AFRL-RY-WP-TR-2010-1095

ALGORITHM CLASSES FOR ARCHITECTURE RESEARCH (ACAR)

Jinwoo Suh, Stephen P. Crago, Karandeep Singh, and Janice O. McMahon

University of Southern California

MARCH 2010
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Defense Advanced Research Projects Agency Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2010-1095 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

*//Signature//

KERRY L. HILL, Project Engineer
Advanced Sensor Components Branch
Aerospace Components Division

//Signature//

BRADLEY J. PAUL, Chief
Advanced Sensor Components Branch
Aerospace Components Division

//Signature//

JAMES LOUTHAIN, Lt Col, USAF
Deputy Chief
Aerospace Components Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
March 2010	Final	23 April 2008 – 31 March 2010

4. TITLE AND SUBTITLE
ALGORITHM CLASSES FOR ARCHITECTURE RESEARCH (ACAR)

5a. CONTRACT NUMBER
FA8650-08-C-7832

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)
Jinwoo Suh, Stephen P. Crago, Karandeep Singh, and Janice O. McMahon

5d. PROJECT NUMBER
ARPR

5e. TASK NUMBER
YD

5f. WORK UNIT NUMBER
ARPRYD09

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Southern California
Information Sciences Institute
3811 N. Fairfax Drive, Suite 200
Arlington, VA, 22203

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

Defense Advanced Research Projects Agency/
Information Processing Techniques
Office (DARPA/IPTO)
3701 North Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYDI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2010-1095

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
PAO Case Number: DISTAR 16545; Clearance Date: 19 Nov 2010. This report contains color.

The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

14. ABSTRACT
The University of Southern California / Information Sciences Institute (USC/ISI) conducted exploratory studies to establish the need for and the value of innovative research on domain-specific architectures, applications, and tools based on the challenges posed by computational bottlenecks in DoD applications. The study was driven by key representative applications limited in performance by current computing performance and tool chains.

There are several related research areas that the ACAR project studied that have significant potential benefits to DoD applications: multi-core application specific architectures, emulation systems, automatic board generation, design patterns for increasing productivity, and domain-specific languages. USC/ISI recommends further research in these areas.

15. SUBJECT TERMS
application-specific architecture, domain-specific architecture, emulation system, evaluation board, board generator, design pattern, domain-specific language

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Kerry L. Hill
Unclassified	Unclassified	Unclassified	SAR	82	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Table of Contents

List of Figures

1 Summary

The University of Southern California / Information Sciences Institute (USC/ISI) conducted exploratory studies to establish the need for and the value of innovative research on domain-specific architectures, applications, and tools based on the challenges posed by computational bottlenecks in DoD applications. The study was driven by key representative applications that are limited in performance by current computing performance and tool chains.

The initial study was to evaluate the performance of domain-specific architectures. To evaluate domain-specific architectures, USC/ISI has developed performance and chip area estimation tools. These tools allowed us to estimate the performances for various chip designs for different kernels and an application. The experimental results showed that different application kernel characteristics and applications demand different processor architectures. Therefore, a general-purpose processor cannot meet all the different processing requirements in different domains. Thus, for different application domains, different architectures fitting the processing requirements in the domain can achieve performance that is not achievable by general-purpose processors.

The second part of the study was on emulation systems. We evaluated existing emulation systems first and then proposed an emulation system that can address DoD needs, such as security and the need to support tools specific to DoD applications. The emulation system provides orders of magnitude faster emulation speed compared to software simulation. The emulation system uses field programmable gate arrays (FPGAs) for various architectures with a flexible software stack to accommodate a wide range of DoD applications, processors, and systems.

Evaluation systems are needed for any newly developed processor to ensure design validity and to support software development until hardware is available. We propose flexible evaluation boards for three new processor approaches: structured ASICs, patterned ASICs, and automatically generated ASICs. These proposed evaluation systems could be made specific to different applications by populating different configurations on the board to cope with different application environments.

Based on the evaluation board research, we also propose an automatic board generator tool. The board generator automates manual board design tasks for related chip designs by exploiting templates and standard interfaces. but the primary benefits are also reduced development time and increased reliability, along with cost reduction.

A third part of the study was on design patterns. Application performance, especially on parallel systems, can be very different depending on the quality and experience of programmers. One way to increase the quality of a novice programmer's code is by using design patterns. The design pattern is a description of an approach that programmers use to learn how to write programs that are likely to map well to parallel architectures. In this study, we evaluated the potential benefits of research on design patterns for DoD applications by applying design patterns to exemplary kernels. The study shows both that DoD could benefit

from design patterns and that there are DoD-specific needs that are not addressed by current design patterns that could be addressed through additional research.

A fourth part of the study was on domain-specific languages. Domain-specific languages, such as SQL and MATLAB, have been used widely in many domains successfully. These domains are highly productive due to the many advantages of domain-specific languages, such as proper abstractions of problems. In this study, USC/ISI conducted a study on domain-specific language to identity potential benefits for DoD applications. As an example, USC/ISI developed a straw-man signal processing language targeting DoD signal processing applications. USC/ISI also implemented an application called complexity ambiguity function (CAF) using the proposed language and demonstrated the potential benefits of the domain-specific language.

2 Introduction

Since the introduction of the first microprocessor, general-purpose processors have been used widely for most computing applications. However, in some applications and domains, the general-purpose processors do not meet computing demands. For example, graphics processing and signal processing often need much more powerful or efficient computation capabilities than general-purpose processors can provide. At the same time, some features provided by general-purpose processors are unnecessary in specific domains. Therefore, many domain-specific processors, such as GPUs and DSPs have been developed for graphic processing and signal processing, respectively, and are widely used.

In this work, we have conducted a series of studies on the architectures, systems, tools, and languages in domain-specific areas. To study the domain-specific architectures, we first surveyed and selected four key representative kernels and applications. The chosen kernels and applications are matrix multiplication, FFT, object detection, and neural networks.

With the chosen kernels and applications, we designed domain-specific architectures. First, we designed models for chip area and application performance. Then, with the models, we searched the design space for high performance for each application to show how domain-specific architectures should be designed to best fit the target domains.

When a new processor with a domain-specific architecture is being developed, it needs to be simulated or emulated to verify that the new design is bug-free. Due to the slow speed of accurate simulation, emulation is used in many cases to expedite the design process. In this study, we evaluated commercial FPGA evaluation boards and commercial emulation systems. Commercial evaluation systems are usually too small to emulate a new processor and its system. Commercial emulation systems are expensive and have limited flexibility.

When a new processor is manufactured, the new processor needs to be tested on an evaluation board. The proposed evaluation board design assumes a template with some common pin layouts such as power pins for reliable design. However, it also supports some flexible pin layouts such as memory interface pins to accommodate different architecture characteristics. Moreover, the board design can be used for multiple application areas. For example, if an application is a memory-intensive application, it can be configured with a large memory. If an application is I/O intensive, the board can be configured for many I/O interfaces.

Traditional board design involves a human design phase especially in the schematic capture and place and route (PAR) stages. Since most schematic capture is done manually, the design cost is high and error-prone and lead times are long. The cost of errors can be very high, especially in lost design time, lost opportunities in the market, and delayed deployment to the warfighter. We have conducted a study to investigate issues related to an automatic board generator tool. Based on our study, we propose a design tool called a board generator that automates the process of designing a board. The board generator can be developed for boards that can house various types of new or old processors.

There are big differences in quality of codes between programmers, which stem from practical knowledge and skill. When a program needs to be written, programmers often need to develop or choose a programming approach, which may not be suitable for the target application. Sometimes, after a long coding time, the programmer discovers the approach's limitations and will need to start again.

However, high-productivity programmers learn programming methods and techniques through past successes and failures. With this experience, the programmer will know the appropriate programming approach and can finish a high quality code in a short development time with decreased risk. Design patterns are one way of helping more programmers to write like productive programmer, which is especially important for the parallel programming required to leverage the potential performance improvements of multi-core processors. Design patterns provide a tested and proven approach. A design pattern is a documented approach to writing a program for a specific problem type. A pattern description typically consists of several sections. The "problem" section describes the problem. The "forces" section describes a trade-off in various approaches. And the "solution" section describes how to program the pattern.

In our study, we have evaluated the use of design patterns for DoD applications using matrix multiplication.

In the application development process, one important tool is the programming language. There are many general-purpose programming languages such as C and FORTRAN that are widely used in application development. However, in some domains such as database processing, general-purpose languages do not meet the programming needs in specific domains. Therefore, several domain-specific languages have been designed and used intensively in those domains. For example, in database management, SQL is mainly used to create, update, and delete records in a database. Using any other general-purpose language is not as efficient as SQL. At the same time, using SQL for another purpose is not efficient, since the SQL is a domain-specific language for database management only.

In this work, we studied the possibility of using a domain-specific language for DoD applications. First, we identified an area that has an inadequate language for DoD applications and proposed a straw-man domain-specific language for the domain. We selected the signal processing area, which is used widely in DoD applications. Currently, there are a few commonly used languages for the signal processing applications. For example, the C language is widely used in the area to obtain high performance. However, the C language is a general-purpose language, and uses a much lower level of abstraction than signal processing domain experts typically work in.

To study the potential benefits of having a domain-specific language in the signal processing domain, we designed a straw-man domain-specific language. The domain language covers many signal processing related constructs. To demonstrate the use of the language and the potential benefits of the proposed language, we implemented a signal processing application called complex ambiguity function using the proposed language. Also, the implementation is compared with other language approaches.

The rest of this report is organized as follows. In section 3, the methods, assumptions, and procedures used in the work are described. In section 4, the results obtained from the work are described in detail. Section 5 concludes the work, and section 6 describes recommendations.

3 Methods, Assumptions, and Procedures

In this section, we discuss methods, assumptions, and procedures used for our study.

3.1 Multi-core Domain-specific Architecture

In this section, the multi-core domain-specific architecture study is discussed. We discuss kernels and an application used in this study followed by a detailed discussion.

3.1.1 Kernels and an Application Considered in this Study

We considered three kernels and one application. The three kernels are matrix multiplication, FFT, and object detection. The one application considered is neural network.

Matrix multiplication is frequently used in many real-world applications including DoD embedded signal processing applications. We considered implementing a matrix multiplication using a commonly known technique--blocked matrix multiplication, which reduces off-chip data transfers. The partitioning is done on the output matrix, and each core computes its own data.

FFT is another commonly used kernel in many applications to transform data in the time domain to the frequency domain and back [32]. We considered the conventional radix-2 FFT with complex floating-point numbers. Each core processes its own data set in parallel.

Object detection detects target objects from input images. We used the Viola-Jones's algorithm [38] to find an object of pre-defined class in a static image or video frame. The call graph for the algorithm is shown in Figure 1.

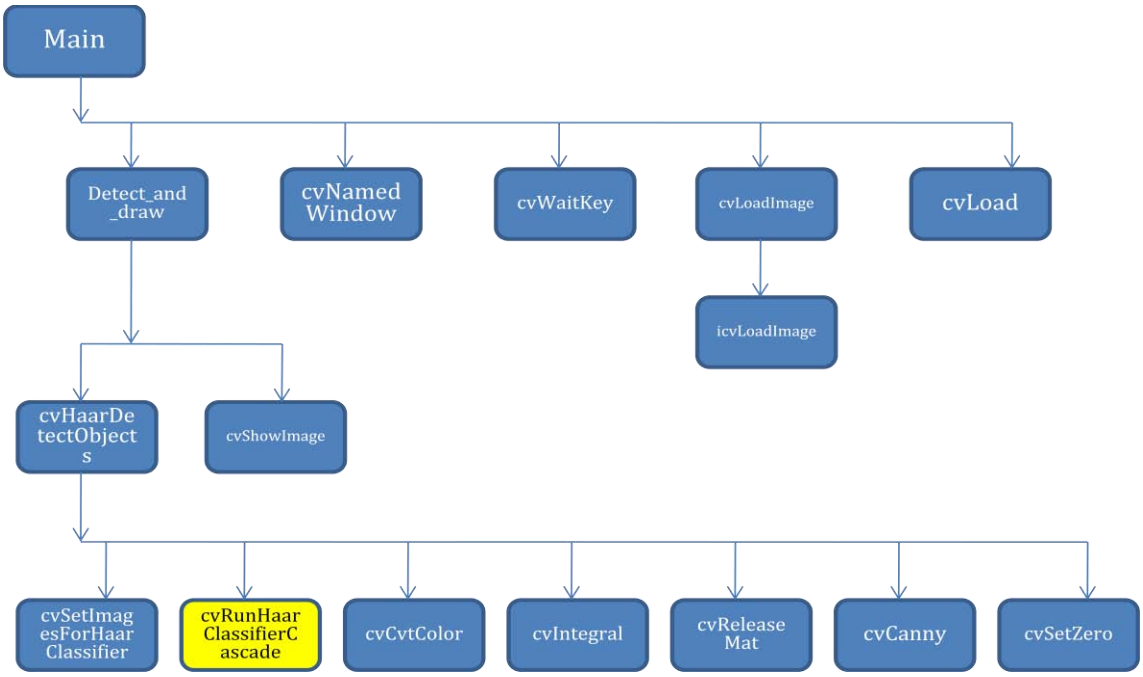


Figure 1. Object detection call graph

Viola-Jones is a robust and state-of-the-art algorithm where feature extraction is done using Haar-like features. Feature selection/classification is done using a variation of the AdaBoost learning method [8], which selects a small number of critical visual features and yields an extremely efficient feature selector/classifier. A multi-scale detection algorithm combines classifiers in a cascade, which allows background regions of the image (typically 1000 times as many non-faces as faces in an image) to be quickly discarded while spending more computation on promising object-like regions.

We analyzed the performance data collected for the algorithm using the Intel VTune Performance Analyzer [9]. It showed that out of all the functions called by the algorithm, the cvRunHaarClassifierCascade function consumed more than 90% of the total execution time. We studied and analyzed this most important function in detail. Figure 2 shows the data flow diagram for this function. The diagram shows the cascade structure of computations.

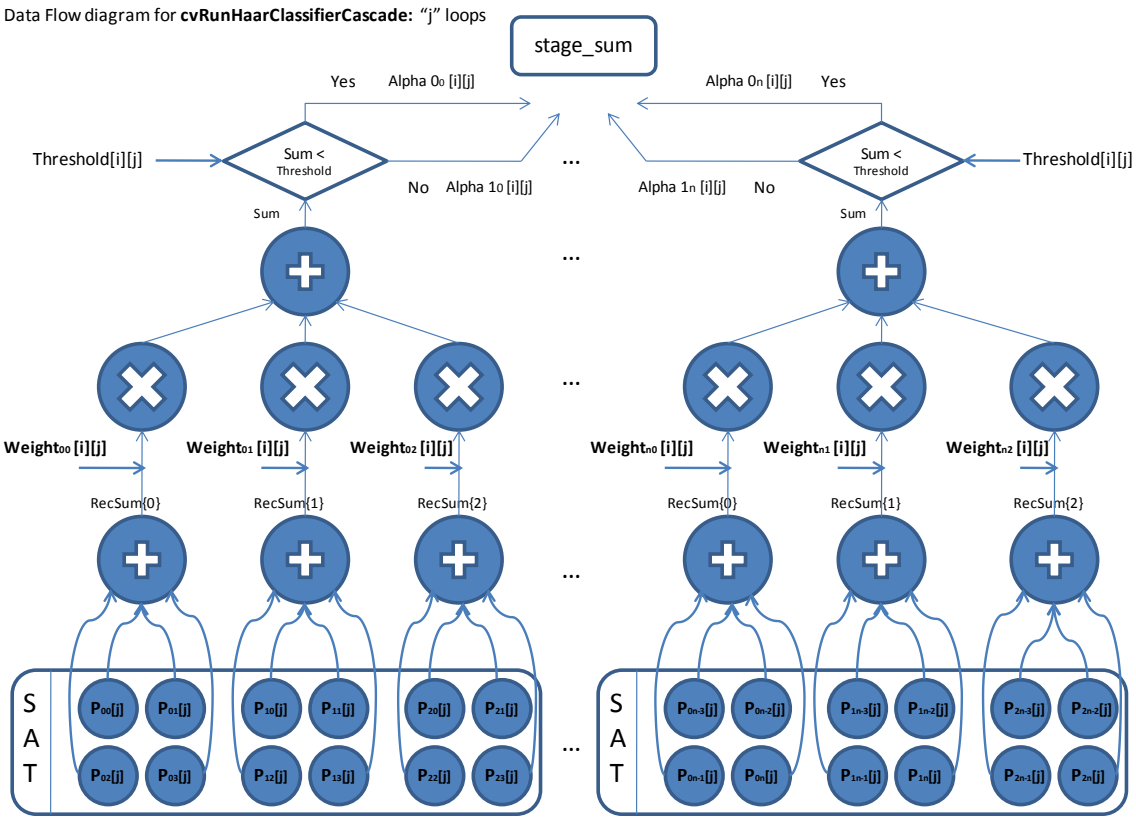


Figure 2. Object detection data flow diagram

One application we considered was a neural network that uses the adaptive resonance theory (ART) algorithm [12]. It is used in a wide spectrum of applications including satellite remote sensing, automatic target recognition, and radar processing.

ART self-organizes stable pattern recognition codes in real-time in response to arbitrary sequences of input patterns. ART2 systems [12] have a pattern matching process that compares an external input with the internal memory of an active code. This matching leads either to a resonant state, which persists long enough to permit learning or to a parallel memory search. If the search ends at an established code, the memory representation may either remain the same or incorporate new information from matched portions of the current input. If the search ends at a new code, the memory representation learns the current input.

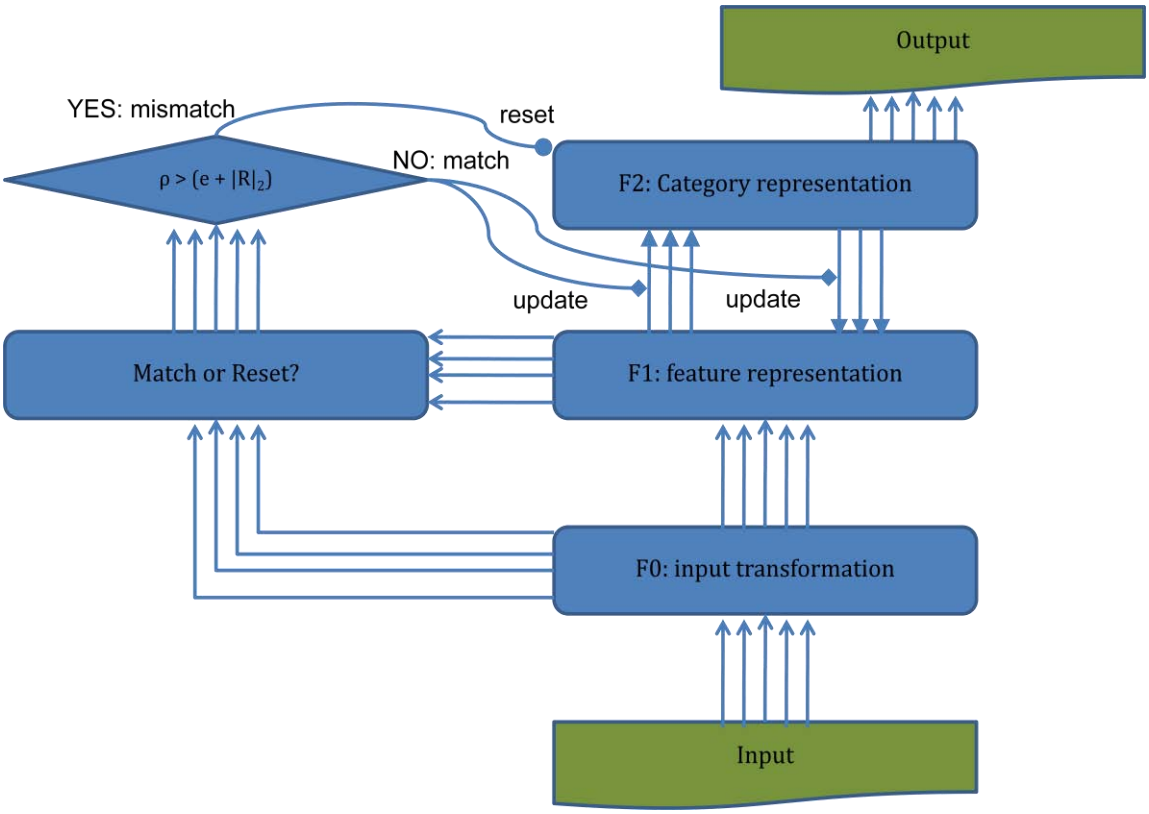


Figure 3. ART2 neural network block diagram

Figure 3 shows the block diagram for the ART2 neural network. Field F0 is for input transformation and the two fields F1 and F2 are for a feature representation and a category representation.

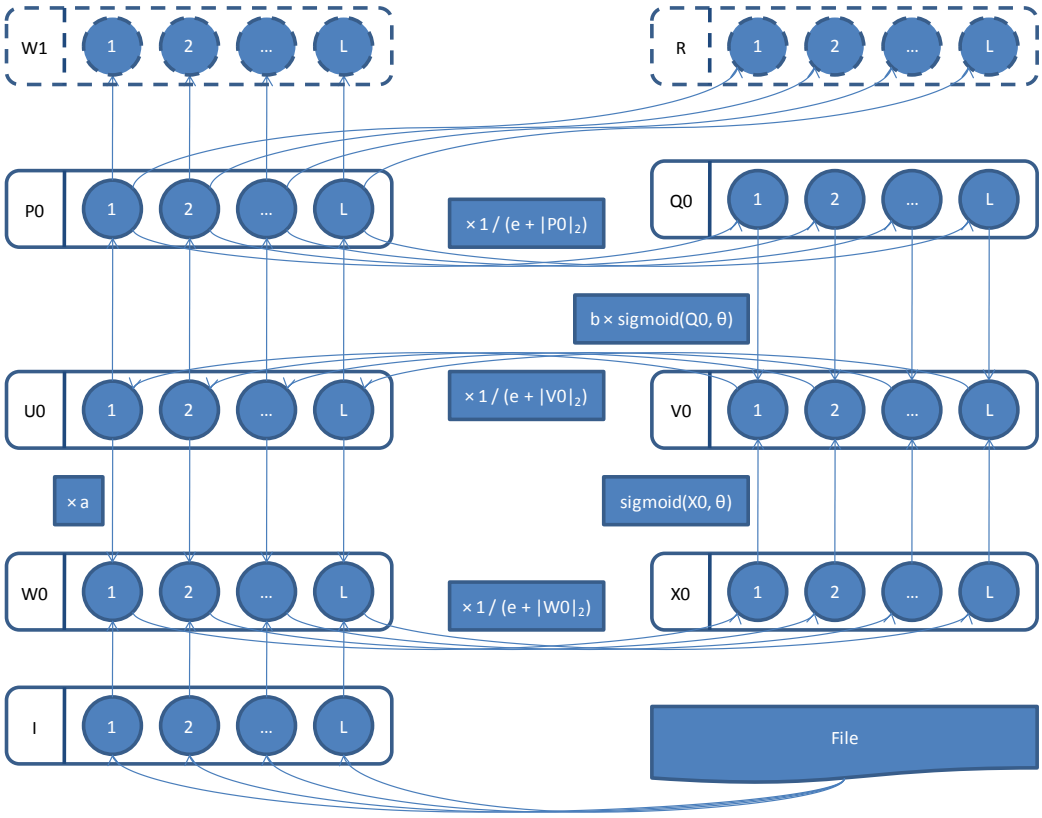


Figure 4. ART2 F0 block diagram

Figure 4 shows the F0 block diagram. It consists of element-wise operations (floating-point multiplications and additions). It has L elements, where L is the length of each input. For our data sets, we have the value of L range from 8 to 30,000.

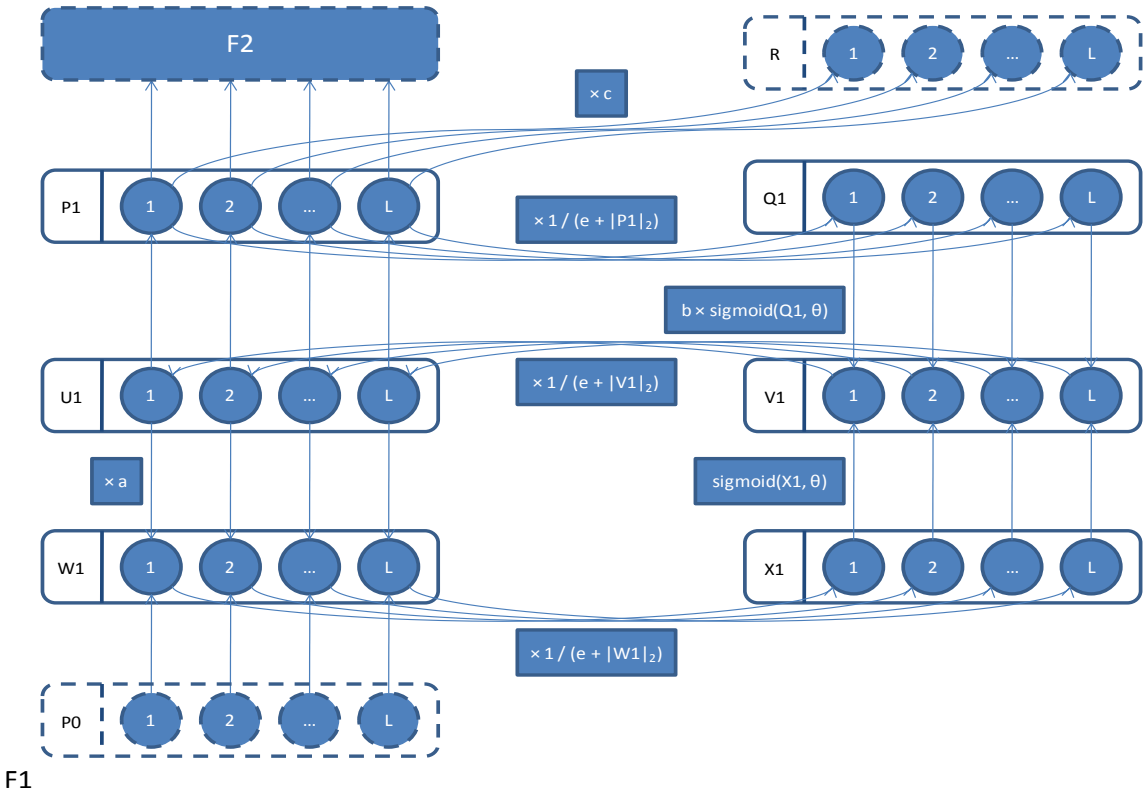


Figure 5. ART2 F1 block diagram

F1, which has an identical structure to F0, is shown in Figure 5. It has element-wise local communication with F0. So to avoid communication between processing elements (PEs), F0 and F1 are computed on the same PE.

```
for j = 0..N-1
    for i = 0..L-1
      T[j] += P[i] * Zij[i][j];
    end
    T[j] = mis[j] * T[j];
    ymax = max(ymax, T[j]);
end
```

Figure 6. ART2 F2 loop

The main compute loop for F2 is shown in Figure 6. It has *N* processing elements (*N* ranges from 12 to 1600). Each PE gets inputs from all *L* F1 PEs. Thus, an efficient broadcast is required.

3.1.2 Performance Model

In this section, we discuss an area estimation model and performance models to do a tradeoff study to find an high-performance processor design. Using the models, we estimated the performance for three kernels and one application.

The area estimation and performance models consider several factors, such as number of cores, vector size, and cache sizes. The area estimation model estimates the chip area needed for a given set of architectural components such as caches and FPUs. Performance models estimate the peak performance for each kernel and application. Although peak performance is usually much higher than real-world performance, it provides upper bounds, which we use for trade-off analysis in the early stages of architecture design. The parameters used in the equations are defined as follows:

m: number of cores
U: CPU area
C: cache area per word
c: cache size in words
F: FPU area
f: 1 if FPU exists, 0 otherwise
M: complex FPU area. The complex FPU computes a complex computation with a single instruction.
p: 1 if complex FPU exists, 0 otherwise
v: vector length
t: 1 if stream unit exits, 0 otherwise. With a stream unit, data read or write operations are initialized in a similar way to DMA. Then, the data is read or received continuously using a register.
S: stream unit area
X: switch processor area (bandwidth independent part). The switch processor acts like a DMA engine for communication. After initialization is done it takes on the task of inter-core data transfer, so that the CPU can continue its computation task.
b: on-chip network width in words
Y: switch processor area per word (bandwidth dependent part)
W: network area per unit length
w: off-chip memory interface width in words
D: off-chip memory controller space per word

The area estimation model is shown below.

$$area = m\left(U + Cc + (fF + Mp)v + tS + X + bY + 2bW\sqrt{U + Cc + (fF + Mp)v + tS + X + bY}\right) + wD$$

The area model estimates the overall space by multiplying the number of cores with the area for a core. The core area includes all features in a core, such as local memory and functional units. Area for a unit that manages off-chip data transfers is also included.

Generally, the performance model is based on the maximum of the arithmetic computation time, data transfer time from and to off-chip memory and other cores, and load/store times. However, each performance model is different from each other since they have different characteristics in computation time, load/store time, and data transfer time.

The number of execution cycles for matrix multiplication is shown below.

$$mm = \max\left(\frac{(1+9(1-f))N^3}{vm}, \frac{\left(\max\left(1, \left(\frac{N}{\sqrt{c(1+h(m-1))/3}} - \left(\frac{N}{\sqrt{c(1+h(m-1))/3}} - 1\right)r\right)\right) + \max\left(1, \frac{N}{\sqrt{c/3}}\right) + 1\right)N^2}{w}, \frac{\left(\max\left(2, \frac{2N}{\sqrt{c/3}}\right) + 1\right)N^2(1-t)}{m}\right)$$

The first term is the number of cycles using the peak computation performance. We assumed that software floating-point computation takes ten times the hardware execution time that is reflected in the equation (1+9(1-*f*)). There are N^3 floating-point operations, and the number of execution cycles is $N^3/(vm)$.

The second term is for off-chip data transfer time. The dividend is the amount of data to be transferred from and to off-chip memory. The divisor is the off-chip bandwidth. Thus, the second term is the number of cycles to transfer data from and to off-chip memory.
The third term is the number of cycles to execute load and store operations. If there is a streaming hardware, the load/store cost is almost zero, which is reflected in the (1-*t*) term. The dividend, except the (1-*t*) term, is the number of load and store operations. Since the data is distributed to cores, the cost is divided by the number of cores.

The number of execution cycles for FFT is shown below:

$$fft = \max\left(\frac{(1+9(1-f))5N\lg N}{v(1+7p/3)}, \frac{4N}{w}, (1-t)10\frac{N\lg N}{2}\right)$$

Since each core computes its own FFT, the execution times for all cores are the same. Thus, the execution model computes the execution cycles for a core. The first term is the number of cycles using the peak computation performance. There are $5N\lg(N)$ floating-point operations. In the divisor, (1+7*p*/3) reflects the performance boost from the complex computation unit. With the complex computation unit, the performance is improved by 3.33 times.

The second term is for off-chip data transfer time. The dividend is the amount of data to be transferred from and to off-chip memory, i.e., input and output data. The divisor is off-chip bandwidth. Thus, the second term is the number of cycles to transfer data from and to off-chip memory.

The third term is the number of cycles to execute load and store operations. If there is streaming hardware, the load/store cost is almost zero, which is reflected in the (1-*t*) term. Then is the number of loads and stores in a butterfly. In a butterfly, ten data elements need to be loaded and stored, *i.e.*, real and imaginary components for two input elements, real and

imaginary components for two output data elements, and real and imaginary data elements for a twiddle factor. There are $N \lg(N)/2$ butterflies in FFT.

The number of execution cycles for object detection is shown below.

$$od = \max\left(\frac{1{,}555{,}200(1+9(1-f)N^2}{vm}, \frac{(1+(1-h)(m-1))\left(1+172800g\left(\frac{N^2}{c(1+h(m-1))}\right)\right)N^2}{w}, \frac{(1-t)1{,}382{,}400N^2}{m}\right)$$

$$g(x) = \begin{cases} 0, & xi < 1f \\ 1, & e \quad 1 \end{cases}$$

The first term is the number of cycles using the peak computation performance. There are 1,555,200 floating-point operations per input data. In the divisor, the product of v and m are used to calculate the number of cycles since the computations are distributed to cores evenly.

The second term is for off-chip data transfer time. The dividend is the amount of data to be transferred from and to off-chip memory. Note that $g(x)$ determines whether or not the whole input data fits in cache. If data does not fit in cache, the data needs to be transferred many times. The amount of the data transfer is divided by the off-chip bandwidth to provide the number of cycles to transfer data from and to off-chip memory.

The third term is the number of cycles to execute load and store operations. If there is streaming hardware, the load and store cost is almost zero, which is reflected in the (1-*t*) term. The dividend, except the (1-*t*) term is the number of load and store operations. Since the data is distributed to the cores, the cost is divided by the number of cores.

The number of execution cycles for the neural network application is shown below:

$$nn = \max\left(\frac{(1{,}064.5+L)(1+9(1-f))NL}{v}, \frac{mNL}{w}, (1-t)143NL\right)$$

Since each core computes its own neural network, the execution times for all cores are the same as in the FFT. The first term is the number of cycles using the peak computation performance. There are (1064.5+*L*)*NL* floating-point operations. The number of floating-point operations is divided by v to provide the number of cycles to compute the floating-point operations.

The second term is for off-chip data transfer time. The dividend, *mNL*, is the amount of data to be transferred from and to off-chip memory.

The third term is the number of cycles to execute load and store operations. There are 143*NL* load and store operations in each core.

3.2 Emulation Board

When a new processor is developed, the cost of manufacturing a processor is very high, and the new architecture goes though a simulation process to minimize the bugs in the design. However, the speed of the simulation is very slow, *i.e*., thousands of times or more. Therefore, there is interest in using FPGAs to emulate the processor for fast verification of the design.

We evaluated vendor-supplied evaluation boards with FPGAs. The study results are shown in Section 4.

We also proposed emulation systems for DoD, particularly for structured ASICs, patterned ASICs, and the chip generator. Although these emulation systems are targeted for chips designed using constrained tools for special-purpose architectures, the emulation systems can accommodate any processor or system. The proposed designs are shown in Section 4.

3.3 Evaluation Board

When a chip is developed, it needs to be tested on an evaluation board. In this study, we designed a high-level emulation board design. To accommodate multiple different chips, we designed a board with flexibility, which comes from using FPGAs and a flexible memory interface. FPGAs provide flexibility by providing programmable interconnections and glue logic.

We also considered configuration for different applications. Some applications need more memory while others need more I/O capability. Our high-level design can be configured differently for different applications. The high-level design is shown in Section 4.

3.4 Board Generator

When a processor is manufactured, it needs to be tested on an evaluation and development boards. Traditionally, these boards have been designed and manufactured in four steps: schematic capture, place and route (PAR), board manufacturing, and assembly. The two last steps, related to manufacture of the board, are outside the scope of this project. The first two steps are not fully automated, especially schematic capture. Instead, they are more labor intensive and are error-prone and cost additional funding and delay. Manual modification, often used to fix design errors, requires human intervention and can reduce performance. Modern complex boards have many layers and small component contacts, making manual rework difficult or impossible.

In this study, we explored the opportunity to automate the schematic capture and place and routing (PAR) steps to design a board quickly with high reliability and performance.

Figure 7 shows the overview of the approach:

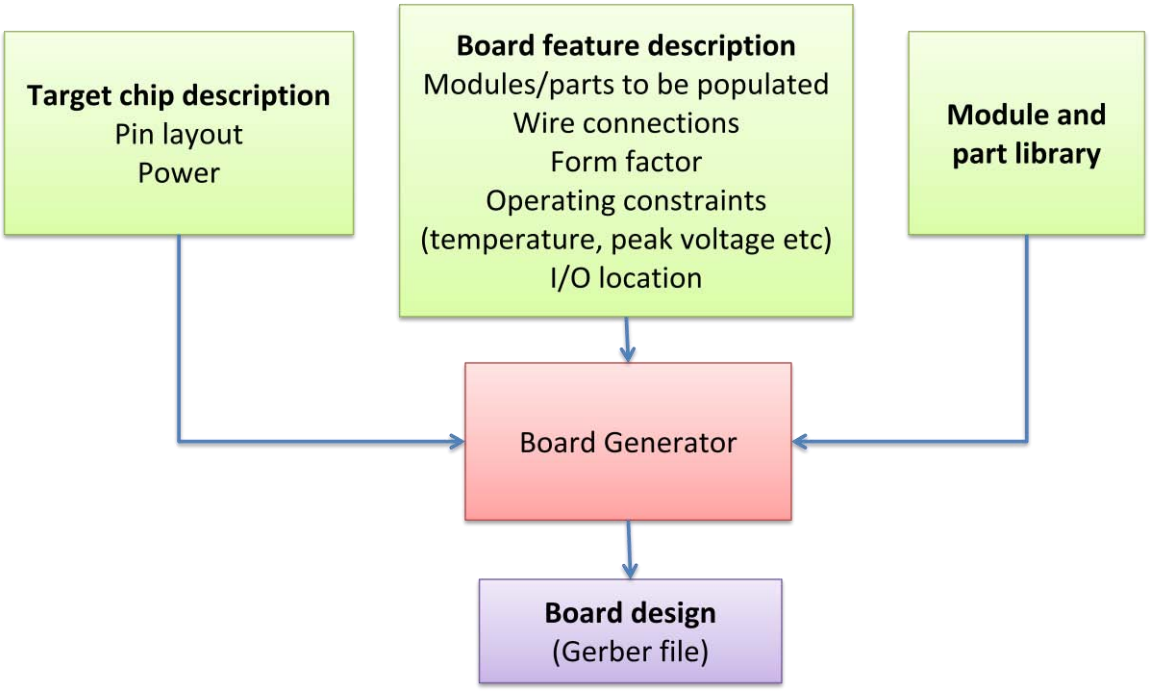


Figure 7. Overview of board generator

The board generator gets three inputs: target chip description, board feature description, and module and part library. The target chip description includes pin layout and power requirements.

The board feature description includes user input on the desired board configuration. One aspect of the board descriptions is a board form factor. It includes size, shape, and thickness of the board. The number of layers may be determined automatically. The board description also includes a list of modules and parts. For example, the description can specify one PCIe-8 and two XAUI interfaces. Another part of the description is location and orientation of modules and parts. Not all module and part locations and orientations need to be described. However, sometimes, the user wants a precise location for a particular module. For example, the user may prefer specific locations for a reset button and or USB interface for easy access.

Connections among modules and parts may need to be specified. Again, not all of them need to be described. For example, if there are two parts that need to be connected and there is no other possible way, a tool can automatically connect them. However, if there are multiple ways of connecting modules, the user must specify exactly how they are connected. Note that the user description is expected to be at a high level, and we expect the tool to determine low-level details.

Some restrictions can be specified. For example, the operating temperature can be specified if the selection of parts should consider the operating environment. Another restriction is supplied power. If the power is limited, for example, then the tool will try to use low-power modules to meet the specification.

The tool needs a part library, which is generally available from vendors. However, the target chip library and some parts may not be available. These library components must be generated manually or by auxiliary tools. The library database for components includes information such as chip layout, pin requirement, required external components and circuit diagram, connection type, and performance information.

Figure 8 shows the expected flow of the automated board generator.

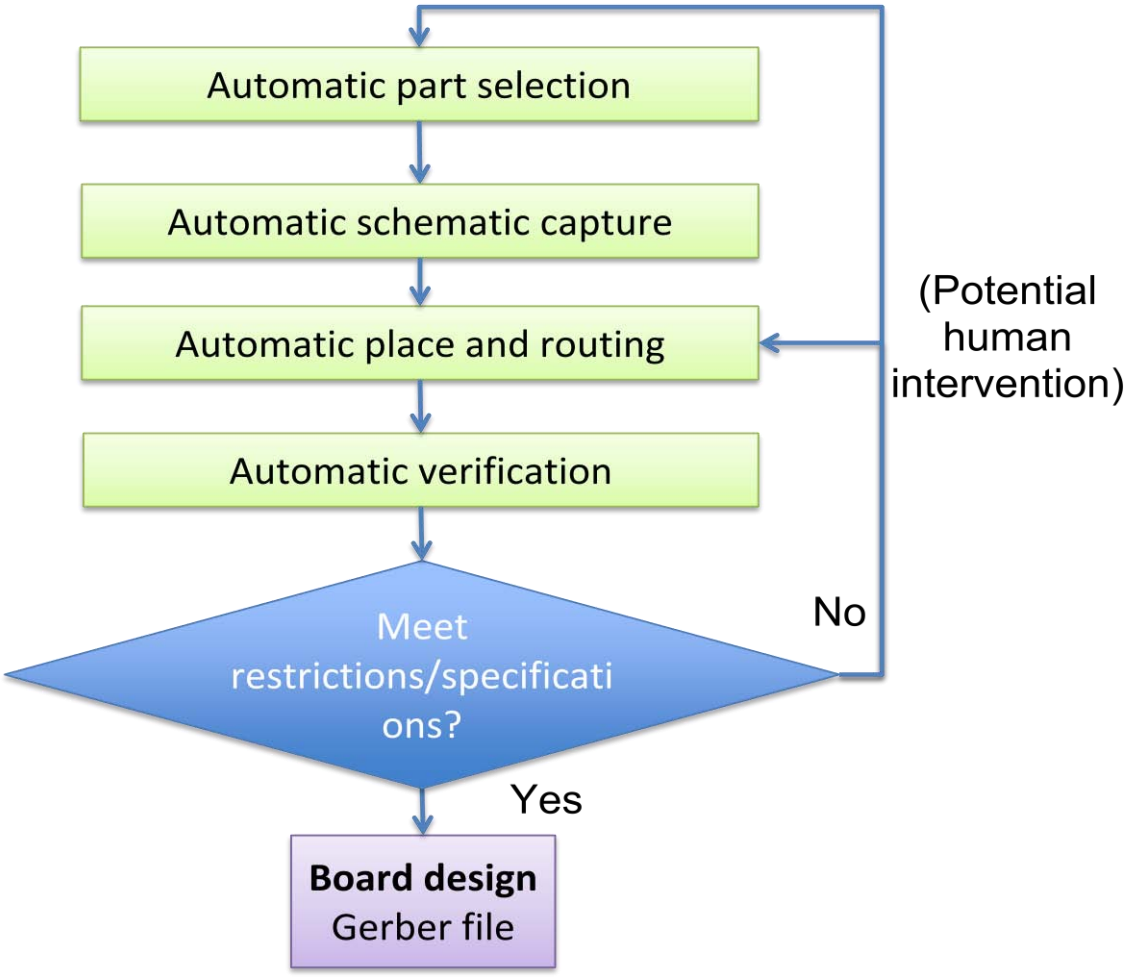


Figure 8. Board generator tool flow

There are several automatic decisions made in the tool. The modules and part selections can be made automatically. The tool searches modules and parts that fulfill the board design needs and also chooses the right combinations. The module selection needs to consider space clearance, power requirement, and performance requirement. Optionally, part cost can be considered as well.

While general automatic board design is probably not feasible, in this case, we have constrained the problem. Since we are generating designs under a set of limited circumstances

and can place limits on the interfaces used by the target chip architectures, we believe that we can come up with a parameterizable, design template. The automated design tools can use this template to generate board designs. If successful, this tool will facilitate research in special-purpose designs by reducing the accompanying board design effort and may also lead to a stand-alone tool that will be useful in its own right, reducing design time and errors in board design.

3.5 Design Patterns

When applications are developed, the programmer's experience and programming ability play big roles in the resulting code quality. However, this problem can be mitigated if new programmers have a way to leverage the methodologies of more skilled programmers. Design patterns provide an approach to facilitate this. The design pattern is a description of an approach that programmers use to learn how to write programs that are likely to map well to parallel architectures. A design pattern is described in simple terms so a less skilled programmer can used the more skilled programmer's pattern as a template. We show matrix multiplication as an example.

3.5.1 Introduction

The advent of multi-/many-core technologies has necessitated the development of new programming paradigms and tools as well as architectures. These new programming models and tools must be developed with two primary goals: achieving high performance on this new class of architectures, and providing a high level of programmability to facilitate software development for key applications.

DoD applications present a unique set of demands in both of these areas. DoD applications require high performance to support the rapid transformation of large amounts of data into usable information by a system user or analyst. Because of form-factor and power requirements, DoD systems must achieve high levels of computational efficiency as well as absolute performance. Because of the need for robustness, DoD systems require extensive system verification prior to deployment. Systems that operate autonomously will also require fault-tolerance, dynamic reconfigurability, and run-time resource management. All of these system goals must be met within the real-time throughput and latency constraints of the system.

The software and programming models for parallel systems must provide this level of performance and must also enable high programmer productivity. High programmer productivity can be achieved by providing a set of abstractions which then give the appropriate functionality and performance to the programmer while isolating him/her from the details of the architecture. DoD applications also require machine independence and portability to support upgrades in processor technologies without requiring large changes to existing software. Finally, software must be scalable to different machine sizes to support changing DoD requirements both between applications and within the same application. These software requirements have been the long-standing challenge for the parallel computing industry, one that is only exacerbated by the arrival of multi- and many-core architectures.

Recurring programming patterns in modern software development form the basis for software abstractions that can be used to improve programmability of complex architectures [35]. These same abstractions can also incur costs in performance and efficiency. Patterns in programming occur at all levels in the software development process. At the higher levels, software architecture patterns such as pipelines, service-oriented architectures, blackboard systems, etc., are often employed for their natural suitability to particular applications. At the lowest levels, tuning patterns and technology-dependent patterns such as those found in compilers and libraries are used to optimize performance on particular kernels and computational structures. Between these are patterns for concurrency such as task and data decomposition; patterns for algorithmic structures such as task parallelism, divide-and-conquer, geometric decomposition, and recursion; and patterns for parallel implementations such as SPMD, master/worker, loop parallelism, and fork/join.

At any level, there is a tradeoff between high-level programmability and architectural performance. These tradeoffs depend on a number of factors, including: the suitability of a high-level pattern for a particular application or algorithm, the suitability of a low-level pattern for a particular machine architecture, the level of generality provided by the pattern (higher levels of generality usually incur more overhead), the method of implementing the pattern (patterns implemented using machine primitives will have higher performance than those implemented in higher-level languages), and the synergy between the different types of patterns that work together to implement an algorithm or application. At any level, there will be several different patterns to choose from, each with their own performance/programmability trade-off, and each potentially affecting the trade-offs at a different level. For example, task parallelism may best suit a particular application, but if the software and hardware primitives for task parallelism are not provided in the architecture, performance may suffer. Similarly, software and hardware primitives that yield the best performance on an architecture may require complex programming and long development cycles (which has often been the case in high-performance embedded parallel programming).

This work examined how parallel design patterns can be used on multi-core architectures to solve some of the software challenges posed by DoD applications. A specific processing kernel, matrix multiplication, and representative architecture, the Tilera TILE64 [5], were selected for detailed study. The design patterns used in implementing the kernel on the architecture were identified and requirements for abstractions based on the design patterns were defined. A prototype set of portable, machine-independent abstractions based on the identified design patterns were designed. An implementation of the kernel on the architecture based on the design patterns, but parameterizeable for specific instances of the pattern, was created. Performance of the generalized version that was consistent with baseline benchmark results was demonstrated. Analysis was performed to demonstrate that the prototype abstractions contain the information needed to generate the parameterized code. Both the existence of DoD application-specific design patterns and the incorporation of DoD constraints into the existing design patterns were considered.

3.5.2 Matrix multiplication and parallel design pattern

Matrix multiplication was chosen for this study because it is widely used in DoD applications and can be implemented using a variety of parallel methods. Many different approaches were found in the body of literature on parallel matrix multiplication including Cannon's algorithm, broadcast-multiply-roll, parallel and scalable universal matrix multiplication algorithms, etc. The purpose of examining these algorithms was to determine any common elements that would be part of a general data distribution scheme; these elements would be candidates for inclusion in a generalized data distribution pattern library.

Figure 9 shows the patterns that have been defined for parallel programs [29]. The patterns that are relevant to matrix multiplication are indicated. This case study briefly mentions each of the relevant patterns and describes the context, and forces and solutions that are specific to matrix multiplication. Specific attention is paid to DoD needs for real-time performance, portability, maintainability, and high efficiency as required for embedded platforms.

Data Decomposition. Data decomposition is the simplest and most natural choice for matrix multiplication since the result matrix can be easily decomposed into chunks that are operated on relatively independently. Since matrix multiplication is computationally intensive, there is a large potential performance benefit to processing the chunks in parallel. There is a lot of flexibility to this type of distribution, since the patterns of data distribution and resulting implementations are similar for different chunk sizes, allowing us to choose chunk sizes that maximize efficiency without re-thinking the data distribution. Array-based computations like matrix multiplication are very common in DoD applications.

However, programming these computations can be challenging and complex if the programmer must explicitly manage all array indices and data communications. DoD applications would benefit strongly from having both a set of abstractions that allow the programmer to specify algorithms and data structures monolithically (*i.e.*, in the aggregate) and a set of tools that generate the correct index patterns from those aggregate data structures.

Group and Order Tasks. Each chunk of the result matrix is computed independently, but needs as input the correct portions of the two input matrices, i.e., the corresponding column block of one matrix and the corresponding row block of the other matrix. Tasks must be grouped and ordered according to the sharing of pieces of the input data. These pieces need to be moved around or shared between tasks as required by the parallel algorithm. Often, these data movements are coded by hand, in complicated programs that are tailored for a particular algorithm task decomposition and/or architecture, and are therefore not re-usable or portable. Data abstractions for DoD applications must include mechanisms to specify the overlap and sharing of data between different chunks in such a manner to be portable between different architectures and data sharing methods.

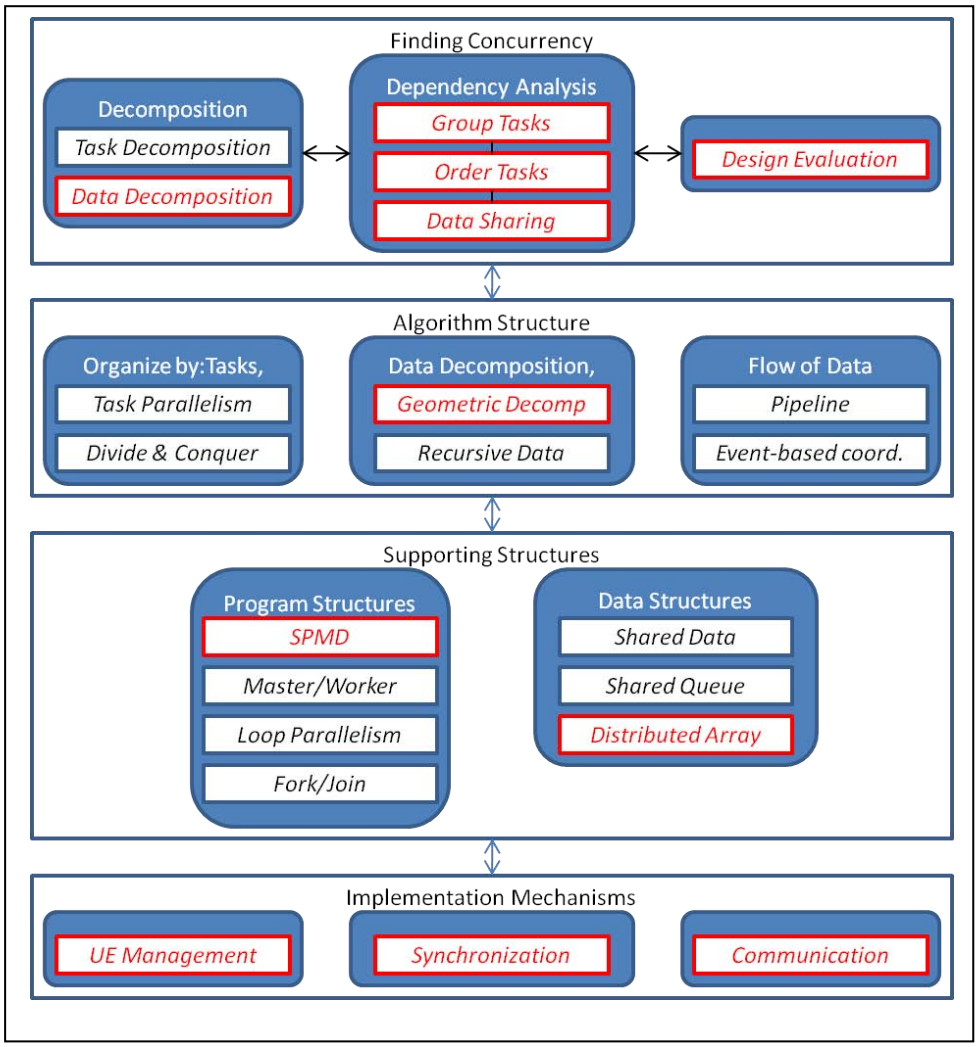


Figure 9. Parallel programming patterns:

Data Sharing. The sharing of input data between different tasks can be achieved in several ways. One possible way is to replicate and broadcast a row block of the first input matrix on all processors which need that row block, and do the same for each column block of the second input matrix. The cost of this method is an up-front aggregate data movement that requires synchronization and may preclude the benefit to efficiency that occurs with overlapped communication and computation. The benefit is simplicity. Another method is to move row and column blocks of the input matrices between processors as the computation proceeds, accumulating results into the output matrix as they are computed. This method potentially allows more overlap of computation and communication, however, is more complicated to program. Often, the more simple solution is tried first, and the second is attempted if the first does not meet real-time requirements for the application. This means that often a full implementation is required to make design trade-offs. DoD programmers would benefit from mechanisms that allow modeling of the performance effects of different data

sharing methods so that they can be compared without requiring extensive programming efforts.

Design Evaluation. For matrix multiplication, the size of each chunk can be easily determined by spreading the matrix over all processors in the system. However, if a matrix is spread over too many processors, there is a point of diminishing returns where the costs of communication and data sharing outweigh the benefits of parallelization. Further, there is an optimal chunk size relative to individual processing element cache size as well. The topology of the network in the system will also affect the number of parallel tasks and processors used for the algorithm, since different topologies will have different effects on communication performance. DoD applications would benefit from abstractions and models that are independent of the number of processors and network topology in order to provide maximal portability and simplicity.

Geometric Decomposition. For matrix multiplication, the chunks into which the input and output matrices are divided are sub-regions of a two-dimensional data object. These sub-regions are square, and therefore a simple geometric decomposition is a natural fit. The data is shared between parallel units by dimension, and then computation proceeds independently. In fact, the geometric decomposition used for matrix multiplication can be viewed as a simple case of a distributed array. DoD applications would benefit from abstractions that allow the generality of distributed arrays but provide simple mechanisms for geometric distributions as a special case (N.B. see distributed arrays).

SPMD. For matrix multiplication with data decomposition, the single-program multiple-data pattern is an obvious and natural choice, since the same operations are carried out once the data has been distributed correctly. The program will perform the data distribution and sharing based on a local identifier that uniquely specifies the execution unit for each chunk in to which the data has been partitioned. Once the local processor identifier is defined, it is used to compute an index into the shared data structure in order to have each execution unit process the correct sub-region of data. This greatly simplifies programming, but does require that the programmer perform the bookkeeping operations of processor identification and index computation, which can be architecture-specific. DoD applications would benefit from abstractions that perform these index computations automatically and independently of the algorithm, in order to ease the burden on the programmer and to provide higher levels of portability.

Distributed Arrays. Distributed arrays are key for any algorithm involving large multi-dimensional data objects. In general, sub-regions of multi-dimensional data objects can be described by a block-cyclic specification, in which we supply three parameters for each dimension of the data: offset, block, and cycle. The offset tells which piece is associated with each chunk, the block tells the size of the chunk, and the cycle tells how often to repeat that chunk on that dimension. The ability to vary these parameters allows us to vary the granularity of the decomposition, thereby allowing a choice that maximizes efficiency of the implementation. The sharing of data between parallel units could also be described dimensionally, since each output chunk needs all rows of one input and all columns of the other. Once data is shared, the computation proceeds independently in parallel. If each

execution unit processes a block of the same size, then the load will be evenly balanced between all processors. DoD applications, which contain many such operations on multi-dimensional data objects, would benefit greatly from abstractions that include block-cyclic partitions as an integral part of the semantics of the language and/or library.

Implementation Patterns: UE (Unit of Execution) management, Synchronisation, Communication. The implementation patterns used in matrix multiply depend strongly on the nature of the patterns used in the previous phases of the algorithm design. For example, the blocks into which the output matrix is divided define exactly what the tasks are, and each process, or execution unit, in the parallel processor can process one task. It remains only to map processes to processors in the system, and there, the data distribution helps us as well. Because rows and columns of the inputs must be shared in a particular way, i.e., locally on a single dimension, the mapping of tasks to processors should place blocks that use the same section of input rows to physically adjacent or local processors in the system, in order to improve communication performance and reduce overhead. On a two-dimensional mesh processor, the one-to-one mapping based on topology is a natural mapping which can be achieved using the architecture-specific structures provided, however, since this is a natural and often-used pattern, the programmer would benefit from abstractions that map mesh processor topologies to multi-dimensional data objects in a portable way. These abstractions could use processor-specific application programming interfaces underneath for performance. Similarly, synchronization and communication in the matrix multiplication algorithm involve detailed scheduling and coordination of data movement with matrix computation. Commonly used operations are barrier synchronization, row and column broadcasts, and shift-like operations where a chunk of a matrix is sent a certain number of hops along a dimension of the mesh. Programming would be greatly facilitated by the use of portable interfaces and routines for commonly-used synchronization and communication patterns.

3.5.3 Prototype abstractions

In general, pattern-based abstractions for DoD applications should provide reusability (generality), portability, isolation from implementation details, and a level of abstraction that makes programming easier without sacrificing performance. One approach toward achieving these goals is to hide the complexity of parallelism, data distribution, and machine configuration inside a re-usable library of objects and functions that implement widely-used patterns. By parameterizing these objects and functions, they can be re-used for different instantiations of the same pattern. The key parameters describe the data (i.e., data size, shape, element type) as well as the parallel machine (e.g., mesh topology, size, and shape). In addition to machine and data parameters, object and function parameters could include directives that make precise the way in which a pattern is to be used as well as processor-specific structures to support parallelism. Matrices and multi-dimensional objects would be implemented with internal state to facilitate sharing and distribution. The goal is for all processor-specific code to reside only inside the functions and objects and not in the application software, resulting in portable, high-level code that is more compact and easier to create. The challenge is in defining the abstraction barriers and parameterized structures that achieve this goal efficiently.

For example, matrix multiplication uses data parallelism, shared data, and SPMD patterns. By definition, each of these patterns has a **context** within matrix multiplication, **forces** that act on the pattern within that context, and **solution** that mitigates those forces. Figure 10 shows these patterns and summarizes the context, forces, and solution for each pattern. For matrix multiplication, the data is decomposed along two dimensions in a block pattern, where the extent of each block on each dimension depends on the size of the original matrix and the number of processors over which that dimension is mapped. The regularity of the computation is such that the same computation is carried out regardless of the block size. For example, processing a block of output requires a matrix multiplication of a row block of one input and a column block of the other input. The size of each block affects only the length of a dimension, or in implementation terms, the number of iterations in a loop over that dimension. Parameterization of a loop bounds is the easy, first step in creating pattern-like code that works for a particular decomposition, irrespective of block size. The challenge with matrix multiplication, and with many matrix algorithms, is the complexity of the data sharing, i.e., making sure that each processor has access to the necessary rows and columns of the inputs. This part of the code is often heavily tailored to a particular way of sharing data, and is difficult to parameterize, abstract, or re-use. Further, implementations are usually tailored to shared memory or message passing.

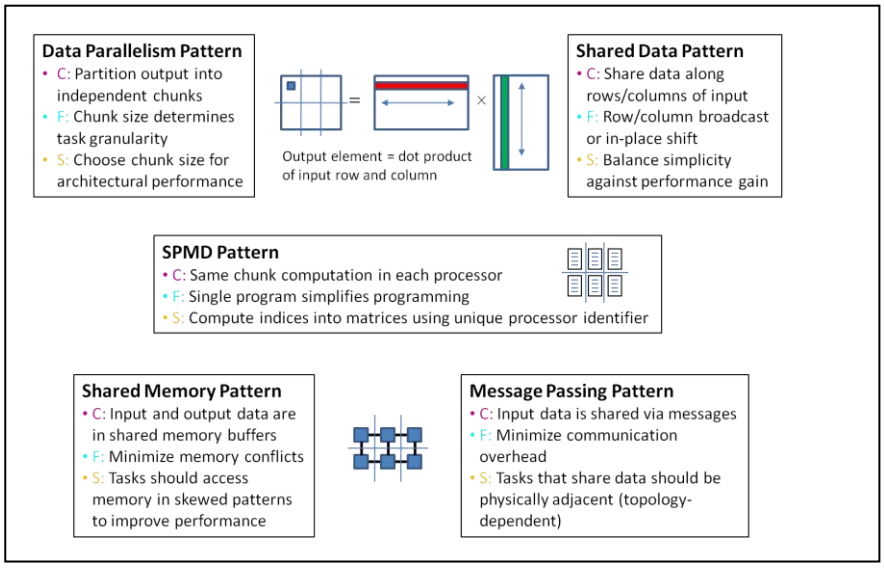


Figure 10. Patterns for matrix multiplication

If the data is in shared memory, we compute the right pointer based on the number of processors, matrix size, and processor identification number. If the blocks are shared via message passing, then we must make sure we receive the shared pieces either all at once in an up-front broad-cast, or in the course of the computation as needed (or some combination thereof). If the output data has not been allocated, then we must allocate buffer space either in shared memory or on the local heap for subsequent re-combination. Whether the implementation is shared memory or message passing, the reusable patterns in the code can be abstracted into a library of tools for the application programmer. In a pattern-based approach,

the implementation details would be hidden at the highest level, and the user is just concerned with specifying what data is moved rather than how it is moved. Additionally, at the highest level, the code should be independent of the execution units, i.e., the number of processors and their topology. Abstractions for parallelism should parameterize the level of parallelism. At the highest level, the code should be independent of the assignment of tasks to execution units.

In the next several paragraphs, we start with a sample high level implementation and then walk through several code transformations that use patterns to achieve a final, parallelized, specific implementation. At each level, we define what information is used by each pattern and how that information would be captured in an object or function parameter. The final result is a template that will translate directly into a specific implementation on the target architecture.

At the highest level, we start with the simplest code: the matrix objects are defined and a matrix multiply function (or method) is called for those objects. Information about the data is stored inside the object and used by the methods. The programmer would invoke the multiplication method as follows:

```
MatrixIntObject A(100,300), B(300,500), C(100,500);
MatrixIntMultiply(A, B, C);
```

The sequential code executed by the multiply method would look as follows:

```
if ((NumberCols(A) != NumRows(B)) ||
    (NumberRows(A) != NumberRows(C)) ||
    (NumberCols(B) != NumberCols(C))
   exit with error
for(i=0; i<NumberRows(C); i++)
   for(j=0; j<NumberCols(C); j++)
      C[i][j] = 0;
for(i=0; i<NumberRows(C); i++)
    for(j=0; j<NumberCols(C); j++)
      for(k=0; k<NumberCols(A); k++)
          C[i][j] += A[i][k] * B[k][j];
```

The first pattern used in the parallel version of matrix multiplication is data parallelism. The data parallelism pattern denotes that the data is mapped over multiple processors and that the matrix multiply task uses that mapping to organize itself into tasks. Abstractions for data parallelism would need two classes of information related to data mapping: firstly, how to divide the matrix into chunks, and secondly, how to assign those chunks to processors in the machine. This information would be supplied in parameters we call here “directives”. In addition to the mapping information, a high-level specification of the parallel machine would be supplied. Ideally, the abstraction barriers between the mapping information and the parallel machine description would allow us to change the mapping independently of the machine, i.e., for a different size machine with the same matrix partitioning, we would change just the machine description and not the directives. Similarly, for a different mapping on the same machine, we would just supply the corresponding directives. Methods for matrix objects would be supplied that would map the data according to the directives and the machine and each matrix object would contain its own internal state denoting its current mapping to processor. The matrix multiplication code would use the information in the matrix object to

organize itself into tasks. A data parallel version of matrix multiplication might look something like this at the highest level:

```
DecomposeIntMatrix A([100 300], Directives, Machine);
DecomposeIntMatrix B([300 500], Directives, Machine);
DecomposeIntMatrix C([100 500], Directives, Machine);
MatrixIntMultiplyDataParallel(A, B, C, Directives, Machine);
```

In this code fragment, the objects *A*, *B*, and *C* are instantiated with their sizes and with other parameters that specify their layout on a parallel machine. The data parallel matrix multiplication task uses the information contained inside the objects to execute the computation. Since the methods for decomposing data and performing the matrix multiplication are parameterized in machine and data size, the code is scalable and portable to other machine configurations.

Inside of data parallelism, other patterns used in matrix multiplication are geometric decomposition, SPMD, and shared data. In our implementation, the geometric decomposition pattern specifies that the matrix is divided into blocks on each dimension and that these blocks are mapped to the corresponding dimensions of an abstract 2-D mesh architecture. The SPMD patterns generate the unique processor identifiers and program components to perform the matrix multiplication task with the given data decomposition. The shared data pattern specifies how the data is shared in each dimension within the main computation loop. A transformed code using pattern abstractions might look something like this:

```
LocalID = UE_CREATE(Machine);        /* Set the unique processor ID using an */
                                     /* abstract parallel machine description */

MyA = GEOMETRIC(A, LocalID, Directives, Machine);      /* Specify geometric */
MyB = GEOMETRIC(B, LocalID, Directives, Machine);      /* distributions for */
MyC = GEOMETRIC(C, LocalID, Directives, Machine);      /* A, B, and C */

DATA_SHARE(MyA);                /* Perform an initial sharing of data (skew) */
DATA_SHARE(MyB);                /* skew definition is inside the object */
While (NOT-DONE(MyC)) {         /* loop over all blocks */
  DATA_SHARE(MyA);              /* get next block of input A */
  DATA_SHARE(MyB);              /* get next block of input B */
                                /* objects know how to access their next block
*/
  MatrixIntMultiplyBlock(MyA, MyB, MyC);     /* perform the block multiply */
                                             /* using the correct loop bounds */
                                             /* and block sizes */
}

MatrixIntMultiplyBlock(MyA, MyB, MyC);       /* Perform final block multiply */
DATA-SHARE(MyC);                             /* put current block of output C */
```

After this level of code, a specific implementation could use either message passing or shared memory patterns. In the message passing model, the matrix objects are physically distributed over the multiple processors. In this case, the directives specify where the different blocks reside in the processor array and the data sharing abstractions implement the message passing required to get the right data to the right place for the inner computations. In the shared memory model, the matrix objects reside in shared memory accessible to all processors. In this case, the directives and the local identifiers are used to compute the indices into the matrix in shared memory and the data sharing abstractions compute the next set of indices

inside the computation loop. The specific choice of implementation should be maintained inside the pattern abstractions to simplify programming and allow the programmer to focus on the algorithm rather than the implementation details. This would also enable design evaluation and performance optimization, since the programmer can use the same block of code to try both shared memory and message passing to find the higher-performing implementation.

The final transformation would produce the specific C code to implement the parallel matrix multiply for either shared memory or message passing. In a shared memory implementation, the geometric pattern abstractions would define the block size from the properties of the matrix object, the mapping directives, and the machine description. The SPMD pattern abstractions would define the local variables to index processors in both dimensions and find the local piece for each matrix in shared memory by calculating a pointer form the processor identifier. The shared data abstractions would get the next shared piece in each iteration and the matrix multiply loops would be implemented using the pointers and indices computed in the earlier patterns. With the shared memory implementation, the blocks are located in different regions of memory, and the code staggers the accesses to improve performance. This is based on Cannon's algorithm, which specifies a template that can be used for both shared memory and message passing. The final matrix multiplication code consists of simple memory transfers and multiply-accumulate loops. Example code for computing the indices and allocating local buffers is shown below; this code assumes that the number of processors is a known parameter and that each matrix object has information about its own size:

```
BlockRows = NumberRows(C)/ProcY;  /* Output rows mapped to processor Y */
BlockCols = NumberCols(C)/ProcX;  /* Output columns mapped to processor Y*/
BlockInner = NumberCols(A)/ProcX; /* Inner dimension mapped to processor X */

rank_X = rank%ProcX;              /* Each processor's unique identifier */
rank_Y = rank/ProcX;

MyRow = rank_Y * BlockRows;       /* Each processor's indices */
MyCol = rank_X * BlockCols;
MyBlockX = rank_X * BlockInner;
MyBlockY = rank_Y * BlockInner;
                                  /* Local buffers for matrix blocks */
local_A = (int *)malloc(BlockRowsY * BlockInnerX * sizeof(int));
local_B = (int *)malloc(BlockInnerX * BlockColsX * sizeof(int));
local_C = (int *)malloc(BlockRowsY * BlockColsX * sizeof(int));
```

Inside the computation loop, the indices into the input matrices are computed as follows:

```
for (shift_by=1; shift_by<ProcX; shift_by++) {
       CurrInner = (rank_X + rank_Y + shift_by) % ProcX;
       A_start = MyRowY * InnerDim + BlockInnerX * CurrInner;
       B_start = CurrInner * BlockInnerX * OutCols + MyColX;
       …
```

Since the abstractions described in this section are all parameterized in machine size and array size, the fundamental and most important requirement for pattern-based matrix multiplication is that the code be generalized, *i.e.*, execute correctly for any machine size and array size. In code that has been tailored for particular cases, this generalization can be difficult to achieve. The code above works correctly only for matrix sizes that are a multiple of the processor array size. However, since this code was developed in a top-down process using patterns, the

generalization to non-uniform data and machine sizes can be easily accomplished using the established code framework.

In a message passing implementation, the geometric pattern abstractions would define the block size from the properties of the matrix object, the mapping directives, and the machine description in the same manner. In this case, the SPMD pattern abstractions would use that information to allocate message buffers and to define processor indices for communication patterns. The actual computations for indices and buffer sizes are exactly the same as the code fragment for the shared memory case. The share-data abstractions would send and receive the local blocks according to the communication patterns for matrix multiplication and the matrix multiply loops would be implemented directly on the message buffers. With the message passing implementation, an initial, up-front data movement is performed to stagger the use of the blocks in the data decomposition; subsequent data movements are between nearest neighbors in a shift pattern that implements Cannon's algorithm. The final matrix multiplication code consists of successive nearest neighbor message passing and multiply-accumulate operations to update the local output block. Because of the need to allocate buffers and fix message sizes, the message passing version of Cannon's algorithm is difficult to generalize to non-square tile arrays and to matrix sizes that are not a multiple of the tile array. Assuming the correct data proportions, the code that could be generated has very similar patterns to that of shared memory, however, the patterns perform different tasks.

3.6 Domain-specific Languages

Domain-specific languages (DSLs) have been used widely in various domains. For example, SQL is a language for handling databases, and MATLAB is a langugate for matrix operations. They are designed specifically for the target domain, and domain experts use them very efficiently.

Despite many advantages of DSLs, some areas lack suitable DSLs that can be used for deployed applications. For example, signal processing applications are being used heavily in DoD platforms. However, there are not widely used DSLs for the domain, and programmers are usually using general-purpose languages such as C. Therefore, a tremendous amount of programming effort is used to provide low level constructs to implement signal processing applications. Note that although MATLAB can be considered a signal processing language, it lacks some important language features that would make it an ideal DSL for signal processing applications.

In this study, we propose an example strawman DSL for signal processing and show the benefits of having a DSL for the signal processing domain. The study results are shown in the result section.

3.6.1 Advantages and Disadvantages

There are several advantages of using a domain-specific language [40].

- It can express a problem at the level of abstraction of the problem domain. Therefore, domain experts can express the problem using the right abstraction for the job. Without a DSL, programmers have to program at a level lower than the problem abstraction and this leads to significant overhead.
- Self-documenting code. Since the code is at the right level of abstraction, the code is very similar to the documentation. Thus, code can serve as documentation without many modifications.
- Improved productivity, quality, reliability, maintainability, portability, and reusability.
- Higher performance. Since the coding is done in the problem domain, the compiler can understand the algorithm itself and the context better than a general-purpose language compiler.

However, there are a few disadvantages of DSLs [40]. One of them is the cost of learning a new language. Note that the cost of learning a DSL by a domain expert is usually not very high since the language is closely coupled with the domain. Another disadvantage is the cost of designing, implementing, and maintaining the language and tools. This cost can be amortized across application deployments, so, for many domains, the benefit of using a DSL is higher than the cost of language development.

3.6.2 Proposed DSL

In this section, a proposed strawman DSL for the signal processing domain is described. The full language definition, compiler, and library are beyond the scope of this project. However, the description of the language shows advantages of using the DSL for DoD signal processing applications.

3.6.2.1 Declaration

- complex double x[1024], y[1K];
  // Declare two complex arrays x and y. As shown for y, the user can use K, M, G, for common power of 2 abbreviations.
- Complex single a;
  // Declare an array. However, no memory is allocated until it is actually needed

3.6.2.2 Dynamic memory allocation

- a = [1, …, 1K];
  // Allocate 1024 elements to array a. The array is initialized with a sequence of data starting from 1 to 1024.
- c = a * b;
  // Allocate an array c and the contents are initialized with the product of two arrays a and b

3.6.2.3 FFT operations

- y = fft on m;
  // Perform FFT on input data m and the results are written in array y
- y = radix-4 fft on m;
  // Perform radix-4 FFT
- y = fft on m with fft-sizeof 1K
  // Perform 1 K FFT on m. If size of m is larger than 1 K, the input data is partitioned into segments of 1 K. One FFT is performed for each 1 K segment of m. If the size of the last segment is less than 1K, zero padding is done.
- y = fft on m with fft-sizeof 1K, input-overlapping 16, output-overlapping 8
  // Similar to the previous example except that the segments of data have overlapping data. The number of elements overlapped in the input data is 16 and that in output data is 8.
- IFFT is similar to the FFT. Use ifft instead of fft.

3.6.2.4 Input data manipulation

- fft on m with fft-sizeof 2K, input-sizeof 1K padded(1K, 0);
  // Partition input data into segments of 1K. Each segment is padded with 1K zero valuesa. Then, perform a 2K FFT on each segment

3.6.2.5 Multiplication

- *
  // Element-wise multiplication. If one array is shorter than the other, the shorter one repeats until the larger one is consumed.

3.6.2.6 Concatenation

- [a b] // horizontal concatenation
- [a; b] // vertical concatenation
- a .= b // same as a = [a b]

3.6.2.7 Complex component manipulation

- real a // real part of array a
- imag a // imaginary part of array a
- complex (a, b) // make complex using a as real and b as imaginary

3.6.2.8 Transpose

- Transpose a // transpose a matrix

3.6.2.9 Change array dimension

- Reshape (a, [3 6]) // change array dimension

3.6.2.10 Maximum

- max a // find maximum value in array a

3.6.2.11 Circular shift

- Circshift (a, shift_amount) // circular shift of array a with the shift amount of shift_amount

3.6.2.12 Compute signal power

- Signal-power() // computes power of signal

3.6.2.13 Iteration

- for x=start:step:end // repeats from start to end with step value of *step*

3.6.2.14 Data access

- a(index) // index data in array a. Index starts from zero.
- a(start:step:end) // data in array a is accessed from start to end with step value of *step*. For example, a(1,2,6) means a(1), a(3), and a(5)
- a[[size]] // access data in blocks of *size* elements. If the last block is not a multiple of *size*, zero padding is automatically performed. For example, if array size of a is 10, a[[4]] means a(0:3), a(4, 7), and [a(8:9) 0 0].
- b[[size]][[size2]] // similar to a[[size]] but with two dimension

3.6.2.15 Multi-core constraints

- Compute using max 20 cores; // The code cannot use more than 20 cores

3.6.2.16 Real-time constraints

Compute in 10 ms latency, 5 ms throughput // Latency of 10 msec and throughput of 5 ms

3.6.3 Related Languages

There are a few related languages with the proposed DSL. These are discussed in this section.

3.6.3.1 SPIRAL [34]

SPIRAL was developed at Carnegie Mellon University and is under active development. It is a library generator for linear transforms. It supports libraries for scalar, vector, and parallel machines. It can support new architectures by regenerating a retuned library. SPIRAL has its own language, Signal Processing Language (SPL). SPL describes a problem in mathematical form. For example, FFT can be described as follows.

$$DFT_n \rightarrow \left(DFT_2 \otimes I_2\right) T_2^4 \left(I_2 \otimes DFT_2\right) L_2^n$$

Another example description for matrix multiplication is

$$C = AB$$

SPIRAL reports excellent performance. For example, they achieved up to about 1.8x speedup relative to FFTW for an FFT size of 215 [7].

However, the SPL is not a purely mathematical language. A user needs to give a more detailed description tailored to transformation generation using a few primitives including recursiveness, permutation, and tensor products. The tensor products are not easy to handle and not intuitive for some new users. The FFT description is provided by the SPIRAL group, and the user can use it as is. However, even understanding the representation is not straightforward to most new users.

Although SPIRAL demonstrated good performance on many signal processing problems, the SPL can have a steep learning curve for new users, and may be more appropriate for expert library developers.

3.6.3.2 Scala [31]

Scala is a functional object-oriented pattern matching language. One of the biggest advantages of Scala is that it can use Java libraries as is. Therefore, a large collection of the Java libraries is available to be used in Scala right away.

It also provides features for DSL extensions. One feature allows any method to be used as an infix or postfix operator. For example, the following code utilizes this feature to extend the language.

```
1 class MyBool(x: Boolean) {
2       def and(that: MyBool): MyBool = if (x) that else this
3       def or(that: MyBool): MyBool = if (x) this else that
4       def negate: MyBool = new MyBool(!x)
5 }
6 def not(x: MyBool) = x negate;
7 def xor(x: MyBool, y: MyBool) = (x or y) and not(x and y)
```

The code extends three operators: “and”, “or”, and “negate”. In line 6, “negate” is postfix and in line 7, the operators are infix. Compare the usage in conventional object-oriented style shown below:

```
1 def xor(x: MyBool, y: MyBool) = x.or(y).and(x.and(y).negate)
```

The conventional code is hard to read and understand. However, the Scala code is very clean and the extended operators look just like a conventional operator.

Scala’s flexibility makes it an interesting starting point for DSLs; the efficiency of languages implemented with Scala will determine its success, and this is a potential topic for future research.

3.6.3.3 Erlang [4]

Erlang is a pattern matching and functional language for fault tolerant parallel processing. However, the language does not provide any special language constructs for specific domains. Also, at this time, the performance from the language is not very high.

3.6.3.4 Sisal [20]

Sisal is a single-assignment functional parallel programming language. However, the language is not being actively developed and it does not provide signal processing domain-specific constructs.

3.6.3.5 LISP [3]

LISP is derived from "LISt Processing" and is widely used in the artificial intelligence field. It is an extendable language. However, it has a fixed S-expression syntax that is in parenthesized list form. An example is that to represent an addition, it uses (+ 1 2 3), where the first "+" indicates an addition operation. The rest of the items are arguments to the addition. Even with extensions, the new form is still in S-expression format, which limits its flexibility.

4 Results and Discussions

In this section, we discuss the results from the multi-core domain-specific architecture, emulation system, evaluation board, design pattern, and domain-specific language studies.

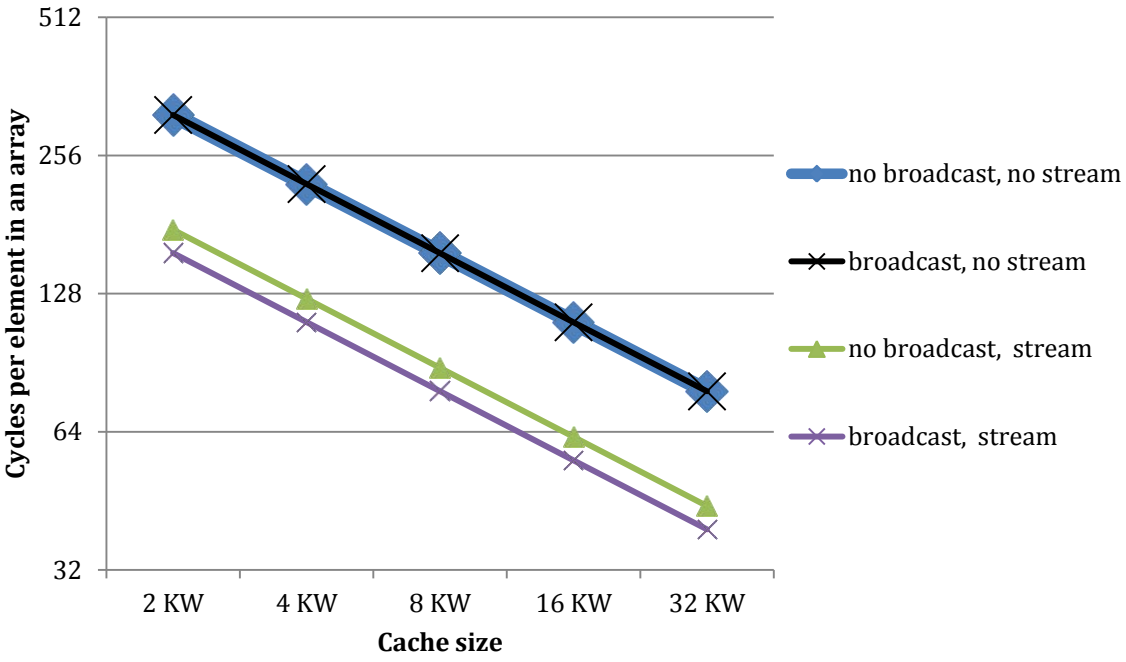
4.1 Multi-core Domain-Specific Architectures

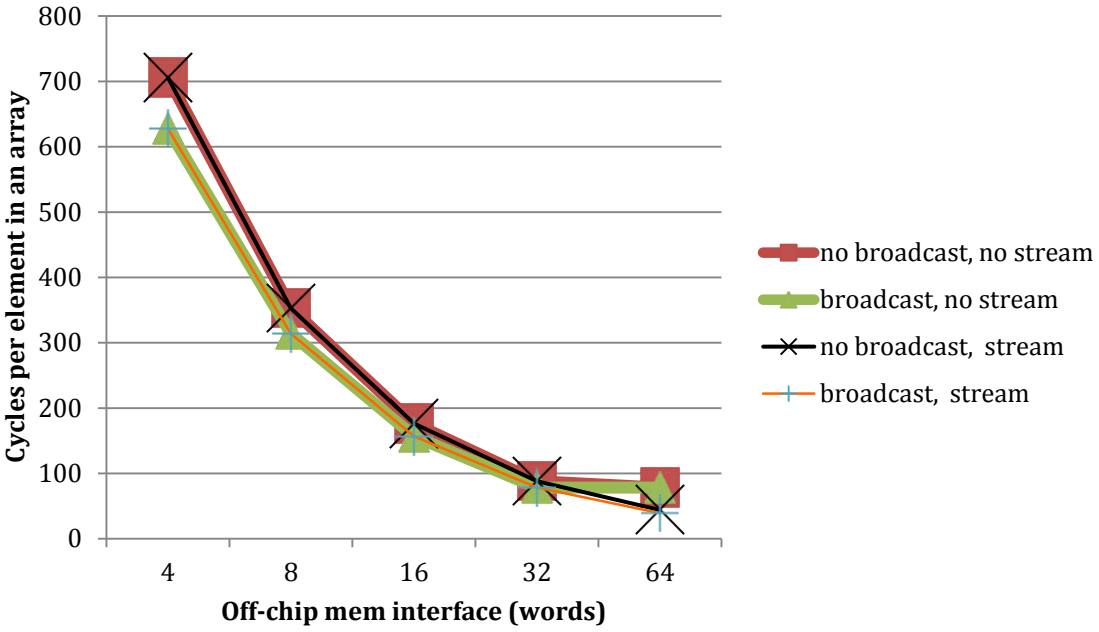
In this section, we present the results of applying our models presented in Section 3.1 and analyze results. In our experiments, we assumed 65 nm technology running at 1 GHz with a total silicon area per chip of 130 mm^2. The chip area was estimated from the footprint of the Texas Instrument TMS320C6474 [37], whose core chip area is similar to that of the 120-mm^2 DSP chip shown in [1].

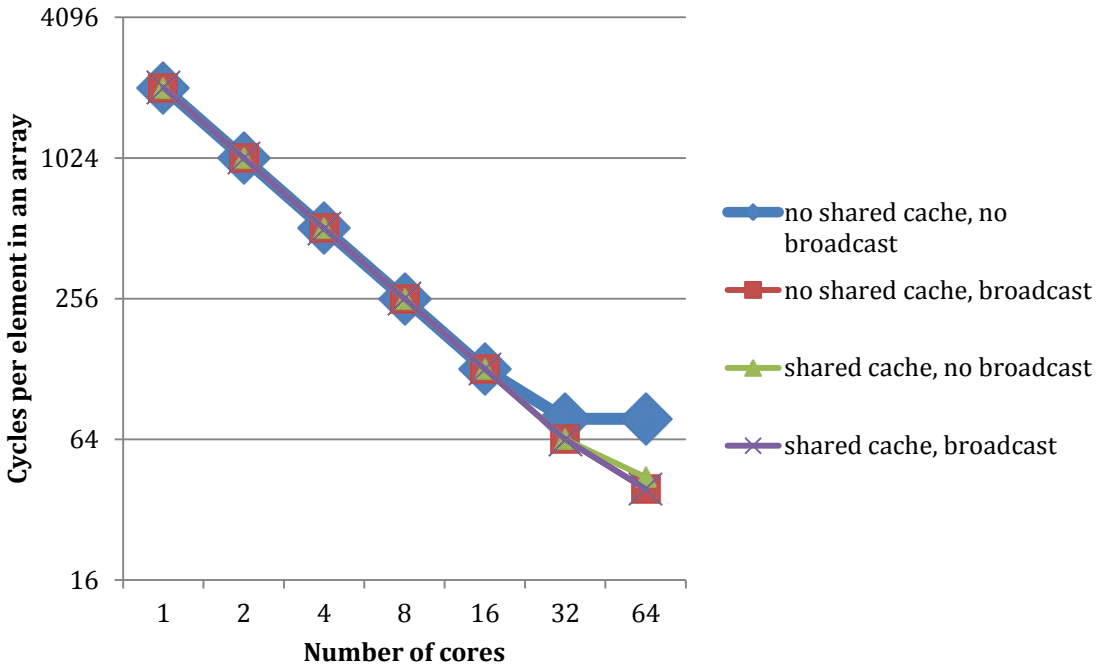
Each component's area was obtained using the InCyte [11] tool from Cadence. The InCyte tool used 130 nm technology, and we extrapolated the area to 65 nm technology by dividing it by four.

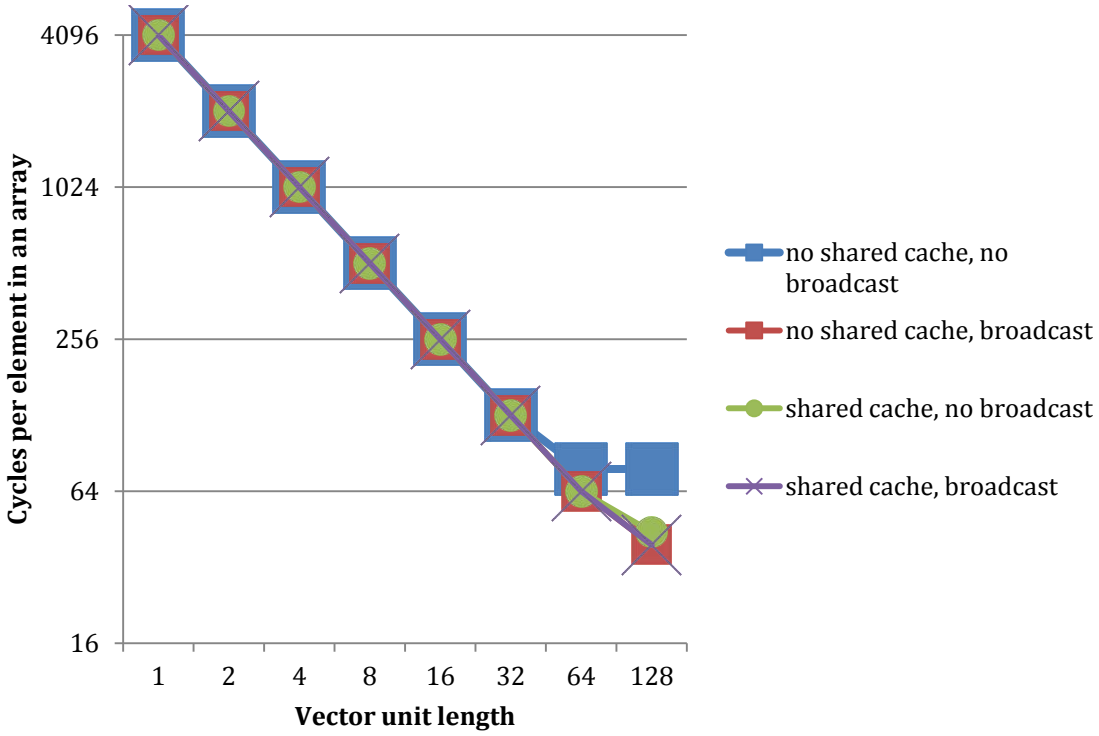
Although our models can calculate the area and performance for any input configurations, the whole design space is too large to explore in a reasonable amount of time. Therefore, in our experiments, we fixed the value of each parameter to a power of two. From the "reduced" parameter space, we use exhaustive search to choose the parameter settings that achieve the best performance under the area constraints. Then, to see the effect of a parameter on the performance, we varied one of the parameters while the others remained fixed and used the models to calculate the corresponding performance. If some parameters do not affect the resulting performance, they are not shown here.

Figure 11 shows the output of the model for matrix multiplication. Each matrix size is 256K by 256K. The graphs show the number of cycles per element in a matrix. Performance depends on hardware support for: broadcast, streaming cache size, off-chip memory interface, vector size, and number of cores. Broadcasting, large cache size, and a large off-chip memory interface reduce the data transfer cost. A large cache size increases the block size used in the matrix multiplication, which results in a reduced number of data transfers per computation. Streaming hardware reduces the load/store cost while an increased number of cores and vector size reduces the computation cost. However, on-chip bandwidth does not change the performance since there is no inter-core communications in matrix multiplication.


512
256
128
64
32
Cycles per element in an array
2 KW
4 KW
8 KW
16 KW
32 KW
Cache size
no broadcast, no stream
broadcast, no stream
no broadcast, stream
broadcast, stream


800
700
600
500
400
300
200
100
0
Cycles per element in an array
4
8
16
32
64
Off-chip mem interface (words)
no broadcast, no stream
broadcast, no stream
no broadcast, stream
broadcast, stream


4096
1024
256
64
16
Cycles per element in an array
1
2
4
8
16
32
64
Number of cores
no shared cache, no broadcast
no shared cache, broadcast
shared cache, no broadcast
shared cache, broadcast


4096
1024
256
64
16
Cycles per element in an array
1
2
4
8
16
32
64
128
Vector unit length
no shared cache, no broadcast
no shared cache, broadcast
shared cache, no broadcast
shared cache, broadcast

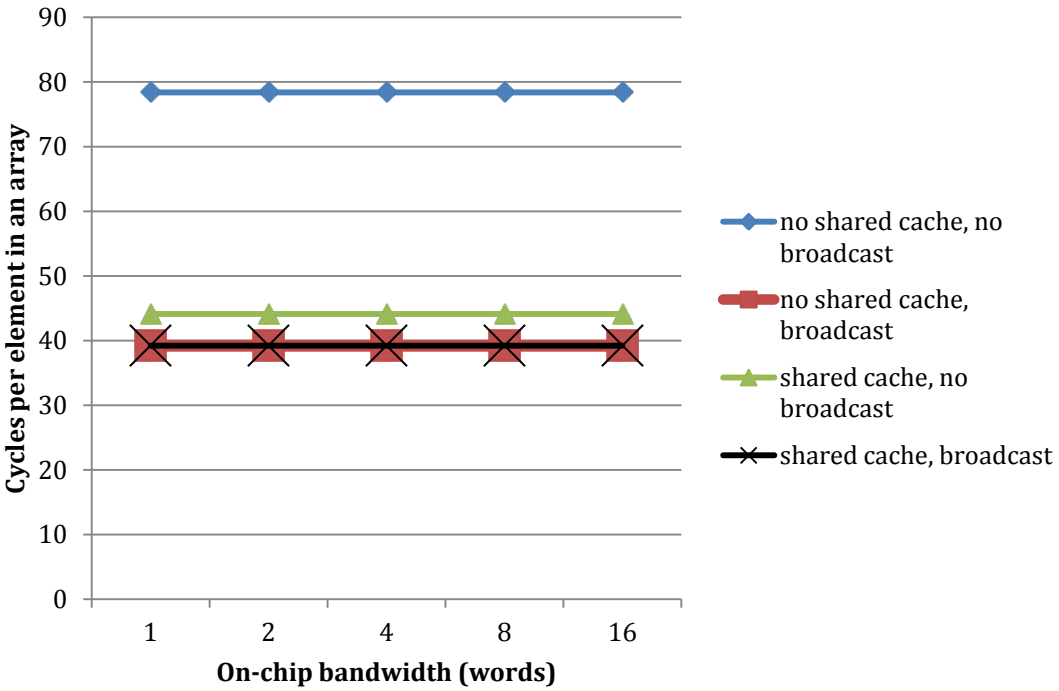
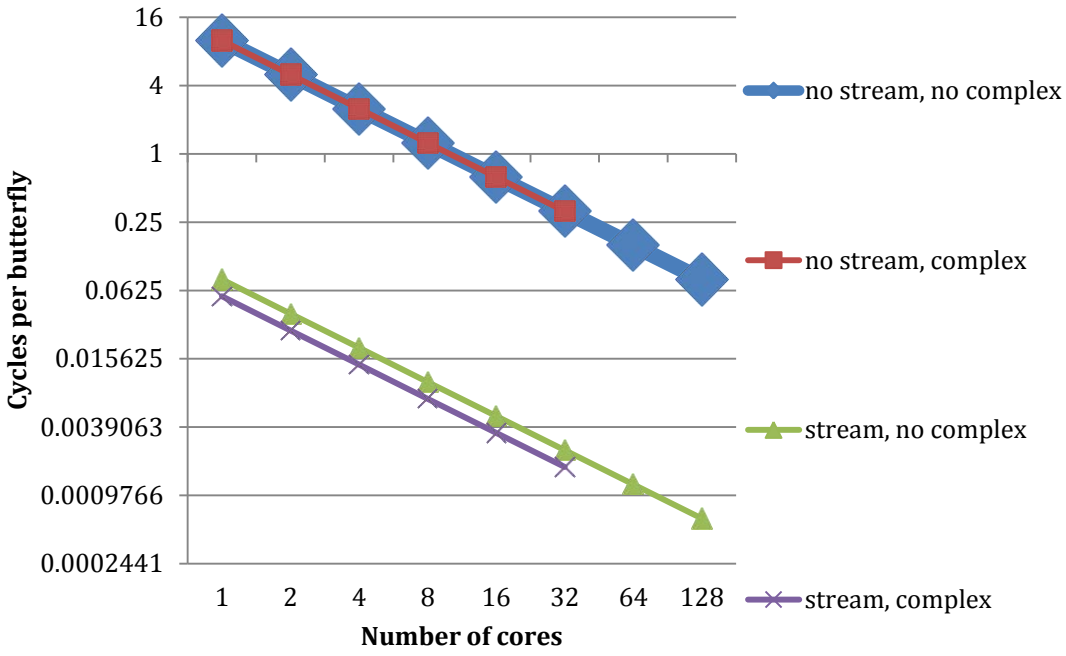
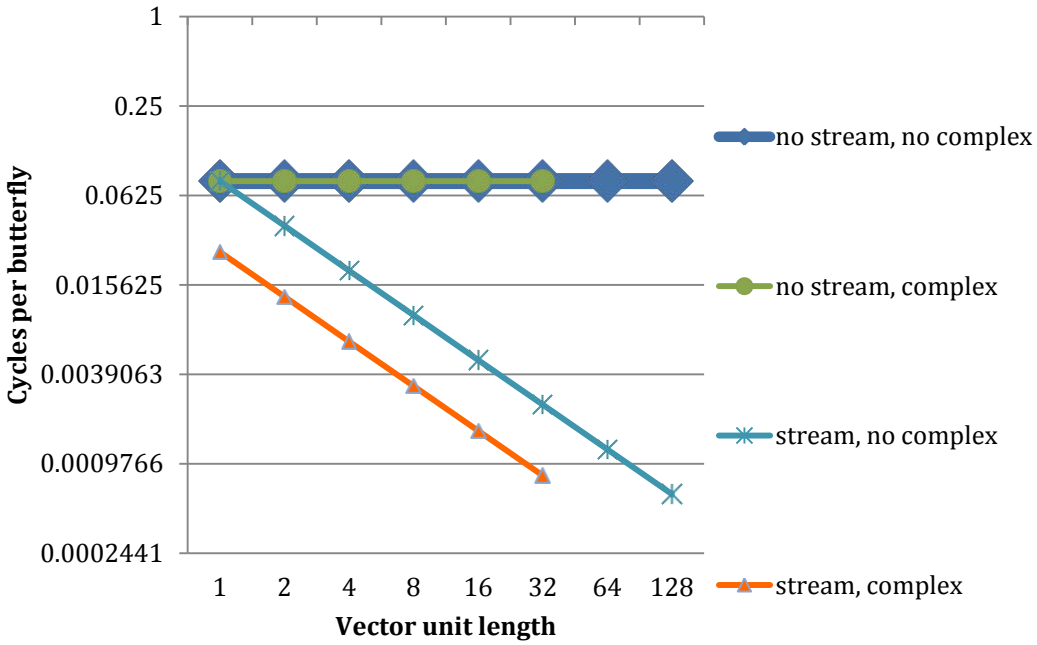


Figure 11. Matrix multiplication results

Figure 12 shows FFT performance. The data size of the FFT is 256K complex floating-point data elements. Each tile computes an independent FFT. The graphs show that the performance depends on number of cores, vector length, and data streaming. Since FFT is a computation-intensive kernel, the performance depends on the computation capacity, which is proportional to the number of cores multiplied by the vector length. The performance also depends on the streaming capability since the load and store operations can be a bottleneck.


16
4
1
0.25
0.0625
0.015625
0.0039063
0.0009766
0.0002441
Cycles per butterfly
1
2
4
8
16
32
64
128
Number of cores
no stream, no complex
no stream, complex
stream, no complex
stream, complex


1
0.25
0.0625
0.015625
0.0039063
0.0009766
0.0002441
Cycles per butterfly
1
2
4
8
16
32
64
128
Vector unit length
no stream, no complex
no stream, complex
stream, no complex
stream, complex

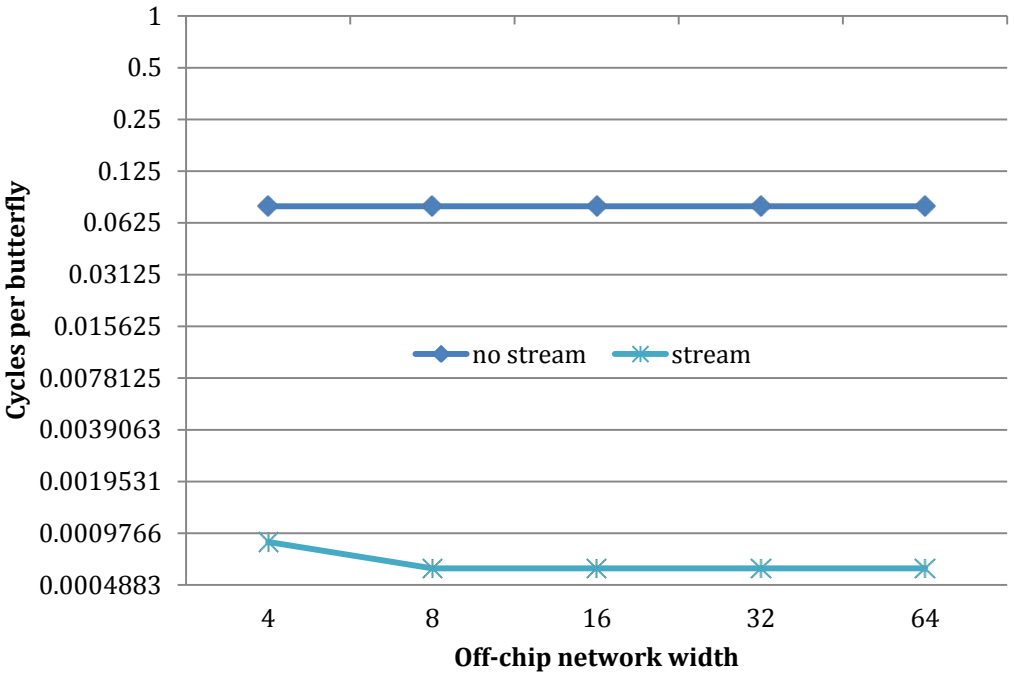


Figure 12. FFT results

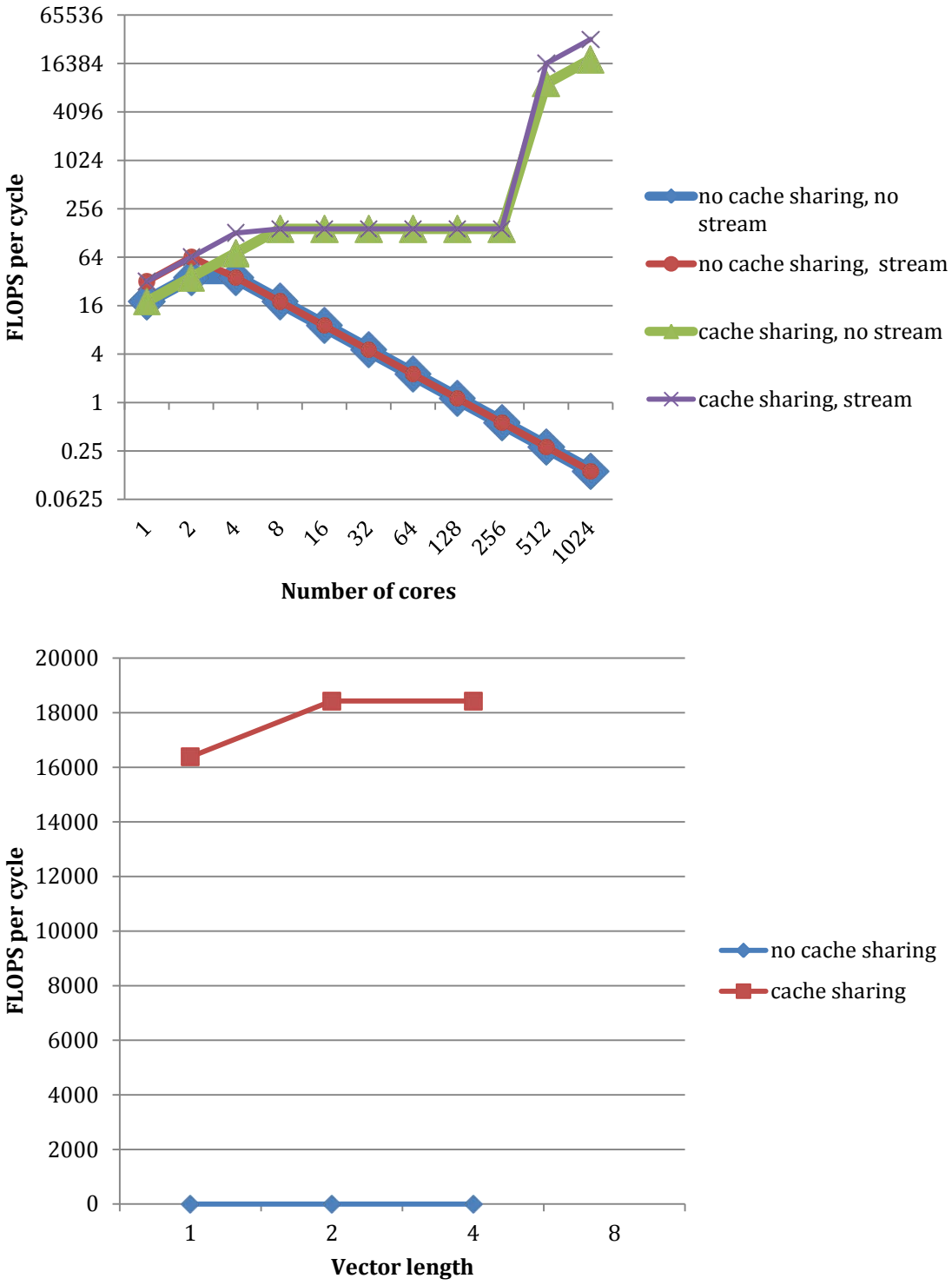


Figure 13. Object detection results

The object detection results are shown in Figure 13. The input image size is 512 by 512. One image is distributed to all cores and each core applies an independent filter, exploiting task parallelism.

The results show that the performance depends on number of cores, vector length, and cache sharing. The performance as a function of the number of cores shows performance affected by different bottlenecks. When the number of cores is one or two, the performance is determined by the product of the number of cores and vector size.

However, when the number of cores is between 4 and 256, performance is determined by off-chip data transfer, since many cores need to bring in a large amount of data at the same time. Also, the data needs to be loaded into the cache many times since the data size is larger than the aggregate cache size.

When the number of cores is 512 and 1024, if the data in cache is shared, the whole input data set can be held in the cache, and the input data needs to be loaded only once at the beginning. Then, the data transfer time is not a bottleneck anymore and performance is determined by the time to execute the load and store instructions. Since the input data is distributed to cores, the larger number of cores means that the smaller load and store time results in higher performance. The stream hardware improves performance. In the case of non-cache-sharing, the off-chip data transfer is still a bottleneck and the performance decreases as the number of cores increases.

The graph as a function of vector length shows that the performance increases when the vector length increases from one to two, but stays the same when the vector size is four. The vector size of four does not help in this case since performance is now limited by the load and store execution time. It also shows that the cache sharing helps the performance significantly.

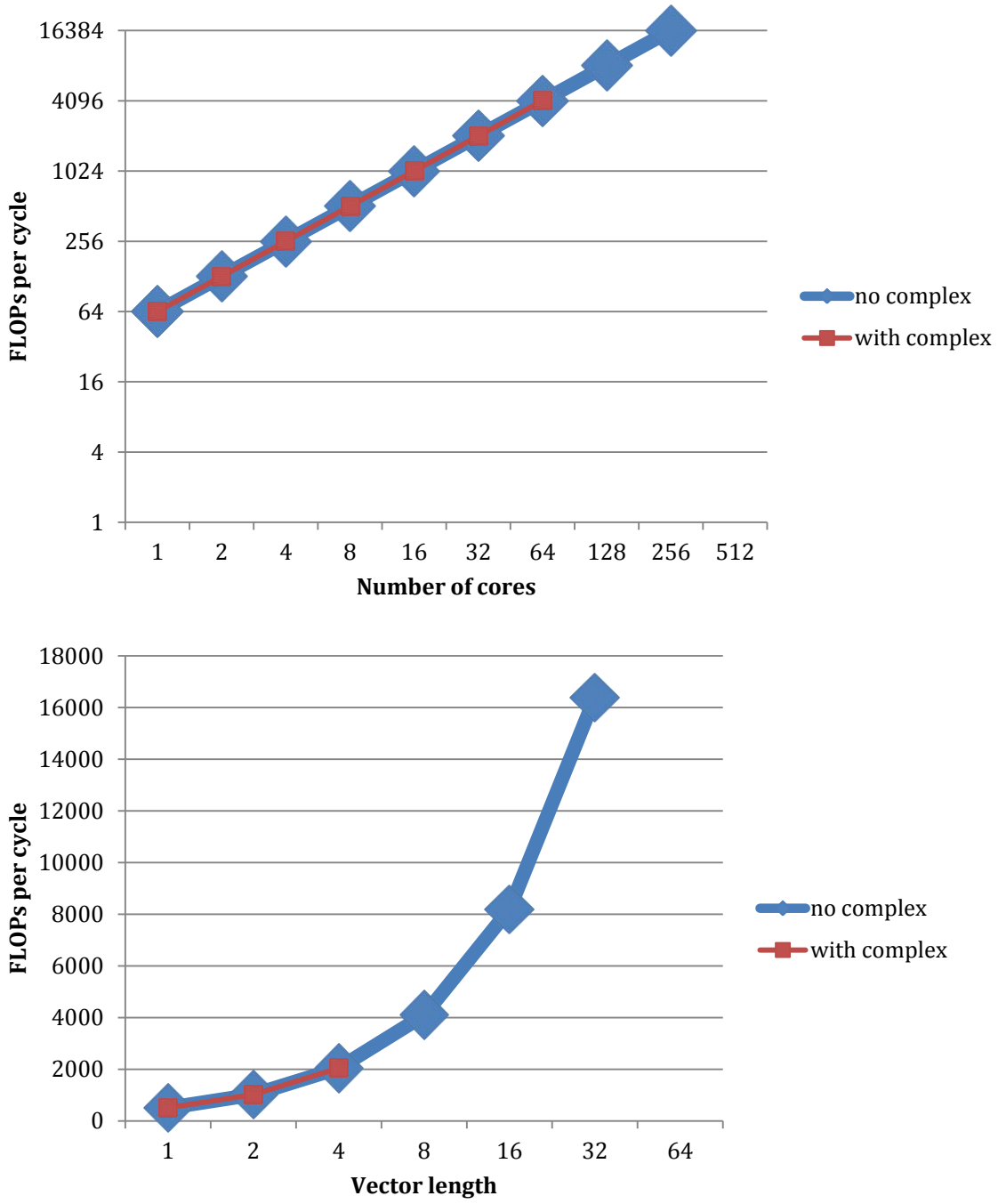


Figure 14. Neural network results

The neural network results are shown in Figure 14. The input data size is 8192 by 8192. Each tile processes its own data in parallel. The results show that the performance is proportional to the number of cores and the vector length since it is a compute-intensive application.

4.2 Emulation System

We studied the design of new emulation systems for structured ASICs [13], patterned ASICs [8], and the chip generator tool [22]. The structured ASIC contains a collection of prefixed building blocks in a fixed ratio. Stanford University is conducting research in this area. Figure 15 shows an example of a structured ASIC. The chip contains an array of microprocessors (P), signal processors (S), and memory tiles (M). If a block is not needed, the building block can be powered down to reduce the power usage.

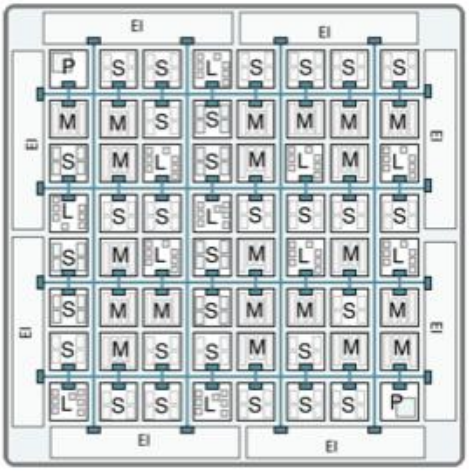


Figure 15. Structured ASIC [13]

When the chip is designed, a user designs building blocks in a HDL. A combination of the building blocks is populated in a chip. The high level output from the design is converted to RTL or gate level physical design by a human or automation tools. When the chip is used for applications, people can map applications to building blocks in the chip. Then, a tool can route connections between building blocks.

There are big differences in manufacturing memory chips and logic chips that typically result in separate packages for memory chips and logic chips. A patterned ASIC [8] allows using memory technology for logic by using a regular pattern grid common to all designs including memory and logics for all layers. This ensures manufacturability and minimizes lithographic variations. However, it does not preserve an identical array of transistors across multiple designs, so these layers and metallization are customized for each design. Figure 16 shows an example chip designed using this technology.

University of Notre Dame and Carnegie Mellon University conducted a study on patterned ASIC technology [8] and developed the Multi-core Power, Area, and Timing (McPAT) tool to evaluate trade-offs and explore the range of possible designs for complex, heterogeneous ASIC designs. It can include many processor cores, interconnection networks, I/O components, and multiple varieties of memory including 3D stacks of quilts.

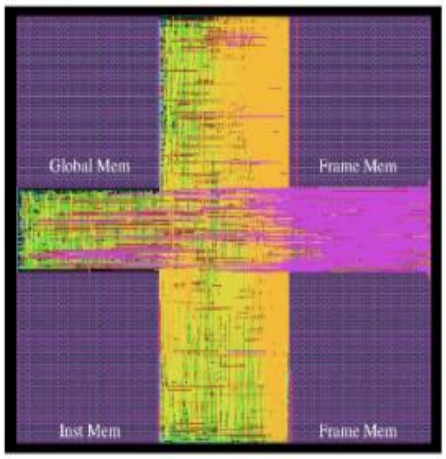


Figure 16. Patterned ASIC [8]

Chip generator [31] tool that uses a flexible, parameterized virtual device which can be tailored for each specific application. Chip design stages are as follows:

- Configure system at high level using the chip generator tool
- Run compiled application on the simulator or emulator containing the chip design generated by the chip generator
- Reconfigure system based on the previous step to achieve better performance
- Run recompiled application on the modified system
- Iterate the previous two steps until a desirable system is designed
- Finally, generate physical design and validation suite

In the validation process, they use a "relaxed" scoreboard. In this scheme, the output predictor knows a small set of possible answers and with a small additional flexibility; the end-to-end check becomes much easier than the traditional strict verification process.

In this study, we designed an emulation system that can emulate any chip that was designed using the previously mentioned approaches. An overall design flow for the emulation system that integrates the ASIC design approach is shown in Figure 17.

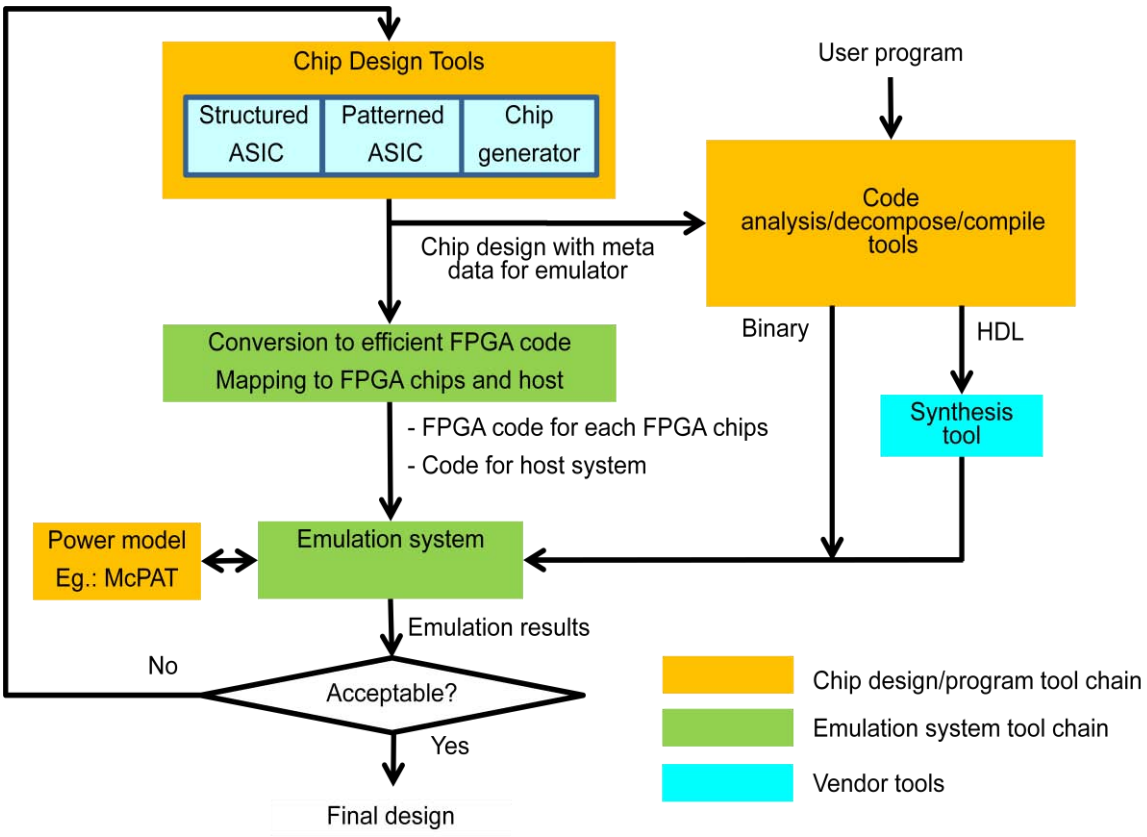


Figure 17. Chip design flow using the proposed emulation system

The chip design tool generates a design output that is converted to FPGA code, and the resulting FPGA code is mapped to FPGA chips. Then, the emulation system emulates the designed chip. In the process, a power model tool such as McPAT can be used. On the software side, the user program is analyzed, decomposed, and compiled. The output can be binary files or HDL files. The output is also fed to the emulation system. If the emulation results do not meet the criteria, the process is repeated until a good chip design is obtained.

Figure 18 shows an enlarged block diagram of the chip design tool use part (left top box in Figure 17). Based on which technology is used in the process, the chip requirement or description is input to one of the chip design tools. The chip design output format is a layout level design for patterned ASIC and RTL for the chip generator. The format for structured ASIC is not known at the time of report.

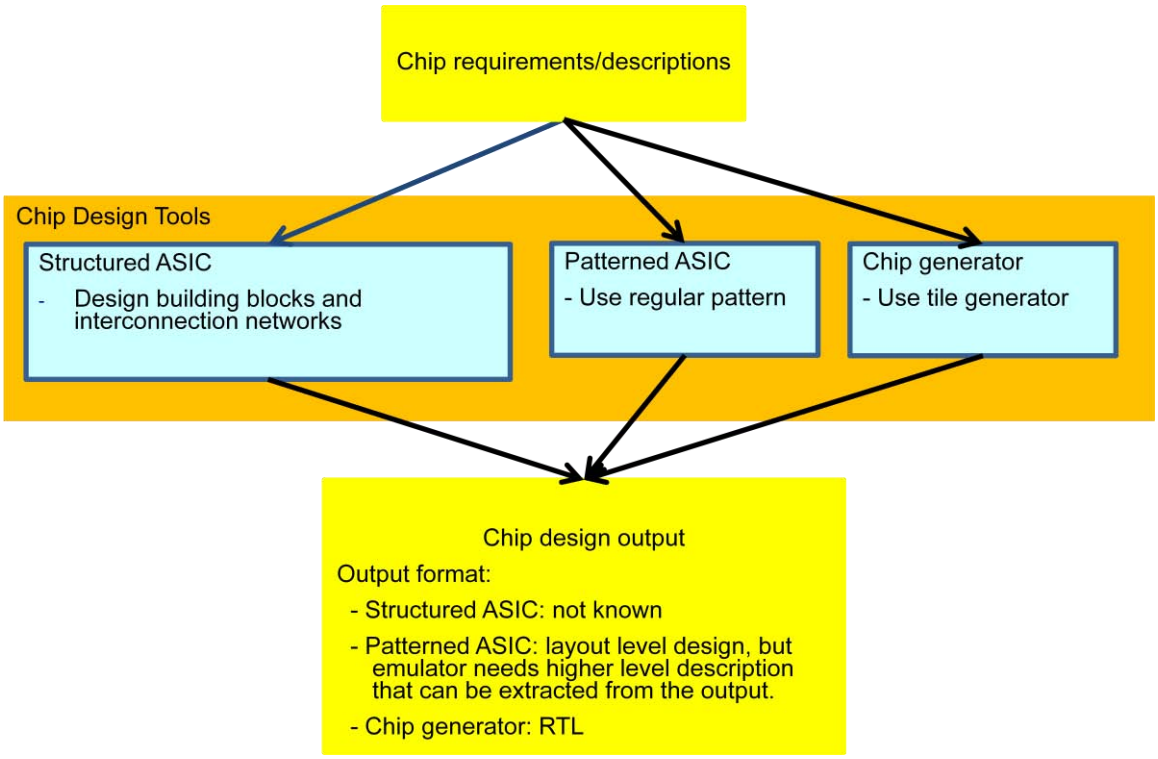


Figure 18. Chip design tools

Figure 19 shows an enlarged diagram of how user program tools are used (top right box in Figure 17). If the tool used is structured ASIC, the tool goes through several stages such as decomposition, compile, and mapping to generate binary file or HDL files. If Patterned ASIC or chip generator is used, the tool-specific compiler is used to generate binary files.

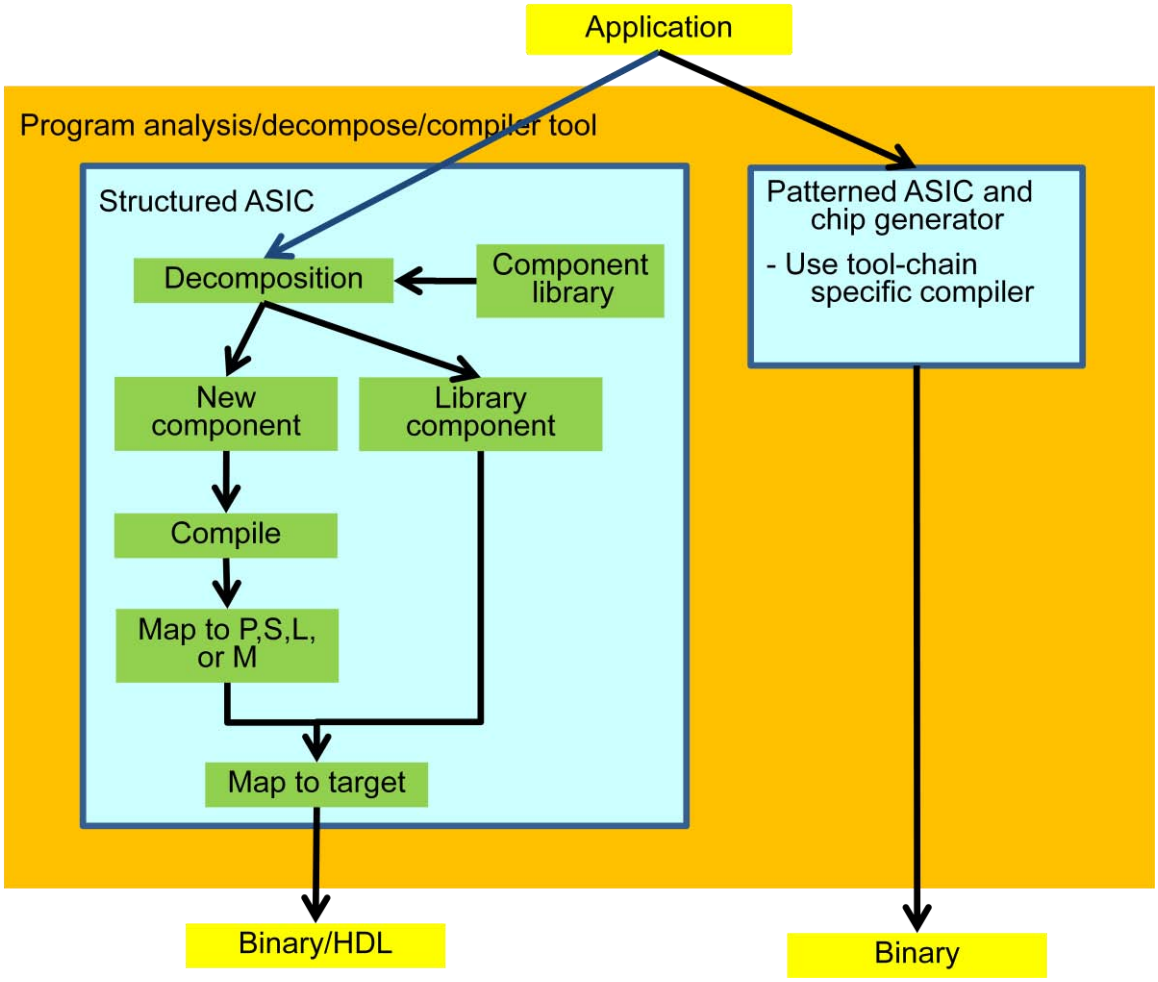


Figure 19. User program tools

The emulation system consists of a host machine, multiple FPGA boards, and a software emulation system. The host machine is a front-end interface to the external world, including user and external tools. We believe Linux is a good candidate for the host machine OS.

Each FPGA board populates multiple FPGA chips and memory chips. FPGA and memory chips are interconnected through an on-board network. Modules frequently used and/or easier to implement in an FPGA, such as a CPU pipeline, cache, interconnection network, or memory, can be emulated in FPGA chips.

The software emulation system uses software to emulate a target module and connects to the FPGA boards. This is for modules infrequently used and/or hard to implement in FPGAs, such as slow I/O and power estimation modules.

Each FPGA contains a part of the target system. The network between FPGAs provide communication. Each FPGA has a wrapper, which provides communication interface to others.

The target system consists of a set of modules and interconnection between modules. The modules may be described in RTL or logic gate level. In structured ASICs, there are four types of modules: processor, memory, signal processor, and configurable logic. In patterned ASICs and the chip generator, modules are more diverse. The final design may be multiple homogeneous or heterogeneous modules.

The modules are mapped to FPGAs. If possible, multiple modules can be mapped to one FPGA. If a module cannot fit in an FPGA, it may be partitioned into multiple sub-modules that map to an FPGA. The partition information can be saved for reuse later. Multiple modules may be emulated using time-sharing. If needed, instrumental modules can be added. For example, power measurement can be added.

Finding the best mapping that minimizes the number of FPGAs and communication network usage is a challenging task. It can start with a very basic algorithm such as a greedy algorithm, but with more research, better mapping algorithms can be used.

To determine the necessary emulation size, the target system size needs to be determined. Then, based on that, the number of FPGA boards is determined. The emulation system needs to be expandable to accommodate various target system sizes. Each board needs to have an inter-board network interface and module identification number, and an FPGA identification number can be used for routing messages among modules.

The emulation system design requirements determine the parts to be used. The higher the target emulation speed, the more difficult to design the system and the more expensive it is. At the time of this report, it seems reasonable to target an emulation speed of 6 - 30 MHz. This is 20 - 100 times slower than the maximum FPGA frequency, which is reasonably achievable for unoptimized designs. The Virtex-6 FPGA has up to 758,784 logic cells and 1,200 user I/O pins that might be used in the emulation system. Memory size can be up to 8 GB per module. The best communication network bandwidth and topology depend on the number of FGPAs and the number of boards. A mesh or ring topology might be used.

The emulation system needs an interface to the host system. For example, PCIe or Ethernet can be used to communicate to the host system.

The emulation system needs to interact with external tools. For this purpose, meta-data embedded in a chip design and compiled code may be used. Some examples are shown below.

- Metadata from chip design
  - ## module_type memory size=4096 Mwords
  - ## module_type cache size=4096 words, set=2-way
  - ## module_type logic
  - ## module_type signal_processor
- Metadata from user program compiler
  - ## code_type executable, module=module_id
  - ## code_type VHDL, module=module_idx

The communication between FPGAs can be done using messages. The communication protocol needs to be defined. For example, a header contains target FPGA identification number, target module identification number, source module identification number, type of communication, and message length.

Instrumentation for the system can help in obtaining run-time statistics of the emulation system execution and target application execution. For example, a switch rate counter for power estimation per unit, amount of communication among modules, ratio of emulation cycles to target cycles for emulation system executions, can be collected. Examples of target application executions include the number of cycles, number of instructions, cache miss rate, amount of data to/from cache, amount of data to/from memory, and so on.

The debugging feature allows users to locate bugs in an emulated system. A debugging module can be populated, and the module can accept user commands and interact with system modules. Each module, then, needs to be able to accept control through the debugging module. The debugging module can control the emulation system with controls such as starting, breaking, and stepping. It needs to be able to report status to the user, such as status of modules, status registers, and activities of modules. Optional features, such as a visualization tool, can help the user significantly.

In the case of the chip generator, the target system is described using a "program," which is an intermediate level description of a chip. Structured ASICs and patterned ASICs may use the same form to describe the target system. If necessary, the intermediate form may be generalized to accommodate all target systems. Further study is needed to identify the fidelity between the intermediate form and the final target. The intermediate form must be usable by high-level domain experts. For example, all coherence and memory ordering rules require a specific set of hardware and protocol rules for correct implementation and often contains a number of tricky corner cases [22].

Validation is needed to ensure emulation system correctness. There are many different types of validation tests; a mix of them would ensure accuracy. Some of these tests are shown below:

- Test of modules
  - Correctness in function and timing
- Test of interactions of modules
- Sample test vector based test
- Exhaustive test vector tests
- Fault detection tests
  - Add in incorrect input and see if it is detected

Validation tests need to be initiated with automatic and manual generation methods. In automatic generation, individual test suites are readily generated from the emulation system configurations. A test generator is needed from each module operation and inter-module interactions. Complex and/or special tests can be generated more efficiently by manual test suite generation.

In the structured ASIC case, it is relatively easy to design a small number of components for application domains in an FPGA, and powering off unused blocks is straightforward. The interconnection network in the structured ASIC is configurable and it can be implemented in an FPGA to allow "soft" configuration of the network.

A large-sized target system can be emulated using the proposed expandable emulation system. Mapping of peripheral functions implemented in libraries may be emulated using the software emulation unit. This issue is common to chip generators as well.

In a patterned ASIC, the processing/memory ratio is higher than in an FPGA. Since an FPGA is generally slower than ASIC, a memory interface module is needed. To match the ratio in the target system, the memory interface module reads and supplies data when needed. Linking McPAT and an emulator is a separate task, but can be linked using meta-data.

We surveyed power estimation tools for use in an emulation system. CACTI [21] from HP is a cache and memory access time, cycle time, area, leakage, and dynamic power model. It provides both a web interface (quid.hpl.hp.com:9081/cacti/) and C++ code. It does not provide short-circuit power modeling. Another similar tool is Wattch [9] from Princeton University. The tool provides a parameterized power model.

The Capet [36] from IBM estimates power due to switching activity. It can estimate power from RTL description. Another tool from Princeton University is Orion [39], which estimates power for on-chip interconnection network and is ongoing research work. It may augment other tools since it focuses on interconnection networks.

We designed an emulation system for an example many-core target chip having up to one thousand cores. The target chip has a 400 mm^2 chip area with a 32 nm technology operating at 1 GHz clock speed. For this example, we estimated that up to 400 million logic gates are necessary based on the extrapolation from the Intel Core 2 Duo E8200 [24]. With an assumption that 50% of the gates are used for cache, we need to emulate 200 million gates by FPGA. Since the Virtex-6 FPGA has up to about 0.75 million logic cells, each of which has 3 gates, about 100 FPGAs are needed.

FPGAs can operate at up to 600 MHz, but the actual target chip speed will be much slower because of inter-module communication. We estimate about a 20 – 100 times slowdown which is still much faster than software simulation.

The emulation system should provide remote access capability since different DoD user teams may be stationed in various locations, and we can amortize the cost over multiple sites.

Several software tools are needed. One of them is a mapping tool that maps modules to FPGAs and is able to partition a module to multiple FPGAs if a module is too large to fit in an FPGA. Another is a wrapper tool that enables communication between modules. Another tool is instrumentation and debugging tool that allows interaction with the user and external tools. The last tool is a power estimation tool that estimates dynamic power using switching activity counter, static power, and short circuit power.

A validation suite is also needed to guarantee that the emulation system works properly.

We surveyed other systems that may work for DoD emulation systems. One of them is the Palladium III [10] from Cadence. It is a software accelerator/emulation system of up to 256 million gates operating at up to 2 MHz. It comes with a large set of software tools, such as a compiler, debugger, and power estimation tool. Further study is needed to determine suitability.

Another system is the Novo-G [19] from University of Florida, which has 96 Altera Stratix FPGA chips. Each PCIe card has four FPGA chips. The boards are made by GiDEL [18] and are interconnected with InfiniBand. We estimate it would take about 50 boards to support the example target chip.

Another FPGA board that may be used for emulation is Xilinx FPGA board from the DINI group [15]. Several boards with different configurations are offered. For example, the DN-DUALV6-PCIe-4 board has two Virtex-6 chips, and the DN-80000K10 has 16 Virtex-5 FPGAs. These systems are smaller than the proposed system, and they may be used to emulate only part of the chip.

There are evaluation boards from FPGA vendors [2][41]. These usually come with a single chip on a board with a relatively low price. These may be used for a subset of a target chip, but are not suitable for full target chip emulation.

Most of these systems provide hardware for emulation, but are lacking the software tools still need to be developed for productive emulation.

4.3 Evaluation Board

In this study, we developed a high-level design for an evaluation board for structured ASICs, patterned ASICs, and the chip generator. The evaluation board uses a host computer to control the board, the latter containing one or two target chips and memory. FPGAs on the board provide glue logic.

The design can have variations to suit different purposes. For example, one variation can serve for memory-intensive applications by providing large-sized memory. Another variation can serve I/O-intensive applications by providing many I/Os with a reduced memory size.

We designed the board to accept any one of the target chips. To allow this, the three target chips need to have a common pin layout for most common pins such as power, host interface, and JTAG. Some different pin layouts can still be accommodated by using FPGAs. For example, memory interface or I/O layouts can be different, but the FPGA can route them to the right place by configuring the FPGA with a corresponding bit file.

Figure 20 shows a general block diagram of the proposed evaluation board.

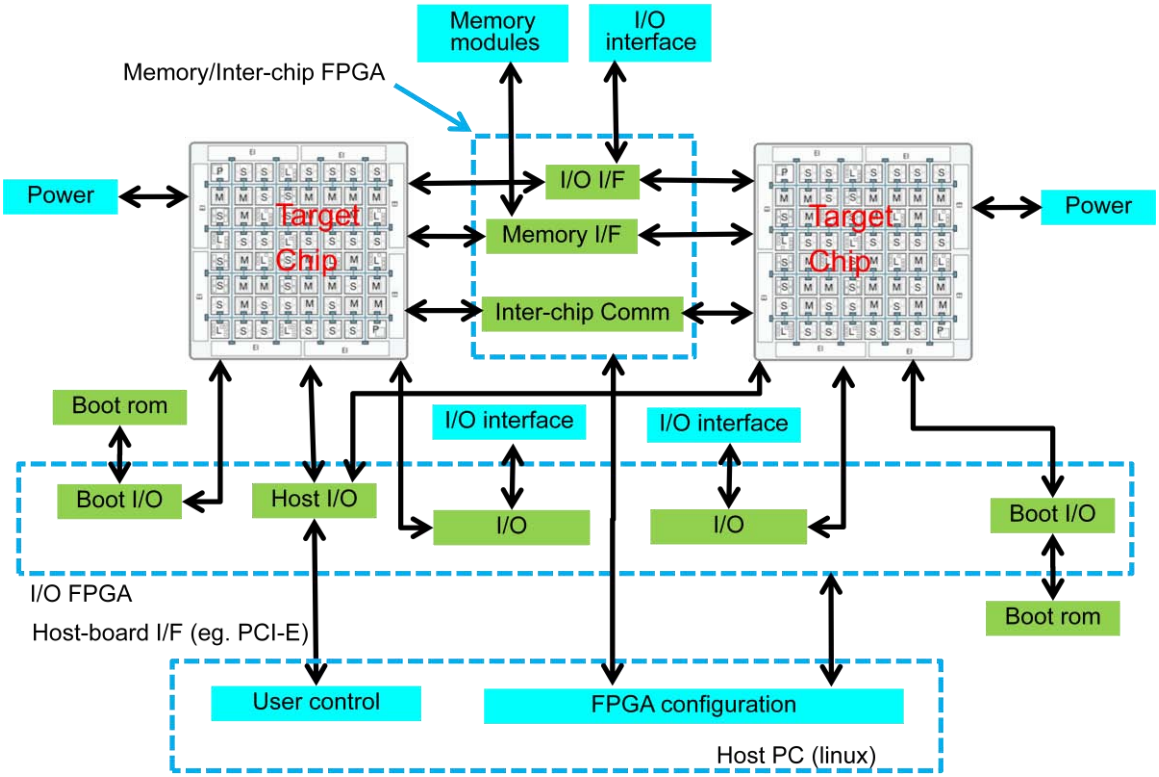


Figure 20. General block diagram for evaluation board

The board uses a PCIe board form factor. It can be populated with two FPGAs, two target chips, and four memory modules. We assume a target chip having a standard 1,536 pin BGA. However, if necessary, even a 2,025 pin BGA can be accomodated. The target chip may be soldered directly on the board or a socket may be used for easy testing. Soldering provides higher reliability, easier handling, and lower cost. Using a socket provides an easier replacement mechanism.

There are a few candidates for host I/O including PCIe [33], which provides 8GB/s bandwidth with a x16 configuration. QuickPath [26] and HyperTransport [23] -- both 25.6 GB/s bandwidth -- are faster but relatively new. PCIe and HyperTransport cores for FPGAs are available and make the board design easier. The host I/O will provide a path for controlling target chips, uploading/downloading data, sending programs to and from target chips and memory, and debugging target chips, etc. External I/O and boot ROM logic will be connected to the target chip through glue logic in the FPGA.

Four memory modules, to support up to 32 GB, can be populated. Each of the modules is a 240-pin 8-GB DDR3.

Memory glue logic in FPGAs provides flexibility for target chip memory configuration. For example, a larger data path with a smaller address space or a smaller data path with larger address space can be configured. It can operate at full memory speed through the FPGA logic. The IP core for DDR3 is available for easier implementation of memory interface.

Some status LCD or LED will be helpful to display the status of the board or debugging information.

Alternative memory interfaces are those memory interfaces not provided by the FPGA. The alternative memory interface can use one of two methods: fixed interface or semi-flexible interface. In a fixed interface, all memory interfaces are constant given that all target chips agree on the same memory configurations. If this method is used, it saves the FPGA, PCB space, and cost. It also increases reliability, and memory latency will be reduced.

In the second method, the semi-flexible interface, four external interfaces are provided as a specification, each of which is connected to a memory module of 8GB. The target chip can choose its own configuration from one of the three possibilities:

1. Four internal interfaces on the chip, each with access to 8 GB of capacity, are connected to the four external interfaces directly. Note: the four external interfaces are connected to the four memory interfaces on the board.

2. The target chip can use two internal interfaces with 16 GB of memory each. Then, the chip internally uses two multiplexers to connect the two internal interfaces to four external interfaces to use the four external interfaces. To do that, the first half of the memory space in one internal interface goes to one external interface and the other half of the memory space in the internal interface goes to another external interface through the multiplexer. Then, each of the two internal interfaces has access to 16 GB of memory.

3. One internal interface is connected to four external interfaces through a two-stage multiplexer on the chip. Therefore, the one internal interface effectively has access to 32 GB of memory.

There are a few issues to be resolved with the target chips. One of them is power. The voltage inputs and number of pins needs to be determined by the target chip designers. Also, what and how many I/Os to populate needs to be determined. The interconnection network between the two chips needs to be decided. The pin layout and memory interface also need to be determined.

Three example configurations are shown below. The first, shown in Figure 21, shows a configuration with one target chip. Note that the green lines represent connections on the board not used in the configuration. Figure 22 shows a configuration with two target chips with a large memory to support memory-intensive applications. Figure 23 shows a configuration with two chips to support I/O-intensive applications.

Figure 21. Example board configuration I

Figure 22. Example configuration II

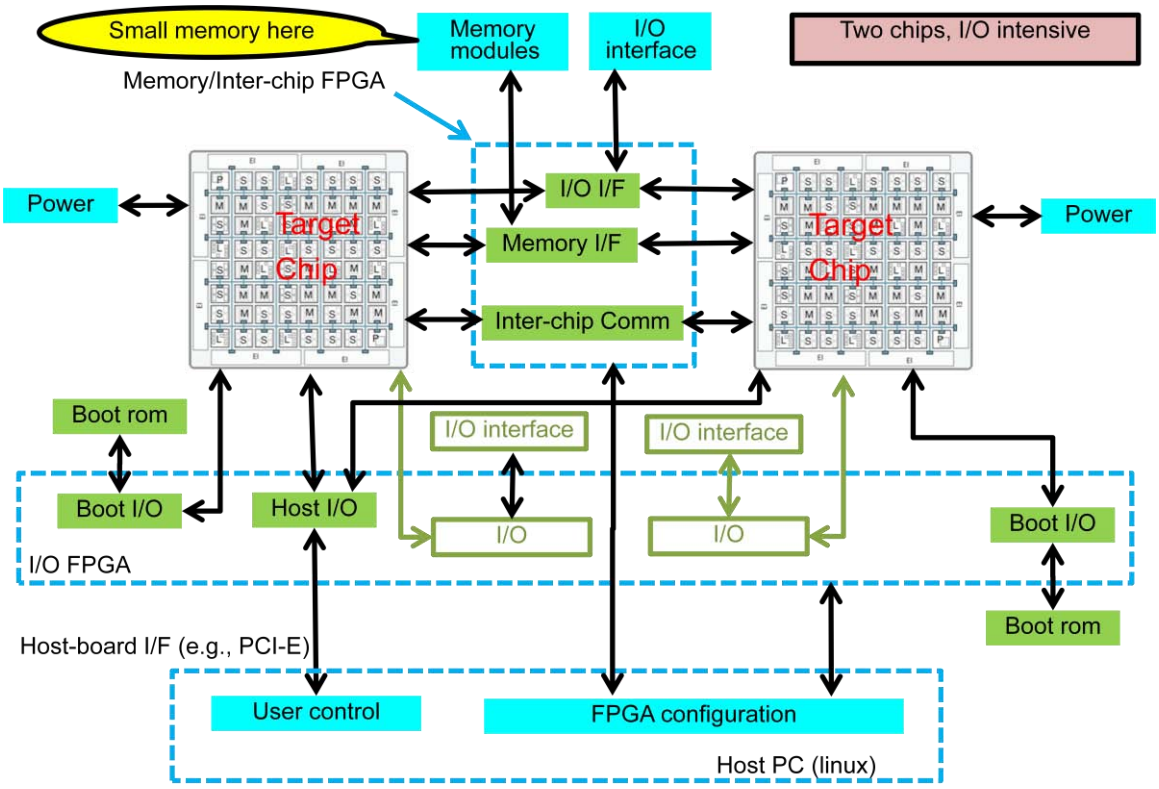


Figure 23. Example configuration III

4.4 Design Patterns

The prototype abstractions presented earlier demonstrate how a pattern-oriented approach for programming could be used to create portable and scalable code using parameterized abstractions to encapsulate the details of parallel programming. In order for this approach to be applicable to DoD applications, we must ensure that the code generated by these hypothesized abstractions attains high performance on a target architecture. To demonstrate high performance, we created a fully generalized shared-memory implementation of matrix multiplication that works for arbitrary matrix and processor sizes and that uses the indices computed from the parameterized patterns.

The final, generalized code was implemented on the Tilera TILE64 [5]. The TILE64 consists of 64 processors on a chip organized into a 2-D mesh fabric. Both shared memory and message-passing mechanisms are provided. Floating-point operations are implemented in software; for this reason, the performance of integer-based matrix multiplication was chosen for study.

A code fragment from the matrix multiplication algorithm is shown in Figure 24. The first part of the code computes the parallel indices based on processor ID. In the pattern-based abstractions hypothesized in the previous section, these computations would be done when the matrix object is instantiated and mapped to an abstract processor. The indices would be recomputed and stored inside the object if the object were re-mapped to the same or a different processor array. Therefore, the index computations are not considered as part of the

matrix multiplication algorithm, and are not included in the performance measurements. The function “StridedMemCopy” copies blocks of shared memory to a local buffer to access the correct block of the matrix. The function “mutiply” implements the loops to multiply the two local blocks.

```
BlockRowsY = (int)ceilf((float)OutRows/(float)ProcY);
BlockColsX = (int)ceilf((float)OutCols/(float)ProcX);
BlockInnerX = (int)ceilf((float)InnerDim/(float)ProcX);

MyRowY = rank_Y * BlockRowsY;
MyColX = rank_X * BlockColsX;
MyBlockX = rank_X * BlockInnerX;
BlockRowsY = MIN(BlockRowsY, OutRows-MyRowY);
BlockColsX = MIN(BlockColsX, OutCols-MyColX);

Leftover = InnerDim - BlockInnerX * (ProcX-1);
if (BlockInnerX > Leftover) Limit = ProcX-1;
else Limit = ProcX;

for (shift_by=0; shift_by<Limit; shift_by++) {
  CurrInner = (rank_X + rank_Y + shift_by)%Limit;

  StridedMemCopy(local_A,
                 BlockInnerX*sizeof(int),
                 &mat_A[MyRowY*InnerDim+BlockInnerX*CurrInner],
                 InnerDim*sizeof(int),
                 BlockInnerX*sizeof(int),
                 BlockRowsY);

  StridedMemCopy(local_B,
                 BlockColsX*sizeof(int),
                 &mat_B[CurrInner*BlockInnerX*OutCols+MyColX],
                 OutCols*sizeof(int),
                 BlockColsX*sizeof(int),
                 BlockInnerX);

  multiply(local_A, local_B, local_C,
           BlockRowsY, BlockColsX, BlockInnerX);
}

if (BlockInnerX > Leftover) {
  StridedMemCopy(local_A,
                 Leftover*sizeof(int),
                 &mat_A[MyRowY*InnerDim+BlockInnerX*(ProcX-1)],
                 InnerDim*sizeof(int),
                 Leftover*sizeof(int),
                 BlockRowsY);

  StridedMemCopy(local_B,
                 BlockColsX*sizeof(int),
                 &mat_B[(ProcX-1)*BlockInnerX*OutCols+MyColX],
                 OutCols*sizeof(int),
                 BlockColsX*sizeof(int),
                 Leftover);

  multiply(local_A, local_B, local_C,
           BlockRowsY, BlockColsX, Leftover);
}

StridedMemCopy(&mat_C[MyRowY*OutCols+MyColX],
               OutCols*sizeof(int),
               local_C,
               BlockColsX*sizeof(int),
               BlockColsX*sizeof(int),
               BlockRowsY);
```

Figure 24. Matrix multiplication generalized code

The results for this code are shown in Figure 25 for increasing machine sizes. Both cycle count and throughput (operations per cycle) are shown. The performance decreases as we scale to larger machine sizes; this can be attributed to increased memory network contention.

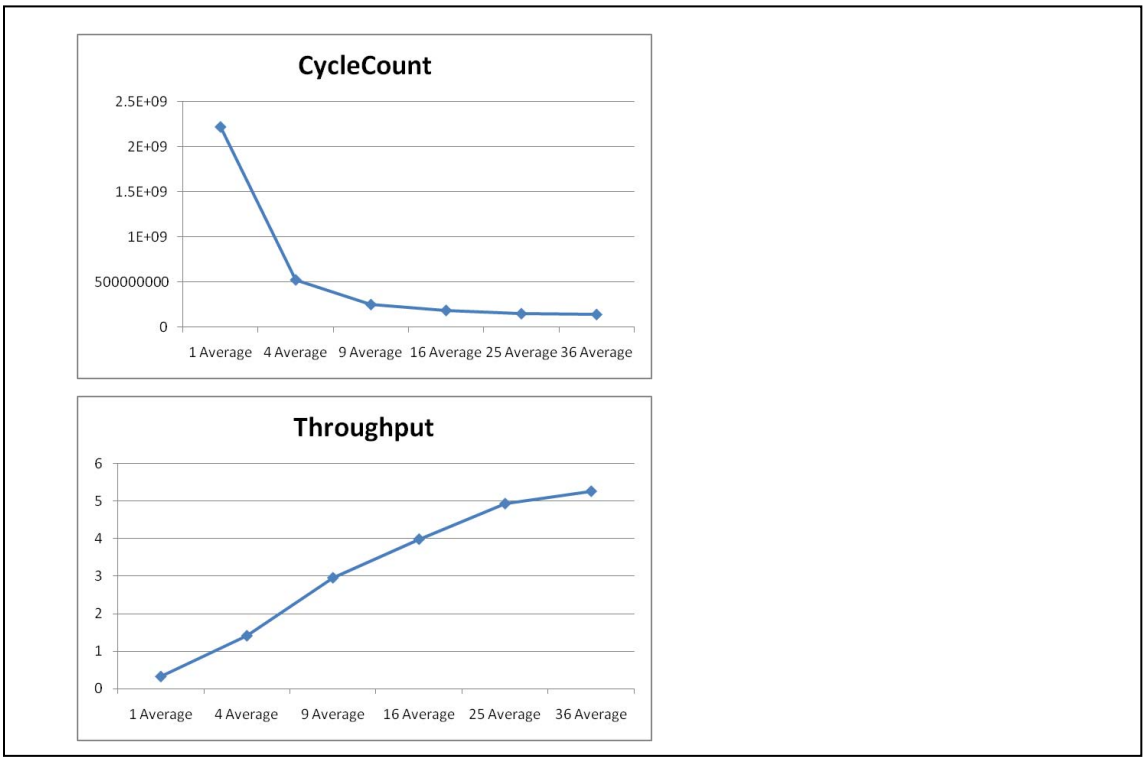


Figure 25. Performance results, generalized dode (problem size: 720x720x720)

For comparison, we generated results for a baseline version of matrix multiplication based on Cannon's algorithm. This baseline version is not generalized: the matrix and block sizes are fixed in the code at compile time, and the code only that works for square matrices and square tile arrays where the matrix size is a multiple of the tile array size. The comparison between the two codes is shown in Figure 26, which shows cycle counts for each processor for each run of the code.

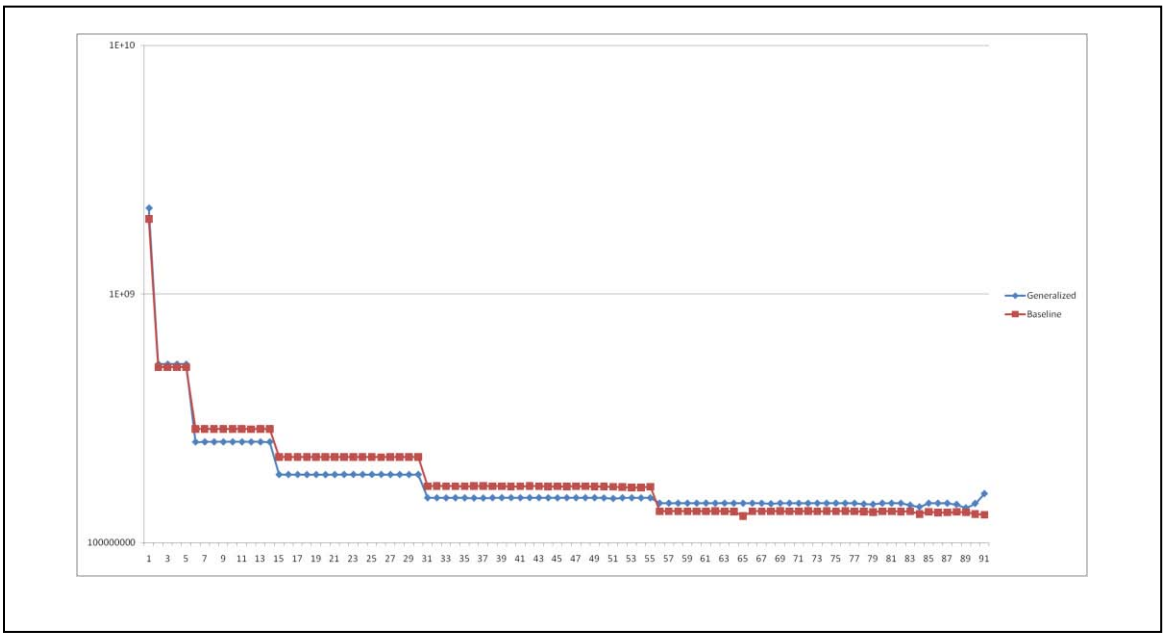


Figure 26. Performance comparison (cycle count)

Multiple runs were performed with different machine sizes; the large changes in cycle count show the boundaries between machine configurations. There is a fundamental difference in the access patterns between the two versions: the baseline code has inputs in skewed order, so that the matrix multiplication accesses memory in regular patterns. In the fully generalized code, the inputs are in normal, C order, and the skew indices must be computed inside the main loop. This difference in access patterns has an interesting effect: the generalized code performs better for some of the smaller machine sizes, but slightly worse for the larger machine sizes. Further experimentation, profiling, and analysis must be performed to determine the cause of the performance discrepancy.

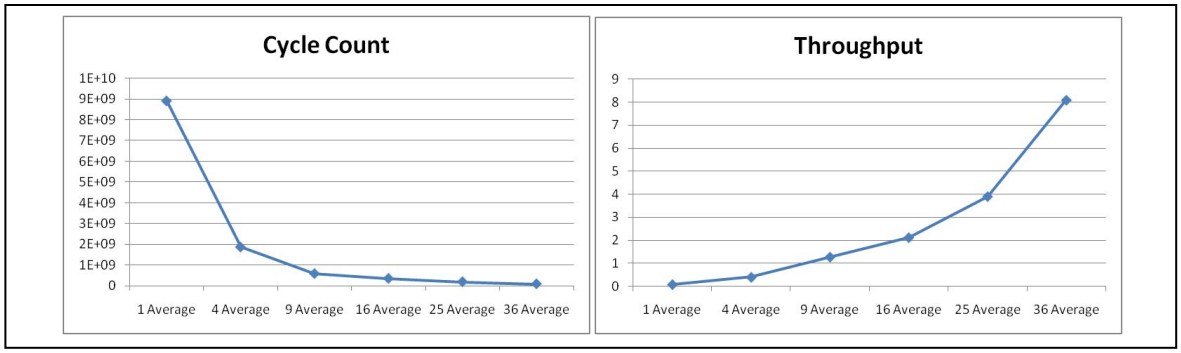


Figure 27. Message passing (iLib) performance results

A second version of matrix multiplication was implemented using iLib message passing. Note that iLib is a message passing and shared memory communication library for TILE64 from Tilera Corporation. Performance results for this code are shown in Figure 27. In this code, the

block sizes and row/column indices are computed in the same manner as the shared memory implementation, but the values are used to set up message buffers and perform transfers. The message passing code performs better for larger tile arrays, but worse for small tile arrays. One possible reason for this is that the overhead for message transfer is relatively higher for smaller tile arrays, but the greater efficiency of message passing starts to dominate in larger tile arrays where memory bandwidth would start to saturate. Further profiling and experimentation is needed to examine the performance behavior of this code.

To demonstrate the synergy of patterns between the message passing and shared memory implementations, a second, non-generalized version of the shared memory version was created. This second version has the same performance characteristics of the generalized version presented above but has the same structure as the message passing code at the high level, suggesting that abstractions could be built which take advantage of these common patterns. To demonstrate the synergy of this version with message passing, this code was not generalized to arbitrary matrix sizes and tile arrays.

The message passing version and the second shared memory version each start with the same index computations and local buffer allocations, shown below with the corresponding pattern abstractions:

```
int rank_X = rank % ProcX;                  /* LocalID = UE_Create(Machine); */
int rank_Y = rank / ProcX;

int BlockRowsY = OutRows / ProcY;           /* MyPiece = GEOMETRIC(Matrix, */
int BlockColsX = OutCols / ProcX;           /*   LocalID, Directives, Mapping) */
int BlockInnerX = InnerDim / ProcX;

int MyRowY = rank_Y * BlockRowsY;
int MyColX = rank_X * BlockColsX;
int MyBlockX = rank_X * BlockInnerX;
int MyBlockY = rank_Y * BlockInnerX;

int A_Buf = BlockRowsY * BlockInnerX;
int B_Buf = BlockInnerX * BlockColsX;
int C_Buf = BlockRowsY * BlockColsX;
local_A = (int *)malloc(2 * A_Buf * sizeof(int));
local_B = (int *)malloc(2 * B_Buf * sizeof(int));
local_C = (int *)malloc(C_Buf*sizeof(int));
for (i=0; i<C_Buf; i++) local_C[i] = 0;
```

After the initial index calculations, the message passing and shared memory versions share a similar structure. Figure 28 shows the message passing code. Inside the functions, calls to iLib implement the message transfers. Figure 29 shows the shared memory code. In each case, high-level patterns are shown in comments.

```
/* DATA_SHARE(MyA) and DATA_SHARE(MyB) */
InitSkew(&send_left, &recv_right, &send_up, &recv_down, rank_X, rank_Y, ProcX, ProcY);
ShiftMsgs(send_left, recv_right, local_A, A_Buf, 1, 0, rank_X, rank_Y);
ShiftMsgs(send_up, recv_down, local_B, B_Buf, 1, 0, rank_X, rank_Y);

Neighbors(&send_left, &recv_right, &send_up, &recv_down, rank_X, rank_Y, ProcX, ProcY);
int compute_flag = 0, recv_flag = 1;
for (i=0; i < ProcX-1; i++) {                /* while (NOT-DONE(MyC) */
  compute_flag ^= 1;

  /* DATA_SHARE(MyA) and DATA_SHARE(MyB) */
  ShiftMsgs(send_left, recv_right, local_A, A_Buf, compute_flag, i, rank_X, rank_Y);
  ShiftMsgs(send_left, recv_right, local_B, B_Buf, compute_flag, i, rank_X, rank_Y);

  /* MatrixIntMultiplyBlock */
  multiply(local_A+compute_flag*A_Buf, local_B+compute_flag*B_Buf,
           local_C, BlockRowsY, BlockColsX, BlockInnerX);

  ClearMessages();
}
compute_flag ^= 1;

/* MatrixIntMultiplyBlock and DATA_SHARE(MyC) */
multiply(local_A+compute_flag*A_Buf, local_B+compute_flag*B_Buf,
         local_C, BlockRowsY, BlockColsX, BlockInnerX);
```

Figure 28. Message passing (iLib) code excerpt

Now, we discuss the abstractions described above in more detail. Specifically, we describe the information that must be contained in each abstraction in order to generate the generalized C code in the results section. This discussion is intended to provide a general framework for future research in this area by setting forth requirements in the context of a pattern-oriented approach to software design. Design and development of a working set of pattern-based software abstractions would be the subject of future research projects.

The fragment of high-level code that uses hypothesized pattern-based abstractions is shown again below:

```
DecomposeIntMatrix A([100 300], Directives, Machine);    // Instantiate 3 matrix
DecomposeIntMatrix B([300 500], Directives, Machine);    // objects, specify
data
DecomposeIntMatrix C([100 500], Directives, Machine);    // decomposition
MatrixIntMultiplyDataParallel(A, B, C, Directives, Machine);
```

```
/* DATA_SHARE(MyA) and DATA_SHARE(MyB) */
CurrInner = (rank_X + rank_Y) % ProcX;
A_start = MyRowY * InnerDim + BlockInnerX * CurrInner;
B_start = CurrInner * BlockInnerX * OutCols + MyColX;
StridedMemCopy(&local_A[A_Buf], BlockInnerX*sizeof(int), &mat_A[A_start],
               InnerDim*sizeof(int), BlockInnerX*sizeof(int), BlockRowsY);
StridedMemCopy(&local_B[B_Buf], BlockColsX*sizeof(int), &mat_B[B_start],
               OutCols*sizeof(int), BlockColsX*sizeof(int), BlockInnerX);

compute_flag = 0, recv_flag = 1;
for (shift_by=1; shift_by<ProcX; shift_by++) {       /* while (NOT-DONE(MyC) */
  compute_flag ^= 1; recv_flag ^= 1;

  /* DATA_SHARE(MyA) and DATA_SHARE(MyB) */
  CurrInner = (rank_X + rank_Y + shift_by) % ProcX;
  A_start = MyRowY * InnerDim + BlockInnerX * CurrInner;
  B_start = CurrInner * BlockInnerX * OutCols + MyColX;
  StridedMemCopy(&local_A[recv_flag*A_Buf], BlockInnerX*sizeof(int), &mat_A[A_start],
                 InnerDim*sizeof(int), BlockInnerX*sizeof(int), BlockRowsY);
  StridedMemCopy(&local_B[recv_flag*B_Buf], BlockColsX*sizeof(int), &mat_B[B_start],
                 OutCols*sizeof(int), BlockColsX*sizeof(int), BlockInnerX);

  /* MatrixIntMultiplyBlock */
  multiply(local_A+compute_flag*A_Buf, local_B+compute_flag*B_Buf, local_C,
           BlockRowsY, BlockColsX, BlockInnerX);
}
compute_flag ^= 1; recv_flag ^= 1;

/* MatrixIntMultiplyBlock and DATA_SHARE(MyC) */
multiply(local_A+compute_flag*A_Buf, local_B+compute_flag*B_Buf,
         local_C, BlockRowsY, BlockColsX, BlockInnerX);
C_start = MyRowY * OutCols + MyColX;
StridedMemCopy(&mat_C[C_start], OutCols*sizeof(int), local_C, BlockColsX*sizeof(int),
               BlockColsX*sizeof(int), BlockRowsY);
```

Figure 29. Code excerpt for shared memory, version 2

In this code fragment, there are two types of objects: data objects such as matrices, and task objects such as the matrix multiplication task.

Task objects have two fundamental properties that are relevant to their coded implementation. Firstly, the decomposition for the task must be specified. If the task uses task decomposition, then the problem must be further divided into sub-tasks. If the task uses data decomposition, then all inputs and outputs to the task must have a decomposition specified. If data decomposition is specified, but inputs and outputs have no decomposition, then the specification is inconsistent, and an error would be generated.

Secondly, the program structure must be specified. If the program structure is SPMD, as in the case of matrix multiplication, then the program components must be provided in a SPMD coding framework, including initialization, UID generation, program execution, data distribution, and finalization. The methods for task objects could change either the decomposition or the program structure, as long as the correct metadata were supplied. Since the matrix multiplication case study in this work uses only data decomposition and SPMD, the development of task object parameters and how they would be used to generate code is left for future efforts.

In addition to size, dimensionality, and element type, matrix objects would require extra parameters and metadata that specify data decomposition in such a manner that a data parallel function could derive the correct code from the information inside the matrix object. The code inside the data parallel matrix multiplication is shown again below.

```
LocalID = UE_CREATE(Machine);        /* Set the unique processor ID using an */
                                     /* abstract parallel machine description */

MyA = GEOMETRIC(A, LocalID, Directives, Machine);      /* Specify geometric */
MyB = GEOMETRIC(B, LocalID, Directives, Machine);      /* distributions for */
MyC = GEOMETRIC(C, LocalID, Directives, Machine);      /* A, B, and C */

DATA_SHARE(MyA);                /* Perform an initial sharing of data (skew) */
DATA_SHARE(MyB);                /* skew definition is inside the object */
While (NOT-DONE(MyC)) {         /* loop over all blocks */
  DATA_SHARE(MyA);              /* get next block of input A */
  DATA_SHARE(MyB);              /* get next block of input B */
                                /* objects know how to access their next block
*/
  MatrixIntMultiplyBlock(MyA, MyB, MyC);     /* perform the block multiply */
                                             /* using the correct loop bounds */
                                             /* and block sizes */
}

MatrixIntMultiplyBlock(MyA, MyB, MyC);       /* Perform final block multiply */
DATA-SHARE(MyC);                             /* put current block of output C */
```

In the first step, a unique local identifier is created for each processor in the parallel machine. The function UE_CREATE returns this local identifier given an abstract description of the machine. In the case of matrix multiplication, the machine is described as a two-dimensional tile array. The actual assignment of physical processors to abstract processors can be achieved either inside the UE_CREATE function or in the system software for the processor. In either case, only the unique processor identifier is needed for the matrix multiplication code.

In the next step, the data parallel matrix multiplication uses the processor ID, the mapping directives, and the abstract machine description to determine which piece of each matrix is mapped to each processor. The function GEOMETRIC performs the geometric decomposition of each matrix using these parameters. For the shared memory code, this function computes the block sizes and pointers into shared memory for each matrix. The mapping directives specify the block-cyclic parameters on each dimension of the matrix and the mapping of dimensions between the matrix and the abstract 2-D processor array. The abstract machine description defines the extent of the tile array in each dimension. The computations performed inside the GEOMETRIC pattern are exactly those in the C matrix multiplication code presented earlier.

For distributed matrices with message passing, the semantics of the GEOMETRIC function are a bit different. In this case, the data distribution is a property of both the matrix and the task. For the matrix, the data distribution specifies how the data is arranged over the processors. For the task, the data distribution specifies a required arrangement of data over processors for inputs and outputs. In some cases, the data distributions required for the task may not be the same as the data distributions of the input matrices. In this situation, the data must be rearranged into the correct distribution for the algorithm. With the correct

information contained inside the data objects, any required data movements can be performed inside the GEOMETRIC pattern and will be invisible to the programmer at the pattern level. Once the data is arranged correctly for the algorithm, message buffers are allocated and source-destination pairs are determined for the various message exchanges in the algorithm.

After the GEOMETRIC pattern, each processor has an object that contains the location of its block for each iteration of the algorithm. Inside the matrix multiplication loop, the DATA_SHARE function uses the information inside this object to retrieve the next block. If the data is in shared memory, the DATA_SHARE function computes a pointer and retrieves the appropriate block. If the data is in distributed memory, then the DATA_SHARE function accesses a message buffer into which a message is received from another processor; the source processor is determined from the mapping information contained in the local object. The local output object contains the information required to determine when the loop is finished; the NOT_DONE function accesses this information.

The abstractions described here would provide a pattern-based approach that isolates details of pattern implementation from the programmer at the high level. The challenge for future work in this area is to define the information contained in the object properties, mapping directives, and abstract machine descriptions such that the abstraction barriers between the different functions can be maintained and high performance code can still be produced. An immediate next step in this research would be to implement the message-passing version of matrix multiplication from the pattern template described above to verify both the abstraction semantics and performance viability.

The idea of encapsulating mapping information inside data and task objects and providing semantics and notation for describing data mappings has been explored in depth in previous efforts. The Embedded Digital Systems Group at Lincoln Laboratory has been developing parallel libraries for more than 10 years [6]. The libraries developed there include STAPL (Space-Time Adaptive Processing Library) [14], PVL (Parallel Vector Library) [28], and pMatlab [27]. All of these libraries have increased the level of abstraction by implementing a map layer that insulates the algorithm developer from writing complicated message-passing code. These libraries introduce the concept of map independence—that is, the task of mapping the program onto a processing architecture is independent from the task of algorithm development. Once the algorithm has been specified, the user can simply define maps for the program without having to change the high-level algorithm. The maps can be changed without having to change any of the program details. The key idea behind map independence is that a parallel programming expert can define the maps, while a domain expert can specify the algorithm.

Lincoln Laboratory has also developed an automatic parallelization framework, called pMapper, which is general with regard to programming languages and computer architectures and which focuses on distributing operations common in signal processing [6]. The pMapper framework globally optimizes performance of parallel programs at runtime. The underlying pMapper technology varies the amount of parallelism according to performance goals and is therefore particularly relevant both to the design evaluation pattern and also to the need for DoD applications to meet performance constraints.

DoD performance constraints act as forces on all patterns in the sense that they affect the way the patterns are used. For example, form-factor constraints in a DoD platform may constrain the number of processors used whereas the need for real-time performance may increase the number of processors. This balancing of forces is an often-repeated pattern in DoD applications, and should be incorporated into all defined parallel patterns, particularly design evaluation. Since they provide key technologies for achieving this goal, the Lincoln Laboratory efforts are directly relevant to pattern-oriented software development and future research efforts in this area would benefit strongly from a detailed examination of these technologies.

This work examined how parallel design patterns can be used on multi-core architectures to solve some of the software challenges posed by DoD applications. This study was performed in the context of a specific kernel and a representative architecture. The design patterns used in implementing the kernel on the architecture were analyzed with the goal of designing a set of portable, machine-independent abstractions to assist in parallel program development. Performance results for a pattern-oriented implementation of the kernel on the architecture were measured to show the viability of the approach. A key result of the work is the definition of requirements for pattern-based abstractions for future DoD software development efforts for multi- and many-core architectures.

4.5 Domain-Specific Languages (DSL)

In this section, an example application is briefly described that is implemented using the proposed DSL.

4.5.1 Complex Ambiguity Function (CAF)

CAF is a radar signal processing application that detects the time and frequency delay between two streams of data [16]. It consists of five stages: 1) a Hilbert transformation, 2) channelization into frequency bands, 3) threshold computation, 4) CAF space construction to produce time and frequency delay bins, and 5) peak detection.

4.5.2 CAF Implemented Using the Proposed DSL

We implemented an application, Complex Ambiguity Function (CAF), using the proposed DSL. Implementing CAF in the DSL reduced code size by more than a factor of 50 compared to a C implementation.

4.5.3 Coding Comparison

In this section, for illustrative purposes, we implemented an arbitrary code snippet using the proposed DSL and compared it with a C-language implementation. This demonstrates the easiness and conciseness of the proposed DSL. As a summary, the proposed DSL can describe the code snippet in two lines of source code, where MATLAB takes ten lines, and C takes 21 lines.

4.5.3.1 Proposed DSL

```
1 y1 = fft_r2c on imag(a) with fft-sizeof 512, input-overlapping 4;
2 z1 = fft_r2c on imag(b) with fft-sizeof 512, input-overlapping 4;
```

4.5.3.2 MATLAB

```
1 offset2 = 1;
2 for offset1 = 1:508:3000000
3         A = imag (input1(offset1: offset1 + 512));
4         C = imag (input2(offset1: offset1 + 512));
5         if (offset + 512) > 3000000
6                   A(512) = 0;
7                   C(512) = 0;
8         B(offset2:offset2+512) = fft(A);
9         D(offset2:offset2+512) = fft(C);
10        offset2 = offset2 + 512;
```

4.5.3.3 C language

```
1 for (i=0;i<nblks;i++) {
2       npts = blksize/2;
3        blkstrt = i*blksize - 2*spad;
4       blkstp = blkstrt + 2*nfft;
5        nstrt = 0;
6        nend = nfft;
7        if(blkstrt < 0) {
8          nstrt = spad;
9          memset(podd,0,nstrt*sizeof(double));
10           memset(peven,0,nstrt*sizeof(double));
11         }
12        else{
13           memset(podd,0,nfft*sizeof(double));
14           memset(peven,0,nfft*sizeof(double));
15         }
16         if(blkstp > 2*ndat) {
17           blkstp = 2*ndat;
18           nend = (blkstp-blkstrt)/2;
19           npts = (blkstp-blkstrt)/2 - spad;
20         }
21         for (j=nstrt;j<nend;j++) {
22           peven[j] = chirp[blkstrt+2*j];
23           podd[j]  = chirp[blkstrt+2*j+1];
24         }
25         fftw_execute_dft_r2c(pf,podd,fodd);
26 }
```

5 Conclusions

In this study, USC/ISI conducted exploratory studies to establish the need for and the value of innovative research on domain-specific architectures, applications, and tools based on the challenges posed by computational bottlenecks in DoD applications.

In the multi-core domain-specific architecture study, we developed chip area and performance models and used them to estimate performance for several kernels and an application. The results show that the performance highly depends on the design of a chip and to obtain high performance on domain-specific application, it is necessary to design the chip based on the characteristics of the target domain. Our results show that very high performance can be obtained using domain-specific architectures. Therefore, it is necessary to develop a particular type of processor for each target domain to obtain the high performance.

USC/ISI proposed a prototype emulation system that can be used for DoD applications readily. The system can be used for any type of a new processor for DoD applications and will provide orders of magnitude faster emulation time compared to a software simulation approach.

USC/ISI proposed prototype evaluation boards for three new processor architectures proposed for DoD applications. The three processor types are structured ASICs, patterns ASICs, and generated chips. A common evaluation system can be used for any type of these processors. Also, the system provides flexibility so that it can be used for wide variety of application environments.

A board generator is a tool to automate the design of an evaluation board for special-purpose architectures. The board generator will be based on a template design, which will make the automated design feasible and the results reliable. USC/ISI proposed a two-phase research program: in the first phase, a limited scope board generator is developed. The board generator will be robust enough for producing designs for structured ASICs, patterned ASICs, and generated chips. In the second phase, the board generator's scope is expanded to include more target processors or systems.

USC/ISI has evaluated design patterns for parallelism in DoD applications. A design pattern is a description of a tested and verified method of programming for specific problems. USC/ISI evaluated the potential benefits of research on design patterns for DoD applications by applying design patterns to exemplary kernels. The results show that design patterns can be very helpful for DoD applications. Therefore, it would be beneficial to the DoD to develop design patterns that address DoD specific needs.

USC/ISI evaluated domain-specific languages for DoD applications. Domain-specific languages are useful in boosting. We identified an area that can benefit significantly by using a domain-specific language. USC/ISI also proposed a strawman domain-specific language and showed its usability and effectiveness. We showed that the proposed domain-specific language can reduce the number of source lines of code by a factor of ten or more compared to the C language.

6 Recommendations

Domain-specific multi-core architectures are a critical component in mission-critical systems since they can provide high performances that cannot be achieved by general purpose architectures. However, design of these architectures is challenging. Therefore, USC/ISI recommends research efforts on architecture design including performance estimation tools, processor design tools including simulators/emulators and application development tools such as compilers, debuggers, and performance profiling/monitoring tools. We also recommend investment in run-time environments and hardware chip and system development. Although the target chip is domain-specific, much of the investment could be shared across multiple domains. For example, an emulation system could be shared by multiple domains.

Emulation systems are a very helpful tool in designing new chips/systems since it provides orders of magnitude faster emulation time compared with a software simulation system. The faster emulation time provides quick feedback to system designers who can reduce design time. USC/ISI recommends development of emulation systems that can be used for multiple DoD systems. The boards would be expandable to accommodate any reasonably sized processor-based systems. The emulation system would be accompanied by a supporting software tool chain environment with support function libraries.

If the structured ASICs, patterned ASICs, and chip generators are funded, evaluation boards for them should be developed as well. Therefore, USC/ISI recommends building a set of evaluation boards to support the chips. The evaluation boards would share common designs such that any chip can be populated on the developed board. Also, the board would provide flexibility, such that different configurations of the boards for different application domains can be configured. By developing the common evaluation boards, duplicated effort could be avoided.

We recommend that a board generator be developed to automatically generate evaluation and development boards. This work could be carried out in two phases, one with a more limited set of target chip interfaces.

Design patterns are a useful means to boost the productivity of programmers. Therefore, USC/ISI recommends research on providing design patterns for DoD relevant topics. In addition, USC/ISI recommends development of a design pattern repository system. The repository system is a central depot to collect design patterns developed for DoD applications such that programmers for DoD applications can easily access and use them for higher productivity. The depot could be divided into two areas: one publicly accessible and the other restricted.

Domain-specific languages are very helpful in achieving high productivity for programmers. The resulting codes are not only much shorter, but also closer to the abstraction level of the domain, which results in self-documentation and fewer bugs and opportunities for optimized implementation. USC/ISI recommends research on domain-specific languages for DoD applications. The research work would investigate DoD application areas that most need domain-specific languages; currently the commercial sector has little incentive to invest in

such research. Once the domains are identified, domain-specific languages would be proposed. Interpreters or compilers would be developed, and the languages and interpreters/compilers would be evaluated continuously with feedback from domain experts.

7 References

[1] B. Ackland, A. Anesko, D. Brinthaupt, S. J. Daubert, A. Kalavade, J. Knobloch, E. Micca, M. Moturi, C. J. Nicol, J. H. O'Neill, J. Othmer, E. Säckinger, Member, IEEE, K.J. Singh, Member, IEEE, J. Sweet, C. J. Terman, and J. Williams, "A Single-Chip, 1.6-Billion, 16-b MAC/s Multiprocessor DSP," IEEE Journal Of Solid-State Circuits, Vol. 35, No. 3, March 2000.

[2] Altera, "All Development Kits," http://www.altera.com/products/devkits/kit-dev_platforms.jsp, March 18, 2010.

[3] American National Standard Institute, "Programming Language Common List (ANSI/X3.226-1994)," 1994.

[4] J. Armstrong, "Program Erlang," Progrmatic Bookshelf, 2007.

[5] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "TILE64 Processor: A 64-Core SoC with Mesh Interconnect," Proc. IEEE International Solid-State Circuits Conference (ISSCC 2008), Feb 2008, pp. 88-98.

[6] N. Bliss, "Addressing the Multicore Trend with Automatic Parallelization," MIT Lincoln Laboratory Journal, Volume 17, No. 1, 2007.

[7] A. Bonelli, F. Franchetti, J. Lorenz, M. Püschel, and C. W. Ueberhuber, "Automatic Performance Optimization of the Discrete Fourier Transform on Distributed Memory Computers," Proc. International Symposium on Parallel and Distributed Processing and Application (ISPA), Lecture Notes In Computer Science, Springer, Vol. 4330, pp. 818-832, 2006.

[8] J. Brockman, P. Kogge, M. Niemier, and L. Pileggi, "Memory-based Structured Application Specific Integrated Circuit (ASIC) Study," Final Report to DARPA/IPTO and Air Force Research Laboratory, AFRL-RY-WP-TR-2008-1229, 2008.

[9] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," ACM SIGARCH Computer Architecture News, Vol. 28, Issue 2, May 2000.

[10] Cadence, "Inciseive Palladium Series," http://www.cadence.com/products/sd/palladium_series/pages/default.aspx, March 18, 2010.

[11] Cadence, "InCyte," URL: http://www.chipestimate.com, March 18, 2010.

[12] G. A. Carpenter and S. Grossberg, "ART 2: self-organization of stable category recognition codes for analog input patterns," Applied Optics, Vol. 26, Issue 23, pp. 4919-4930, 1987.

[13] W. Dally, J. Balfour, D. Black-Schaffer, and P.Hartke, "Structured Application-Specific Integrated Circuit (ASIC) Study," Final Report to DARPA/IPTO and Air Force Research Laboratory, AFRL-RY-WP-TR-2009-1042, 2009.

[14] C.M. DeLuca, C.W. Heisey, R.A. Bond, and J.M. Daly, "A Portable Object-Based Parallel Library and Layered Framework for Real-Time Radar Signal Processing," Proc. 1st Conf. International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE '97), pp. 241–248, 1997.

[15] DINI GROUP, http://www.dinigroup.com/, March 18, 2010.
[16] D. P. Enright, E. M. Dashofy, M. AuYeung, R. S. Boughton, J. M. Clark, R. Scrofano, Jr., "Multicore Acceleration of the Complex Ambiguity Function," HPEC 2008, Lexington, MA, Sept. 2008.
[17] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," Journal of Computer and System Sciences, 55(1):119-139, 1997.
[18] GiDEL, http://www.gidel.com/, March 18, 2010.
[19] George, "Novo-G: Reconfigurable Supercomputing for the New Decade," http://www.ece.ufl.edu/announcements/news/2010/Novo-G.html, March 18, 2010.
[20] M. Haines and W. Bohm, "Towards a distributed memory implementation of Sisal," Scalable High Performance Computing Conference, 1992.
[21] Hewlett-Packard, "CACTI," http://www.hpl.hp.com/research/cacti/, March 18, 2010.
[22] M. Horowitz, D. Stark, Z. Asgar, O. Azizi, R. Hameed, W. Qadeer, Ofer, Shachasm, M. Wachs, "Chip Generators Study," Final Report to DARPA/IPTO and Air Force Research Laboratory, AFRL-RY-WP-TP-2009-xxxx, 2008.
[23] HyperTransport Consotium, "HyperTransport™ I/O Link Specification Revision 3.10," http://www.hypertransport.org/docs/twgdocs/HTC20051222-00046-0028.pdf, March 18, 2010.
[24] Intel, "Intel© Core™2 Duo Processor E8000 and E7000 Series," June 2009.
[25] Intel, "VTune Performance Analyzer," URL: http://software.intel.com/en-us/intel-vtune/, March 18, 2010.
[26] Intel, "Intel QuickPath Technology," http://www.intel.com/technology/quickpath/, March 18, 2010.
[27] J. Kepner and N. Travinin, "Parallel Matlab: The Next Generation," Proc. High Performance Embedded Computing Workshop (HPEC 2003), Lexington, Mass., Sept. 23–25, 2003.
[28] J. Lebak, J. Kepner, H. Hoffmann, and E. Rutledge, "Parallel VSIPL++: An Open Standard Software Library for High Performance Parallel Signal Processing," Proceedings of the IEEE, Vol. 93, No. 2, February 2005.
[29] T. Mattson, B. Sanders, and B. Massingill, "Patterns for Parallel Programming", Addison-Wesley, 2005.
[30] J. Nayfach-Battilana and J. Renau, "SOI, Interconnect, Package, and Mainboard Theremal Characterization," International Symposium on Low Power Electronics and Design (ISLPED), August 2009.
[31] M. Odersky, "A Scalable Language," Free and Open Source Software Developers' European Meeting (FOSDEM 2009), February 2009.
[32] A. V. Oppenheim, R. W. Schafer, Prentice Hall, Discrete-Time Signal Processing, 1989.
[33] PCI-SIG, "PCIe Base Specification 2.0," http://www.pcisig.com/specifications/pciexpress/base2/, March 18, 2010.
[34] M.s Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," Proceedings of the IEEE, special

issue on Program Generation, Optimization, and Adaptation, Vol. 93, No. 2, pp. 232- 275, 2005.

[35] A. Shalloway and J. Trott, "Design Patterns Explained," 2nd Ed., Addison-Wesley, 2005.

[36] D. Stasiak, R. Chaudhry, D. Cox, S. Posluszny, J. Warnock, S. Weitzel, D. Wendel, and M. Wang, "Cell Processor Low-Power Design Methodology," IEEE Micro, Vol. 25, Issue 6, 2005.

[37] Texas Instrument, URL: http://focus.ti.com/docs/prod/folders/print/tms320c6474.html, March 18, 2010.

[38] P. Viola and M. Jones, "Robust real-time object detection," International Journal of Computer Vision, 2001.

[39] A. Kahng, B. Li, L.-S. Peh, and K. Samadi, "ORION 2.0: A Fast and Accurate NoC Power and Area Model for Early-Stage Design Space Exploration," In Proceedings of Design Automation and Test in Europe (DATE), Nice, France, April 2009.

[40] Wikipedia, URL: http://en.wikipedia.org/wiki/Domain_Specific_Language, March 18, 2010.

[41] Xilinx, "Boards and Kits," http://www.xilinx.com/products/boards_kits/index.htm, March 18, 2010.

List of Symbols, Abbreviations, and Acronyms

ART	Adaptive resonance theory
ASIC	Application-specific integrated circuit
BGA	Ball grid array
CAF	Complex ambiguity function
CMP	Chip-level multiprocessing
CPU	Central processing unit
DDR	Double data rate
DMA	Direct memory access
DoD	Department of Defense
DSL	Domain-specific language
DSP	Digital Signal Processor
FIFO	First in first out
FFT	Fast Fourier transformation
FGPA	Filed programmable gate array
FPU	Floating point unit
GPU	Graphic processing unit
HDL	Hardware description language
HP	Hewlett Packard
IBM	International business machine
I/O	Input/output
ISA	Instruction set architecture
JTAG	Joint test action group
LCD	Liquid crystal display
LED	Light emitting diode
LISP	List processing
MATLAB	Matrix laboratory
McPAT	Multi-core power, area, and timing
OS	Operating system
PAR	Place and route
PCB	Printed circuit board
PCI	Peripheral component interconnect
PVL	Parallel vector library
RAM	Random access memory
ROM	Read only memory
SLOC	Source line of code
SMP	Symmetric multiprocessing
SPL	Signal Processing Language
SPMD	Single program multiple data
SQL	Structured query language
STAPL	Space-time adaptive processing library
TM	Transactional memory
UPC	Unified parallel C
UE	Unit of execution
USC/ISI	University of Southern California / Information Sciences Institute

VHDL	VHSIC hardware description language
VHSIC	Very high speed integrated circuit
XAUI	X Attachment Unit Interface