AFRL-IF-RS-TR-2003-110
Final Technical Report
May 2003

DEFACTO: A DESIGN ENVIRONMENT FOR ADAPTIVE COMPUTING TECHNOLOGY

University of Southern California

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. D002

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** MAY 2003	**3. REPORT TYPE AND DATES COVERED** Final Apr 98 – Jun 02

4. TITLE AND SUBTITLE
DEFACTO: A DESIGN ENVIRONMENT FOR ADAPTIVE COMPUTING TECHNOLOGY

5. FUNDING NUMBERS
C - F30602-98-2-0113
PE - 62301E
PR - D002
TA - TC
WU - 02

6. AUTHOR(S)
Mary Hall and Pedro Diniz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Southern California
4676 Admiralty Way
Marina Del Rey California 90292-6695

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency AFRL/IFTC
3701 North Fairfax Drive 26 Electronic Parkway
Arlington Virginia 22203-1714 Rome New York 13441-4514

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2003-110

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Martin J. Walter/IFTC/(315) 330-4102/ Martin.Walter@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***
This report describes the activities of the DEFACTO project, a Design Environment for Adaptive Computing Technology funded under the DARPA Adaptive Computing Systems and Just-In-Time-Hardware programs. The goal of DEFACTO is to derive system-level implementations of mappings to FPGA-based systems, from a high-level algorithmic description in standard C. We have demonstrated synthesis time reductions to 100-10000X with the automated design space exploration algorithm. The current reduction in design time, including human effort, has been approximately 40-60X for two case studies, SLD from Sandia ATR and Sobel edge detection. We have demonstrated end-to-end mapping on the Annapolis Wildstar board for both examples, with no minimal intervention, at DarpaTech on 31 July 2002.

14. SUBJECT TERMS
FPGAs, Design Tools, Parallelizing Compliers, Behavioral Synthesis

15. NUMBER OF PAGES
35

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Table of Contents

List of Figures

List of Tables

Abstract

This report describes the activities of the DEFACTO project, a Design Environment For Adaptive Computing TechnOlogy, funded under the DARPA Adaptive Computing Systems and Just-In-Time-Hardware programs. The goal of DEFACTO is to derive system-level implementations of mappings to FPGA-based systems, from a high-level algorithmic description in standard C. We have demonstrated synthesis time reductions of 100-10000X with the automated design space exploration algorithm. The current reduction in design time, including human effort, has been approximately 40-60X for two case studies, SLD from Sandia ATR and Sobel edge detection. We have demonstrated end-to-end mapping on the Annapolis Wildstar^{TM} board for both examples, with no manual intervention, at DarpaTech on July 31, 2002.

I. Introduction

The DEFACTO project is a high-level design tool for developing application-specific hardware in FPGA-based systems through a collaboration between parallelizing compiler technology and high-level hardware synthesis. The original goal of the project was to derive system-level designs in an end-to-end open design environment. Estimation and module generators were considered essential to reducing the synthesis time, to be used in an automated design space exploration algorithm. The design environment was to be retargetable, so that multiple input languages and multiple hardware platforms were supported. We anticipated that these techniques would reduce synthesis time by 10X, and overall design time, including human effort, by several person-years.

We planned several demonstrations. First, an end-to-end demonstration of the SLD code from Sandia ATR, specified in C and mapped to the Annapolis Wildstar^{TM} board. Later, we would show retargetability of the system by starting with a MATLAB^{TM} frontend, and subsequently, mapping to the SLAAC 1-V board.

This report describes the accomplishments of the DEFACTO project. In terms of reducing synthesis time, we have exceeded our goal of 10X, demonstrating reductions of 100-10000X with the automated design space exploration algorithm. The current reduction in design time, including human effort, has been approximately 40-60X for two case studies, SLD from Sandia ATR and Sobel edge detection. We have demonstrated end-to-end mapping on the Annapolis Mirosystems Inc.'s Wildstar^{TM} board for both examples, with no manual intervention, at DarpaTech on July 31, 2002. The end-to-end demonstrations use the external host, multiple memories and a single FPGA. We have also demonstrated in simulation automatically-generated multi-FPGA designs, and have mapped a multi-FPGA design for the SLD code from Sandia ATR to the Wildstar^{TM} board with modest manual intervention. We were redirected by DARPA to limit the scope of the project, and eliminate the MATLAB^{TM} frontend and SLAAC 1V demonstrations, so this work was not completed.

The DEFACTO project will continue under NSF funding, and we plan to complete automatically-generated multi-FPGA designs to the Wildstar^{TM} board in the near term, as well as explore algorithms to partition computation and data across multiple FPGAs and memories.

In the remainder of this report, we highlight the key technical results from DEFACTO in the next section. These results are discussed in more depth in the five publications that accompany this report. We then discuss the complex infrastructure developed for this project in Section III. In Section IV, we present subcontract activities as well as auxiliary ISI activities. Section V presents technology transfer activities. We conclude with a summary and a list of publications supported by this effort.

II. Key Results

1. Automated Mapping from C to Annapolis WildStar^{TM} Board

We have developed a complete compilation and synthesis infrastructure that successfully maps applications written in sequential C to the Annapolis Microsystems Inc.'s FPGA-based WildStar^{TM} board. **This tool flow is fully integrated and automated**. We have developed a suite of compiler analysis and transformations in SUIF that are specific to adaptive computing, as well as auxiliary tools to derive an integrated design flow.

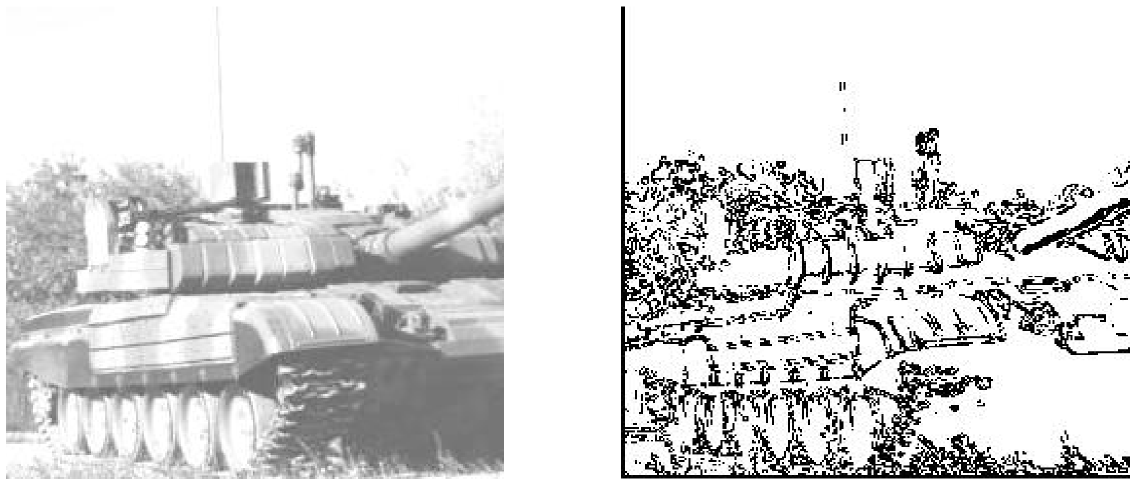

Input Image 256x256 8-bit gray-scale
Output Image 256x256 8 -bit gray-scale

Figure 1: Input and output sample Images for the automatically mapped Sobel computation on the WildStar^{TM} board.

Using the current DEFACTO compilation system we have automatically mapped the Sobel Edge Detection application code written on C to the WildStar^{TM} board. This mapping takes a total of 42 minutes, 40 of which accounted for by the logic synthesis and Place&Route of the FPGA design. The remaining 2 minutes are spent by the DEFACTO compiler performing internal analysis and intermediate code generation. The automatic mapping is attained in a single compilation cycle and is a correct implementation of the original Sobel computation. For comparison purposes, the same design was manually mapped to the same board in about 2 weeks. Although the automated mapping using the DEFACTO compiler is about 59% slower than the manual mapping, it is achieved in 42 minutes, a roughly two orders of magnitude reduction in design time. This modest increase in execution time is a small price to pay for a fully automated design approach. Figure 1 below depicts sample input and output images using the automated mapping on the WildStar^{TM} board. For this particular mapping, the compiler uses 2 memories.

2. Automated Design Space Exploration

A significant contribution to design time for FPGA systems is the cost of *design space exploration*, which is the iterative process of selecting a design among a set of candidates. The standard approach to design space exploration is shown in Figure 2 below. The designer specifies the implementation in structural VHDL and synthesizes the design. This process is iterative in that the design may not be valid, and even if it is, may not meet the designer's area and speed requirements. Each design iteration usually requires a minimum of a few hours and possibly as much as a week.

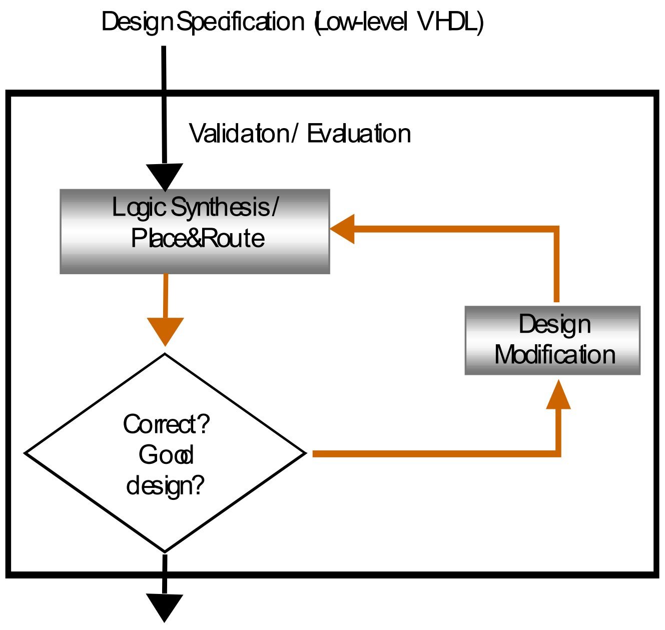

Design Specification (Low-level VHDL)
Validaton/ Evaluation
Logic Synthesis/ Place&Route
Design Modification
Correct? Good design?

Figure 2: Manual design space exploration.

In our approach, shown in Figure 3, we have reduced design time in several ways: (1) the designer specifies a much higher-level algorithmic specification; (2) validation is guaranteed, assuming tools work correctly; and, (3) we avoid numerous iterations of place-and-route, which is the costliest component of synthesis. The automated algorithm relies on behavioral synthesis, which is one of the distinguishing features of our system as compared to other ACS programs. It quickly provides a rough estimate of area and performance, which can be used to guide the compiler in selecting a design without requiring full place-and-route. It also derives structural VHDL, by performing scheduling, allocation and binding of resources, a process that other ACS programs are doing manually or are building into their tools.

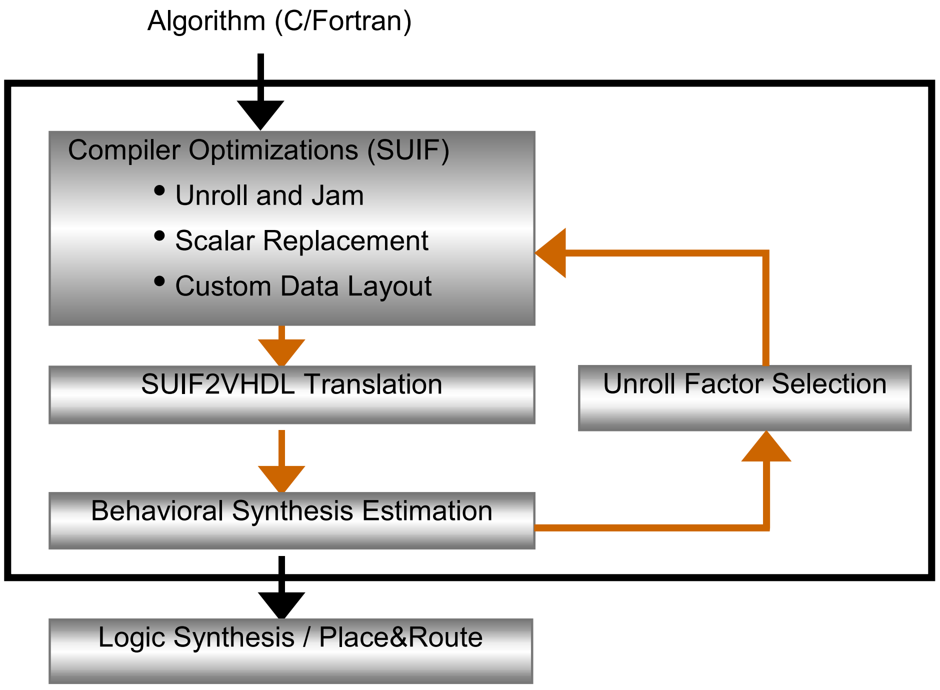


Figure 3: Our approach to automatic design space exploration.

The automated design space exploration algorithm uses a collection of compilation techniques to exploit the functional and memory parallelism in a computation, as well as exploit reuse of data on chip, to avoid costly accesses to external memory, whenever possible. *Unroll-and-jam,* which is a loop nest optimization whereby outer loops are unrolled and resulting inner loop bodies are fused, exposes parallel operations and memory accesses to behavioral synthesis optimizations and scheduling. *Scalar replacement* replaces accesses to array variables with scalar temporaries, to signal to behavioral synthesis that variables can be placed in registers, avoiding off-chip accesses to memory. *Custom data layout,* described in Section II.4.2 below, lays out data in multiple memories such that memory parallelism can be maximized.

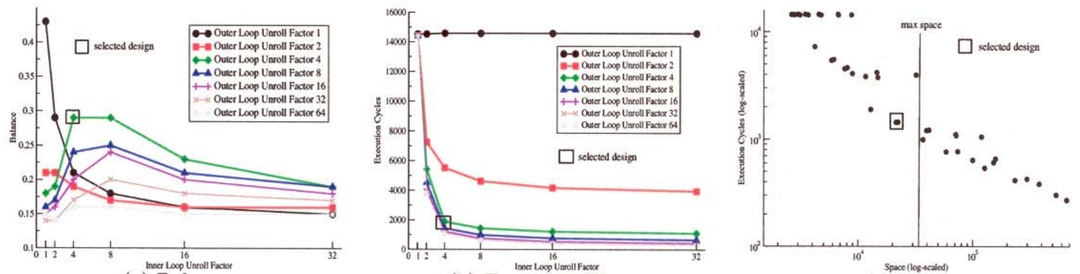
The algorithm employs a compiler-derived metric called *balance* to determine unroll factors for loops in a nest, such that the computation rate on the FPGA and the fetch rate to external memories are matched. Balance is important, because it allows us to derive the most space-efficient design among the set of designs that have the best performance. Conserving space has two direct benefits. First, it frees up FPGA hardware that can then be used for other computations. Second, a less complex design can achieve a faster clock rate, due to simpler routing. Thus, it may actually yield better performance than a larger design. We also use another metric, called the memory *saturation point*, to define a solution where the rate of memory accesses matches or exceeds the bandwidth of the architecture platform.

We have demonstrated this approach on the following five multimedia kernels: FIR filter, Matrix multiply, Sobel edge detection, String pattern matching, Jacobi 4-point stencil relaxation. The compiler automatically derives the best design for a single FPGA with multiple memories (see the next section) among a set of candidates.

The experimental results are shown in Figures 4 through 10. The graphs show a large number of points in the design space, substantially more than are searched by our algorithm, to highlight the relationship between unroll factors and metrics of interest. The first set of results in Figures 5 through 8 plots balance, execution cycles and design area in the target FPGA as a function of unroll factors for the inner and outer loops of FIR and MM. Although MM is a 3-deep loop nest, we only consider unroll factors for the two outermost loops, since through loop-invariant code motion the compiler has eliminated all memory accesses in the innermost loop. The graphs in the first two columns have as their x-axis unroll factors for the inner loop, and each curve represents a specific unroll factor for the outer loop.

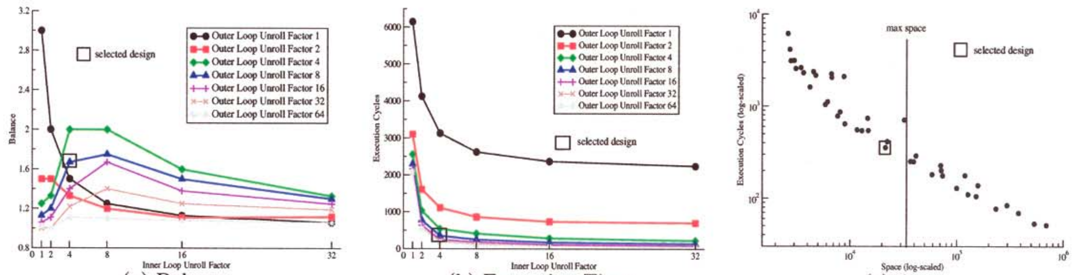
For FIR and MM, we have plotted the results for pipelined and non-pipelined memory accesses to observe the impact of memory access costs on the balance metric and consequently in the selected designs. In all plots, a squared box indicates the design selected by our search algorithm. For pipelined memory accesses, we assume a read and write latency of 1 cycle. For non-pipelined memory accesses, we assume a read latency of 7 cycles and a write latency of 3 cycles, which are the latencies for the Annapolis WildStar^{TM} board. In practice, memory latency is somewhere in between these two as some but not all memory accesses can be fully pipelined. In all results we are assuming 4 memories, which is the number of external memories that are connected to each of the FPGAs in the Annapolis WildStar^{TM} board. In these plots, a design is *balanced* for an unrolling factor when the y-axis value is 1.0. Data points above the y-axis value of 1.0 indicate *compute-bound* designs whereas points with the y-axis value below 1.0 indicate *memory-bound* designs. A compute-bound design suggests that more resources should be devoted to speeding up the computation component of the design, typically by unrolling and consuming more resources for computation. A memory-bound design suggests that less resources should be devoted to computation as functional units that implement the computation are idle waiting for data.

The design area graphs represent space consumed (using a log scale) on the target Xilinx Virtex® 1000 FPGAs for each of the unrolling factors. A vertical line indicates the maximum device capacity. All designs to the right side of this line are therefore unrealizable.



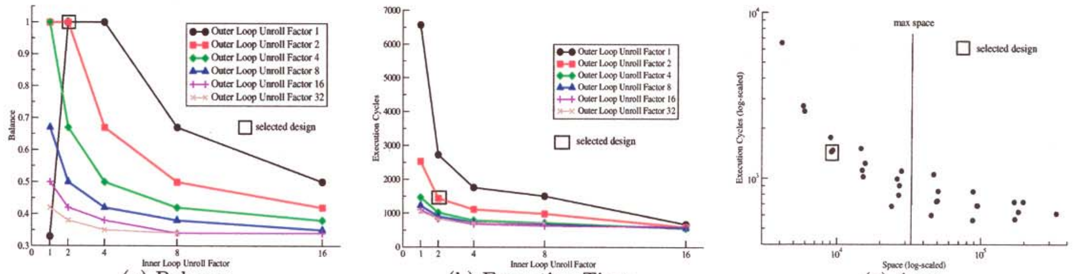
(a) Balance (b) Execution Time (c) Area

Figure 4: Balance, Execution Time and Area for Non-pipelined FIR.



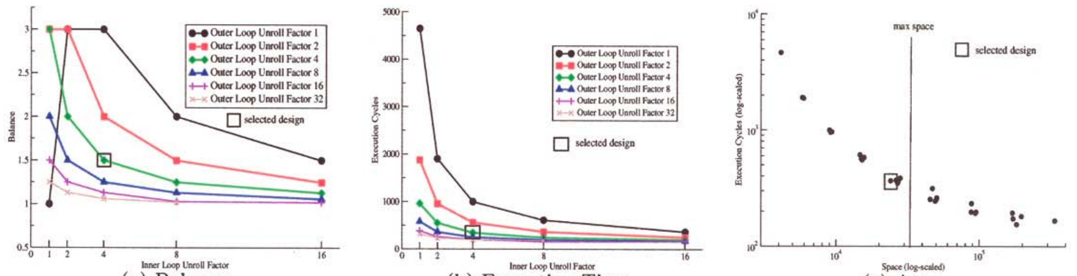
(a) Balance (b) Execution Time (c) Area

Figure 5: Balance, Execution Cycles and Area for Pipelined FIR.



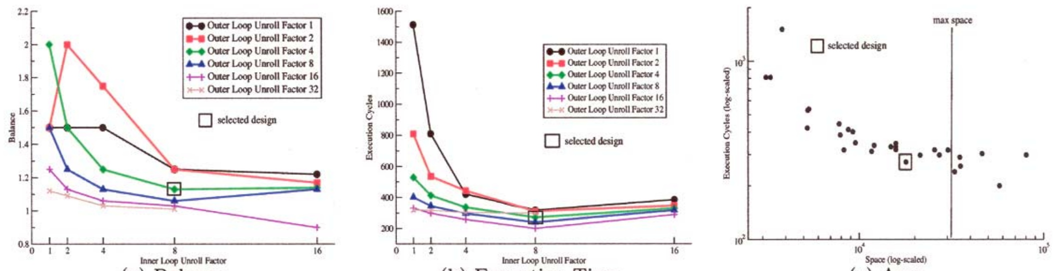
(a) Balance (b) Execution Time (c) Area

Figure 6: Balance, Execution Cycles and Area for Non-pipelined MM.



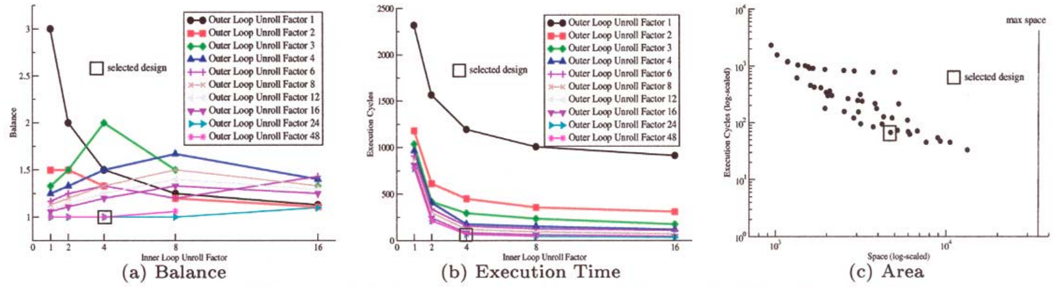
(a) Balance (b) Execution Time (c) Area

Figure 7: Balance, Execution Cycles and Area for Pipelined MM.



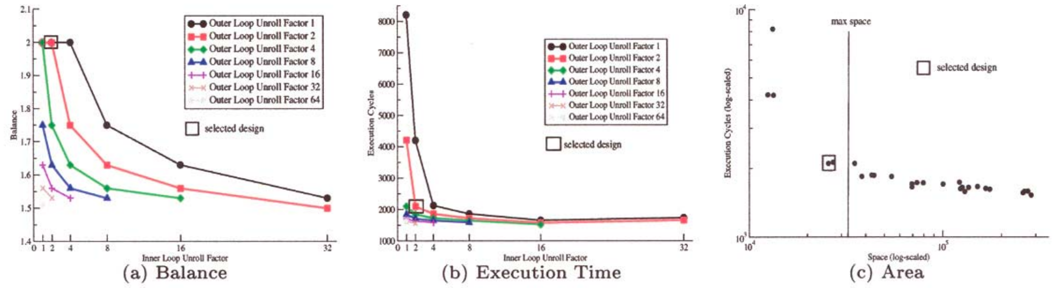
(a) Balance (b) Execution Time (c) Area

Figure 8: Balance, Execution Time and Area for Pipelined JAC.



(a) Balance (b) Execution Time (c) Area

Figure 9: Balance, Execution Cycles and Area for Pipelined PAT.



(a) Balance (b) Execution Time (c) Area

Figure 10: Balance, Execution Time and Area for Pipelined SOBEL.

With pipelined memory accesses, there is a trend towards compute-bound designs due to low memory latency. Without pipelining, memory latency becomes more of a bottleneck leading, in the case of FIR, to designs that are always memory bound, while the non-pipelined MM exhibits compute-bound and balanced designs.

The second set of results, in Figures 8 through 10, show performance of the remaining three applications, JAC, PAT and SOBEL. In these figures, we present, as before, balance, cycles and area as a function of unroll factors, but only for pipelined memory accesses due to space limitations.

We make several observations about the results. First, we see that Balance is monotonic, increasing until it reaches a saturation point, and then decreasing. The execution time is also monotonically nonincreasing. In all programs, our algorithm selects a design that is close to best in terms of performance, but uses relatively small unroll factors. Among the designs with comparable performance, in all cases our algorithm selected the design that consumes the smallest amount of space. As a result, we have shown that our approach meets our optimization goals. In most cases, the most balanced design is selected by the algorithm. When a less balanced design is selected, it is either because the more balanced design is before a saturation point (as for non-pipelined FIR), or is too large to fit on the FPGA (as for pipelined MM).

Table 1 below presents the speedup results of the selected design for each kernel as compared to the baseline, for both pipelined and non-pipelined designs. The baseline is the loop nest with no unrolling (unroll factor is 1 for all loops) but including all other applicable code transformations.

Kernel	Non-Pipelined	Pipelined
FIR	7.67	17.26
MM	4.55	13.36
JAC	3.87	5.56
PAT	7.53	34.61
SOBEL	4.01	3.90

Table 1: Speedup results for pipelined and non-pipelined design using DSE.

Although in these graphs we present a very large number of design points, the algorithm searches only a tiny fraction of those displayed. Instead, the algorithm uses the pruning heuristics based on the saturation point and balance. This reveals the effectiveness of the algorithm as it finds the best design point having only explored a small fraction, only 0.3% of the design space consisting of all possible unroll factors for each loop. For larger spaces, we expect the number of points searched relative to the size to be even smaller.

To speed up design space exploration, our approach relies on estimates from behavioral synthesis rather than going through the lengthy process of fully synthesizing the design, which can be anywhere from 10 to 10,000 times slower for this set of designs. To determine the gap between the behavioral synthesis estimates and fully synthesized designs, we ran logic synthesis and place-and-route to derive implementations for a few selected design points in the design space for each of the applications. We synthesized the baseline design, the selected designs for both pipelined and non-pipelined versions, and a few additional unroll factors beyond the selected design.

In all cases, the number of clock cycles remains the same from behavioral synthesis to implemented design. However, the target clock rate can degrade for larger unroll factors due to increased routing complexity. Similarly, space can also increase, slightly more than linearly with the unroll factors. These factors, while present in the output of logic synthesis and place-and-route, were negligible for most of the designs selected by our algorithm. Clock rates degraded by less than 10% for almost all the selected designs as compared with the baseline, and the speedups in terms of reduction in clock cycles more than made up for this. In the case of FIR with pipelining, the clock degraded by 30%, but it met the target clock of 40ns, and because the speedup was 17X, the performance improvement was still significant. The space increases were sublinear as compared to the unroll factors, but tended to be more space constrained for large designs than suggested by the output of behavioral synthesis.

The very large designs that appear to have the highest performance according to behavioral synthesis estimates show much more significant degradations in clock and increases in space. In these cases, performance would be worse than designs with smaller unroll factors. Our approach does not suffer from this potential problem because we favor small unroll factors, and only increase the unrolling factor when there is a significant reduction in execution cycles due to memory parallelism or instruction-level parallelism.

Selecting the desired behavioral design requires less than 5 minutes for our system, and then at most another 2 hours to fully synthesize the result. To produce a comparable design by hand for Sandia ATR SLD required up to 2 months. Thus, this algorithm has demonstrated several orders of magnitude improvement in design time. A more detailed description of automated design space exploration can be found in reference [11].

3. Tapped Delay Line Analysis

As part of the DEFACTO project we developed a code transformation to exploit reuse of data on chip and avoid accesses to external memory. The compiler generates VHDL code with links to hardware modules that exploits that reuse by storing the data read from memory in a hardware structure called *tapped delay line*. Figure 11 below illustrates the use of tapped delay lines in an implementation of the *Sobel* edge detection algorithm as 3 sets of interconnected registers on the left hand side.

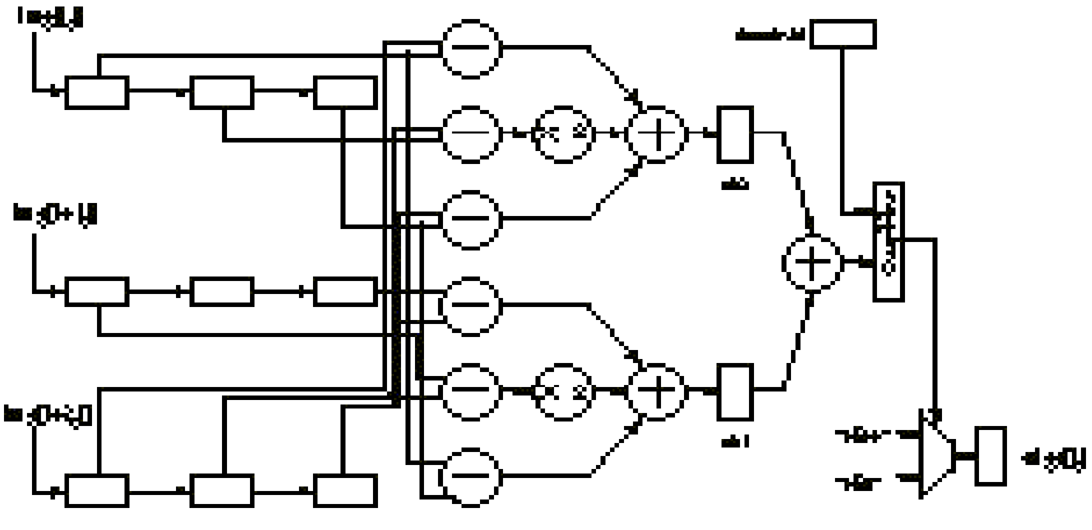


Figure 11: Example of usage of tapped delay lines.

Table 2 below presents characteristics for 3 kernels. For each of the kernel codes we report on the number of source code lines for both the C input and the generated VHDL (excluding comments and blank lines). We report the number of loop nests in each application and the number of loops the compiler selected for hardware execution. For the generated VHDL source code we report on the number of distinct components and instances used. Finally we report on the compilation analysis and synthesis speed.

Table 2: Compilation and synthesis results.

	Source Code Metrics			VHDL Code Metrics			Analysis & Synthesis Time		
Kernel	Code Lines	Loop Nests	Loop Hard	Code Lines	Num Comps	Num Inst	Analyzes Time	Emit Time	Synthesis Time
Sobel	80	3	1	2,340	39	134	< 1 sec	< 1 sec	10 min
Pattern	98	4	1	2,445	32	111	< 1 sec	< 1 sec	8 min.
ATR	300	5	3	4,400	38	386	< 1 sec	< 1 sec	780 min.

Table 3 presents the results of the compiler analysis. For each kernel we report on the number and length of the data queues the algorithm has identified and the unrolling factor that has the lowest memory access metric.

Table 3: Data reuse analysis results.

Kernel	Unrolled Loops	Reuse Vector	Data Queues	Length Queues	Data Reuse	Mem bandwidth
Sobel	---	(x,y) = (0,1)	3	3	6	1.0
Pattern	{i}	(i) = (1)	2	16	31	0.5
ATR	{i,j}	(m,n) = (0,1)	2	32-by-32	2016	8.25

For the *Sobel* kernel the compiler recognizes the opportunities of two reuse directions. The compiler implementation chooses the lowest memory access. As for *Pattern* the analysis recognizes that unrolling loop *j* is highly profitable as the *pat* variable becomes loop invariant. The resulting implementation should have a single queue of length 16 for the *str* variable and another queue of the same length for the *pat* variable. Finally the *ATR* application has 3 loops in which there is a substantial amount of reuse. We present only the results for the unrolling of loop *i* and *j*, which reveals a maximum reuse for a 32-by-32 queue for the *mask* variable which is loop invariant and a 32-by-32 queue for the *img* variable. Unfortunately the design corresponding to the full unrolling of the two inner loops is too large to fit on a single FPGA. Instead we partially unroll each of the two inner loop by a factor of 16, therefore creating 16-by-16 data queues. This consumes less FPGA resources but significantly increases the control complexity and therefore the simulated execution time.

Table 4 shows the simulated performance results for the generated designs. It includes the overall simulated clock speed, the number of flip-flops and latches used and the number of LUTs and equivalent gates counts. For raw performance comparison we included the number of execution cycles required to complete the entire loop nest computation of each application. Finally we report on the area fraction used on the FPGA by the P&R tool.

Table 4: Performance metrics of simulated target designs.

Kernel	Core Datapath Clock (MHz)	Global Design Simulated Clock (MHz)	Number FF & Latch	Number LUTs	Equiv. Gates	Simulated Execution Cycles	Virtex1000 Area
Sobel	56.5	26.5	840	727	11,375	2,196,480	5%
Pattern	56.5	26.0	782	771	11,239	287	5%
ATR	52.5	25.7	9,163	9,649	145,759	182,272	55%

Table 4 reveals the three designs attain respectable clock rates for automatically derived designs. Recall these designs were generated automatically by manually using the results of the analysis with the library of code generation functions we have implemented using generic, and simple, parameterized modules (*e.g,* adders, sub, comparators, multiplexors, etc.). These results reveal the compiler is able to identify the opportunities for data reuse and generate the data required to automatically generate complete VHDL design. Because of their relative small size, the generated designs for *Sobel* and *Pattern* are synthesized and routed fairly quickly. The design corresponding to the ATR application uses 55% of a single FPGA resource and takes much longer to synthesize (even with hierarchical P&R). We attribute this discrepancy to memory trashing effects.

The performance results (simulated clock rates) also reveal that the limiting factor is the control datapath as the core datapaths are capable of much higher clock rates. Several factors contribute to this. First the generality of the modules used. As an example our memory access subunit is fairly generic as it can handle both SRAM and DRAM modules. This clearly introduces latency in terms of clock cycles. Our interfaces allow for the presence of multiple memory interface module for distinct memory banks with a common pipelining control unit. Several other design aspects have not been described here as our focus was the design of a compiler analysis algorithm to allow the automatic generation of hardware implementations. We have not explored the trade-offs in the design of the pipelining control unit and the possible refinements of pipelined memory access unit. As such our simulation performance results can be improved by using control units with more advanced features, which we will describe in subsequent sections of this report.

4. Multi-Memory Designs

Each FPGA on the WildStar^{TM} board has several external SRAM memory chips from/to which it can read and write its data. A straightforward mapping based on a conventional compilation approach would map all data to a single memory. To exploit the multiple memories, and therefore increase the memory bandwidth of the FPGAs, we have developed and automated several techniques, used to derive the results in the previous sections.

4.1 Custom Memory Controllers

We have developed a code generation approach for handling the scheduling of external memory accesses in a parameterized way and with application-specific knowledge. In this approach, the compiler generates a customized memory controller for a particular application, exploiting the ability to pipeline memory accesses in a streamed mode as well as reduce the number of states in the memory controller. Our preliminary experimental results reveal these transformations yield a substantial reduction in the cycles for accessing memory in external memories, of great importance for FPGA-based computing engines. We first describe the methodology used in these experiments and then present the results obtained for three image processing kernel computations running on a real FPGA-based computing board.

We have mapped three (3) computation kernels from C to VHDL using DEFACTO: Sobel Edge Detection (SOBEL), a kernel from Sandia ATR, and MAZ, a multiply-accumulate-zero kernel. Next, we have manually modified the memory channel interface for the target architecture to allow the implementation of two distinct flavors of the round-robin memory access scheduling strategy, namely **naïve (N)**, **pipelined(P), group (G)** and **pipelined with grouping(P+G)** scheduling. We then compare the performance of the designs using the different strategies. This performance comparison was carried out in a functional simulator, ModelSim, where we are able to extract more precise clock cycle counts. We also confirmed the performance improvements via real executions on the WildStar board. To exacerbate the problems of memory access scheduling, we mapped all of the data onto a single memory module. This approach allowed us to determine the severity of the memory scheduling issue. Techniques such as the custom data layout for multiple memory banks as discussed below are orthogonal to this scheduling approach.

We begin this discussion by first characterizing the execution of each of the kernels using the default round-robin naive memory access scheduling strategy. Figure 12 below presents a breakdown of the execution time for the steady state of the main computation loop in each of these applications for a single memory bank implementation.

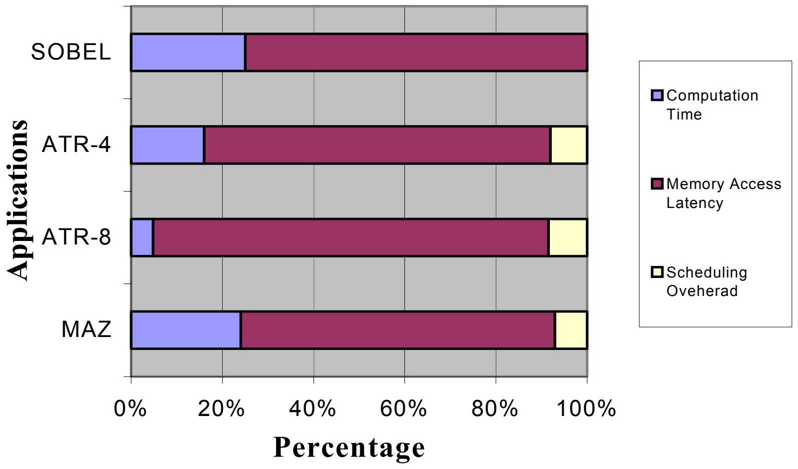


Figure 12: Execution time breakdown for tested applications.

As expected, and given that in these experiments memory accesses are blocking, the bulk of the execution time (60% to 80%) is spent stalling on memory accesses. Approximately 8% to 9% is spent checking the status of the input/output FIFO queues.

Table 5 shows the performance results for all applications for the different scheduling strategies. These results exclude the initial data loading and final data retrieval from the board. We report the overall design size in terms of FPGA slices; the maximum allowed clock rate for the design; the simulated execution time using a 25 MHz clock and the speedup measured as the ratio of the execution time of each version with respect to the computation using the **naïve** scheduling strategy.

Table 5 reveals that all designs are small (12.5% maximum FPGA occupancy) and therefore exhibit good performance characteristics in terms of maximum attainable clock rates. Table 5 also reveals the performance advantages of pipeline with an average speedup of 1.9 over the four tested kernels. Group scheduling by itself yields modest performance improvement with an average speedup of 1.1. When combined with pipelined, group scheduling boosts the average speedup to 2.05. This improvement is most noticeable for ATR-8 where the number of channels with the same input/output behavior is the largest.

Table 5: Synthesis and timing results.

Kernel		Slices (out of 12,288)	Max. Freq. (MHz)	Simulation Time (nsecs)	Speedup
SOBEL	**N**	1,144	30.1	1,312,020	1.00
	P	1,061	31.5	738,540	1.78
	G	1,160	31.7	1,312,140	1.00
	P+G	1,068	31.6	697,660	**1.88**
ATR-4	**N**	1,968	25.9	120,040	1.00
	P	1,980	25.6	66,600	1.82
	G	1,974	33.9	102,280	1.17
	P+G	1,984	26.9	59,600	**2.00**
ATR-8	**N**	2,771	25.9	188,440	1.00
	P	2,707	25.9	71,840	2.62
	G	2,718	30.8	163,440	1.15
	P+G	2,730	25.9	69,480	**2.71**
MAZ	**N**	1,027	30.4	85,760	1.00
	P	1,191	36.2	62,360	1.38
	G	1,226	31.5	78,680	1.09
	P+G	1,003	29.6	55,520	**1.55**

Table 6 shows the synthesis metrics for the synthesis of the channel controllers for each design. Overall the more sophisticated group-scheduling controller has clock rates in the 100MHz range and therefore appears not to impact the critical path of the whole design. By itself, the implementation of the group-scheduling controller requires no more than 21 additional slices than the simpler naïve controller does for a total a maximum of 75 slices barely 5% of the designs.

Table 6: Synthesis metrics for channel controller (N: Naïve, P: Pipelined, G: Group P+G: Pipelined with Grouping).

Applications		CLBs	Gates	Clock Rate (MHz)
SOBEL	N	29	381	140.1
	P	25	333	130.3
	G	35	431	87.5
	P+G	35	448	134,7
ATR-4	N	35	458	127.3
	P	42	568	125.0
	G	40	531	105.2
	P+G	46	635	120.7
ATR-8	N	54	679	106.2
	P	74	937	82.2
	G	57	758	77.9
	P+G	75	1,010	69.4
MAZ	N	30	383	120.5
	P	34	450	112.8
	G	36	450	115.6
	P+G	45	559	116.4

The experimental results, not surprisingly, reveal that pipelining techniques substantially improve the overall design performance. The implementation of group-scheduling techniques marginally increases the performance for the whole design with negligible impact in terms of area and very little influence on the maximum clock rate.

While we are able to eliminate almost all the memory overhead by pipelining and aggressive group scheduling there are several techniques that have been explored in other contexts and could be explored for the context of FPGA-based designs, namely:

- Reducing the sharing of physical bus channels will reduce the memory latency.
- Assigning multiple memory modules to disjoint input array for concurrent accesses.
- Aggressive pre-fetching and overlapping memory accesses with computations.

In this work we have focused exclusively on application-level techniques that impact the design of the memory controller, rather than on architecture related approaches for reducing memory latency. We focused on the scheduling of memory accesses within a single computational task where memory accesses are statically scheduled. The scheduling in the context of multiple tasks may require a more flexible run-time scheduling strategy to minimize memory access contention. In the future we plan to address the implementation of dynamic, run-time scheduling, where a schedule is setup only at run-time rather than statically for both single and multiple tasks.

Given the trade-off between generality and performance, we have estimated the performance gap between the currently automated applications in this empirical study and what a designer could achieve exploiting the overlapping of computation in the core datapath with the communication with external memory. In Table 7 we compare the performance of the generated designs against an optimal solution where the memory accesses are perfectly scheduled and are fully overlapped with the computation in a zero latency scenario.

Table 7: Performance expectation for hand designs (P+G+O: Pipelined with Grouping and Overlapping, OPT: Optimal Scheduling Design).

Kernel		Speedup	Kernel		Speedup
SOBEL	**N**	1.00	*ATR-8*	**N**	1.00
	P+G	**1.88**		**P+G**	**2.71**
	P+G+O	3.60		**P+G+O**	4.41
	OPT	7.99		**OPT**	7.00
ATR-4	**N**	1.00	*MAZ*	**N**	1.00
	P+G	**2.00**		**P+G**	**1.55**
	P+G+O	3.56		**P+G+O**	2.91
	OPT	7.50		**OPT**	6.48

While Table 7 reveals there is still a substantial performance gap between the automatically generated codes and the possibly infeasible optimal version, the effort and time investment for a hand design is still substantial, in particular for a novice programmer. While our designs take a few seconds to generate and about 30 minutes to synthesize and download onto the board, a hand design can take days if not weeks to design and verify its correctness. This work is described in more detail in reference [8].

4.2 Custom Data Layout

To maximize parallelism of memory accesses, we have developed techniques for custom data layout, whereby elements of an array are spread across multiple memories according to their access patterns. In this way, we can potentially achieve the full external memory bandwidth of the FPGA. An example of the data layout resulting from this approach for Sobel edge detection, where the inner and outer loops are unrolled by a factor of 2, is shown in Figure 13, below. As compared to a naïve layout, where only a single memory is used, the custom layout can take full advantage of the four available memories (as is possible on the Wildstar^{TM}). We also compare with modulo unrolling, a technique used in the *Raw* compiler (another ACS program), where only the leading dimension of an array is laid out across multiple memories. Our approach is more effective in the presence of multi-loop code transformations. This work is distinguished from data layout solutions designed to map well to cache-based architectures, such as that of the DARPA DIS Adaptor project, in that in the latter, the emphasis is on eliminating conflict misses from the cache. It is also distinguished from the large body of work on data partitioning for large-scale multiprocessor systems, where the goal is to promote coarse-grain parallelism and avoid communication (vs. increased instruction-level parallelism in our approach).

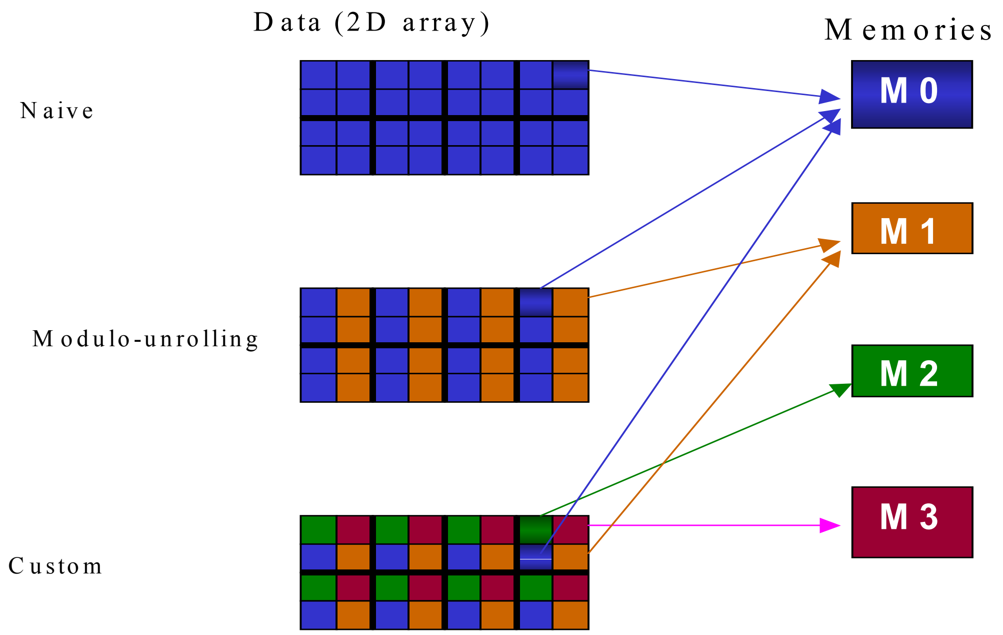


Figure 13: Comparison of custom layout with naïve (single-memory) and modulo unrolling.

The results of a study of the effectiveness of this technique for five multimedia benchmarks (described in Section II.2 above), is shown in Figures 14. Figure 14 shows the time (in cycles) spent accessing memory, for each of the three layout schemes as a function of unroll factors for the loops in each application. With higher latencies, the benefits of memory parallelism increase, so we conservatively assign a low memory latency for both read and write of one cycle each. We assume all memory accesses are pipelined.

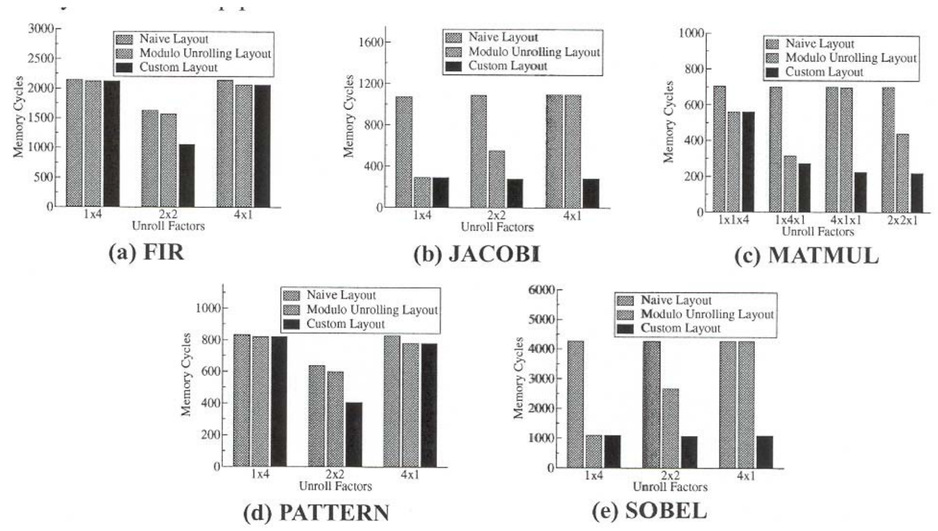


Figure 14: Memory access times versus unroll amounts.

For both the FIR 1x4 case in Figure 14(a) and the PATTERN 1x4 case in Figure 14(d), all three layout schemes perform approximately the same. This is due to the fact that the bulk of the memory accesses in the kernel are associated with multiple induction variable (MIV) array accesses. Scalar replacement cannot eliminate these accesses without further unrolling. The argument is the same for the 4x1case for both kernels as well. There are slightly more array accesses whose access expressions contain the outermost loop *i* in the lowest dimension and this accounts for the slight decrease in memory cycles.

In the 2x2 cases for FIR and PATTERN, in the custom data layout, we are able to take advantage of the unrolling in two dimensions, affecting not only the lowest order dimension in array access expressions but any dimension related to the *i* or j loop, and derive a custom layout to achieve the maximum parallelism available to the system. Therefore, our layout outperforms modulo unrolling which can only take advantage of unrolling in the lowest dimension of a given array access expression. The better performance of our custom layout is also attributed to the scalar replacement of further exposed MIV array accesses not exposed in modulo unrolling. Via our design space exploration algorithm, we would choose an optimal unroll amount of 8x4 and 24x4 for FIR and PATTERN, respectively, to expose further reuse and thereby decrease memory access time.

For both the JACOBI and SOBEL 1x4 cases in Figure 14(b) and 14(e), we see that modulo unrolling and the custom layout decrease the memory access time by 3/4 over the naive layout time. This is because unrolling the inner *j* loop by four allows for the maximum parallel data layout to be used for arrays with j in their lowest dimension access expression. For both 4x1 cases for these kernels, the custom data layout outperforms *modulo unrolling* since our algorithm is powerful enough to detect and take advantage of unrolling in any array dimension. For the JACOBI and SOBEL 2x2 cases, modulo unrolling decreases the memory access time by 1/2 over the naive layout due to unrolling by two in the *j* loop as opposed to our custom layout which again decreases the memory accesses by 3/4 over the naive layout. Since there is no array distribution in the dimension corresponding to the *i* loop for modulo unrolling, the maximum available memory parallelism is not exploited as it is for our custom layout where we distribute across memories for the dimensions corresponding to both the *i* and *j* loops.

Although MATMUL is a three-deep loop nest, we only consider unroll factors for the two outermost loops, since the compiler eliminates all memory accesses in the innermost loop through loop-invariant code motion. A subtle point is that the same mechanism does not eliminate all memory accesses in the peeled loops. Looking at the 1x1x4 case in Figure 14(c), there is a decrease in memory access cycles in modulo unrolling and custom layout over the naive case due to unrolling in the lowest array dimension. This win is due to the now parallelized memory accesses in the peeled loops. For the MATLMUL 1x4x1, 2x2x1, and 4x1x1 cases, the arguments are similar to those for the JACOBI and SOBEL 1x4, 2x2 and 4x1 cases in that our custom data layout outperforms modulo unrolling when unrolling occurs in an array dimension other than the lowest order dimension or a combination of lowest and non-lowest order dimensions.

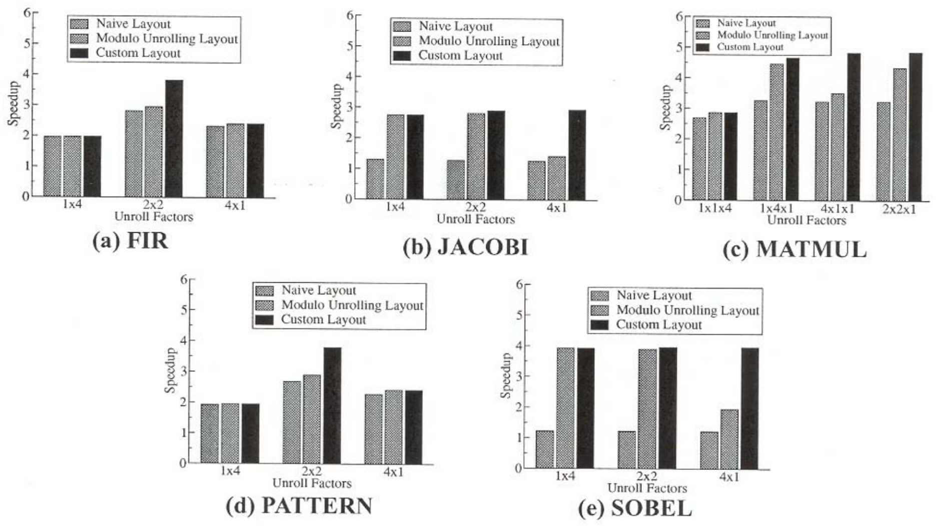


Figure 15: Speedups.

For the speedup results, the execution times are normalized to the naive 1x1 (*i.e.* not unrolled) time for each kernel. Scalar replacement was also performed in this baseline case. The speedups for FIR and PATTERN are shown in Figure 15(a) and 15(d). The speedup, due mostly to additional parallel computation, is equivalent for all three layout schemes in the 1x4 cases. The slightly higher speedups in the 4x1 cases are attributed to the fact that there are slightly more array accesses whose lowest order dimension contains the outermost loop *i*. The speedups for the 2x2 cases reflect the win from an increase in useful parallel memory accesses and are directly proportional to the decreases in overall memory cycles as shown in the earlier graphs.

The speedups for JACOBI and SOBEL are shown in Figure 14(b) and 14(e). The speedups for the 1x4 cases are proportional to the decreased number of memory cycles for each layout. In the 2x2 cases, while our custom layout is able to decrease the time spent accessing memory by a factor of three over the naive scheme, this is not reflected in the speedups. This is because JACOBI and SOBEL are highly compute bound kernels and are not able to take advantage of the additional memory parallelism exposed by our custom layout. For the 4x1 cases, modulo unrolling does exhibit a speedup over the naive scheme even though the memory access cycles are the same for these two layouts. This is due to the fact that we assume in our memory access accounting model that a read and write to different memories may not occur in parallel. This is important in JACOBI and SOBEL because many memory accesses are in the form of A(i,j) = B(i,j). In reality, the schedule that Monet generates does allow for a memory read and write to different memories to occur in parallel and coupled with the assumption that entire arrays are spread across multiple memories in modulo unrolling, we see a speedup over the naive case. The speedup for MATMUL is shown in Figure 15(c). In all cases, the speedups are proportional to the decreases in memory accesses shown in Figure 14.

Overall, we demonstrated a 3-5X improvement in performance with this technique as compared to mapping all data to a single memory. This work is described in more detail in reference [12].

5. Multi-Task Designs

As part of our focus on memory optimization, we have developed several techniques for multiple tasks, possibly mapped to distinct FPGAs. We designed and implemented data reorganization engines that reorganize data between tasks with different data access patterns. For example, in Inverse Discrete Cosine Transform, two tasks access the same array data, but one loop accesses the data in a transpose order as compared to the other loop. We have experimented with a suite of reorganizations common in DSP applications.

5.1 Data Reorganization Engines

We have implemented code generation functions in C that emit templates in structural and behavioral VHDL for a set of standard data reorganization engines described below. Each is parameterizable in terms of the number of the memory modules to which it interfaces. Each memory controller is parameterizable in terms of the number of entries and sizes of the base address and offsets. In addition we have generated selected data patterns in the switching network module and integrated it with the data engine design.

We compared the resources required for different units with distinct data access patterns. For each comparison we present the maximum achievable clock rate, number of CLBs (Configurable Logic Blocks) the unit uses in an FPGA implementation and the maximum achievable bandwidth between the input and output ports.

In this application experience we have focused on a small set of kernel data patterns, namely, transposing, row-wise and column-wise accesses and data packing and unpacking. For the purpose of the experiments we have focused on particular implementations for these operations in terms, with different parameters, of the number of channels and bit widths used, namely:

- Merging (MG-4/8) – In this kernel we merge 4 8-bit input streams into a single 32-bit output stream by interleaving each of the 8-bit data elements in a single word.
- Transpose (TP) – In this kernel the output stream is a transpose of the input data stream for a known fixed stride for both the input and the output. Both streams have a 32-bit width format.
- Stripping (ST-4/8) – In this kernel a single 32-bit word input stream is striped across 4 8-bit outputs.
- Replication (RP-8/32) – In this kernel a single 32-bit data stream is replicated across 8 outputs.

We have encoded these data access patterns and generated the VHDL descriptions for the data engines that implement them, and synthesized the result. From the implementation, we observed its metrics as presented in Table 8 and Table 9 below. We then tested the implementation of these kernel data reorganizations on an existing real FPGA-based board by applying the data reorganization to copy and reorganize data between two external memories of the same FPGA device on the WildStar^{TM} board .

Table 8 presents the size in terms of number of slices used in the FPGA for the memory controllers and network required for each of the data reorganization engines as well as the total for the data engine (along with the corresponding percentage occupancy of the FPGA). For each of the data reorganizations (except for the transpose) we have included results from different parameters. For example, we include results for the merging operation for 4 streams of 8 bits wide each (MG-4/8) and results for merging with 2 streams of 16-bit wide (MG-2/16).

Table 8: Size breakdown for the various kernel data reorganization engines.

Kernel	**Memory Controllers**	**Network**	**Total** (%)
MG-4/8	487	38	525 (4.2)
MG-2/16	325	36	361 (2.9)
TP-32	195	37	232 (1.9)
ST-4/8	477	39	516 (4.2)
ST-2/16	252	39	291 (2.4)
RP-8/32	475	103	578 (4.7)
RP-2/32	323	59	382 (3.1)

Table 9 addresses the performance of each of the variants of data engines by presenting the maximum clock rate and the sustained rate transfer in MegaBytes per second (MB/sec).

Table 9: Implementation metrics for selected set of data reorganization patterns.

Kernel	**Rate** MHz	**Bandwidth** MB/sec
MG-4/8	40.3	80
MG-2/16	40.1	80
TP-32	40.2	80
ST-4/8	40.4	80
ST-2/16	43.4	86
RP-4/32	40.3	20 to 80
RP-2/32	40.8	20 to 80

For the replication operation the sustained rate varies between 20 to 80 MB/sec depending on the number of distinct target memory modules. A 20MB/sec rate is imposed on single memory reorganization operations due to contention on the particular FPGA memory bus interface used in these experiments.

5.2 Communication and Pipelining Analysis

We have also developed a communication analysis compilation algorithm that aims at automating the remapping of data across multiple tasks and possibly globally in the context of pipelined execution. This work can determine more sophisticated remapping transformations other than transpose. It also identifies opportunities for task pipelining, and determines communication placement to maximize pipeline overlap. Currently this communication analysis implementation is able to generate correct simulation results but has not been demonstrated on the WildStar board. It has been demonstrated on five programs, SLD from Sandia ATR, MVIS, a machine vision code, Inverse Discrete Cosine Transform, and Histogram, a global histogram equalization calculation. For two of the programs, IDCT and Histogram, the analysis identifies that data requires reorganization. For the other two, the data is accessed in the same order across tasks, and so data is communicated as it is produced, in a fully pipelined fashion.

6. Multi-FPGA Designs

In the last phase of the project, we have concentrated on extending the infrastructure and the compiler analysis to adequately support, in an automated fashion, multi-FPGA designs. We have currently upgraded most of our interfaces and target VHDL abstractions to allow the compiler to specify which portions of the computation should be mapped to which computing elements and how should the data be partitioned across the various local memories on the target board.

6.1 Mapping Multi-FPGA Designs to the Wildstar^{TM} Board

We have taken a staged approach to the mapping of computations to multiple FPGAs. We have extended the current infrastructure and have successfully mapped one computation to multiple FPGAs and multiple memories, with some manual intervention. This computation is a pared-down version of the original Sandia SLD ATR computation and has been mapped manually by hand to the Annapolis WildStar^{TM} board. Our preliminary results indicate that the extended infrastructure is capable of supporting a wide variety of mappings and offers all the flexibility a compiler analysis requires. Looking forward, the next step will be to bring together what we have done with the automated approach and the version requiring manual intervention so that we can automatically map multi-FPGA designs to the WildStar^{TM} board.

6.2 Partitioning Across Multiple FPGAs

For designs with more tasks than the number of FPGAs, an additional task is to partition the tasks across the FPGAs so that system-level performance is maximized. To address this problem, we combined the automated design space exploration algorithm with communication analysis and the data reorganization analysis previously described. For a machine vision code excerpt, we show how these results can guide the compiler to perform computation and data partitioning across multiple FPGAs by matching producer and consumer rates in pipelined computations, in reference [10].

The following set of results is derived automatically for each of the three pipeline stages from the MVIS machine vision code example in Figure 16. Each pipeline stage has access to 4 memories, and the data for each stage is automatically mapped to these memories to exploit memory parallelization. For the purposes of this experiment, we assume that communication and memory latency are the same: 1 cycle for either reads or writes. This optimistically assumes full pipelining of memory accesses or communication, which is possible but not guaranteed on the Annapolis Wildstar^{TM}.

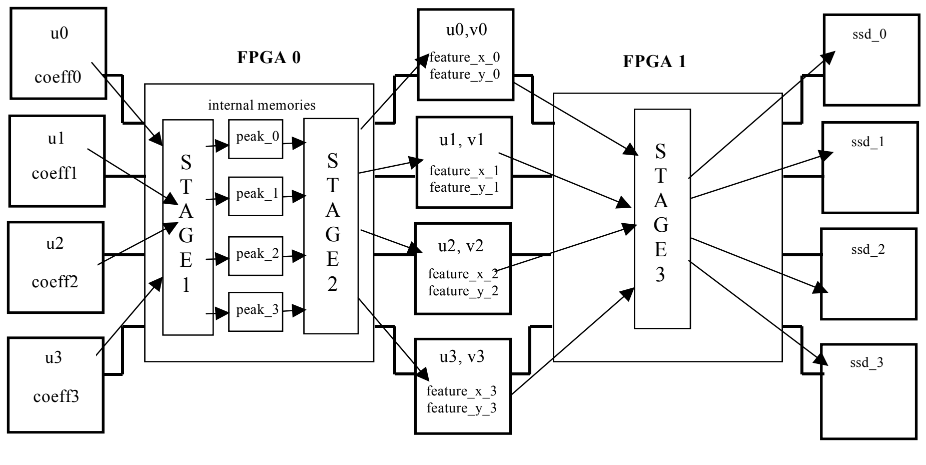
Our implementation of communication and pipelining analysis automatically derived the communication requirements and appropriate placement of communication to enable pipelining between stages. We have not yet implemented computation and data partitioning, but we plan to generalize the solution for the example presented here. In the remainder of this section, we will examine the selected design for the individual stages, and show how an automated approach would arrive at a fully integrated solution.

```
signal(host);

//  Step 1. Extract features with SOBEL
 for(x = 0; x < IMAGE_SIZE-2; x++){
  for(y = 0; y < IMAGE_SIZE-2; y++){
   // u is read only
   peak[x][y] = …;
   write(peak[x][y]);
  }
 }

 //  Step 2. Select features above threshold
 for(x=0; x < IMAGE_SIZE-2; x++){
  for(y=0; y < IMAGE_SIZE-2; y++){
   read(peak[x][y]);
   if(peak[x][y] < threshold){
    features_x[x][y] = …;
    features_y[x][y] = …;
   } else {
    features_x[x][y] = …;
    features_y[x][y] = …;
   }
    send(features_x[x][y]);
    send(features_y[x][y]);
  }
 }

 //  Step 3. Compute Distance Across Images
 for(i = 0; i < IMAGE_SIZE-2; i++) {
  for(j=0; j < IMAGE_SIZE-2; j++) {
   receive(features_x[x][y]);
   receive(features_y[x][y]);
   ssd[i][j] = 0;
   if((features_x[i][j] != 0) &&
      (features_y[i][j] != 0)) {
       ssd[i][j] = …;
   }
   send(ssd[i][j]);
  }
 }
 receive(host);
```

(b) Application Mapping to a Two FPGA Architecture

```
int U0[66][17],U1[66][17],U2[66][17],U3[66][17];
int V0[66][17],V1[66][17],V2[66][17],V3[66][17];
int SSD0[66][17],SSD1[66][17],SSD2[66][17],SSD3[66][17];
int FEATURE_X0[66][17], FEATURE_X1[66][17];
int FEATURE_X2[66][17], FEATURE_X3[66][17];

/* intialize registers
  v_0_33,v_1_33,v_2_32,v_0_17,v_1_17,v_2_16,v_3_32,
  v_3_16,v_0_32,v_0_16,v_1_32,v_1_32 */
  for (i = 0; i <= 63; i++) {
    /* intialize registers v_0_1,v_1_1 */
    for (j = 0; j <= 15; j++) { /* unrolled by 4 */
      /* unroll section 0 */
      ssd_0_0 = 0;
      u_0_0 = U0[i][1+j];  v_2_0 = V2[2+i][1+j];
      if (FEATURE_X0[i][1+j] != 0)
       ssd_0_0 = (u_0_0 - v_0_33)*(u_0_0 - v_0_33) +
          (u_0_0 - v_1_33)*(u_0_0 - v_1_33) +
          (u_0_0 - v_2_32)*(u_0_0 - v_2_32) +
          (u_0_0 - v_0_17)*(u_0_0 - v_0_17) +
          (u_0_0 - v_1_17)*(u_0_0 - v_1_17) +
          (u_0_0 - v_2_16)*(u_0_0 - v_2_16) +
          (u_0_0 - v_0_1) *(u_0_0 - v_0_1) +
          (u_0_0 - v_1_1) *(u_0_0 - v_1_1) +
          (u_0_0 - v_2_0) *(u_0_0 - v_2_0);
        SSD0[i][1+j] = ssd_0_0;
        /* unroll section 1 */
           .....
        /* unroll section 2 */
           .....
        /* unroll section 3 */
        ssd_3_0 = 0;
        u_1_0 = U3[i][1+j];  v_1_0 = V1[2+i][2+j];
      if (FEATURE_X3[i][1+j] != 0)
        ssd_3_0 =
         (u_1_0 - v_3_32)*(u_1_0 - v_3_32) +
          (u_1_0 - v_0_32)*(u_1_0 - v_0_32) +
          (u_1_0 - v_1_32)*(u_1_0 - v_1_32) +
          (u_1_0 - v_3_16)*(u_1_0 - v_3_16) +
          (u_1_0 - v_0_16)*(u_1_0 - v_0_16) +
          (u_1_0 - v_1_16)*(u_1_0 - v_1_16) +
          (u_1_0 - v_3_0)*(u_1_0 - v_3_0) +
          (u_1_0 - v_0_0)*(u_1_0 - v_0_0) +
          (u_1_0 - v_1_0)*(u_1_0 - v_1_0);
        SSD3[i][1+j] = ssd_3_0;
        shift_registers(v_0_0,....,v_0_33);
        shift_registers(v_1_0,....,v_1_33);
        shift_registers(v_2_0,....,v_2_32);
        shift_registers(v_3_0,....,v_3_32);
      } /* end of for j */
    } /* end of for i */
```

(c) Optimized Loop Nest in Stage 3.



(d) Mapping of Example Code to Pipelined Architecture

Figure 16: Machine vision example.

The following tables show behavioral synthesis estimates (augmented by compiler models) of the compiler-optimized pipeline stages for each of the three stages of the MVIS example. These results were derived automatically by our system. Tables 10-12 contain three rows. The first is Balance, as defined in Section II.2. If Balance < 1, then the design is memory bound; otherwise, it is compute bound. Assuming computation and memory access can be overlapped, Balance permits the design space exploration to limit unroll factors. For example, if a design is memory bound and memory bandwidth is fully utilized, there will be little or no performance advantage to increasing the unroll factor, and in fact, the increase in design complexity might degrade the achievable clock rate.

The second row provides the number of clock steps on a Xilinx Virtex 1000-BG560 FPGA. The third row provides an estimate of space; this number does not directly correspond to CLBs on the Virtex; in our experience with place-and-route following Monet estimation, we have determined that a number above about 32000 exceeds the capacity of the Virtex. The results are presented for different unroll factors of the innermost loop for these three stages. Note that in Tables 10 and 12, an unroll factor of 32 represents full unrolling due to loop peeling used to initialize registers for scalar replacement.

Table 10: Stage *S1* results for different unroll factors

Unroll Factor	1	2	**4**	8	16	32
Balance	1.50	1.50	**1.50**	1.25	1.13	0.95
Cycles	12344	6297	**3177**	2653	2385	2554
Space	12470	9745	**15619**	25583	45889	79928

Table 11: Stage *S2* results for different unroll factors

Unroll Factor	1	2	**4**	8	16	32	64
Balance	1.00	1.00	**1.00**	0.83	0.75	0.71	0.69
Cycles	12352	6208	**3136**	3136	3136	3136	3136
Slices	249	327	**384**	532	717	1116	1470

Table 12: Stage *S3* results for different unroll factors

Unroll Factor	1	2	4	8	16	**32**
Balance	2.00	1.67	2.00	1.63	1.44	**1.34**
Cycles	16632	10266	8229	6699	5919	**5420**
Space	16817	31954	45912	83340	132656	**262054**

The columns that are in bold face represent the desired unroll factor. To see how we arrived at this result, let us first consider the optimal unroll factors selected for each individual stage. Without unrolling, stage *S1* is compute bound. The best solution for stage *S1* in isolation is the balanced solution obtained with an unroll factor of 16. Stage *S2* without unrolling is balanced, but performance improves up to the unroll factor of 4, where the memory bandwidth is fully utilized, which we call the *saturation point.* Beyond the saturation point, the design is memory bound, and performance does not improve. In stage *S3*, the design is compute bound because, as shown in Figure 15(C), scalar replacement has eliminated most of the memory accesses. By fully unrolling the loop nest, the design is nearly balanced.

To arrive at the final solution, we make several observations:

- Stage *S3* is the slowest pipeline stage, so it is possible to reduce unroll factors for stages *S1* and *S2* without slowing down overall performance. This leads us to select unroll factors of 4 for both stage *S1* and stage *S3*.
- Stage *S3* is much larger than the other stages, so it is placed on an FPGA by itself. Stage *S1* and stage *S2* are placed on the same FPGA.
- Accordingly, the *v*, *features_x* and *features_y* arrays are placed in memories that may be shared by stages *S2* and *S3*. Array *u* is replicated.

Assuming this implementation meets the capacity constraints of the FPGA, we will derive the mapping shown in Figure 15(d).

Note that on a Virtex, capacity may be limited for Stage 3, and an unroll factor of 2 is the upper limit. In this case, we would also limit the unroll factor for Stages 1 and 2 to a factor of 2 as well. We are currently working to automate this partitioning (the remainder of the implementation is complete).

III. Infrastructure

In this project we have addressed system-level issues that were not addressed by industry or any other academic project. We have focused on design space exploration using estimation techniques as an approach for the tool to generate designs that will physically fit on the target FPGA devices. In addition we have also focused on memory access optimization and customization as a technique to increase the effective data bandwidth for the designs – a major concern for important application domains such as digital image processing. In this section, we describe the complex infrastructure we have developed for this project.

The DEFACTO compilation and synthesis tool was and still is a major undertaking. The current infrastructure contains about 100K lines of C/C++ code in the SUIF system, several thousand lines of structural VHDL, and additional few thousand lines of scripts to run the synthesis tools. It includes numerous internal code generation tools interfacing two compilers, a behavioral synthesis tool as well as logic synthesis and place-and-route (P&R) tools.

DEFACTO uses the SUIF (Stanford University Intermediate Format) front end to convert C and FORTRAN programs to its internal representation. It also leverages data dependence and other parallelization analyses, and code transformations from SUIF. In addition, DEFACTO includes commercially available EDA tools and also interfaces with other standalone C compilers on PC platforms.

As outlined in Figure 16, the system starts with a high-level algorithm description in standard C and generates two sets of files. One file contains the original program, with the portion of the computation that executes on the FPGAs removed and replaced with library calls that manage execution on the FPGA as well as transfer data to and from the configurable architecture. Another set of files specifies the computation to be executed on the FPGAs and contains numerous internal components. These components interact with the hardware to effectively transfer data to and from its internal memories and synchronize its execution with the C code. These files must then be compiled using the native C compiler of the target processor (in our case a PC using the Visual C++ compiler) as well as commercially available logic synthesis tools. We have used Mentor Graphics Monet to translate a behavioral VHDL file into a netlist formatted file which we feed to Synplify's Synplicity logic synthesis tool and P&R tools to generate the final bit-stream executable file for programming each FPGA

We now describe the fundamental infrastructure building blocks of the DEFACTO compilation system. The front end consists of the C/Fortran SUIF compiler front-end that translates the input code to the SUIF intermediate format.

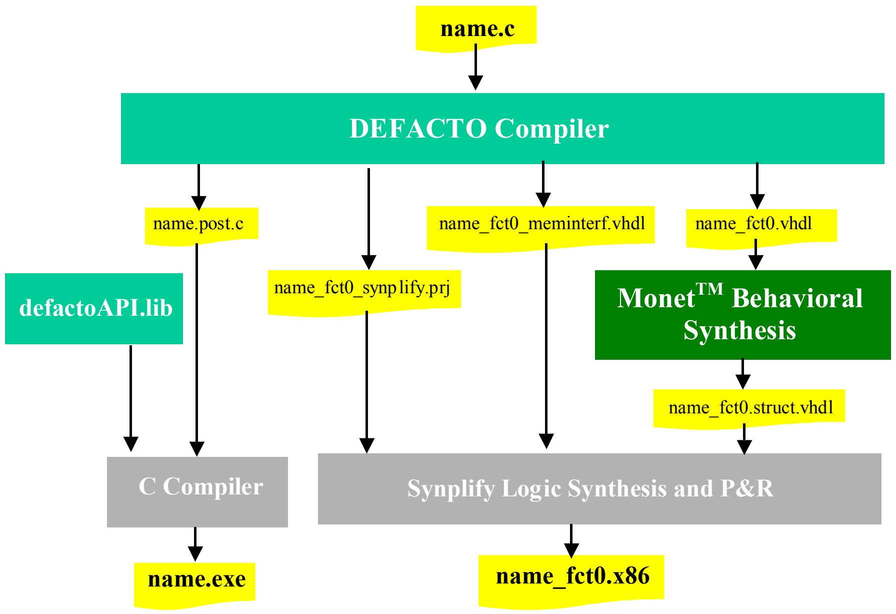


Figure 17: The DEFACTO design flow.

As part of the DEFACTO front-end, we have developed numerous analysis passes beyond the SUIF v1.2 release, briefly described below.

1. **Data Reuse Analysis.** This analysis identifies when a data value accessed through a memory reference is accessed by another reference, across all loops of a given loop nest. This analysis also permits elimination of unnecessary memory accesses.

2. **Delay Line Analysis.** This analysis scans each inner loop data access pattern attempting to identify simple data reuse opportunities. It then converts these opportunities into tapped-delay lines in the generated code to reduce the number of memory accesses.
3. **Communication Analysis.** This analysis pass identifies the set of loop nests that share data either in the form of complete arrays or in the form of array section. It is used to determine which loop nests (or loops within each nest) should be aggregated into hardware tasks to match the data partition strategy.
4. **Pipelining Analysis.** This analysis identifies the data access patterns across different loops. The compiler uses this information to determine good data partition and placement strategies in conjunction with communication analysis.
5. **Memory Interface Library and Annapolis Library Interface.** This pass is responsible for generating a series of structural VHDL definitions that are used by the designs in each FPGA to access memory. It has been integrated with the pipelined memory access features offered by the target board for enhanced performance. This pass can also generate various memory controller configurations for increased memory subsystem performance, as discussed in Section II.4.
6. **Script Generation Tools.** This set of tools is responsible for the generation of EDA tools scripting languages and encapsulate most of the naming schemes used to identify bit-stream and VHDL files the compiler generates.

To interface with behavioral synthesis, we have also developed a tool that translates SUIF constructs to behavioral VHDL amenable to be processed by the Mentor Graphics Monet behavioral synthesis tool. This tool called SUIF2VHDL generates a set of VHDL constructs that perform the computation of a loop or set of loops and interfaces with a set of VHDL abstractions we have developed to allow the implementation to actually read and write data to the memories on the target board. The main abstraction of the VHDL generated code is a *task*, which reads and writes data to and from a set of memories via *channels*. Tasks can also communicate via direct links as directly supported in the Annapolis WildStar^{TM} board.

A significant component of the infrastructure has to deal with providing a set of VHDL abstractions the compiler can use to generate behavioral VHDL. We have developed the notions of hardware channels and channel controllers. These abstractions allow for the compiler and its analysis algorithms to perform a wide variety of optimizations leading to an effective increase in the available memory bandwidth to the design (see Section II.4).

A significant challenge of this project was making design execute consistently and reliably on the target architecture, the Annapolis WildStar^{TM} board. In this context we have faced and overcome several challenges:

1. **Annapolis Hardware problems**: An earlier board had intermittent memory errors causing application failure and the inability to develop other components of the system. Eventually the board was replaced by Annapolis without any report of the cause of the problem.

2. **Annapolis Board Interface problems**: Annapolis interfaces have been hard to use due to their proprietary nature. The initial version of the board interfaces were later replaced to adequately support the Mezzanine Cards that allowed us to access more memories and effectively exploit more FPGA computing capabilities. Once the Annapolis libraries were revised, we had to update our entire design flow, which resulted in a significant delay in advancing to implementations of multi-FPGA designs.
3. **EDA Tool Integration**: Because our goal was to use behavioral synthesis and the Annapolis WildStar^{TM} board, we were forced to integrate tools from four different vendors, and encountered many incompatibilities. In the last year, we had to accommodate a complete release overhaul of our synthesis flow – Monet^{TM} v4.2 and Synplicity v7.0 to comply with the Annapolis compatibility and tool requirements.
4. **Behavioral VHDL idiosyncrasies**: We spent a significant amount of time and energy understanding the style of VHDL constructs that would map well to hardware to be generated by the SUIF2VHDL tool.

Overall all these challenges substantially hampered our progress in the early and intermediate phases of the projects. Over the course of the last 18 months, we have progressed the capabilities at a rapid and steady pace, now that we have a reliable hardware and software design flow. As we continue this work under separate funding, we will build on this complex infrastructure and continue to enhance its capabilities.

IV. Additional Activities

The UCLA group supported the DEFACTO effort in two ways. They contributed significantly to debugging the design flow presented in the previous section, working through case studies, iterating with the synthesis tool vendors, and deriving scripts to run the tools. They also developed module generators and determined how to integrate them into the design flow. Among the modules developed by UCLA include an RC_adder/subtractor, AdderTree, On-line Adder, On-line DA 4 coefficient MAC, borrow-save adders, vector register modules, and FIR filters. In addition, in conjunction with the previously described compiler work on generating code for tapped delay lines, UCLA developed tapped delay line libraries, for representing 1-dimensional and 2-dimensional arrays. The need for a library was motivated by the observation that synthesized designs with tapped delay line structures led to inefficiencies. By using placement constraints on the tapped delay line library in the Xilinx Foundation tool (vertical placement), we achieved an implementation of the library that used a 36% faster clock and 25% fewer slices. These libraries tie in nicely with the compiler analysis that identifies when the libraries can be used.

Angeles Design Systems supported DEFACTO by participating in the early system design. Subsequently, Angeles developed the S2VHDL tool, which derives behavioral VHDL from a SUIF intermediate representation. Angeles worked with ISI and UCLA on case studies to derive a flavor of VHDL that could be supported well by the synthesis tools. Angeles also provided a specification of the channel library implementation.

Integrated Sensors' task for DEFACTO was to integrate the HRTExpress / MATLAB programming environment to provide MATLAB level FPGA programmability for the DEFACTO system. Integrated Sensors' efforts involved definition of the required MATLAB-to-SUIF conversion process and HRTExpress GUI changes required to support DEFACTO. DEFACTO compatible, annotated SUIF vector functions can be obtained by leveraging existing C math library functions and by using the Math Works MATLAB-to-C translator. Through modification of the HRTExpress pre and post translator passes, GUI selection and specification of the required hardware information is passed, through SUIF annotations, to the VHDL backend. After DARPA redirection in fall 2000, the MATLAB frontend for DEFACTO was discontinued.

ISI completed an evaluation of requirements to map to System C rather than Behavioral VHDL, as stated in last quarter's project summary. What we found was that mapping the datapath to System C instead of Behavioral VHDL was straightforward, and we completed a prototype implementation. However, a number of automatically-generated Structural VHDL libraries are included in our current designs. We are not aware of any good approach to mixing VHDL and SystemC code in synthesis, so a move to SystemC does not appear feasible at this time. Under direction from DARPA, we revised our plans and will not implement the SystemC port.

V. Technology Transfer

Most recently, on July 31, 2002, an important technology transfer activity was a demonstration by Pedro Diniz and Mary Hall of the end-to-end mapping and design space exploration at the DarpaTech conference. There were a large number of people who dropped by our demonstration, but there was significant interest from Randy Carlson, the CTO of Red Hawk Inc., who expressed an interest in establishing a collaboration.

In the last years of the project, we met with a number of industrial representatives to discuss our technology and how it might be incorporated into their future product lines. In particular, Mary Hall and Pedro Diniz gave a company-wide seminar at Synopsys, and spoke with Synopsys CTO Raul Camposano about how DEFACTO technology might be used in mixed hardware/software designs, particularly if the emerging SystemC standard is used as the input to synthesis tools rather than VHDL. Pedro Diniz and Mary Hall also visited C-level and spoke with Vice President of Marketing David Park about how DEFACTO technology could be used in conjunction with higher level languages, such as using C-level as a target output (instead of VHDL) for DEFACTO.

We have had discussions with many other interested individuals from industry, including Nick Dragiewicz from CoWare, John Glosner from Lucent Technologies, J. Tresher from Philips Research and Mike Vahey from Raytheon. Recently, we had a request from Raytheon for copies of our software.

VI. Summary

This report describes the results of the DEFACTO project, which is an end-to-end design environment deriving application-specific hardware for FPGA-based systems from a high-level algorithm specification in C. This project has demonstrated reductions in synthesis time of 100-10000X with the automated design space exploration algorithm. The current reduction in design time, including human effort, has been approximately 40-60X for two case studies, SLD from Sandia ATR and Sobel edge detection. We have demonstrated end-to-end mapping on the Annapolis Wildstar^{TM} board for both examples, with no manual intervention, at DarpaTech on July 31, 2002. The end-to-end demonstrations use the external host, multiple memories and a single FPGA. We have also demonstrated in simulation automatically-generated multi-FPGA designs, and have mapped a multi-FPGA design for the SLD code from Sandia SLD ATR to the Wildstar^{TM} board with modest manual intervention. The DEFACTO project will continue under NSF funding, and we plan to complete automatically-generated FPGA designs to the Wildstar^{TM} board in the near term, as well as explore algorithms to partition computation and data across multiple FPGAs and memories.

The products of the DEFACTO project were a complex infrastructure for end-to-end design, and a set of technical results obtained from the resulting system. We produced 12 papers in a broad range of conferences, including synthesis, FPGA, compiler and parallel computing conferences. Details on the key technical results can be found in the publications listed.

Publications (in chronological order)

[1] Kiran Bondalapati, Pedro Diniz, Phillip Duncan, John Granacki, Mary Hall, Rajeev Jain, Heidi Ziegler, "DEFACTO: A Design environment for Adaptive Computing Technology," In Proc. of the Reconfigurable Architectures Workshop, held in conjunction with the International Parallel and Distributed Processing Symposium, April, 1999.

[2] Byoungro So, Heidi Ziegler and Mary Hall, "Parallelizing Compiler Technology for Adaptive Computing Systems," In Proceedings of the Workshop on Reconfigurable Computing, held in conjunction with the Parallel Architectures and Compilation Techniques Conference, October, 1999.

[3] Pedro Diniz and Joonseok Park, "Automatic Synthesis of Data Storage and Control Structures for FPGA-Based Computing Engines," In Proceedings of the IEEE Symp. on FPGAs for Custom Computing Machines (FCCM'00), IEEE Computer Society Press, Los Alamitos, CA, April, 2000.

[4] Pedro Diniz and Ashok Venkatachar, "A Behavioral Synthesis Estimation Interface for Configurable Computing", In Proceedings of the IEEE Symp. on FPGAs for Custom Computing Machines (FCCM'01), IEEE Computer Society Press, Los Alamitos, CA, April, 2001.

[5] Joonseok Park and Pedro Diniz, "An External Memory Interface for FPGA-based Computing Engines", To appear in the Proceedings of the IEEE Symp. on FPGAs for Custom Computing Machines (FCCM'01), IEEE Computer Society Press, Los Alamitos, CA, April, 2001.

[6] Pablo Moisset, Pedro Diniz and Joonseok Park, "Matching and Searching Analysis for Parallel Hardware Implementation on FPGAs" In the Proceedings of the ACM Symposium on FPGAs (FPGA'2001), ACM Press, New York, Feb. 2001, pp. 125-133.

[7] Pedro Diniz, Mary Hall, Joonseok Park, Byoungro So, Heidi Ziegler, "Bridging the Gap Between Compilation and Synthesis in the DEFACTO System," In Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC '01), August, 2001.

[8] Joonseok Park and Pedro Diniz, "Synthesis of Pipelined Memory Access Controllers for Streamed Data Applications on FPGA-Based Computing Engines," In Proceedings of the 14th International Symposium on System Synthesis (ISSS '01), Montreal, Canada, October, 2001.

[9] Pedro Diniz and Joonseok Park, "Data Reorganization Engines for the Next Generation of System-On-a-Chip FPGAs," *In Proceedings of the ACM Symp. on Field-Programmable-Gate-Arrays (FPGA'02)*, ACM Press, New York, Feb. 2002, pp. 237-244.

[10] Heidi Ziegler, Byoungro So, Mary Hall and Pedro Diniz, "Coarse-Grain Pipelining for Multi-FPGA Architectures," In *Proceedings of the ACM Symposium on FPGAs for Custom Computing Machines* (FCCM '02), April, 2002.

[11] Byoungro So, Mary Hall and Pedro Diniz, "A Compiler Approach to Fast Design Space Exploration for FPGA-Based Systems," In *Proceedings of the ACM Symposium on Programming Language Design and Implementation,* June, 2002.

[12] Byoungro So, Heidi Ziegler and Mary Hall, "A Compiler Approach for Custom Data Layout," In Proceedings of the Workshop on Languages and Compilers for Parallel Computing, July 2002.