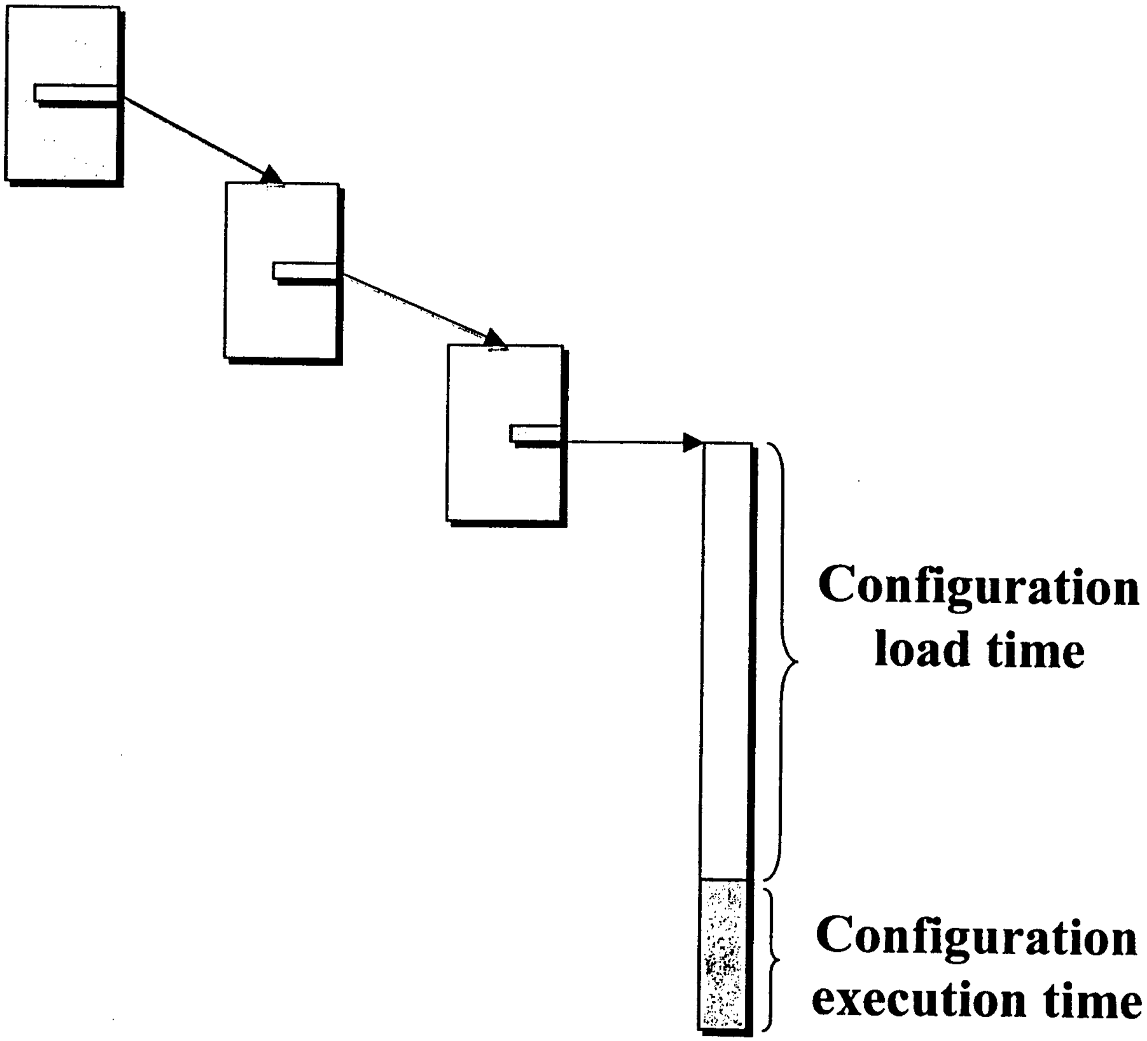
ILP Core

RCSP

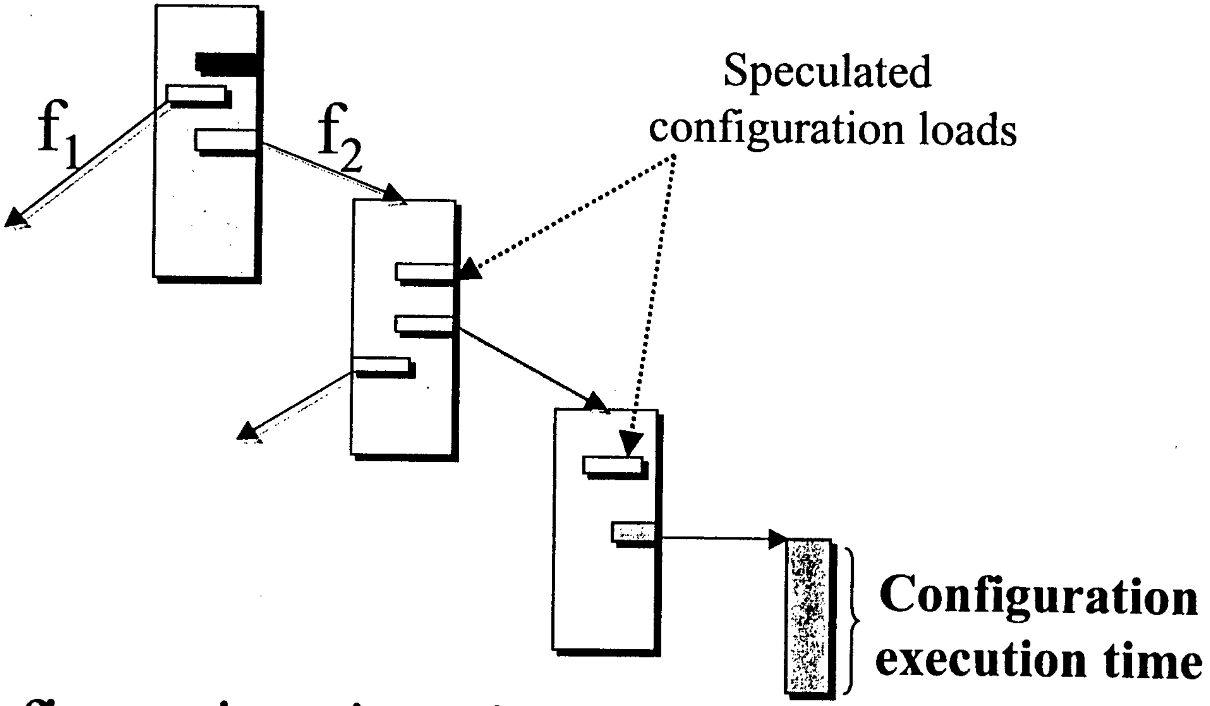
Δ cycles through profiling

A.Leung, K.Palem , A.Pneuli, S. Talla“A Fast Algorithm for Scheduling Time-constrained Instructions on RCSP”.

The Problem With Long Reconfiguration Times



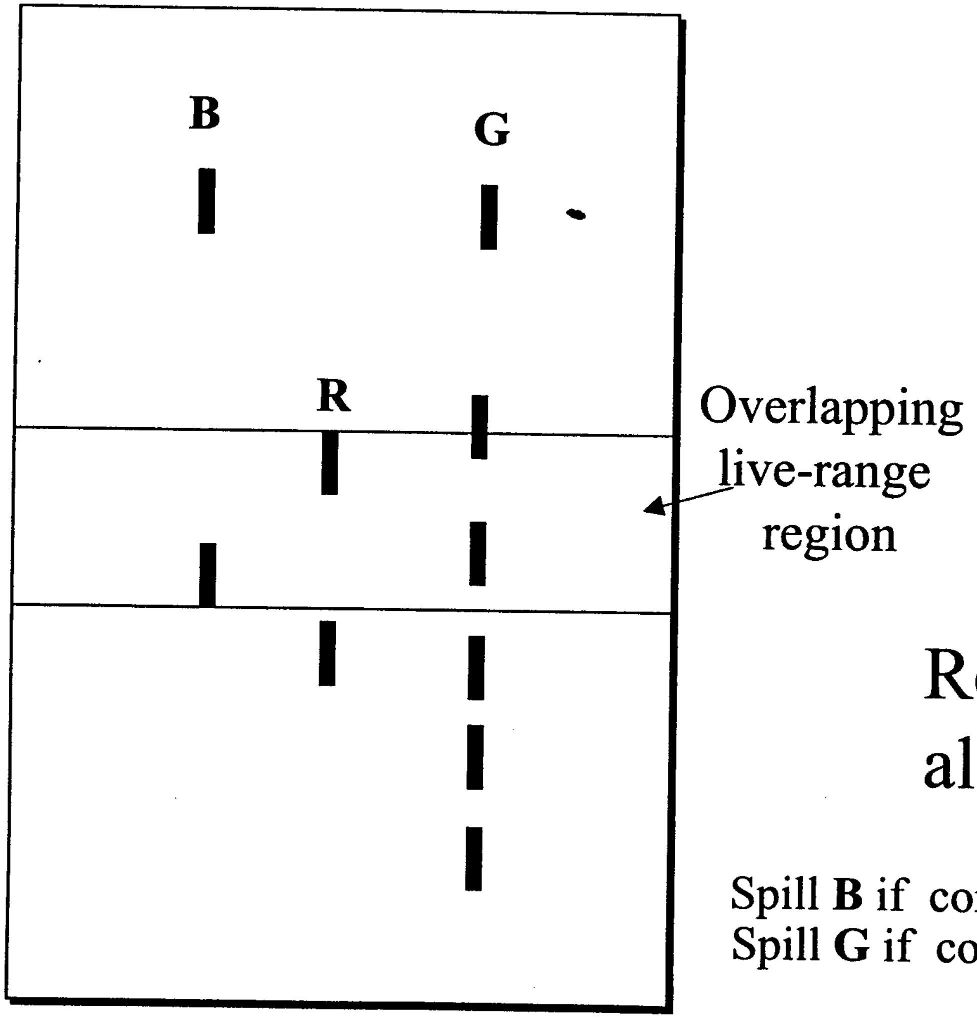
Speculating Configuration Loads



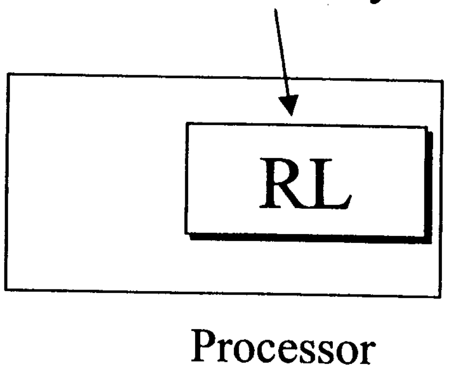
- Mask reconfiguration times!
- Need to know when and where to speculate
- If f1>>f2 do not speculate to “red” empty load slot

Resource allocation for configurations

Record of execution



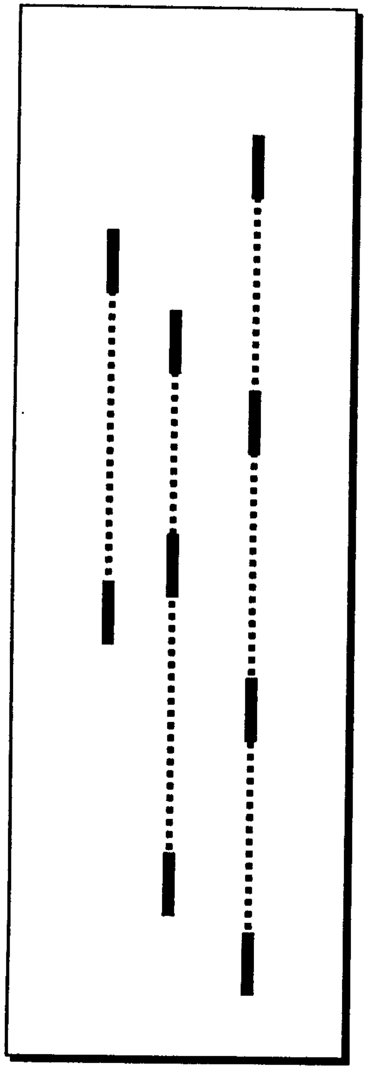
Reconfigurable logic can accommodate only two configurations simultaneously



Reduced to register allocation framework

Spill **B** if configuration size of **B** is same size as **G**
Spill **G** if configuration size of **B** is much larger

Managing Reconfigurable Resource (contd.)



- Configuration Live Ranges
- Directly relates to register allocation problem
- New parameters
  - No need to spill (no self-modifying mappings)
  - Load costs different for different configurations
  - prioritize based on configuration sizes and usage

Spill **B** if configuration size of **B** is same size as **G**
Spill **G** if configuration size of **B** is much larger

Machine Description Enhancements for AEPIC's

NYU

Machine Description Requirements

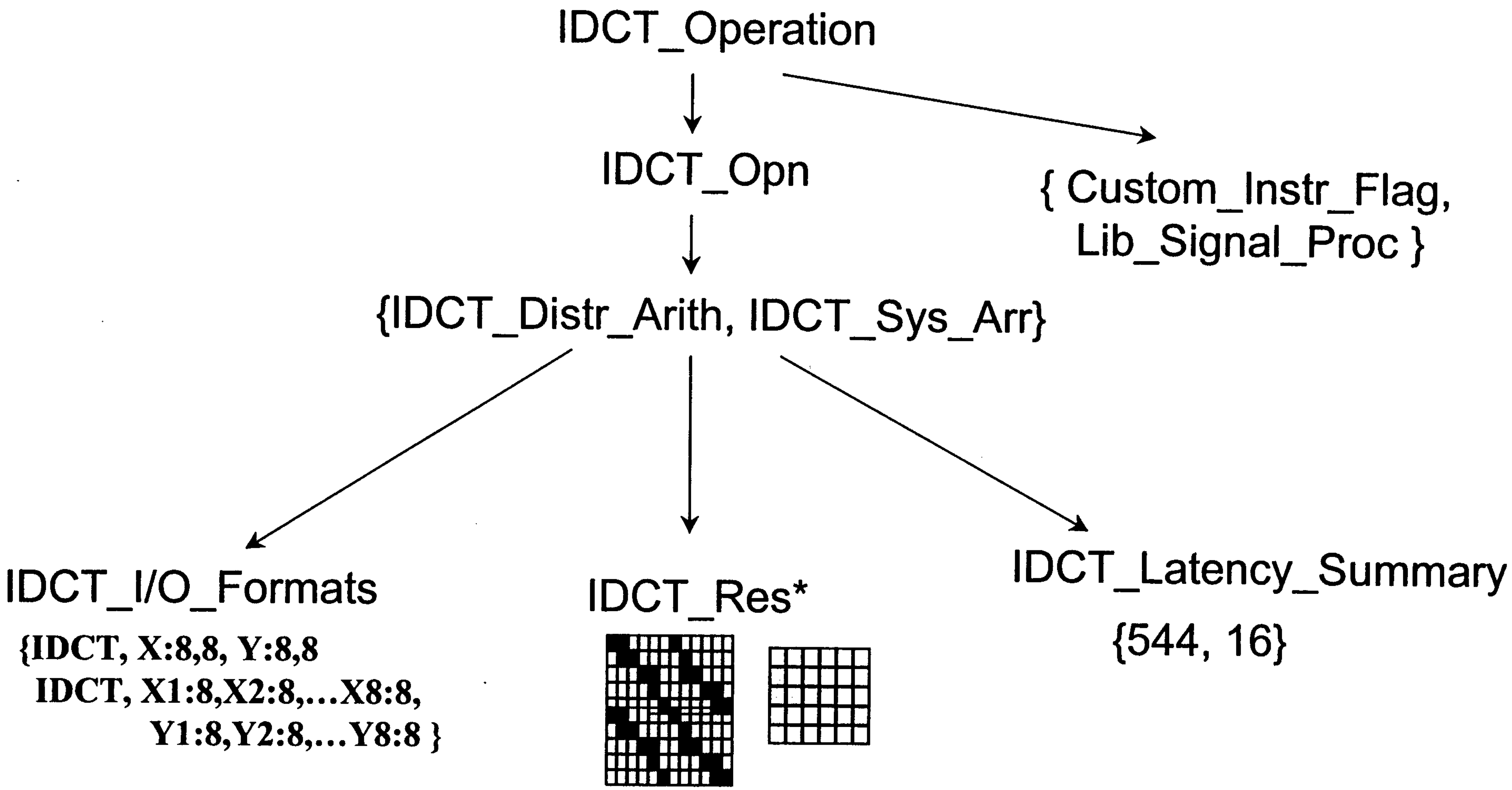
- Retargetability : No built in assumptions about machine in the compiler!
- Easy to understand/modify both by compiler writer and processor architect.
- Language must allow specification of a wide variety of architectures.
- Support automatic generation of tools like assemblers, simulators, verifiers, cost-modelers, etc.
- Support efficient query interfaces to external (compiler) modules.

NYU

Requirements for Adaptive EPIC's

- In addition to what is required of a machine description for an EPIC processor, it should provide
  - an ability to describe reconfigurable resource
  - an external interface to the reconfigurable logic resource
- The machine description mechanism should provide a dynamically (during compile time) changing ISA to the compiler

Proposed Extensions (contd.)



IDCT_Operation

IDCT_Opn

{ Custom_Instr_Flag, Lib_Signal_Proc }

{IDCT_Distr_Arith, IDCT_Sys_Arr}

IDCT_I/O_Formats

{IDCT, X:8,8, Y:8,8
IDCT, X1:8,X2:8,…X8:8,
Y1:8,Y2:8,…Y8:8 }

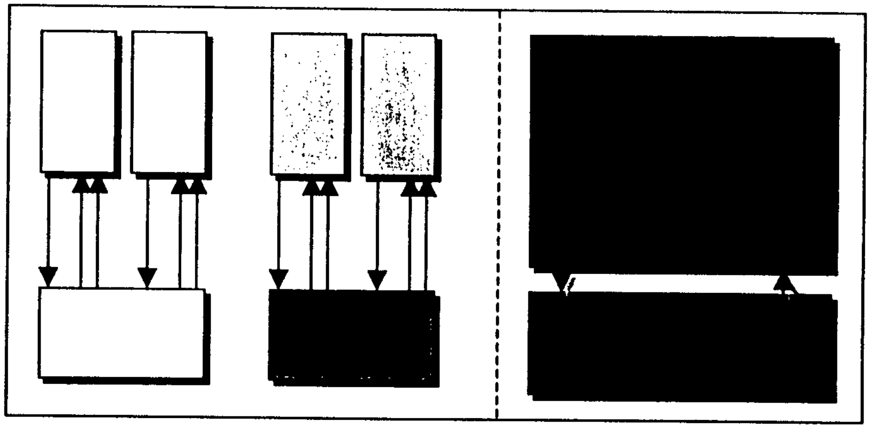
IDCT_Res*

IDCT_Latency_Summary

{544, 16}

* from configuration template files

AEPIC target description



```
GRP_SIZE  64
FPR_SIZE  64
INT_UNITS 2
FLOAT_UNITS 2
RLA_PARAMS  64 128 2
C_CACHE  128 32 1
```

```
r1 = L.W.C2.V1 r2
r1 = ADD r1, r2
ST.C1 r2, r3
CFG_LD  cr2, ADDR
CFG_INP  cr1, r1
CFG_EXEC  cr4
```

NYU

MDES Extensions for Adaptive EPIC's

- Custom instructions and base instructions will be handled similarly
- Unlike base instructions, custom instruction resource usages will be pointers to configuration-files

Summary

AEPIC's : A Summary

- AEPIC's
  - at the right granularity for automated compilation
  - leverage wealth of ILP experience
  - simple architectures
- Adaptive computing specific compiler optimizations
  - automatic partitioning and mapping : extremely important but poorly understood - currently library based
  - other problems such as configuration scheduling and allocation can be related to well known optimization problems
- Machine descriptions for retargetability
  - necessary for architectural exploration
  - easy migration path for compiler (and other tools)

Future Directions

Compiler Framework For AEPIC's

- AEPIC specific compiler optimizations
  - incorporate and validate several configuration scheduling/allocation optimizations
  - tackle partitioning and mapping (semi-automatic)
  - optimization modules need to use profile feedback
  - optimization modules to be parameterized by MDES
- Machine description framework
  - extend MDES with AEPIC architectural features
  - hooks to configuration library
- Simulation, emulation and performance monitoring
  - gather statistics specific to execution on adaptive extension
  - simulation environment adapts to MDES variations

NYU

Architectures

- Adaptive targets with DSP/RISC cores
  - use DSP or RISC cores instead of EPIC cores in AEPIC
  - compare cost/performance with respect to EPIC cores
- Extend AEPIC model to scalable AEPIC's
  - just as extra RAM is added to improve memory performance add extra reconfigurable logic and raise processor performance
  - needs innovations in processor-memory interconnect
- Explore adaptivity in other areas of architecture (not just customization of functional units)
  - adaptive cache control logic, malleable caches
    - goes beyond explicit control of caches as in HPL-PD

NYU

Emerging Constraints

- Power sensitive adaptation
  - so far, most adaptation geared towards performance gains
  - can we develop techniques to adapt architecture to consume optimal (low) power for a given application?
    - need metrics for instruction combinations that are power friendly
    - information useful for power sensitive scheduling
    - need architectural enhancements to enable power sensitive adaptation
- Other constraints : timing, size and weight
- Compiler-architecture co-design
  - fine tune architecture for specific application domains
  - tune for performance/power
  - potential for high impact in embedded systems

NYU

Compiler Infrastructure

- Trimaran infrastructure and architectural model ideal
  - easy to generate MDES variations rapidly
  - retargetability built into the compiler infrastructure
- Compiler - architecture co-design (with Georgia Tech.)
  - VHDL/Verilog structural descriptions for Trimaran targets
    - for area, power estimates
    - path to place and route tools for targeting Xilinx, Altera, ...
  - domain specific customization
- Target commercial architectures
  - e.g., code generators for IA-64
- Extensions to intermediate representations (IR)
  - to handle timing constraints
  - annotations for power/area sensitive optimizations

Summary

NYU

Technical Achievements

- Developed compiler models for EPIC processors extended with reconfigurable logic
- Initial design of suitable extensions to current machine description framework to target AEPIC processors
- Efficient algorithms for instruction scheduling
  - in the presence of long latency instructions
  - under user specified time constraints
- Developed Time-C for specifying time
  - translate source (time) constraint specifications to constraints on intermediate graphs
  - developed algorithms for scheduling time critical instructions
- Released compiler infrastructure for EPIC processors
  - conducted performance tests for simplified AEPIC model

NYU

Impact to the Community

- Model for targeting a modern optimizing compiler
  - vendor neutral parametric processor
  - proof of performance improvements
  - fast compilation
- Dynamic configuration
  - configuration caches
  - compiler optimizations e.g. speculation
- Notation for expressing constraints
  - timing
  - partitioning and mapping (to be done)

Contact Information

ReaCT-ILP
New York University

719 Broadway,
New York, NY 10003
react-ilp.cs.nyu.edu