AFRL-RI-RS-TR-2007-268
Final Technical Report
December 2007

ABSTRACT MACHINES FOR POLYMORPHOUS COMPUTING

University of Southern California

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. L171

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

NOTICE AND SIGNATURE PAGE

AFRL-RI-RS-TR-2007-268 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/	/s/
MARK NOVAK Work Unit Manager	WARREN H. DEBANY, Jr. Technical Advisor, Information Grid Division Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)* DEC 2007	**2. REPORT TYPE** Final	**3. DATES COVERED** *(From - To)* Jun 01 – Jun 07

4. TITLE AND SUBTITLE ABSTRACT MACHINES FOR POLYMORPHOUS COMPUTING (AMP)	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER FA8750-01-C-0171
	5c. PROGRAM ELEMENT NUMBER 62712E
6. AUTHOR(S) Stephen Crago	**5d. PROJECT NUMBER** AMPC
	5e. TASK NUMBER SN
	5f. WORK UNIT NUMBER 01
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Southern California, 10920 Wilshire Blvd 1200, Los Angeles CA 90024-6523	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency, 3701 North Fairfax Dr., Arlington VA 22203-1714; AFRL/RIGB, 525 Brooks Rd, Rome NY 13441-4505	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-RI-RS-TR-2007-268

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. WPAFB PA# 07-0580

13. SUPPLEMENTARY NOTES

14. ABSTRACT
The Abstract Machines for Polymorphous Computing (AMP) project made important contributions to the understanding of the programming of polymorphous computing architectures. Polymorphous architectures are a promising technology for exploiting explicit on-chip parallelism, which the microprocessor industry has recognized is necessary for the survival of the industry. The AMP project developed software tools to explore programming methodologies for polymorphous architectures, including Morphware. The AMP project mapped application to polymorphous architectures and demonstrated significant, scalable speedups. Finally, the AMP project developed prototype hardware that enabled the development of realistic and complex applications for polymorphous architectures

15. SUBJECT TERMS
Multiprocessor architecture, polymorphous computing

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Mark E. Novak
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UL	57	**19b. TELEPHONE NUMBER** (*Include area code*) N/A

Table of Contents

List of Figures

1. Introduction

Polymorphous computing elements provide very high performance across a range of applications by providing morphable microarchitectures. The increased level of programmability available through morphing has become necessary to overcome potential bottlenecks to performance and to improve efficiency. As feature sizes shrink, the level of microparallelism increases, and off-chip communication becomes more expensive. While allowing more parts of the microarchitecture to be programmed increases performance potential across a wide range of applications, it also has the potential to increase the complexity of programming polymorphous devices. The goal of the AMP project is to provide a programming environment for programming polymorphous architectures than allows the performance potential to be exploited while minimizing programming overhead. Goals of the AMP project also include demonstrating end-to-end application performance on polymorphous computing architectures and developing board and system-level hardware prototypes.

The original focus of the AMP project as proposed was to develop abstract models for programming polymorphous architectures. The abstract machine models were to allow an application to be expressed in a form that is natural to the application domain and to allow an efficient mapping to polymorphous computing architectures. An abstract machine model represents the view of a computer architecture or class of architectures that is presented to the application programmer or compiler. The purpose of the abstract machine is to present a simplified, yet accurate, view of the architecture to the programmer. While an abstract machine does present a view of the architecture to the programmer, an abstract machine can have more than one hardware implementation. Figure 1 shows an Abstract Machine B, which is mapped to Hardware Machine 1 and Hardware Machine 2.

Figure 1 also shows that different applications can be mapped to different abstract machines, even on the same (morphable) hardware architecture. Different applications domains are naturally best expressed using different abstract machine models. Streaming applications are best described using abstract machines that support streaming, and data parallel programs are best described using abstract machines that support SIMD or vector abstractions. The flexibility of polymorphous computing architectures allows the same hardware architecture to implement different abstract machines.

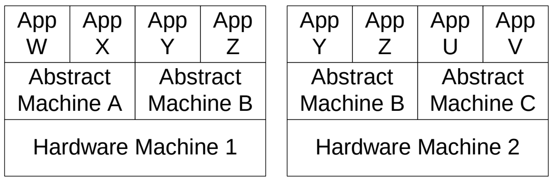


Figure 1. Abstract Machine Models

In a performance-critical environment, the abstract machine must satisfy two additional properties. First, it must allow the application to be expressed so that properties of an application that the architecture can exploit are not lost. For example, existing parallelism should not be hidden if an architecture is to exploit parallelism. Second, the abstract machine must allow the sophisticated application programmer the ability to control performance-critical aspects of the program execution on the hardware. Because many embedded systems require predictable computing performance, the abstract machine cannot be so abstract as to make the performance of an application unpredictable.

DARPA's PCA (Polymorphous Computing Architectures) program formed a group called the Morphware Forum, consisting of PCA contractors and government representatives, to agree on programming interfaces for the PCA architectures [1]. The Morphware Forum was given the charter to define, in a consensus process including all PCA participants, stable interfaces (abstract machines) to be used for all PCA architectures. The Morphware Forum agreed to a two-level compilation approach, which defined one standard interface at the application specification level and another as an intermediate compiler level. The SAPI (stable application programming interface) defined a common programming interface used by the high-level compiler to target PCA architectures. The SAAL (stable architecture abstraction layer) defined a back-end for the high-level compiler, which exposes parallelism data movement in an architecture-neutral way. This SAAL includes a model for multi-threaded applications, called the Thread Virtual Machine (TVM), and another model for streaming applications, called the Stream Virtual Machine (SVM). These virtual machines are instantiations of the abstract machines envisioned by the AMP project.

The formulation of the Morphware Forum and its products, which the AMP project participated in, allowed the AMP project to shift its focus to contribute complementary technologies for the PCA program. In addition to the hardware prototyping work and application performance analysis and demonstration work that was part of the original charter of the AMP project, the project focus shifted toward experimentation with low-level communication implementations for PCA architectures, which were used to implement the Morphware SAAL and performance analysis and optimization of the Morphware SVM. Work in the first phase of AMP also included a modeling framework used to investigate run-time morphing capability. Because the MIT Raw platform [2] was much more mature while much of the AMP development was conducted, most of the work on the AMP project was conducted on the Raw architecture.

The abstract machines described above allow mapping to multiple PCA nodes, but are focused on static compiler-driven mapping. To enable construction of effective application development environments for dynamic systems, we developed an application modeling framework and hierarchical verification methodology for robust design of polymorphous systems [3]. The modeling framework, called the Configuration Modeling Framework (CMF), provides a platform-independent approach to incorporating run-time morphing capability into application development environments. This capability leads to automated co-synthesis of parameterized hardware and software structures and their configurations; synthesis of run-time configuration management software; and hierarchical, formal verification of key configuration properties, such as communication consistency requirements, deadlock avoidance, and parameter range validity. Our verification approach is based on systematic decomposition of execution into local regions of time in which subsystems satisfy relevant properties. By evaluating compositions of these properties through hybrid static and run-time techniques, the entire polymorphous system can be verified so that errors are detected as soon as they occur.

The Navy Radar System Signal Processing application is a large system, multi-mission problem that will enable Theater Ballistic Missile Defense (TBMD) capability to the

Navy. The AMP project included a system prime, Lockheed Martin, which provide representative high-level application code and evaluation of AMP software. The US Navy surface combatants face a new set of challenges over the next few decades. A Navy designed years ago for open ocean combat now finds itself confronted with combat situations in the littoral waters across the globe. With this new mission comes a new set of threats. Small, stealthy and fast threats at low altitude as well as ballistic missile threats at high altitude that require a new degree of discrimination. These new threats place a new and significant load on the shipboard sensors that are required to detect, track and classify across the threat space and motivate polymorphous computing architectures.

The Sanders Polymorphic Channelized Receiver (PCR) is an implementation of polymorphic computing that has application to real-time resource constrained platforms, in particular space and airborne platforms performing RF receiver tasks such as EW or communications. The PCR morphs into different channelized receiver configurations in real time to improve the probability and fidelity of detection. Based on a set of primitives, the PCR tailors its receiver configuration in terms of the quantity of channels, channel bandwidth, and channel center frequency to optimally address the signal of interest. Each channel can have a unique configuration, applying a tailored set of frequency components based on the signal quality, providing the opportunity to attenuate or eliminate those components attributed to noise that could degrade the output quality. An efficient architecture for this class of problem has application far beyond the RF receiver user community. For example, a potential application of the PCR structure in a spatial domain (rather than in the frequency domain) results in a unique adaptive beamformer. BAE SYSTEMS participated in the AMP project to provide feedback to ensure relevance of AMP technology the PCR.

The AMP project also developed system-level prototype hardware, primarily for the Raw architecture. The development of system-level prototype hardware serves several purposes. First, it allows a validation of the design principles and performance metrics for the architecture under development. Second, it permits the execution of realistically sized benchmarks, which would be infeasible to run on a simulator. Third, it provides a platform on which technology demonstrations can be developed. The hardware developed by the AMP project achieved all three of these objectives.

The AMP team developed boards that could be tiled together to create a 64-node multiprocessor, called the Raw Fabric, based on the MIT Raw processor. The AMP team mapped DoD applications to the Raw chip and Raw Fabric. Having an independent team map DoD applications to the Raw architecture provided several advantages. First, and most importantly, the MIT Raw team did not focus on DoD applications. The second advantage is that having an independent team use the Raw software and architecture provided objective feedback to both the MIT team and DARPA about the Raw architecture. This additional feedback contributed to the ultimate success of the Raw architecture.

In addition to the primary body of work the AMP project did under the PCA program, there were two supplemental parts of the project, added by other programs. First, the AMP team performed performance analysis for the Knowledge Aided Signal and Sensor Processing with Expert Reasoning (KASSPER) program [5]. Second, the AMP team developed concepts that contributed to the formulation of the Architectures for Cognitive Information Processing (ACIP) program.

In this final report, we describe the work completed by the AMP project. Section 2 discusses the software tools developed by the project and the application mapping results. Section 3 discusses the hardware prototyping. Section 4 describes the work the AMP project did for KASSPER, and Section V discusses the work the project did laying groundwork for ACIP. Finally, Section 6 summarizes the conclusions.

2. Software

a. Communication Application Programming Interface

The AMP project developed a toolset and application programming interface (API) to help automate programming the complex inter-tile communications on Raw. In order to make the programming and debugging inter-tile communication easier, the AMP team developed: 1) a programming model, which enables the programmer to describe and debug the communication pattern in an architecture-independent way, and 2) a toolset, which schedules the inter-tile communications and generates switch and tile codes automatically.

The purposes of the API and the tools for the static network are as follows:

- Provide a higher-level message passing interface to Raw programmers (at the expense of some performance)
- Ease debugging for multi-tile applications
- Automatic partitioning and routing of inter-tile communications into a sequence of sets of collision-free communications
- Automatic generation of switch code and transparent integration of switch and tile code

An overview of the AMP approach is shown in Figure 2. Our approach consists of two phases, a PROFILE phase and RUN phase. The API for the static network provides the toolset with information about inter-tile communication in the PROFILE phase and configures the Raw switch processors in the RUN phase.

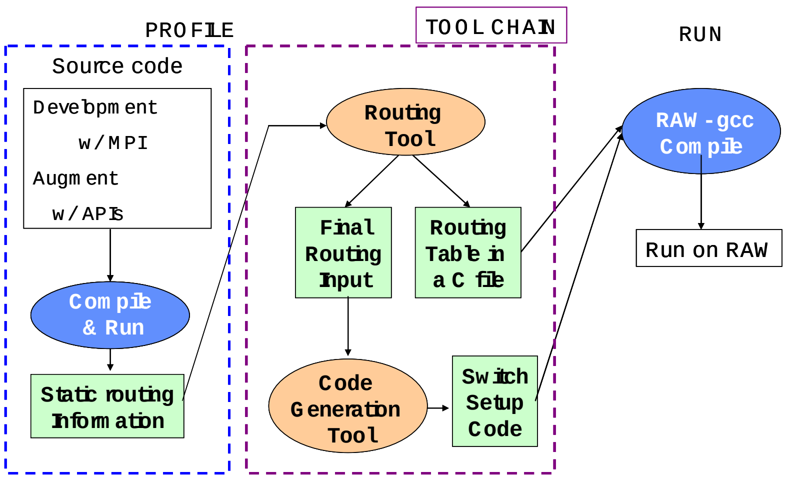


Figure 2. Communication API Overview

Each inter-tile communication is described with four parameters: a unique communication identification number, source tile number, destination tile number, and stage number. The stage number allows ordering to be enforced upon inter-tile communications when there are data dependencies. The end of a stage acts as a barrier synchronization point. Communications within a stage do not have any dependencies and can be executed in parallel. It is the application programmer's responsibility to group the communications into stages and to specify the sequence of the stages. The toolset checks the schedulability of the communications in a stage and partitions the stage into schedulable sub-stages.

In the profile stage, inter-tile communication is profiled by the communication library, which implements the communication API on MPI (Message Passing Interface) [4]. The user can program and debug the program using MPI, which makes debugging easier because the MPI/Linux programming environment is more mature than the Raw programming environment. This stage can be carried out on any machine where MPI runs. For example, an x86/Linux workstation with MPI can be used to compile and execute at the profile stage.

Profile information is used by the routing tool to schedule the communication and to generate routing information about the communication. The routing tool also generates a routing table and includes it in the C header files. In compiling, "ROUTE_MODE" is assumed to be declared in the code, which makes conditional compilation possible. Any profile-specific code can be added by using a conditional compilation statement in the source code.

The original code can then be cross-compiled and linked by the Raw cross-compiler with switch code and the routing table to generate the Raw executable code. Any run-phase-specific code can be added using a conditional compilation statement "#ifdef RUN_MODE" in the source code. Code changes from profile phase to run phase are minimal, but hardware-specific routines and actual communication routines need to be changed. Some examples are summarized in Table 1.

PROFILE phase	RUN phase

`MPI_Comm_rank();`	`raw_get_abs_pos_x();`
	`raw_get_abs_pos_y();`

`MPI_SEND()`	`static_send_from_mem();`
`MPI_RECV()`	`Static_recv_to_mem();`

Table 1. PROFILE and RUN Phase Code Differences

For programming large applications on Raw, it is difficult to write switch and tile code manually that utilizes the static network on Raw. The AMP project developed a model for programming applications such that switch and tile communication code could be generated automatically. This model is implemented with the *route2switch* tools. The *route2switch* tools need a routing pattern for the communications involved in the application as input. There may be some ordering enforced upon inter-tile communications in the application because of data-dependency or resource contention. Therefore, the routing pattern for communications in the application may be divided into several stages, where these stages act as barrier synchronization points. So in the application code, we are assured that if in the routing pattern a tile X appears in stages p and q such that $p < q$, then, in tile X's code, stage p code should be executed before code for stage q code.

For large applications, manually finding a feasible routing arrangement for all communication can be a very difficult task and is prone to human error. The AMP routing tools can take all the communications in the application, which may be divided into stages as an input, and then route them so that the resulting routing pattern does not deadlock and is acceptable.

Routes for several communications are acceptable when all the tiles involved in the routes can be programmed so that each tile in the communication pattern receives or sends the desired data. At each stage, a communication circuit monopolizes the links (but not the switches) in the path from the source tile to the destination tile. Therefore, the AMP routing tool must resolve resource contention within stages. For example, in a 4x4 tile system, the routing pattern of 0→1→2 and 1→2→6 is not an acceptable routing pattern for communications 0→2 and 1→6 because the link from 1→2 is overutilized (time multiplexing on the links is supported by the architecture but beyond the scope of the AMP tools). An acceptable routing pattern for the same set of communications may be 0→1→2 and 1→5→6. If the input to the routing tool is a set of communications such that some communications in a stage cannot be resolved, then the routing tool can split one stage into several stages.

There are two versions of the router that make different trade-offs of optimization versus the solution-execution time. The version with lower run-time and less optimization works by first making repetitive passes through all the still unrouted communications in the current stage in the input. When all the communications in a stage are routed, the next stage is read and routed in a similar fashion. A pass involves trying to route the first still un-routed communication in the stage and, if it is routable, then committing the route found for it. The next communication in the current stage is then read, and the router tries to find a route so that it can be routed in the same stage as the first one. If it not possible, that communication is thrown into the list of unrouted communications. The next communication is then read and processed in a similar manner and so on until all the communications in the current stage are exhausted. If unrouted list is not empty, the current stage is then initialized and the process is continued until the list is empty. A new stage from the input file is then read into the current stage. There is no backtracking in this algorithm and the route for a communication is not changed once it is committed. This leads to faster execution time. The complexity of the algorithm increases linearly as the number of communications increases.

The longer-running version of the router exhaustively tries to route all the communications in a single stage of the input file into a single output stage. If no routing pattern exists where this may be possible, it partitions the input stage into two stages of equal size and then tries to exhaustively route all in each of these two stages so that the output is only two stages. The stage that is not fully routable is again partitioned and this process is repeated. There is a lot of backtracking involved in exhaustively finding the optimal solution and this leads to high execution times. The complexity of this algorithm is exponential with the number of communications.

The AMP project also developed an API for using the Raw dynamic network. This API and library are much simpler, because routing is done automatically by the hardware. The industry-standard MPI communication library also provides communication on multiprocessors with dynamic routing, but provides higher level functionality. This higher level functionality incurs additional overhead and a more sophisticated implementation that may not be appropriate for a single-chip multi-tile architecture like Raw, which can provide much lower latency communication, which motivates lower latency message passing software. Table 2 shows some of the trade-offs for various communication methods, including writing fully manual, optimized code and the static and dynamic communication APIs discussed above.

	Fully Manual	**Static Comm. API**	**Dynamic Comm. API**
Application Partitioning	Manual	Manual	Manual
Network Used	Static Net.	Static Net.	Dynamic Net.
Communication Scheduling	Manual	Automatic by the routing tool	No scheduling
Integration of computation / communication code	Loose	Tight	Tight
Tools used	Routing tool, Switch code generator tool	Routing tool, Switch code generator tool	Dynamic network library
Dynamic communication	No	No	Yes
Communication performance	Fastest	Fast	Not as fast

Table 2. Communication API Trade-offs

Figure 3 summarizes the performance of the communication APIs compared to manual static coding on Raw. In manual static coding, the Raw tile switches can be pre-programmed with a known interconnect pattern, and network operands can be referenced without overhead, using network register names as operands in arithmetic operations. By

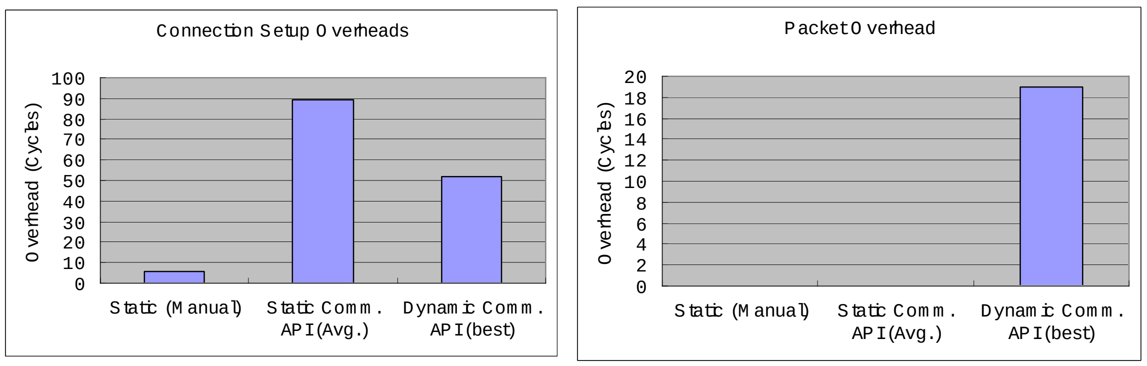


Figure 3. Communication API Overhead Summary

comparison, the static communication API incurs higher overhead to set up a given communication pattern, which is primarily due to the need to check on where the global application is before doing communication to avoid interfering with communication that might be passing through a given node. The table look-up to determine the current location can take a cache miss, which causes higher overhead. The static communication API incurs no overhead on each packet, because no header is needed since the communication pattern is programmed into the switch, just as in the static manual case. The dynamic communication API must synchronize between communication tiles before initiating communication, but this communication is always on-chip, so it is less than the setup overhead for static communication. The dynamic communication API necessarily incurs overhead for each packet, because the header, which specified the destination and packet length, must be set up for each packet. While this per packet overhead appears high relative to static communication, it is orders of magnitude lower than the per packet overhead incurred by a heavyweight message passing API like MPI.

Figure 4 shows the throughput of the communication APIs. As expected, manually optimized static communication has the highest throughput, achieving over 50% of peak throughput for packets of less than 10 words and near peak for packets of above 100 words. The static API performance is very close to dynamic API performance for packet sizes below 500 words because of the setup overhead. However, for large data transfer sizes, performance approaches that of manually optimized static code.

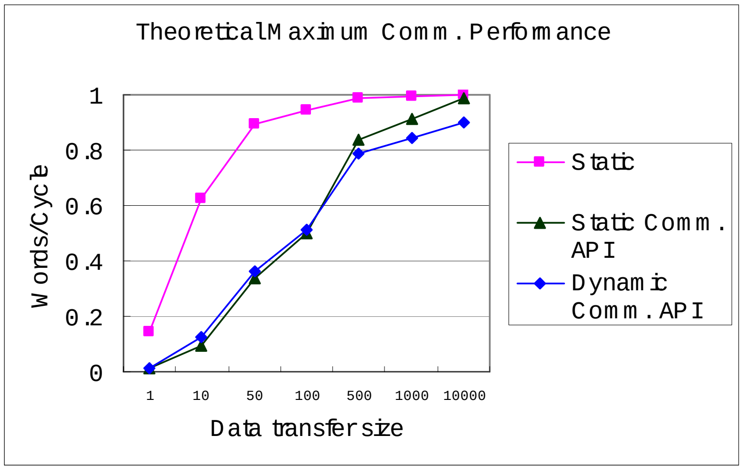


Figure 4. Communication API Throughput

In order to test the performance of the communication of APIs on real applications, we implemented a corner turn and a distributed FFT (Fast Fourier Transform) using the communication APIs and compared them to hand-optimized static mappings. Figure 5 shows the normalized performance for the corner turn kernel for three data sizes using hand optimization, the static API, and the dynamic API. As expected, performance for the manually optimized code is the highest, which is shown in the blue bar, normalized to execution time of 1. Execution time for the static communication API is between 23% and 73% higher, with smaller normalized execution time for larger data sets, since the overhead of setting up the communication paths is amortized over more words. Most of the performance disparity between the static API and manually optimized code for large data sets is due to the non-optimal communication pattern discovered by the router. This non-optimality is a function of the run-time of the router itself; it is infeasible to allow the router enough execution time to find guarantee that the optimal communication pattern is found. Execution time for the dynamic communication API is about three times that of the optimized code. While this performance disparity between the dynamic communication API and manually optimized code is significant and would be unacceptable for many applications, corner turn is a communication-only kernel with no computation to overlap for the computation. So in that sense, the performance overhead incurred by the APIs represents an upper bound for a real application. This upper bound is not absolute since pathological communication patterns could incur higher overhead.

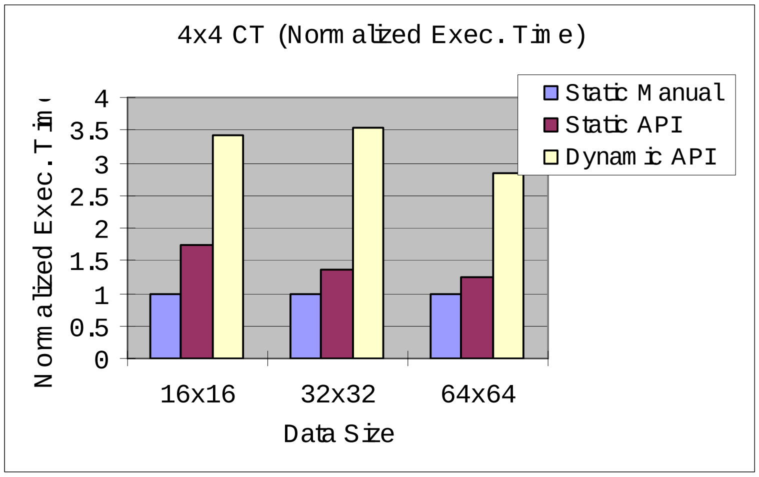


Figure 5. Communication API Performance for Corner Turn

To measure the performance of the communication APIs on more realistic mixes of computation and communication, we also implemented a distributed FFT. Results are shown in Figure 6, which shows that there is very little performance difference between communication methods for the FFT, which is relatively compute intensive, but is also an important kernel central to many embedded signal processing applications. Performance varies by less than 25%, which the largest variation on the smallest FFT size, where the overhead is highest.

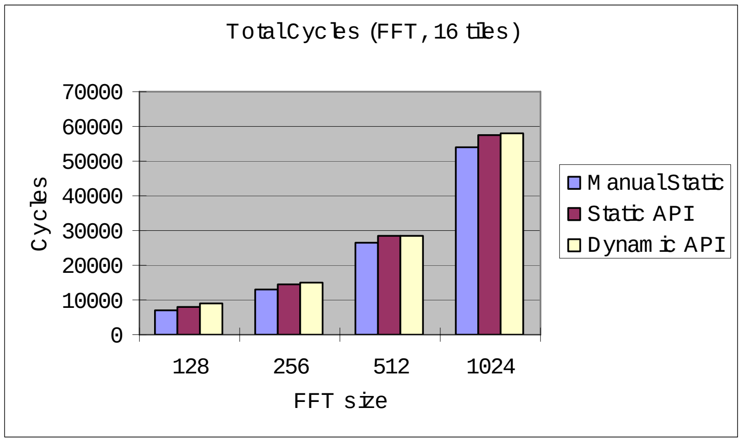


Figure 6. Communication API Performance for FFT

b. Configuration Management Software

(1) Configuration Management Overview

It is useful to view polymorphous computing architectures as having configurable attributes of the architecture and software that are adapted in response to dynamically changing needs. Such attributes may include items such as inter-processor message routing, caching policies, scheduling policies, processor voltages, resource allocation to computing units, and synchronization protocols. A polymorphous computer can be a particularly useful platform for developing a computing system where applications and performance requirements change at run-time as one can adaptively configure the architecture to suit the dynamic constraints and objectives.

This work takes a step towards bridging techniques for scheduling and system synthesis with reconfigurable processing platforms and the dynamically-changing application requirements that drive these platforms. The AMP team first formulated the problem of executing application dataflow graphs on a polymorphous computing architecture such that specified performance requirements are satisfied, where the requirements may vary over time and the application may have tasks with non-deterministic execution times (e.g., due to data dependencies or unpredictable events such as cache misses and interrupts). We analyzed key properties of this problem and the complexity of some relevant sub-problems. We then developed a flexible heuristic framework for guiding the run-time configuration adaptation process, and show through simulation experiments that this approach can efficiently handle both dynamics in performance requirements and dynamics in task execution time behavior.

In the application model addressed in this work, computational tasks (actors), which are represented by dataflow graph vertices, in the application are allowed to have stochastic execution times with static distributions or distributions that may vary slowly over time. The computing unit is a reconfigurable multiprocessor architecture, and the objective is to find a mapping of the actors in the application onto the processors in the multiprocessor and the configuration that the architecture should assume such that performance-related constraints (e.g., constraints on power, resource usage or throughput) are satisfied and objectives (e.g., maximizing throughput or minimizing latency) are optimized effectively. Furthermore, the constraints and objectives may vary over time, and thus, overall solution quality can be viewed in terms of how efficiently reconfiguration of the architecture tracks changes in the application's requirements. Henceforth, we will refer to this problem as the polymorphous mapping problem. As can be seen, the polymorphous mapping problem is quite general in nature and even very restricted special cases can be proved to be NP-complete.

The approach suggested in this work is correspondingly general and can handle diverse applications and performance requirements. All the reported experiments were performed on an abstraction of the Raw architecture that incorporates salient features of the architecture such as the programmability of interconnects between processors. For experiments, the self-timed execution of applications on this abstracted Raw architecture was simulated using the inter-processor communication (IPC) graph model [6].

The emphasis in this work is on coordination of the on-line configuration management process for reconfigurable networks of processors, rather than the development of specialized configuration optimization techniques (such as fixed-objective scheduling and allocation) [7]. Our work is complementary to such existing efforts and also to work on multiprocessor system synthesis [8] [9], which can be used to derive the store of pre-computed configurations that is input to the techniques developed in this work.

(2) Configuration Management Problem Formulation

A set of relevant metrics, such as latency, throughput, average power, peak power, and number of resources, is denoted by *M*. If a certain metric appears as a constraint with a value to be satisfied when the application executes, then this metric is referred to as a *constraint metric* and the value as a *constraint value* for that particular metric. A constraint value belongs to the set of real numbers. A pair of constraint metric and constraint value is called a *constrain pair*. A sequence of constraint pairs in turn is referred to as a *constraint vector*, and is denoted by

$$V = [(m_1, c_1),(m_2, c_2),\ldots,(m_K, c_K)]$$

where m_1, m_2, …, m_k represent any K metrics in M, and c_1, c_2, …, c_K represent the corresponding constraint values, for $K \in \{0,N\}$, where N is the number of all constraint pairs. This (possibly empty) sequence of constraint pairs in a constraint vector is prioritized such that (m_i, c_i) is a higher priority constraint pair than a constraint pair (m_j,c_j) if $i < j$, for $i, j \in \{1, 2, \ldots, K\}$ in a constraint vector $V = [(m_1, c_1),(m_2, c_2),\ldots,(m_K, c_K)]$. A metric m_R that is to be optimized after all constraints have been satisfied is called a *residual objective*. A goal *g* is an ordered pair (V, m_R), where *V* is a constraint vector and m_R is a residual objective. If there is no residual objective, then the goal is composed of only a constraint vector and can be represented by $(V,\ )$. Here, the symbol   represents the absence of a residual objective. Also, without loss of generality, the metrics are such that the associated optimization problems are to minimize the metric (i.e., a lower value of a metric is always better than a higher value). Metrics for which higher values are more desirable must thus be transformed into corresponding metrics for which lower values are better. For example, in iterative applications, the throughput (average rate of completion of application iterations) can be re-cast as the average iteration period, which is the reciprocal of the throughput.

Example 1: Consider a set of relevant metrics $M=\{L,P,T\}$, where L is the latency, P is the average power consumption, and T is the iteration period. Consider the goal $g=[(L, 50),(P, 100),(L, 40),(P, 70), T]$. In g, the constraint pair $(L, 50)$ has higher priority than the constraint pair $(P, 100)$, which in turn has higher priority than the constraint pair $(L, 40)$. The metric T is the residual objective.

This definition of reconfiguration goals as prioritized lists with optional residual objectives leads to a view of dynamic reconfiguration as a sequence of one-dimensional optimization problems. This simplification is useful because run-time adaptation techniques must be of relatively low complexity, and thus, one-dimensional optimization is a better match. Additionally, it allows us to leverage existing libraries of single-dimensional synthesis techniques, which are more abundant than multidimensional techniques. Third, it provides an intuitive and unambiguous format for designers to prioritize multidimensional application requirements. Note, however, that this formulation applies only to run-time reconfiguration, and multi-dimensional optimization techniques, such as Strength-Pareto Evolutionary Algorithm-based methods [10], can be used off-line in arbitrary ways to compute caches of pre-computed configurations. Use of such caches will be discussed further below.

For example, in Example 1, we initially have an unconstrained latency optimization problem (since the first constraint involves latency). As we adapt the system configuration with techniques that address this problem, we will in general improve the latency. Once the latency improves to 50 time units, the current constraint is satisfied, and we switch to a power-optimization problem subject to a constraint of $L = 50$. The optimization process may continue in this manner until the last constraint is satisfied (in this case, $P = 70$), at which point run-time adaptation stops (if there is no residual objective) or reaches a terminal mode of optimizing the residual objective subject to all constraints in the constraint vector. This mode then continues until the system shuts down or the application's goal changes.

Mapping an application to a multiprocessor architecture includes defining a task-to-processor mapping along with defining the configuration of the reconfigurable architecture. In this paper, the scope of the word "configuration" is expanded to include also the mapping of the application onto the reconfigurable architecture. Therefore, a configuration consists of two components 1) task-to-processor mapping and 2) configuration of the architecture. Henceforth, the word "configuration" is used in the above sense, unless stated otherwise. A given application, goal, and resource set define an instance of the PCA mapping problem. Input to the model is an instance that may change with time. We define the design space as the set of all feasible combinations of an instance and a configuration. The solution space for a feasible instance is the set of all feasible configurations for that instance. Latency, throughput, average power and peak power are some of the commonly encountered metrics. With many metrics of simultaneous relevance, the goal space is too vast to be fully explored before run-time, and run-time adaptation of configurations is generally advantageous.

Figure 7 illustrates a general model for solving the PCA mapping algorithm with a combination of off-line and on-line techniques. The main components of the model are the off-line component, the configuration store (CS), and the on-line component. The off-line component, whose objective is to pre-compute a set of efficient candidate mappings for various run-time scenarios, can be constructed using existing methods for scheduling, system synthesis, and multi-objective optimization. The focus of this paper is thus on the on-line refinement component and its interaction with the configuration store.

For a given instance, not every configuration is suitable as some configurations may violate constraints or may not adequately address residual objectives. As the goal changes for a given application, the system needs to derive a suitable adaptation of the run-time configuration. Optimally solving this problem is undecidable in many contexts. Also, reconfigurability of the architecture and the stochastic variance of execution times greatly complicates the solution space consisting of all possible configurations for the input of a goal and a given application. Since computing a suitable configuration is performed during the execution of an application, one can not apply exhaustive or relatively sophisticated search strategies as those techniques will take away excessive computational resources away from the application itself. To address this trade-off (thoroughness of dynamic optimization vs. resources drained from the application), our model of the PCA mapping problem also accounts for the time spent in computing efficient adaptations of mappings at run-time on the basis of feedback obtained from execution and identification of bottlenecks, and hence always tries to move towards an optimal solution. This is taken care of in the on-line refinement part of the model, which consists of low-complexity algorithms that find and refine configurations for a given instance. It also consists of feedback units shown by the "Identify bottlenecks" block in Figure 7 that takes feedback from the execution of the configurations and modifies the configurations so as to better suit the active goal. The OnlineStats unit in the on-line refinement part of the model stores short-term statistics that can be used by on-line algorithms.

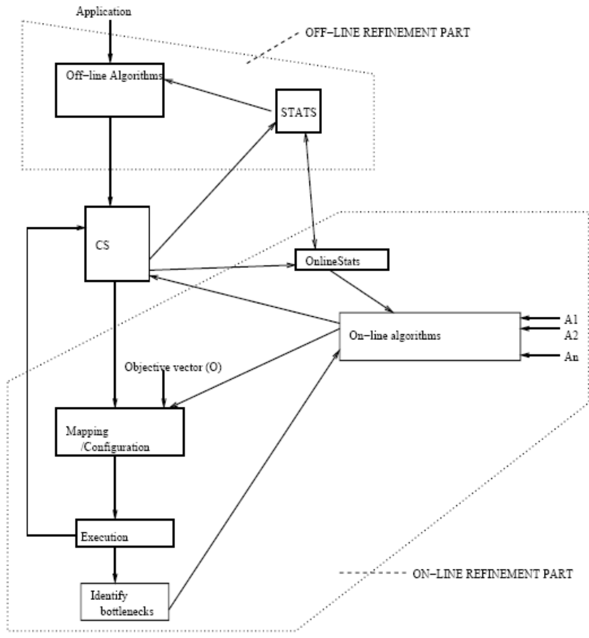


Figure 7. System-Level Reconfiguration Framework

(3) Configuration Management Model

Our PCA system synthesis model is very adaptive in nature and hence is suitable for applications with stochastic execution times and time-varying goals. A configuration store serves as a repository of alternative configurations. A configuration store can be divided into several sub-stores (sub-CSs), one for each relevant application. Each sub-CS has some configurations stored in it, one for a specific combination of goal and resource set. In the later part of this section, we assume that we are dealing with a fixed application and a fixed resource set, unless stated otherwise. This does not detract from the generality of the ideas developed later as they can be generalized to include various applications and resource sets using the hierarchical model of configuration store explained above. Assuming a fixed application and resource set, selecting the goals whose corresponding configurations should be stored in the configuration store depends on various factors such as the size of the configuration store; the optimality of the stored configuration; computational resources drained from the application during execution by the on-line refinement algorithms; and the expected or observed frequency of specific goals.

Notions of *acceptability* and *cover* emerge naturally from this concept of configurations stores, and guide the construction and adaptation of the configuration store in our model. For example, one can envision the reconfiguration process as selecting an acceptable configuration, and gradually tightening the notion of acceptability to guide the on-line refinement process.

In this section, we define an on-line configuration management framework called *CMF* that defines how to choose an initial configuration for a particular instance, and how the on-line adaptation for that configuration should proceed. We also formulate problems related to storage of configurations in the configuration store. These problems and our models to solve them provide fundamental analysis of the complexity of configuration management and provide feasible, low-complexity solutions to this problem.

```
function CMF

/* Global variables accessed: the current goal and the set
of pre-computed configurations, respectively.
*/

global goal $g_c = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K), m_R]$
global configurationStore $C$

stack $S$ = emptyStack ;
goal $g, g_o$;
$g = g_o = g_c$

/* The current optimization metric and the current
constraint to satisfy
*/
objective $objective_c = m_R$
constraint $constraint_c$ = null

$\sigma = \{c \in C | c \text{ satisfies } g\}$
while ($\sigma = \emptyset$) {
        ($g$, $constraint_c$, $objective_c$) = demoteConstraint($g$, $S$)
        $\sigma = \{c \in C | c \text{ satisfies } g\}$
}

/* Select an admissible configuration from σ using some
heuristic or specialized, optimal algorithm.
*/
$configuration_c$ = select($\sigma$)

/* Keep trying to refine the current configuration accord-
ing to the current goal until the goal changes ($g_c$ may
change at any time under external control).
*/
while ($g_c = g_o$) {
        while ($constraint_c$ is not satisfied by $configuration_c$) {
                onLineAdaptation($objective_c$, $constraint_c$,
                                 $configuration_c$)
        }
        /* Move to the next unsatisfied constraint or to
        the residual objective */
        ($g$, $constraint_c$, $objective_c$) = promoteConstraint($g$, $S$)
}
end function
```

Figure 8. Configuration Management Framework

A pseudocode outline of the CMF approach is shown in Figure 8. The objective is to provide a framework that imposes minimal constraints on how reconfiguration is actually performed, while providing systematic support for managing the reconfiguration process in terms of configuration stores, performance constraints, and optimization objectives. CMF is a meta-algorithm because specific details of the architecture, the application, and the on-line adaptation algorithms are left unspecified, and can be customized based on the relevant classes of applications and architectures. This meta-algorithm maintains a *current objective* at all times, where the goal is always to improve the current objective without violating any of the previously satisfied constraints. The function *onLineAdaptation* takes an objective metric, a constraint value, and a configuration as inputs, and keeps refining the configuration in an effort to continually improve its quality. This function would typically be called within an enclosing loop that performs any system-dependent re-initialization and re-invokes the function immediately after the previous invocation of the function terminates (observe that the function terminates when the current goal is changed). Pseudocode for the related functions is given in Figure 9.

```
function promoteConstraint
input goal $g = [(m_1, c_1), (m_2, c_2), \ldots, (m_{K-1}, c_{K-1}), m_K]$, stack $S$
output goal, constraint, objective
goal $g'$
constraint $v = S.pop()$
objective $m = S.pop()$
constraint $x = S.pop()$
$S.push(x)$
$g' = [(m_1, c_1), (m_2, c_2), \ldots, (m_{K-1}, c_{K-1}), (m_K, v), m]$
return ($g'$, $x$, $m$)
end function


function demoteConstraint
input goal $g = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K), m_R]$, stack $S$
output goal, constraint, objective
goal $g'$
$S.push(m_R)$
$S.push(c_K)$
$g' = [(m_1, c_1), (m_2, c_2), \ldots, (m_{K-1}, c_{K-1}), m_K]$ .
return ($g'$, $c_K$, $m_K$)
end function
```

Figure 9. Functions *promoteConstraint* and *demoteConstraint* from Figure 8

Before proceeding with discussion of our experiments with CMF, we first studied some fundamental versions of the problems related to configuration management, their complexity, and related aspects of them to well-studied problems. Two related problems regarding the size of the configuration store are as follows.

P1. Find the minimum size configuration store and the goals that should be stored in it such that all the relevant goals are covered.

P2. If one has a well-defined measure of "distance" between goals and the goal-pace is a metric space [11] , then for a given fixed size configuration store, find the goals whose configurations should be stored such that the sum of the distances of those goals that are not present in the configuration store, from the distance-wise nearest goal present in the configuration store, is minimum.

P1 and P2 can be viewed, respectively, in terms of the well-known problems of minimum dominating sets and k-medians. To reduce P1 from the minimum dominating set problem [12] , for every vertex in the dominating set problem, instantiate a goal, and for every edge, instantiate a condition that the goal corresponding to the source vertex is acceptable to the goal corresponding to the sink vertex. The problem P1 related to this set of goals and the acceptability relation among goals is equivalent to the given minimum dominating set problem instance. The vertices in the given minimum dominating set problem instance, corresponding to the goals that should be stored in the configuration store (found by solving P1) constitute a minimum dominating set for the given minimum dominating set problem instance. This can be used to show that the problem P1 is NP-hard [3]. However, if the acceptability relation is a partial order, then the minimum dominating set can be found in polynomial-time by picking up all the vertices with no incoming edges in the graph of the minimum dominating set problem [3][9][13]. This is in accordance with Theorem 1, and further underscores the advantage of using acceptability relations that are partial orders. If the associated distance function is defined between any two goals and the goal-space is a metric space, then problem P2 can be modeled in terms of the *k-median problem*. For the simple case of a two-dimensional goal space, a polynomial-time approximation algorithm with a 3-approximation factor exists for the k-median problem.

Configuration management problems P1 and P2 can be viewed as extreme cases in the sense that in one of them we want to cover all feasible goals without considering how large the minimum size configuration store would be (P1), and in the other case, we have a fixed size configuration store and we are trying to find out the maximum number of goals that can be covered using that configuration store even though that number could be much less than the total number of relevant goals (P2). A more elaborate formulation would be one in which we have to pay extra cost for increasing the size of the configuration store, but we would be gaining some additional service by that by being able to store more goals in the configuration store. This way we can explore various trade-offs between the size of the configuration store vs. the number of goals stored in a well-defined way. For the specific case when a distance function is defined between any two goals and the goal-space is a metric space, these trade-offs can be explored by modeling this problem as a facility-location problem [3][11][13][14]. A polynomial-time algorithm with an approximation guarantee of 1.74 exists for the facility location problem [11].

(4) On-Line Adaptation

In this section, we focus on the metrics of throughput and power consumption, and develop low-complexity, on-line strategies based on heuristics for throughput optimization and power optimization as implementations of the function *onLineAdaptation* in Figure 8. The objective is to demonstrate the efficacy of the CMF model, and show that it can produce efficient tracking of time-varying application requirements.

The approach of taking feedback from the execution of the application makes these on-line methods able to handle even applications with stochastic execution times that have time-varying distributions, in addition to applications with fixed execution times, and applications with stochastic attributes that have stationary distributions. In general, this on-line refinement formulation can thus be viewed as an approach to tracking the dynamics of the goal and the characteristics of the application.

To experiment with CMF, we used a simple heuristic based on load balancing [15] to optimize throughput during online adaptation. Pseudocode for this heuristic is represented by function *adaptThroughput* in Figure 10. In the pseudocode, *moveTask* is a function that chooses *n* tasks from a maximally loaded processor in a configuration *c*, and randomly, moves them to appropriate locations on a minimally loaded processor, and returns the modified configuration. Randomization in choosing tasks from the maximally loaded processor provides a low-complexity approach to increase the explored region of the design space and to calibrate the configuration to dynamic application characteristics. The function *executeTr* is a function that executes the application according to configuration *c* for a time interval of length *l*, and returns the throughput of the application during that interval. The value of *l* to use depends on the non-determinacy of the application.

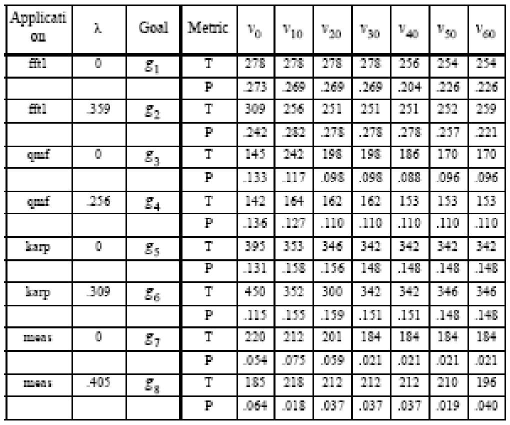
```
/* This function adapts the given input configuration
      while executing the application.
*/
function adaptThroughput
input configuration $c$
global constant time $timelimit$, time $l$

time $t_{old}$ = executeTr($c$, $l$)
time $t$
configuration $c_{old}$ = $c$
$n$ = 1
while (clock < $timelimit$) {
      $c$ = moveTask($c_{old}$, $n$)
      if (exhausted all $n$-task movements
                  without improvement){
            $n$ = $n$ + 1
            $c$ = moveTaskTr($c_{old}$, $n$)
      }
      $t$ = executeTr($c$, $l$)
      if ($t \geq t_{old}$) {
            $c_{old}$ = $c$
            $t_{old}$ = $t$
            $n$ = 1
      }
      clock = clock + $l$
}
end function
```

Figure 10. Online Adaptation Approach

Generally, the more non-deterministic the application is, the longer it needs to be executed to determine an accurate value of average throughput. The function *adaptThroughput* returns a configuration that it deems most appropriate for throughput maximization. Note that if moving any single task from the maximally loaded processor to the minimally loaded processor does not improve performance then the heuristic chooses a *pair* of tasks to be moved to another processor. This approach of progressively increasing the number of tasks to be moved continues whenever all combinations for a particular number of tasks have been exhausted. This approach thus attempts to make small low-complexity changes first and if that does not improve performance, the approach gradually reaches towards higher-complexity changes. The higher complexity changes are larger in number than small, low-complexity changes, and help the system in escaping from local minima. In our experiments, inter-processor communication (IPC) per time unit during the execution is taken as an estimate for relative power consumption. Since IPC consumes relatively large amounts of power, it is a reasonable approximation for comparing the power consumption levels of alternative configurations on a homogeneous multiprocessor. To find a configuration that reduces the power consumption, we use an approach (called *adaptPower*) similar to the *adaptThroughput* approach used for throughput optimization, except that the probability of a task on a maximally loaded processor being transferred to a minimally loaded processor depends upon the IPC associated with that task. The higher the IPC associated with a task, the higher its chances are of being transferred to another processor.

Figure 11 shows the performance of our implementation of CMF using the heuristics for throughput optimization and power optimization based on various goals applied to several DSP benchmarks, including fast Fourier transform, filter bank, music synthesis, and measurement applications. The starting configuration that is refined is found by using standard critical path scheduling. The critical path length is computed in terms of average execution times of actors. The set of relevant metrics M for our experiments is $M = \{T, P\}$, where T denotes the average iteration period of the execution and P denotes the average power consumption. In Figure 11, the column titled “Goal” represents the goal that is applied to the application. Also, for a non-negative integer k, column v_k denotes the value of a metric of the best configuration found by the on-line adaptation scheme, after configurations have been assessed by executing them for some time.



Application	λ	Goal	Metric	v_0	v_{10}	v_{20}	v_{30}	v_{40}	v_{50}	v_{60}
fft1	0	g_1	T	278	278	278	278	256	254	254
			P	.273	.269	.269	.269	.204	.226	.226
fft1	.359	g_2	T	309	256	251	251	251	252	259
			P	.242	.282	.278	.278	.278	.257	.221
qmf	0	g_3	T	145	242	198	198	186	170	170
			P	.133	.117	.098	.098	.088	.096	.096
qmf	.256	g_4	T	142	164	162	162	153	153	153
			P	.136	.127	.110	.110	.110	.110	.110
karp	0	g_5	T	395	353	346	342	342	342	342
			P	.131	.158	.156	148	.148	.148	.148
karp	.309	g_6	T	450	352	300	342	342	346	346
			P	.115	.155	.159	.151	.151	.148	.148
meas	0	g_7	T	220	212	201	184	184	184	184
			P	.054	.075	.059	.021	.021	.021	.021
meas	.405	g_8	T	185	218	212	212	212	210	196
			P	.064	.018	.037	.037	.037	.019	.040

Figure 11. Experimental Results for CMF

c. Stream Virtual Machine

In this section, we describe our implementation of a streaming model of execution using a proposed standard streaming API, and evaluate the performance results for the implementation running on the Raw processor [16]. In our streaming execution model, finite-duration tasks running on programmable hardware computational cores read operands from local on-chip memories, and write results back to local memories. We expect the access patterns to local memory to be regular, e.g., sequential, which assists with getting performance from the hardware cores. Data movement between the on-chip local memories and off-chip memories is via explicit initiation of transfers. If there are multiple cores on the chip, it is also possible to set up direct links to transfer sequences of data directly from core to core.

Based on the hypothesis that this execution model is general and reflects the means by which contemporary processors are programmed to achieve high performance when executing programs with substantial amounts of data parallelism as are found in signal processing, a standard way of describing computations in this model was developed. This is the Stream Virtual Machine (SVM) framework [1][17]. The SVM framework consists of two parts, a C-idiomatic API for expressing computations in this execution model, along with a standard way for writing machine models (in an XML syntax) that describe hardware targets. One use of the SVM framework is as an intermediary between two compilers; a High Level Compiler (HLC) that is responsible for analyzing the input code and performing coarse grain transformations of the application, such as extracting parallelism and coarse grain load balancing, and a Low Level Compiler (LLC) that maps the tasks of the SVM to the hardware computational cores, performing standard compiler transformations such as instruction selection, register allocation, and instruction scheduling.

The output of the High Level Compiler is the application code after it has been mapped into the streaming execution model as expressed in the SVM form. Like all compilers, HLC uses an abstract machine model as a basis for the feasibility constraints and cost functions that drive the transformations and optimization. A HLC has been developed for SVM called R-Stream, and machine models and LLCs have been developed for Raw, MONARCH [18][19], TRIPS [20][21], and Smart Memories [22][23]. These research projects were conducted with the coordination of the Morphware Forum.

In this work, we experimented with the R-Stream 2.1 HLC and a low-level compiler for the Raw architecture, as shown in Figure 12. We implemented the SVM LLC by using Raw's C compiler and implementing the rest of the SVM API as library. Then, we implemented several stream applications: matrix multiplication, FIR filter banks, and Ground Moving Target Indicator (GMTI) [24].

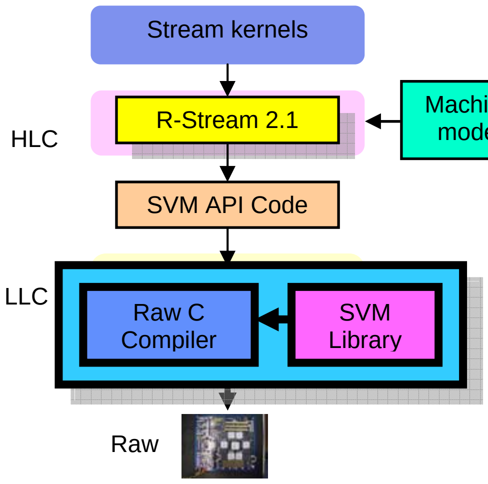


Figure 12. HLC and LLC for Raw

In our implementation of LLC, instead of building standalone compiler, we leveraged the available gcc compiler tailored for Raw by using a library approach to SVM construction. In this approach, we build a library for SVM APIs that is used for an SVM program. Although this approach does not provide the full performance that a SVM compiler might provide, it enabled us to assess the SVM framework in a short time period with minimal cost. In our implementation, all SVM functions in the specification are implemented.

The library provides several functions to support the SVM APIs. One of them is maintaining the kernel data structure. When a kernel is started, the kernel data structure is passed to the secondary master. However, since the kernel start function is non-blocking, it may return before the kernel starts. If the caller returns from a function in which the API is called, the memory space for the kernel data structure can be freed, and, when the secondary master is ready to execute the kernel, the kernel data structure is not available. To solve this problem, our library maintains a copy of the kernel data structure in library memory space.

Another function that the library provides is handling of data buffering for streams through dynamic networks. The dynamic network guarantees the order of data between two tiles. However, if two sender tiles send data to a destination tile, the order of the data in the receiver tile is not known at compile time. Thus, the library needs to identify the source of data whenever a packet of data arrives. Also, if a data being received belongs to a stream that is to be received later, then the data needs to be stored in a buffer. The library keeps buffers for storing such data for proper operation of the dynamic network. We implemented two kernels and one compact application: matrix multiplication, FIR bank, and GMTI using our LLC/SVM implementation. Matrix multiplication, C = AB, calculates an output matrix C from A and B, where A, B, and C are matrices. We implemented systolic matrix multiplication. The A matrix data travels from left to right, and the B matrix data travels from top to bottom in the Raw architecture. Each tile where A and B intersect computes matrix multiplication using incoming data from A and B. The result data travels to the right. The matrix sizes for A and B are 3 by 128 and 128 by 256, respectively.

The FIR bank is from the kernel benchmark suite [24] specified by Massachusetts Institute of Technology Lincoln Laboratory for the PCA program. The FIR bank implements a set of M FIR filters and each FIR filter m, $m \in \{0, 1, \ldots, M\text{-}1\}$, has a set of impulse response coefficients $wm[k]$, $k \in \{0, \ldots K\text{-}1\}$. It is mathematically specified as:

$$y_m[i] = \sum_{k=0}^{K-1} x_m[i-k]w_m[k], \text{ for } i = 0,1,\ldots,N-1 .$$

The filters are distributed over tiles such that each tile has M/T filters, where T is the number of tiles. Each tile computes its own filters. Therefore, there is no communication between tiles. The FIR is implemented in the frequency domain which is more efficient than the time domain when data size is large. Therefore, FIR requires an FFT, a complex product, and an IFFT operation. In this implementation, a few optimizations at the application level were performed: i) the bit-reversal operations are eliminated by bit-reversing filter coefficients and ii) using radix-4 FFT and radix-4 IFFT.

GMTI is a compact application that represents airborne radar applications used to locate a target on the ground [24]. It consists of several stages: Time Delay and Equalization (TDE), adaptive beamform, pulse compression, Doppler filter, space-time adaptive processing, target detection, and target parameter estimation. TDE and pulse compression stages are mainly convolution in frequency domain. Thus, they consist of FFT, multiplication, and IFFT operations. Other stages mainly include matrix operations and FFT operations.

To assess the SVM framework for Raw, we used matrix multiplication, FIR bank, and GMTI. The matrix multiplication and FIR bank were hand-coded using the SVM API so that the HLC is bypassed, which allowed us to isolate issues related to HLC. The code was then compiled using the LLC that consists of our SVM library and the Raw C compiler. The code was executed on the Raw hand-held board. GMTI is written in C-dialect code ("Gumdrop" code) that provides hints to the HLC, and then is compiled with the HLC. The output code from the HLC is expressed in the SVM API, is compiled with LLC, and then is executed on the Raw processor.

The performance results for matrix multiplication are shown in Figure 3. The figure shows the number of cycles used for computation of each multiplication-addition pair, i.e., the total number of execution cycles is divided by the number of multiplication-addition pair in the matrix multiplication. On Raw, multiplication and addition each need one cycle to execute. Thus, the lower bound of the number of cycles for a multiplication-addition pair is two. Figure 13 shows the number of cycles for a multiplication-addition pair as a function of the number of words per communication. The number of words per communication is the message size in words when a tile sends a message to a neighbor tile or a tile in the rightmost column for result data sending.

The curve in Figure 13 named "SVM Library" shows the performance when a full implementation of SVM API is used. The curve shows that the initial cost of the communication using the library approach can be amortized over a long sequence of data.

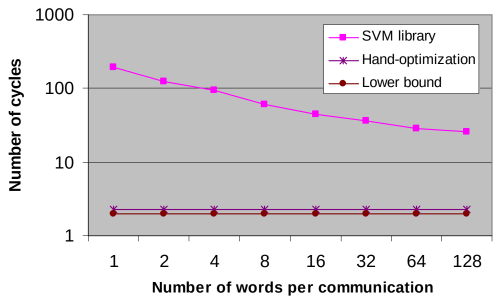


Figure 13. SVM Matrix Multiplication Results

In optimized communication for matrix multiplication in several ways. One way is utilizing available multiple networks so that each network handles only one stream. Then, the overhead for managing multiple streams in a network can be eliminated. Another optimization was using hand-assembly code for critical section of the code. This allows us to use the minimal number of instructions and optimal instruction scheduling. The last optimization is using network ports as operands, a unique feature of Raw. The curve named "Hand-optimization" in Figure 13 shows the performance when these optimizations were applied. The most optimized results show that it takes only about 10% more overhead than the theoretical lower bound, and the main reason for the overhead was due to the loop outside of the deepest loop and software pipeline overhead.

The FIR bank was specified for two data sets in Polymorphous Computing Architecture (PCA) kernel benchmark specification. The first set is a large data set: the number of filters is 64, the number of input data is 4096, and the number of filter coefficients is 128. The second data set is a small size: the number of filters is 32, number of input data is 1024, and the number of filter coefficients is 12.

We implemented the FIR bank manually using the SVM API. We performed several optimizations including all three optimizations applied to matrix multiplication. An additional optimization applied to the FIR bank implementation is using broadcast capability of the Raw switch processor. In the broadcasting scheme, when a switch processor receives data from a source, it duplicates the data and sends one copy to the compute processor and sends another copy to a destination switch processor that performs the same operation.

The FIR bank is also optimized in several ways at the algorithmic level: using radix-4 FFT, elimination of bit-reversal, using overlap-save method, minimization of address calculations using offsets, and preventing register spilling by restricting the number of registers used. The implementation results are shown in Figure 14 and Figure 15. In Figure 14, UB denotes the upper bound of performance considering only useful floating point operations, not overhead operations. Since there are 16 tiles, each of which can compute one floating point operation, the upper bound is 16 floating point operations per cycle.

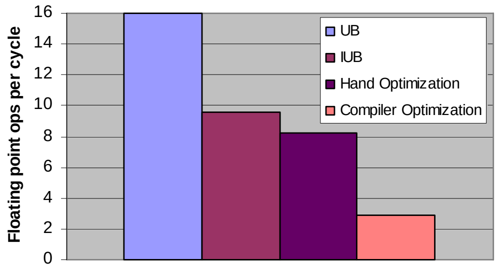


Figure 14. SVM Throughput Results for FIR Bank

IUB (implementation upper bound) denotes the upper bound when load and store operations are considered as well as the number of floating point operations. Since the load and store operations cannot be eliminated in our FIR bank implementation, the IUB is a "practical" upper bound. The performance in Figure 14 is obtained when input and output data are in cache so that time to access the external memory is not considered. Our results show that the hand-optimized results are very close to the "practical" upper bound with only about 10% overhead.

The result denoted as "compiler optimization" is obtained using the LLC with algorithmic optimizations only. It shows about three times difference between hand-optimized performance, which is mainly due to additional instructions and non-optimal instruction schedules. Figure 15 shows the effect of accessing data from memory. It takes about 16% additional cycles when data is accessed from memory. The additional cycles are not significant since in the FFT computation, data re-use is high.

Figure 16 and Figure 17 show the results of our GMTI implementation. Figure 16 shows the execution schedule of each processor including a primary master, secondary masters, and stream processors. Note that tile 0 is mapped to the primary master, one secondary master, and one stream processor in time-shared mode. Other tiles are mapped to one secondary master and one stream processor. The execution schedule shows the

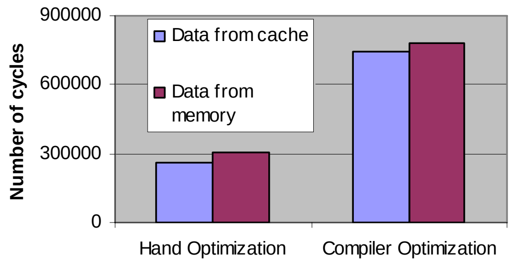


Figure 15. SVM Latency Results for FIR Bank

parallelization achieved by the HLC.

The application is parallelized using up to 12 processors. Note that the HLC actually can parallelize up to the maximum number of tiles on Raw processor (16). However, there was a bug in the program execution on four of the 16 tiles that prevented the execution of 16-tile code. Figure 16 shows that some portions of the application are not parallelized due to the serial nature of those portions of the application. However, the available slots may be utilized using software task pipelining. Utilization (expressed as fraction floating point operations per cycle achieved compared to peak floating point performance) is about 0.5%. The low utilization is due to the empty schedule slots shown in Figure 16, load and store operations, and some redundant data movement operations that could be optimized away.

Since GMTI is a large application, optimizing the entire application for execution on the SVM under this effort was not feasible. Therefore, we chose one stage, Time Delay and Equalization (TDE), to focus on and performed optimizations for that stage. In Figure 17, the x-axis shows several steps in TDE. In each step, the first bar marked as "R-Stream" shows the performance of using the HLC and LLC with only algorithmic optimizations used. The next bar, marked as "direct copy," shows the performance when data is moved without using SVM calls. Then, hand-assembly performance shows when critical sections of the code are optimized using hand-assembly.

The ILB denotes the "practical" lower bound that includes load and store operations as well as floating point operations. Note that the hand-assembled code performance is close to the ILB.

The bar marked as "FLB" shows the lower bound when only floating point operations are considered. The results show that the hand-optimized code obtains very close performance to the "practical" lower bound that is expected to be obtained if the HLC and

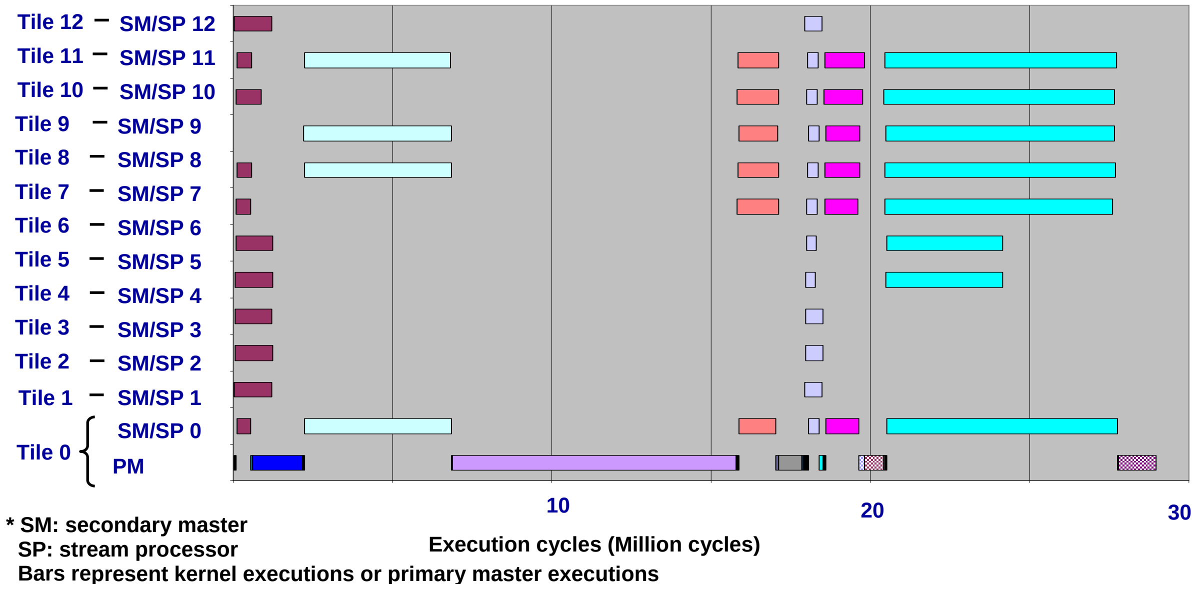


Figure 16. GMTI Execution Schedule

LLC incorporate the optimization techniques we applied. The performance of the R-Stream is also encouraging as the difference between the R-Stream and hand-optimization code is only about three times for major computational parts even though R-Stream is still in research development.

The results also reveal where the current tool chain needs improvement. One of the improvements needed is better parallelization capability. Although the current HLC parallelizes up to the maximum number of processors, in some stages in GMTI, it fails to parallelize because of memory constraints. We expect the HLC to parallelize these sections as better code analysis capability is developed.

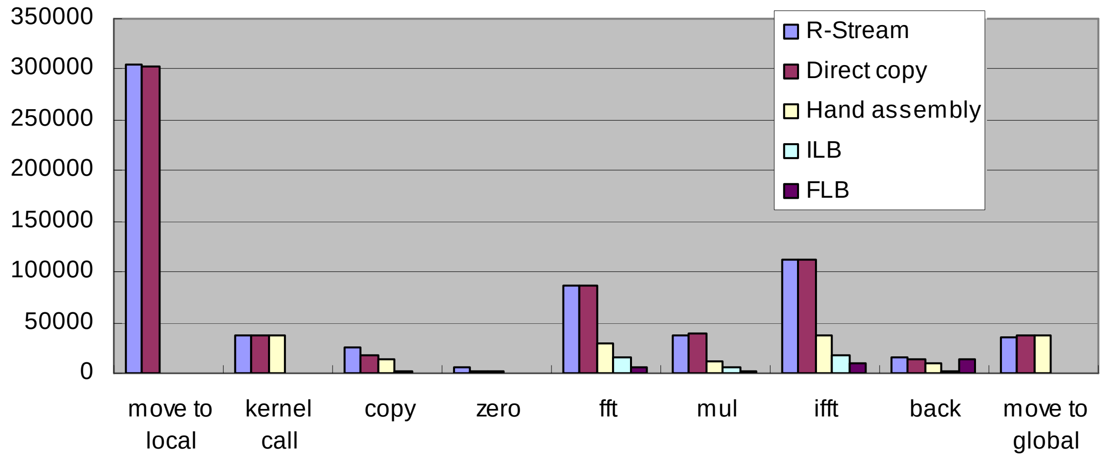


Figure 17. GMTI Cycle Breakdown

d. Application Performance

The Integrated Radar Tracker (IRT) is the primary compact application specified for performance evaluation for the PCA program. Within IRT, most of the computation is in the Ground Moving Target Indicator (GMTI) [24]. GMTI, for which the block diagram is shown in Figure 18, contains many algorithms, divided into nine stages: subband analysis, time delay and equalization, adaptive beamforming, pulse compression, Doppler filter, space-time adaptive processing (STAP), subband synthesis, target detection, and target parameter estimation. The AMP team mapped GMTI to the Raw architecture as a tool to understand the issues that arise in mapping a full-scale application to a polymorphous computing architecture. We mapped GMTI to a 64-tile (4-chip) Raw system because 64 tiles is enough to expose larger scale application issues, including programmability and performance scalability and to run a realistically-sized application, and because it allows us to simulate individual stages in a reasonable amount of time.

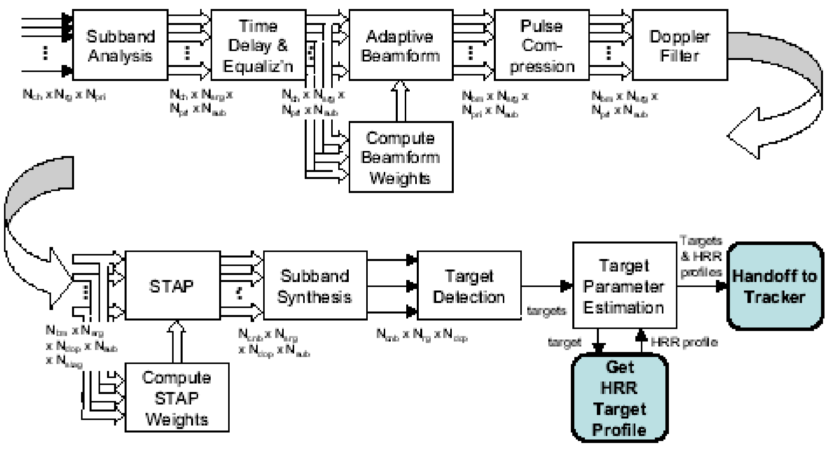
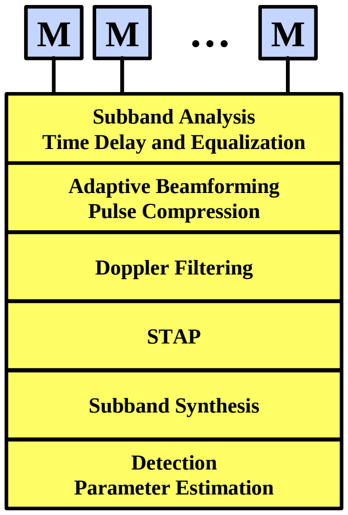


Figure 18. GMTI Block Diagram

The first step in mapping GMTI to Raw was to identify parallelism, which is necessary to map any application to a parallel architecture. Since the application was specified as a pipelined application with task-level parallelism, we chose to exploit that task-level parallelism first, since each task can be executed independently. However, in order to achieve the performance goals of the GMTI specification, we needed to extract additional parallelism from individual stages. The left part of Figure 19 shows the tasks divided into stages, with input data coming from the memory on top, which shows our initial coarse task-level parallelism. The right side of Figure 19 shows the GMTI parameters within each stage that provide parallelism that can be exploited for further performance gains.



Stage	Parallelism
SA/TDE	**PRI, channel, subband**
AB/PC	**Subband, PRI**
DF	**Subband, beam**
STAP	**Doppler**
SS	**Dopper, clutter-nulled beams**
Det/PE	**Doppler, beams**

Figure 19. GMTI Parallelism

Next, we used the computation requirements of each stage to determine how many Raw tiles needed to be dedicated to processing at each stage. The number of tiles needed for each stage is determined by the processing and memory bandwidth requirements for each stage. We used empirical data to determine assumptions about how many operations per second each tile could deliver. Figure 20 shows the number of tiles that were needed for each stage of the GMTI application. The tiles were then mapped to rows of an 8x8-tile Raw system, so that data could flow between tasks without interference from other tasks.

Running the full 64-tile GMTI application on the Raw simulator was not feasible.

GMTI

PRF (Hz)	1,000
transmit duty factor	10%
sampling frequency (Hz)	500,000
number of channels	8
number of PRIs	48
number of PRI staggers	2
number of subbands	2
number of post-ABF beams	4
number of post-STAP beams	2
number of range gates	450
number of doppler bins	47

CPI length (s) 0.048

	FLOPs/second	RAW Tiles	RAW Chips	System Rows
subband analysis	8.30E+08	22	2	1
time delay & equ	3.52E+08	10	1	1
ABF	1.17E+08	3	1	1
PC	3.52E+08	10	1	1
doppler filtering	1.44E+08	4	1	1
STAP	1.65E+08	5	1	1
subband synthesis	9.35E+07	3	1	1
detection	9.71E+06	1	1	1
parameter est	0.00E+00	0	0	0
subtotal	**2.06E+09**	**58**	**4**	**2**

Figure 20. GMTI Computation Balance

However, we did run separate simulations of the inter-stage communication and computation for each stage. We exploited data parallelism within each stage, so the communication within each stage is negligible. The inter-stage communication did not take more than 20% of the computation cycles for any stage, so we focused on application performance within each tile.

Figure 22 shows single-tile performance on Raw, measured in cycles, compared to an x86 architecture (represented by an Intel Pentium III processor). Since each tile of Raw is a single-issue core with one floating point unit, the fact that single-tile performance is roughly equivalent to a commercial x86 processor is encouraging. This means that the complexity of the x86 architecture devoted to extracting instruction-level parallelism is not gaining much performance, and the silicon area on the processor is better devoted to additional tiles (processors). To validate the simulation results, we implemented a 4x4 version of narrow-band GMTI and found that the simulation cycles counts were within less than 2% of the actual cycle counts measured on hardware.

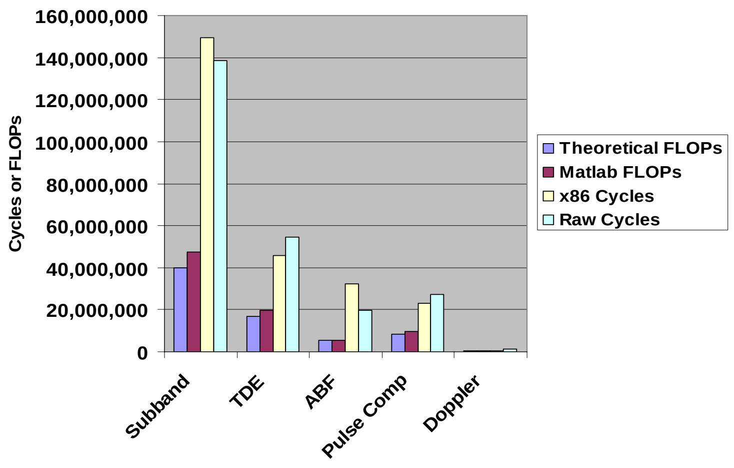


Figure 22. GMTI Single-Tile Raw Performance

e. Coherent Sidelobe Canceller and DARPATech Demonstrations

The AMP team demonstrated compatibility of PCA architectures with existing DoD software standards by implementing advanced embedded DoD signal processing algorithms, such as radar Pulse Compression, on multiple Polymorphous Computing Architectures (PCAs) using the VSIPL industry standard signal processing API. Critical embedded, high performance radar processing algorithms for shipboard ballistic missile defense have been

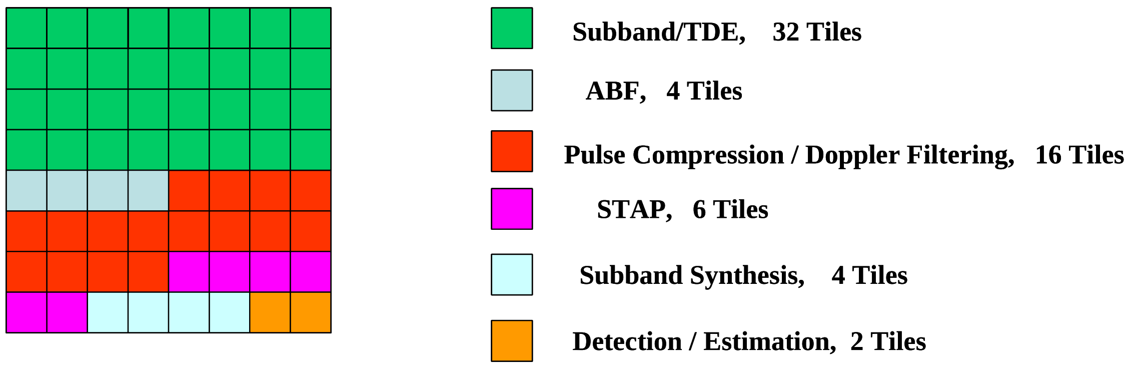

Subband/TDE, 32 Tiles
ABF, 4 Tiles
Pulse Compression / Doppler Filtering, 16 Tiles
STAP, 6 Tiles
Subband Synthesis, 4 Tiles
Detection / Estimation, 2 Tiles

Figure 21. GMTI Resource Mapping

implemented at Lockheed Martin MS2 in COTS PowerPC architectures using equipment from various vendors, including CSPI and Mercury Computers, utilizing standard API libraries such as VSIPL and MPI. Of particular interest is whether we can port these embedded applications from conventional COTS architectures for execution on various embedded PCA morphable processor architectures. This successful demonstration of application portability is key to DoD acceptance of non-conventional computing architectures, such as PCA, for embedded tactical processing applications. The AMP team successfully demonstrated:

- A critical embedded DoD signal processing benchmark (frequency domain radar pulse compression) using industry standard VSIPL, executing on Raw and TRIPS
- A successful morph from C-VSIPL Streaming Pulse Compression to a C-VSIPL threaded tracking function, on an actual Raw processor and a TRIPS simulator
- A successful port of the Raw C-VSIPL streaming pulse compression code and C-VSIPL threaded track processing code to the TRIPS architecture
- A Java-based GUI to display the input data, output data and PCA processor status while running the C VSIPL benchmarks

By providing industry standard APIs, such as VSIPL and MPI, for new PCA architectures, we have demonstrated legacy applications implemented in new, morphable embedded PCA hardware and middleware architectures with industry standard C and C++ programming methodology.

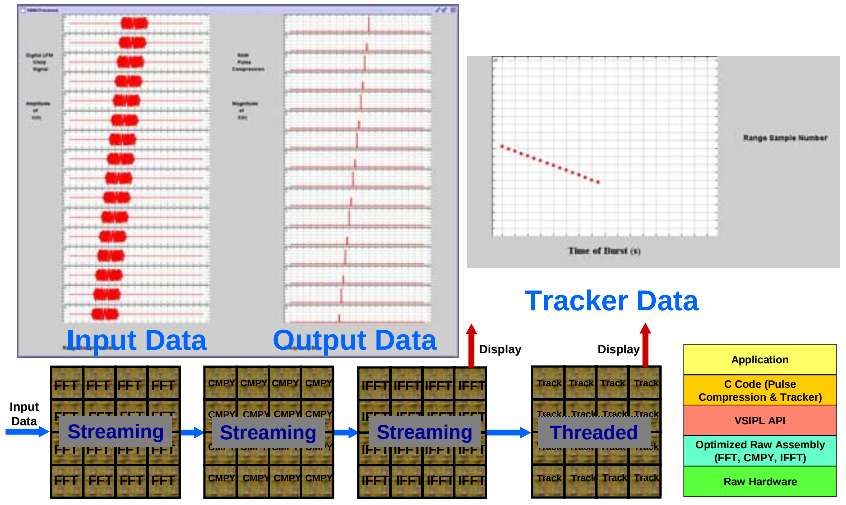


Figure 23. Raw VSIPL Demonstration

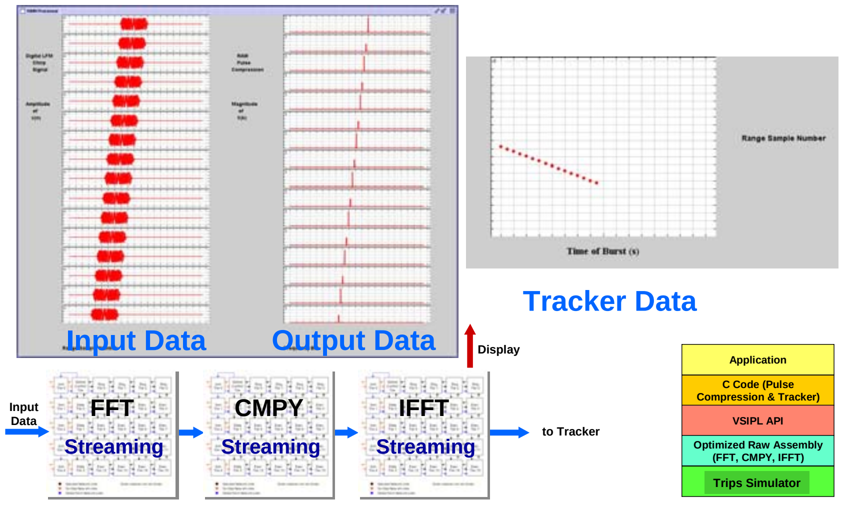


Figure 24. TRIPS VSIPL Demonstration

f. Comparison to Data-Intensive Architectures

In this section, we compare the Raw PCA processor to architectures developed under the Data Intensive Systems program. Performance of these kernels is obtained by using cycle-accurate simulators provided by the VIRAM [25], Imagine [26][27], and Raw teams [28].

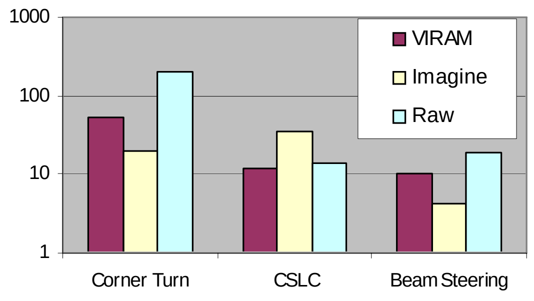
For comparison purposes, actual measurements of performance were taken using a single node of a 1 GHz PowerPC G4-based system (Apple PowerMac G4 [29]). An implementation using AltiVec technology was used for speedup comparisons. The Apple cc compiler was used with timing done using the MacOS X system call mach_absolute_time(). We manually inserted Altivec vector instructions.

Figure 25 summarizes key parameters of each processor and the performance results. Note that the PowerPC is a highly optimized chip in performance implemented with custom logic. The other processors are research chips implemented using standard cells and very small design teams. Thus, if the same level of design effort were applied to these research architectures, we would expect much higher clock rates and density to be achieved. Figure 26 shows the speedup in terms of cycles and execution time.

	PPC G4	VIRAM	Imagine	Raw
Clock (MHz)	1000	200	300	300
# of ALUs	4	16	48	16
Peak GFLOPS	5	3.2	14.4	4.64

	Corner Turn	CSLC	Beam Steering
PPC	34,250	29,013	730
Altivec	29,288	4,931	364
VIRAM	554	424	35
Imagine	1,439	144	87
Raw	146	357	19

Figure 25. Architecture Parameters and Performance Summary



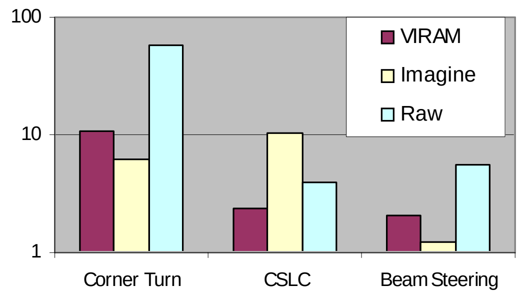


Figure 26. Speedup Relative to PowerPC Measured by Cycles (Left) and Time (Right)

All three architectures provided speedups of more than 20 compared with a PowerPC system in terms of number of cycles on the corner turn benchmark. Corner turn performance is primarily a measure of memory bandwidth, which is not a direct property of an architecture, but rather a function of the number of pins in the package. However, the corner turn does demonstrate an architecture's ability to leverage memory bandwidth that does exist. Since VIRAM has on-chip DRAM, the kernel measures on-chip bandwidth. On the Imagine and Raw architectures, we stress off-chip memory bandwidth.

The performance of corner turn on VIRAM is about half of what would have been expected from peak memory bandwidth. About 21% of the total cycles are overhead due to DRAM pre-charge cycles (which would be mostly hidden with sequential accesses) and translation look-aside buffer misses, and 24% are due to a limitation in strided load performance imposed by the number of address generators.

On Imagine, we assume the memory clock is the same frequency as the processor clock. Imagine has two address generators that provide two words per clock cycle. Note that the number of address generators is a processor implementation choice and is not a limitation of the stream architecture. Since the goal of the Imagine project was to demonstrate how memory traffic could be reduced, the Imagine team chose not to implement a high-bandwidth memory interface. If a network port were used to transfer data between SRF and an external memory connected to network port for corner turn, the performance would be the same since the network port has a peak performance of two words per cycle.

87% of the cycles in the Imagine corner turn are due to memory transfers, which operate close to the maximum theoretical performance. The remaining 13% of the execution cycles are due to non-overlapped cluster instructions. Conceptually, the kernel instructions should be fully overlapped with memory accesses, but a limitation induced by the stream descriptor registers prevented full software pipelining in our implementation.

The Raw chip implementation actually provides enough main memory bandwidth that it is not the performance limiter for our corner turn implementation. Load/store issue rates and local memory bandwidth limit performance. An aggregate of 16 instructions per cycle are executed on the Raw compute tiles, and the static network and DRAM ports are not a bottleneck. The performance we achieved is nearly identical to the maximum performance predicted by the instruction issue rate. Memory latency is fully hidden (except for negligible start-up costs).

CSLC consists primarily of FFTs and matrix-vector multiplication. Since the FFT length is 128, the working set fits into local memory, the performance of the CSLC depends primarily on ALU performance for Imagine and Raw. Our VIRAM CSLC implementation takes about 3.6 times longer than what is predicted by peak performance. The first factor reducing performance is overhead instructions (those instructions that do not directly move input or output data or perform necessary floating point operations). Overhead instructions are needed to perform the FFT shuffles and increase the number of cycles by a factor of 1.67. The second factor that reduces FFT performance is ALU utilization. Since the second vector arithmetic unit in VIRAM cannot execute vector floating point instructions, performance on the FFT is reduced by a factor of 1.52. Finally, memory latency and vector startup costs increase performance by a factor of 1.41.

The CSLC implementations on Imagine and Raw use the radix-2 FFT. On Raw, we used radix-2 to avoid register spilling encountered in the radix-4 FFT. On Imagine, the radix-4 FFT provides better performance (about 34%), but a complete CSLC implementation was not available and was beyond the scope of the AMP project. The number of operations (including loads and stores) in the radix-2 FFT is about 1.5 the number in the radix-4 FFT. So care should be taken when comparing the performance of the CSLC on Raw and Imagine with CSLC performance on other architectures.

Imagine has the best performance of the three architectures on CSLC. This is because it is a computation-intensive kernel for which the working sets fit in the stream register files. Although the data access patterns for FFT are challenging for any architecture, the streaming execution model of Imagine is able to reduce memory operations and Imagine functions as intended on this kernel. Overall, performance achieved on CSLC on Imagine is about 39% of what is predicted by peak performance. While this is much lower than those achieved for many media benchmark kernels, it still allows Imagine to perform about 16 useful operations per cycle; much better than can be achieved on today's traditional superscalar architectures. Performance is reduced by 35% because inter-cluster communication is used to perform parallel FFTs. An alternative implementation, which was beyond the scope of this study, would execute independent FFTs in parallel to eliminate inter-cluster communication overhead.

For the FFT kernel, ALU utilization (as measured by minimum FFT computations / total ALU cycles available) is 34.0%. If we exclude the divider, which is not useful for the FFT, then the utilization is 40.8%. Note that the utilization is on the lower side of the more than 40% obtained in other processing intensive applications. The reason for the relatively low utilization is that the small size of the FFT reduces the amount of software pipelining and increases start-up overheads.

On Raw, we implemented a data parallel version of CSLC. The local memory on Raw successfully caches the working sets, and less than 10% of the execution time is spent on memory stalls. Note that most of this stalling could have been eliminated by implementing a streaming DMA transfer to the local memory that is overlapped with the computation.

One problem with our data parallel implementation of CSLC on Raw was load balancing. The CSLC is easily parallelized for 16 tiles. However, since the number of data sets is 73, which is not a multiple of the number of tiles, some tiles processed five sets while others processed four sets. About 8% of CPU cycles are idle due to load balancing. However, the number of sets in a real environment is not fixed at 73. In a real implementation, the input data sets would arrive continuously. Therefore, it is reasonable to assume that Raw could have nearly perfect load balancing in a real implementation. Thus, we report the performance numbers for CSLC on Raw based on an extrapolation that assumes perfect load balancing.

Raw achieves about 31.4% of the peak performance on CSLC. About 26% of the cycles on Raw are consumed by load and store instructions. Cache stall takes 7.4% of the total cycles. The remaining cycles are consumed by address and index calculations and loop overhead instructions. If FFT is implemented using the stream interface that uses the static network, cache miss stalls are hidden, and load and store operations are not needed. A primitive implementation result suggests about 70% FFT performance improvement.

Beam steering has few memory accesses (two reads and one write) and computations (five additions and one shift) per output data. On VIRAM, the lower bound of the computation time is 56% of the simulation time. The difference between the expected time and simulation cycles (15,412) comes from waiting for the results from previous vector operations and the cycles needed to initialize the vector operations.

On Imagine, the computations and memory accesses for beam steering are overlapped. The performance is limited by memory bandwidth due to the relatively low number of computation per memory access. The load and store operations take 89% of the simulation time. The remaining 11% of execution time is due to the software pipeline prologue. In an actual signal processing pipeline, the beam steering kernel would stream its inputs from the proceeding kernel in the application (e.g., a poly-phase filter bank) and stream its outputs to the following kernel (e.g., per-beam equalization). In such a pipeline the performance of beam steering will not be limited by memory bandwidth, as in the case of this isolated kernel, but rather will be limited by arithmetic performance. On such a streaming application Imagine is expected to achieve a high fraction of its peak performance. If table values were read from the stream register file rather than memory on our kernel, performance would be increased by a factor of about two. The performance of a beam steering algorithm with more computation per data (which is a realistic assumption) could be much higher.

On Raw, we used the static network to stream data from memory while hiding memory latency. In this implementation, loads and stores are not necessary and ALU utilization is very high. It attains 96.6% of the peak performance. The Raw beam steering implementation has the best performance of the three architectures because of the combination of memory bandwidth and high ALU utilization.

VIRAM's primary advantage comes from the high bandwidth between the vector units and DRAM without paying the cost (in terms of pins and power) that are required to achieve high bandwidth between chips. VIRAM is especially suitable for vectorizable applications that can utilize the high bandwidth interface and that are small enough to fit in the on-chip memory. VIRAM outperformed the G4 Altivec by more than a factor of 10 on all three of our kernels and showed especially good performance on the kernels that emphasize memory bandwidth. For embedded applications with reasonably sized data sets, the VIRAM can be used as a one-chip system. If the application size is larger than the on-chip DRAM, the data needs to come from off-chip memory and VIRAM would lose much of its advantage.

Imagine's high peak performance can be utilized in streaming applications where main memory accesses can be avoided or minimized. The CSLC kernel demonstrates that even when the Imagine ALUs are not fully utilized, performance can be quite high, especially when compared to a commercial microprocessor like the G4 Altivec. Imagine's stream-based architecture is designed for scalability and power efficiency and the Imagine architecture has the highest peak performance of the architectures in this study.

Raw also performs best on streaming applications since load and store operations can be eliminated and the static networks provide tremendous on-chip bandwidth. The kernels used in this study do not fully exploit this mode of execution. But we have shown that the tile structure of Raw can be used to utilize the memory bandwidth available from the external ports of Raw. The tile structure also provides flexible support for MIMD and ILP applications, which is the primary goal of polymorphous computing architectures.

g. Knowledge Aided Processing

A recent thrust of algorithm enhancements in radar processing centers around a theme of augmenting traditional digital signal processing algorithms (DSP), such as space-time-adaptive-processing (STAP) and constant-false-alarm-rate (CFAR), with better knowledge-based (KB) situational information, thus enabling them to make better use of the available real-time radar returns to improve signal-to-interference and noise-ratio (SINR). While the core traditional STAP and CFAR algorithms remain largely untouched, there is substantial pre-processing and co-processing of different data sources, such as global positioning system (GPS), Digital Elevation Maps (DEMs), Land Use Land Cover (LULC) maps, Satellite Imaging, Synthetic Aperture Radar maps (SAR), and previously collected radar data that can be used to improve the accuracy of the traditional radar algorithms. This mixing between the KB and traditional radar signal processing requires, however, different types of computations not usually seen in traditional signal processing radar architectures. This pre-/co-processing must be characterized in terms of crucial hardware and software design constraints for computing architectures.

While the use of knowledge-based and traditional signal processing algorithms is not expected to drastically increase the total number of floating-point-operations per second [5], the micro-architecture hardware control logic required to implement these knowledge-based representations and optimally blend the traditional radar signal processing with KB processing, imposes noticeable constraints pertaining not only to processing, but also to the embedded architectural parameters, such as memory latency, I/O throughput, threaded performance, data locality, etc. These new radar performance requirements have strong implications on the underlying hardware architecture, requiring schemes capable of adapting in real-time with the agile KB and DSP algorithms that are necessary to fully realize the expected gains in radar performance.

Traditional GMTI and SAR processing has yielded highly optimized, static ASIC computing architectures such as parallel systolic arrays capable of achieving up to tera-FLOPS performance in radar kernels such as QR factorization. These architectures are dependent on pre-defined computational schema, where the number of streaming input channels, data throughput, and memory access rates are constant. Adding intelligent signal and KB processing techniques to this system may provide cleaner input data to feed such a systolic processor, however it creates dynamic arrival latency. In KB processing, filter weights selected by the in-situ environment reside in large databases, which depending on the effectiveness of the memory hierarchy can result in nanoseconds to milliseconds worth of delays. For the throughput rates required in modern GMTI/SAR radar systems, this will violate real-time constraints, resulting in dropped data and performance degradation. This processor versus memory speed gap issue is becoming more critical, as traditional microprocessor performance has been doubling every 18-24 months, while DRAM (main memory) latency has only been improving 7% per year [30].

The first step of this study was to analyze the total memory, processing throughput, and memory bandwidth required in order to mix the execution of conventional signal processing with knowledge-aided algorithm data accesses. These specifications are driven by what type of database is being referenced, the database access rate, and how much pre-processing needs to be performed on the a priori data. We considered three types of databases: 1) a database of pre-computed weights for each cell the platform is located in 2) a database of intermediate data, such as eigenvectors, for each platform cell, and 3) a database of raw data such as SAR or LULC. Using reasonable radar parameters [31] and assuming a database large enough for a typical mission of 100,000 sq km, we found range of total database sizes assuming fine resolution, typical of SAR at 2.5 m x 2.5 m, and coarse resolution, typical of LULC at 30 m x 30 m. This analysis showed that storing pre-computed weights for each cell is infeasible as it requires 10’s of Peta Bytes of storage. The second scenario is more realistic as it requires 10’s of Tera Bytes of storage, but is still unwieldy. The last scenario requires only about 100 Giga Bytes, or a single rack. So from a total memory standpoint, the third scenario, storing the original databases, seems the most desirable. Considering that we have only analyzed 2 dimensional a priori data, and 3 dimensional databases, such as Digital Elevation Maps (DEM), will increase each scenario’s memory requirements by at least another order of magnitude, strengthens the case for storing the raw databases. Another appealing aspect of storing the raw databases is that it simplifies the logistics of fielding such a system as new databases can be uploaded into the architecture immediately without the need for intermediate processing. The trade-off is that a significant amount of on-platform processing must now be subsumed in order to process the filter weights in real-time.

Now that the database type has been determined, we determined the processing throughput and memory bandwidth requirements. The weights for the combined STAP with Knowledge-Aided Pre-Whitening algorithm are defined [32] as:

$$\mathbf{w} = \frac{\left(\mathbf{R_{xx}} + \beta_{\mathbf{d}}\mathbf{R_c} + \beta_{\mathbf{L}}\mathbf{I}\right)^{-1}\mathbf{v}}{\mathbf{v^H}\left(\mathbf{R_{xx}} + \beta_{\mathbf{d}}\mathbf{R_c} + \beta_{\mathbf{L}}\mathbf{I}\right)^{-1}\mathbf{v}} = \frac{\left(\mathbf{R_{xx}} + \mathbf{Q}\right)^{-1}\mathbf{v}}{\mathbf{v^H}\left(\mathbf{R_{xx}} + \mathbf{Q}\right)^{-1}\mathbf{v}} \qquad (1)$$

where $\mathbf{R_{xx}}$ is the input data covariance matrix, $\mathbf{R_c}$ is the a priori knowledge-based covariance matrix, with loading factor $\boldsymbol{\beta_d}$, $\boldsymbol{\beta_l}\mathbf{I}$ is a standard diagonal loading term and $\mathbf{v}$ is the steering vector. To reduce the number of compute operations in the system, this calculation is typically performed in the data domain, where the input data $\mathbf{x}$ is defined as

$$\mathbf{R}_{\mathbf{xx}} = \mathbf{xx^H} \qquad (2)$$

For the Knowledge-Aided case, this now becomes

$$\mathbf{R_{CL}} = \mathbf{x}'\mathbf{x}'^{\mathbf{H}} = [\mathbf{x} \quad \mathbf{C}] \begin{bmatrix} \mathbf{x^H} \\ \mathbf{C^H} \end{bmatrix} = \mathbf{R_{xx}} + \mathbf{Q} \qquad (3)$$

where $\mathbf{Q} = \mathbf{CC^H}$ is the Cholesky decomposition of **Q**. In this manner, the Knowledge-Aided processing can be seen as formulating **Q** and computing its Cholesky **C**, where **C** is then passed on to a traditional systolic array STAP processor much as is done with diagonal loading terms currently.

In summary, the Knowledge-Aided Pre-filtering Architecture's tasks are to 1) access the relevant database locations, 2) formulate **Q** by finding the desired eigenvectors in the current database set, and 3) compute the Cholesky of **Q**. These tasks are to be performed in real-time without adding non-deterministic latency to the system, as these loading factors must be synchronized with the correct Coherent Pulse Interval (CPI) of radar data. The computational kernels that comprise these tasks are multiple matrix-matrix multiplications, eigenvalue decomposition (for this study, SVD was used, but as Knowledge-Aided algorithms mature, other methods may be found suitable), and Cholesky decomposition. In order to bound the latency and real-time requirements, we assume that computations were to be completed in one CPI. Using expected radar parameters [31], this leads to a full Degree of Freedom (DoF) computational throughput of 215 GFLOPS, when N = 352 are the DoF and M = 1800 is the database array length.

It should be noted that 215 GFLOPS constitutes one thread or "guess" of the a priori contributing factors involved and that it may be desirable to execute several threads or guesses in parallel utilizing different types, or mixes of databases and then select the best fit. Such a multi-threaded system will increase the throughput requirements linearly. Also from this data set, we can derive a memory bandwidth specification. In this scenario a matrix of 352 x 1800 complex 32 bit values are used to formulate Q every CPI, resulting in a system memory bandwidth of 316.8 MB/s.

Next, we provide a brief overview of the approaches used to map the Knowledge-Aided Pre-Whitening algorithm to each of these architectures. As the kernels can be broken into two types, data independent (matrix multiply) and data dependant (SVD, Cholesky), we focused on mapping one kernel of each type, matrix multiply and Cholesky.

The Imagine matrix multiplication kernel uses a blocked version in which the input matrix is partitioned into eight by *L* size blocks. Two blocks are read by the ALU clusters. First, each cluster reads the data in the left-most input matrix column. Then, the data in the first ALU cluster is sent to all other clusters. Next each cluster performs the complex multiplication on its own data and the received data. Then, the data in the second ALU cluster is sent to all clusters and complex multiplication is performed. This is repeated until all data in the eight clusters are sent to other clusters and computed. After all the communication pairings are exhausted, this loop is continued on each column of input data. This approach makes full utilization of the high-bandwidth inter-alu cluster network, without having to access deeper memory.

Since the number of the multiplications and additions are the same for complex matrix multiplication, the two multipliers become a bottleneck. Thus, two multiplications and two additions are performed per cycle in each cluster. Software pipelining is used to overlap these computations with the data transfers between memory and the SRF.

In this implementation, the maximum value of L that can be processed without losing performance was 128. When the size is larger than this, the data does not fit in the SRF, which results in significant performance degradation. The number of columns of the data in this application is larger than this, i.e., 352 and 1800. Thus, the matrices need to be partitioned into many 128-sized blocks to obtain optimal performance. This causes interim data to be used, increasing the data transfers.

On Imagine, the optimal algorithm for Cholesky is a blocked out-of-core algorithm [33]. This algorithm minimizes the amount of data transferred between memory and disk. In mapping to the Imagine architecture, the same technique is used to reduce the amount of data transferred between memory and the SRF. Furthermore, the algorithm is tweaked to make best use of the SRF and ALU cluster sizes maximizing the advantage of stream processing.

The input matrix is partitioned into 8 by 8 blocks to correspond to the number of ALU clusters. The matrix is represented as (4) and the algorithm steps are as follows:

$$A = \begin{pmatrix} L_{00} & * & * \\ L_{10} & A_{11} & * \\ L_{20} & A_{21} & A_{22} \end{pmatrix} \qquad (4)$$

```
1.  For all diagonal blocks from left-top to right-bottom
2.      Read $A_{11}$
3.      $A_{11} \leftarrow A_{11} - L_{10} * L_{10}^T$
4.      $A_{11} \leftarrow L_{11} = \text{chol}(A_{11})$
5.      Write $A_{11}$
6.      $A_{21} \leftarrow (A_{21} - L_{20} L_{10}^T) L_{11}^{-T}$
```

In (4), A_{11} is an 8 by 8 block. At first, A_{11} is the left-top block, and L_{00}, L_{10}, and L_{20} are null. In the next iteration, A_{11} is the second left-top diagonal block, L_{00}, L_{10}, and L_{20} are 8 by 8, 8 by 8, and (N-16) by 8 blocks, respectively, where N is the number of rows of A. Most of the computation is in step 6 which is a blocked matrix multiplication. Thus, the same method as in the matrix multiplication is used here.

In our approach on Raw, two key concepts are used to fully leverage the Raw architecture [34]. First, the tile grid is viewed as a decoupled systolic array (DSA) where typically the border tiles are used to store and load data to memory and feed the interior tiles, which are used to perform the compute processing. This has the effect of removing loads and stores from the critical computation path of an algorithm. This approach is especially attractive to KASSPER architectures, because it allows direct scheduling and control of the database memory. The second key concept is to program a Raw chip in the context of streams. In this approach the interior computational tiles utilize the register-mapped high bandwidth tile interconnect to full advantage by performing ALU operations where an input register may be from an input port of the network and the output destination may be an output port. In this manner a typical serial RISC program will be pipelined and divided over several tiles. It is important to note that as we consider a multi-processor environment, or Moore's Law enables more tiles per chip, this approach becomes more efficient as the number of processing tiles increases by N^2 while the number of memory tiles increases by N.

For a matrix multiplication of C = AB, the memory tiles arrange the data streaming into the processing tiles such that the rows of matrix A are aligned along the left hand tiles and are passed from left to right, while the columns of B are aligned along the top tiles and are passed from top to bottom. The processing tiles receive these input data on their network ports and compute the partial product. The input data is then passed along to the neighboring tiles, while the partial product is accumulated.

The Cholesky implementation on Raw follows the same algorithm that was used for Imagine, [33] coupled with the streaming approach outlined in the matrix multiplication. Here, a similar matrix multiplication approach is utilized for steps 3 and 6 of the algorithm. Step 4 consists of the standard Cholesky formula [35] of taking the square root of the diagonal element, dividing the remaining elements in the row by this value, and performing some vector multiplications to update the block. This step can also be implemented in the streaming method, however here Raw experiences some inefficiencies as the square root function is implemented in software and has a relatively large latency.

The results of the Knowledge Aided Pre-Whitening algorithm were obtained by using cycle-accurate simulators provided by the Imagine and Raw teams. For comparison purposes, actual measurements of performance were taken using a single node of a 500 MHz PowerPC G4 based system. Table 3 summarizes the results.

Kernel	PPC G4	Imagine	Raw
Clutter Covariance Formation	0.680 sec	0.186 sec	0.372 sec
SVD	7.05 sec	1.91 sec	1.77 sec
R_{known} Formation	0.159 sec	0.0364 sec	0.0209 sec
Cholesky Decomposition	0.123 sec	0.00953 sec	0.0316 sec
Total	**8.01 sec**	**2.14 sec**	**2.19 sec**

Table 3. KASSPER Performance Results

The experimental processors each take different approaches, but for this application the results are similar. Imagine provides a 3.8x throughput increase and Raw provides a 3.7x throughput increase over the PowerPC. In large part, this is because the PowerPC can operate near its peak throughput of 4 GFlops/sec using Multiply-Accumulate instructions on matrix multiplication type kernels, which constitute over 54% of the floating point operations in this application. For these data independent kernels, clutter covariance formation and R_{known} formation, Imagine and Raw achieve a 3.8x and 2.1x speedup, respectively. It is important to note that the reason for Raw's lower performance is that it does not have a Multiply-Accumulate instruction. This simple improvement would nearly double the performance of Raw.

For irregular, data-dependent kernels like SVD and Cholesky, the experimental processors perform better. For Cholesky, Imagine and Raw achieve a 12.9x and 3.9x improvement over PowerPC respectively. For SVD, Imagine achieves a 3.7x speedup and Raw achieves a 4.0x speedup. In these kernels, Imagine benefits from being able to keep large blocks of inter-dependant data readily available in its SRF. Raw increases its performance by pipelining serial computations over several tiles, keeping all its ALUs busy each clock cycle. However, the PowerPC has idle Altivec ALUs while calculating data dependent operations.

The KASSPER architecture is a system-level architecture. Though single-chip performance for these experimental architectures could drastically reduce the size of a KASSPER system, a multi-processor solution must also be evaluated. Although for single chip performance Imagine and Raw have similar performance, in multi-processor systems the benefits of Raw are expected to increase as the efficiency increases as the number of tiles and the general purpose nature of each individual tile provides a homogeneous system that allows easier adaptation to system-level overhead routines.

3. Hardware

a. Raw Hardware

(1) Raw Handheld

The AMP team, in collaboration with MIT, designed and fabricated the Raw Handheld board. This board was used to test the Raw chip, validate the simulated performance results, and to provide a platform for software development for Raw and PCA. Figure 27 shows a block diagram of the board, which contains one Raw chip, PC133 SDRAM memory, a PCI bus with three slots, a programmable clock source, and several I/O connectors (including UART, A/D and D/A, and an LCD). The board uses FPGAs to implement glue logic between the Raw chip and memory and I/O devices. The Raw Handheld board is capable of being used as a stand-alone board with boot-up information residing in the flash memory and the board can also be booted by loading a program from a host PC (running Linux) through the expansion connector. The MIT team has ported the dbx debugger to the board. Peak computational power of the Raw chip is 4.6 GFLOPS (at 290 MHz) and the board supports up to 4 GB of memory in standard DIMM packages.

A photograph of the board running in the laboratory is shown in Figure 28. The board was integrated into a PC case and was used for a DARPATech demonstration in 2004. Raw Handheld boards were also deployed to users including MIT Lincoln Laboratory, Lockheed Martin Advanced Technologies Lab, Utah State, and MIT. The board was used to collect application performance results, which closely matched results collected on the simulator.

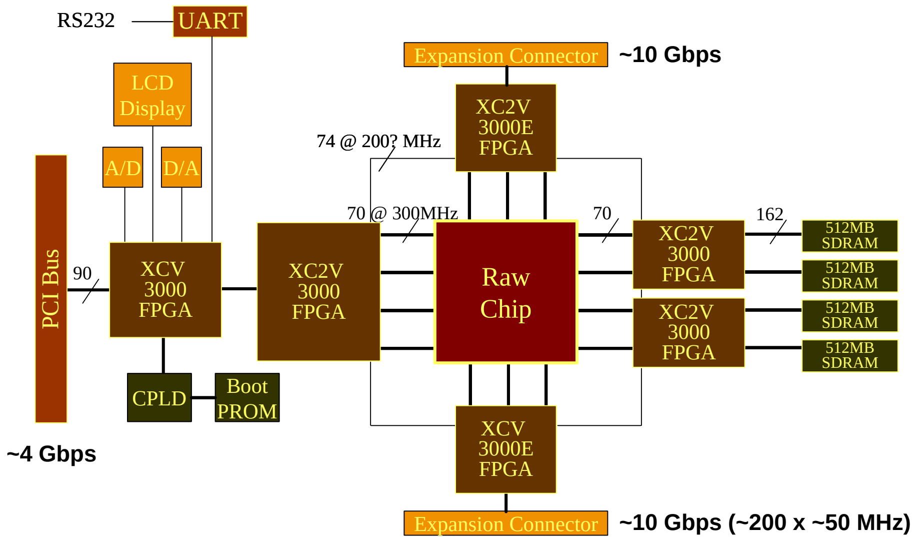


Figure 27. Raw Handheld Block Diagram

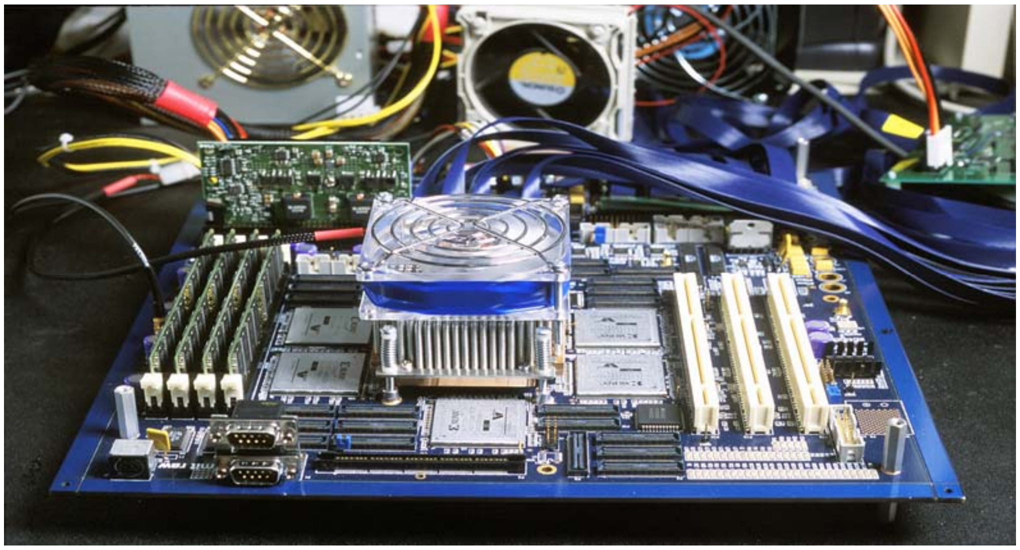


Figure 28. Raw Handheld Laboratory Photograph

(2) Raw Fabric

The AMP team also designed and fabricated a Raw Fabric system. The Raw Fabric is implemented as a mesh of smaller 2x2 boards with connectors to facilitate short ribbon cables between boards. These smaller boards allow testing and replacement of smaller subsystems and are also independently usable as small Fabric systems. A Fabric system consists of two types of boards. The Raw Fabric concept is shown in Figure 29. The processor boards each have four Raw chips, implemented in a 2x2 array, along with DC-DC converters and clock synchronization circuitry. The DC-DC converters are necessary because power is distributed at 48 volts to reduce power loss across the power distribution system and then converted to the voltages needed at the chip. The second kind of board in a Raw Fabric system is an I/O board. I/O boards are distributed around the perimeter of the Fabric system, and implement the main memory controllers and DRAM, PCI, and other connectors.

The AMP team intended to build a 64-processor Raw Fabric. However, due to unexpected problems in manufacturability of the boards, we built two working Raw Fabric systems, each with four processors (with accompanying I/O boards). Unreliability of the connectors between boards made the cost of developing a working, 64-processor infeasible with the remaining scope and resources of the project. A working Raw Fabric system is shown in Figure 30.

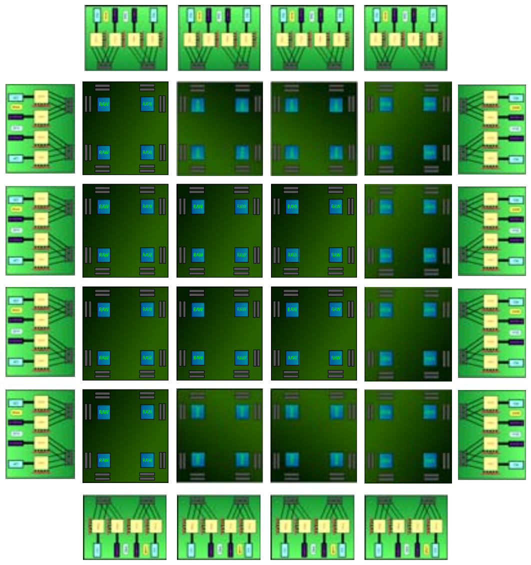


Figure 29. Raw Fabric System Concept

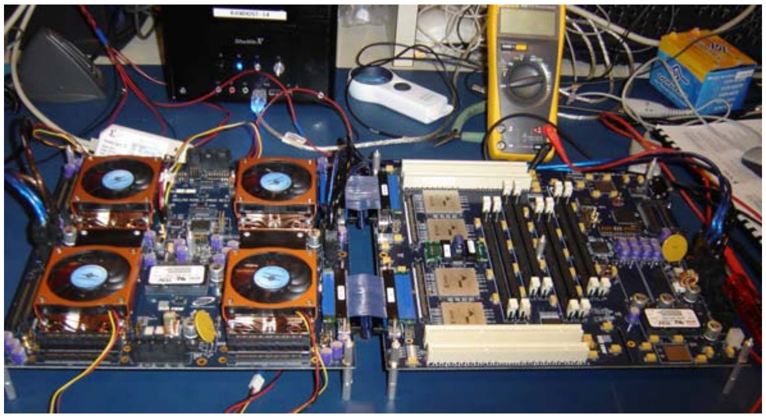
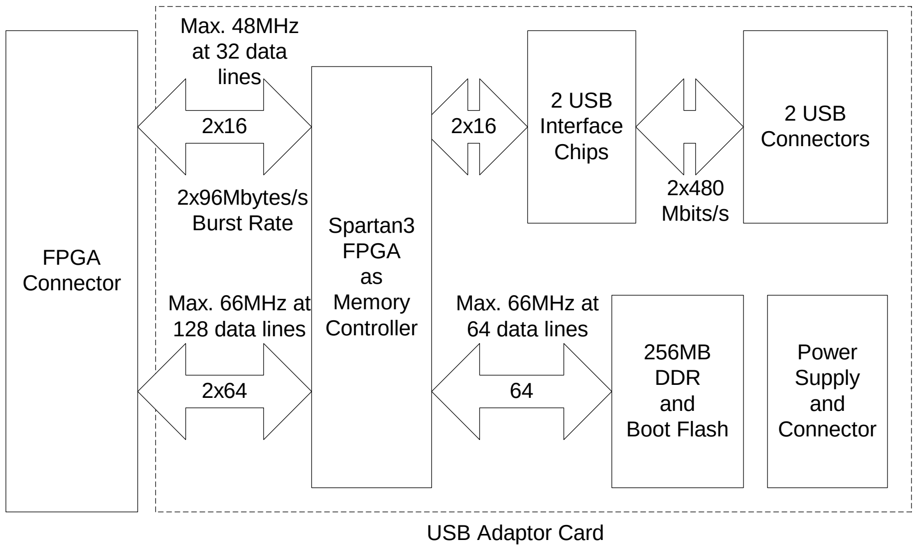


Figure 30. Raw Fabric System Running in Laboratory

b. TRIPS USB Interface

The AMP team also designed a USB interface board for the TRIPS system, to complement work being done under the TRIPS/XTRIPS effort. The design for this board was completed under the AMP project, while fabrication and testing will be completed by the TRIPS team (which includes USC/ISI). Figure 31 is the block diagram of the TRIPS USB Interface Board. The USB Adaptor Board is connected to the TRIPS Motherboard through one of the 190-pin FPGA Connectors. All the signal pins on the FPGA Connectors are connected to the Xilinx FPGA (XC2VP40) on the TRIPS Motherboard. There are two main sub-systems on the USB Adaptor Board.

The USB sub-system contains two separate USB channels. Each channel consists of one USB Interface Chip and one USB connector. Each USB Interface Chip has one 16-bit parallel port. This control port is mapped into the memory space of the PPC405 Core inside the FPGA, which handles all the low-level device control and provides an API for the applications running on the TRIPS chips.



FPGA Connector
Max. 48MHz at 32 data lines
2x16
2x96Mbytes/s Burst Rate
Max. 66MHz at 128 data lines
2x64
Spartan3 FPGA as Memory Controller
2x16
2 USB Interface Chips
2x480 Mbits/s
2 USB Connectors
Max. 66MHz at 64 data lines
64
256MB DDR and Boot Flash
Power Supply and Connector
USB Adaptor Card

Figure 31. TRIPS USB Interface Board Functional Block Diagram

The memory sub-system consists of a 4MB Flash and a DDR SODIMM socket supporting at least 256MB of DDR memory. A Xilinx Spartan3 FPGA is used to provide the Memory Controller function to the PPC405 Core on the TRIPS motherboard. The Flash stores a stripped-down version of Linux to support the low level USB control tasks. The DDR provides the needed memory space to run the Linux on the PPC405 Core.

USB may operate at any speed from 10kbps to 480Mbps in one of three speed modes. A Slow-Speed mode of 10kbps to 100kbps is used for devices such as a USB keyboard or USB mouse. Full-Speed mode is used by most devices and allows a transfer rate of 500kbps to 10Mbps. High-Speed mode (defined by USB 2.0) allows rates of up to 480Mbps, with a speed range of 25Mbps to 480Mbps.

The USB Adaptor Board contains a Cypress Semiconductor CY7C68013A EZ-USB Microcontroller chip. This chip is the same USB interface chip used in the Opal Kelly XEM3010 board, chosen as a demonstration vehicle for collaborators at AFRL. This chip has an integrated USB Transceiver, a Serial Interface Engine, and an 8051 Microcontroller core. Each channel of USB port contains one CY7C68013A USB Controller chip and one USB connector. The CY7C68013A supports up to 480Mbits/s at its USB serial interface and 96Mbytes/s burst rate at its parallel interface. The FPGA Interface Connector Bus supports at least 100MHz of operation. There are two USB channels on this board, each with a 16-bit data interface. With a maximum burst rate of 96Mbytes/s, the maximum required data toggling rate is 96M x 8 / 16 = 48MHz, which is well under what the FPGA connector can support. Each USB channel’s maximum data rate is 480Mbit/s, which translates into 480M / 8 = 60Mbytes/s, which is under the maximum burst rate supported by the USB chip parallel interface.

4. Conclusions

The AMP project made important contributions to the understanding of the programming of polymorphous computing architectures. Polymorphous architectures are a promising technology for exploiting explicit on-chip parallelism, which the microprocessor industry has recognized is necessary for the survival of the industry. Software developed under the PCA program is sufficient for developing some applications on these architectures, but programming is still a significant unsolved challenge for many types of applications. Architecture-specific tools help, but require specialized architectural expertise that many application developers will not have. The Morphware two-level compiler is intended to work at a higher level, but still has performance challenges to be solved, including extracting parallelism and targeting polymorphous architectures more efficiently. The tools that the AMP project developed are invaluable for exploring these programming paradigms, but are not primarily intended for use by application programmers outside of the architecture and compiler research communities. The porting of standard interfaces like MPI and VSIPL will help these architectures to gain acceptance, but these standards were not developed specifically for polymorphous architectures and will not sufficiently exploit the capabilities of these architectures by themselves. The application performance studies that the AMP team has conducted have shown that polymorphous architectures do have significant performance potential for a wide variety of important applications, so the continued investment in software for these architectures is critical. Additional hardware research also would likely lead to improved programmability.

The prototype hardware that the AMP team developed has enabled the development of realistic and complex applications for polymorphous architectures. Simulations are inherently too slow to run realistically sized applications. Large applications must be developed to motivate and test the software infrastructure needed to make polymorphous computing useful to the DoD community. The AMP hardware has provided a platform that facilitates this development.

5. References

[1] The Morphware Forum, "Introduction to Morphware: Software Architecture for Polymorphous Computing Architectures," Version 1.0, Feb. 23, 2004. Available at http://www.morphware.org.

[2] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, J. W. Lee, P. Johnson, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs," *IEEE Micro*, 22(2), pp. 25-35, March/April 2002.

[3] S. Lohani, and S. S. Bhattacharyya, "System Synthesis for Polymorphous Computing Architectures," Technical Report UMIACS-TR-2002-12, Institute for Advanced Computer Studies, University of Maryland at College Park, February 2002. Also Computer Science Technical Report CS-TR-4330.

[4] W. Gropp, E. L. Lusk, N. E. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing,* 22(6): 789-828 (1996).

[5] M. Wicks, "Incorporating Knowledge Base Techniques in Radar Signal Processing – Past, Present and Future," KASSPER Industry Day, Washington, DC, April 3, 2002.

[6] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe and A. Agarwal, "Baring it all to Software: Raw Machines," *IEEE Computer*, September 1997, pp. 86-93.

[7] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, Marcel Dekker, 2000.

[8] S. S. Bhattacharyya. *Hardware/Software co-synthesis of DSP systems*. In Y.H. Hu, editor, *Programmable Digital Signal Processor: Architecture, Programming, and Applications*, pp.333-378. Marcel Dekker, Inc., 2002.

[9] T. Blickle, J. Teich, and L. Thiele, "System-level Synthesis Using Evolutionary Algorithms." *Journal of Design Automation for Embedded Systems*, 3(1):23-58, 1998.

[10] E. Zitzler and L. Thiele, "Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto approach." *IEEE Transactions on Evolutionary Computation*, 3(4): 257-271, November 1999.

[11] F. Chudak, "Improved Approximation Algorithms for Incapacitated Facility Location," In R. E. Bixby, E. A. Boyd, and R. Z. Rios-Mercado, eds., *Integer Programming and Combinatorial Optimization*, Springer LNCS Vol. 1412, 180-194, 1998.

[12] T. Cormen *et al*., *Introduction to Algorithms*, McGraw Hill, 2000.

[13] K. Jain and V. V. Vazirani, "Approximation Algorithms for Metric Facility Location and K-median Problems Using the Primal-dual Scheme and Lagrangian Relaxation," *Proc. Foundations of Computer Science*, 1999.

[14] D. B. Shmoys, E. Tardos, and K. I. Aardal, "Approximation Algorithms of Facility Location Problems." *Proc. 29th ACM Symp. On Theory of Computing*, 265-274, 1997.

[15] A. Y. Zomaya, "Parallel and Distributed Computing: The Scene, the Props, the Players," *Parallel and Distributed Computing Handbook*, A.Y. Zomaya, ed., ppl. 5-23, New York: McGraw-Hill, 1996.

[16] M. B. Taylor, et. al, "Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams," *Proceedings of International Symposium on Computer Architecture, München, Germany*, June 2004.

[17] P. Mattson, W. Thies, L. Hammond, and M. V. Raytheon, "Streaming Virtual Machine Specification," Version 1.0.1, http://www.morphware.org, March 2005.

[18] J. Suh and J. O. McMahon, "Implementations of FIR for MONARCH Processor," *10th High Performance Embedded Computing Workshop*, Boston, MA, Sept. 2006.

[19] M. Vahey, *et al.*, "MONARCH: A First Generation Polymorphic Computing Processor," *10th High Performance Embedded Computing Workshop*, Boston, MA, Sept. 2006.

[20] D. Burger, S. W. Keckler, and K. S. McKinley, *et al.*, "Scaling to the End of Silicon with EDGE Architectures," *IEEE Computer*, 37 (7), pp. 44-55, July, 2004.

[21] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. C. Burger, and K. S. McKinley, "Compiling for EDGE Architectures," *International Conference on Code Generation and Optimization* (CGO), March, 2006.

[22] F Labonte, P. Mattson, I. Buck, C. Kozyrakis and M. Horowitz, "The Stream Virtual Machine," *13th International Conference on Parallel Architectures and Compilation Techniques*, September 2004.

[23] K. Mai, R. Ho, E. Alon, D. Liu, Y. Kim, D. Patil, and M. Horowitz, "Architecture and Circuit Techniques for a Reconfigurable Memory Block," *International Solid State Circuits Conference*, February 2004.

[24] W. Coate and J. Lebak, "Morphing Scenarios For The GMTI Portion Of The PCA Integrated Radar Tracker," Massachusetts Institute of Technology Lincoln Laboratory, 2004.

[25] C. Kozyrakis, "Scalable Vector Media-Processors for Embedded Systems," Ph. D. dissertation, UC Berkeley, May 2002.

[26] B. Khailany, W. J. Dally, S. Rixner, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, and A. Chang., "Imagine: Media Processing with Streams," *IEEE Micro*, March/April 2001, pp. 35-46.

[27] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. R. Mattson, and J. D. Owens, "A Bandwidth-Efficient Architecture for Media Processing," 31st Annual International Symposium on Microarchitecture, Dallas, Texas, November 1998.

[28] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, W. Lee, A. Saraf, N. Shnidman, V. Strumpen, S. Amarasinghe, and A. Agarwal, "A 16-issue Multiple-Program-Counter Microprocessor with Point-to-Point Scalar Operand Network," *Proceedings of the IEEE International Solid-State Circuits Conference*, February 2003.

[29] Apple, http://www.apple.com/powermac/, 2002.

[30] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd Edition, Morgan Kaufman Publishers, Inc., 1996.

[31] J. Bergin and P. Techau, “High Fidelity Site-Specific Radar Simulation” KASSPER ’02 Workshop Datacube,” ISL Tech. Report ISL-SCRD-TR-02-105, May 2002.

[32] C. Teixeira, J. Bergin, and P. Techau, “Reduced Degree-of-Freedom STAP with Knowledge-Aided Pre-Whitening,” 2003 KASSPER Workshop, April 2003.

[33] B. C. Gunter, W. C. Reiley, and R. A. Van de Gejin, “Parallel Out-of-Core Cholesky and QR Factorizations with POOCLAPACK,” IPDPS, San Francisco, CA, April 2001.

[34] H. Hoffman, V. Strumpen, and A. Agarwal, “Stream Algorithms and Architecture,” Laboratory for Computer Science,” MIT, Technical Memo MIT-LCS-TM-636, March 2003.

[35] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing,* Cambridge University Press, 2002, pp 96-98.

6. Acronyms

AB, ABF	Adaptive Beamforming
ACIP	Architecture for Cognitive Information Processing
A/D	Analog-to-Digital
AFRL	Air Force Research Laboratory
ALU	Arithmetic Logic Unit
AMP	Abstract Machines for Polymorphous computing
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CFAR	Constant False Alarm Rate
CMF	Configuration Modeling (or Management) Framework
COTS	Commercial Off-The-Shelf
CPI	Coherent Processing Interval
CPU	Central Processing Unit
CS	Configuration Store
CSLC	Coherent SideLobe Canceller
D/A	Digital-to-Analog
DARPA	Defense Advanced Research Projects Agency
DC	Direct Current
DDR	Double Data Rate
DEM	Digital Elevation Map
DF	Doppler Filtering
DIMM	Dual In-line Memory Module
DMA	Direct Memory Access
DoD	Department of Defense
DRAM	Dynamic Random Access Memory
DSA	Decoupled Systolic Array
DSP	Digital Signal Processing
EW	Electronic Warfare
FIR	Finite Impulse Response
FFT	Fast Fourier Transform
FLB	Floating-point Lower Bound
FLOPS	FLOating point Operations Per Second
FPGA	Field Programmable Gate Array
GFLOPS	Giga-FLOating point Operations Per Second
GMTI	Ground Moving Target Indicator
GUI	Graphical User Interface
HLC	High-Level Compiler
IFFT	Inverse Fast Fourier Transform
ILP	Instruction Level Parallelism
I/O	Input/Output
IPC	Inter-Processor Communication

IRT	Integrated Radar Tracker
ISI	Information Sciences Institute
IUB	Implementation Upper Bound
KASSPER	Knowledge Aided Signal and Sensor Processing with Expert Reasoning
KB	Knowledge Base
Kbps	Kilo-bits per second
LCD	Liquid Crystal Display
LLC	Low-Level Compiler
LULC	Land Use Land Cover
MB	MegaByte
Mbps	Mega-bits per second
MHz	Mega-Hertz
MIMD	Multiple Instruction Multiple Data
MIT	Massachusetts Institute of Technology
MONARCH	MOrphable Networked ARCHitecture
MPI	Message Passing Interface
PC	Pulse Compression, Personal Computer
PCA	Polymorphous Computer Architecture
PCI	Peripheral Component Interface
PCR	Polymorphic Channelized Receiver
PE	Parameter Estimation
PPC	PowerPC
RISC	Reduced Instruction Set Computer
RF	Radio Frequency
SA	Subband Analysis
SAAL	Stable Architecture Abstraction Layer
SAPI	Stable Application Programming Interface
SAR	Synthetic Aperture Radar
SIMD	Single Instruction Multiple Data
SINR	Signal to Interference and Noise Ratio
SODIMM	Small Outline Dual In-line Memory Module
SRF	Stream Register File
SS	Subband Synthesis
STAP	Space-Time Adaptive Processing
SVD	Singular Value Decomposition
SVM	Streaming Virtual Machine
TBMD	Theater Ballistic Missile Defense
TDE	Time Delay and Equalization
TRIPS	Tera-op Reliable, Intelligently adaptive Processing System
TVM	Treaded Virtual Machine
UART	Universal Asynchronous Receiver/Transmitter
UB	Upper Bound
USAF	United States Air Force
USB	Universal Serial Bus

USC	University of Southern California
VIRAM	Vector Intelligent Random Access Memory
VSIPL	Vector/Signal/Image Processing Library
XML	eXtensible Markup Language