Analytical Modeling of High Performance Reconfigurable Computers:

Prediction and Analysis of System Performance

A Dissertation Proposal for

the Doctor of Philosophy Degree in Electrical Engineering

Melissa C. Smith

March 6, 2002

The University of Tennessee, Knoxville

Major Professor: Dr. Gregory D. Peterson

Table of Contents

Table of Figures

1 Introduction

1.1 Motivation

Integration of methodologies and techniques from parallel processing or High Performance Computing (HPC) with those of Reconfigurable Computing (RC) systems presents the potential for increased performance and flexibility for a wide range of problems. HPC architectures and RC systems have independently demonstrated performance advantages for applications such as digital signal processing, circuit simulation, and pattern recognition. By exploiting the near "hardware specific" speed of RC systems in a Beowolf cluster there is potential for significant performance advantages over other software-only or uniprocessor solutions.

1.1.1 What is HPC?

High Performance Computing or HPC is the use of multiple processors or processing nodes collectively on a common problem. The primary motivation for HPC is to overcome the speed bottleneck that exists with a single processor. The most popular taxonomy for classifying computer architectures was defined by Flynn in 1972 [30]. As shown in Figure 1.1, Flynn's classification is based on the two types of information flow into a processor: *instructions* and *data*. Machines are classified according to the number of streams for each type of information. The four combinations are *SISD* (single instruction stream, single data stream), *SIMD* (single instruction stream, multiple data streams), *MISD* (multiple instruction streams, single data stream), and *MIMD* (multiple instruction streams, multiple data streams). HPC systems can also be classified based on their memory structure. The common programming models are *shared memory*, *distributed memory (*message passing) and *globally addressable*. We will discuss these more in a later section and focus on MIMD architectures.

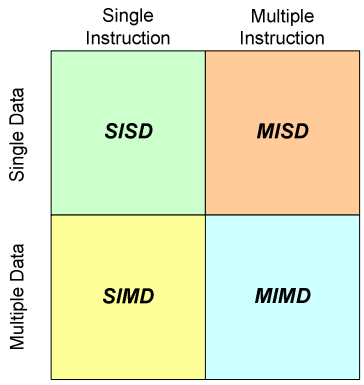


Figure 1.1 Flynn's Taxonomy

1.1.2 What is RC?

Reconfigurable Computing is the combination of reconfigurable logic (normally in the form of FPGAs) with a general-purpose microprocessor. The architectural intention is to achieve higher performance than typically available from software-only solutions with more flexibility than achievable with hardware ASICs. In RC architectures, the microprocessor performs those operations that cannot be done efficiently in the reconfigurable logic such as loops, branches, and possible memory accesses, while computational cores are mapped to the reconfigurable hardware [23]. We will discuss more details regarding RC systems in a later section.

1.1.3 What is HPRC?

High Performance Reconfigurable Computing or HPRC is the architectural combination of High Performance Computing and Reconfigurable Computing. The proposed HPRC platform as shown in Figure 1.2 consists of a number of distributed computing nodes connected by some interconnection network (the architecture can be a switch, hypercube, systolic array, etc.), with some or all of the computing nodes having RC element(s) associated with them. The HPRC platform will potentially allow users to exploit the performance speedups achievable in parallel systems in addition to the speedup offered by reconfigurable hardware coprocessors. Many applications stand to benefit from this architecture: image and signal processing, simulations, among others.

The addition of a configurable network connecting RC elements offers many applications such as discrete event simulation even more performance advantages. The additional network would provide a less inhibited route for synchronization, data exchange, and other communications between processing nodes.

Research by Chamberlain indicates the potential performance improvement from a dedicated synchronization network for synchronous, discrete-event simulations [20,64,65]. The *parallel reduction* network (PRN) proposed by Reynolds et. al., demonstrated the performance advantages from dedicated hardware to support synchronization in parallel simulations [73-75]. The PRN separates the synchronization computation from the application computation, offloading the synchronization overhead from host processors and the host communication network. This additional network could vastly improve performance for not only applications with barrier synchronization events but any application requiring the exchange of large amounts of data. Other research by Underwood et. al. [84], confirms the performance benefits of a specialized configurable network interface card in a Beowulf cluster. The Intelligent Network Interface Card (INIC) assists with both network communications and computational operations. The results presented for a 2-D FFT and an integer sorting algorithm show significant performance benefit for both applications as the cluster size increases.

HPC and RC individually are challenging enough to program and utilize effectively. Combining these powerful domains will require new analysis tools to aid us in understanding the design space. A performance modeling framework with models describing this new architecture will not only help in understanding and exploiting the design space but will be a building block for many of these tools. The performance of the system is affected by architecture variables such as number of nodes, number of FPGAs, FPGA size, processing power, memory distribution, network performance and configuration, just to name a few, and the available permutations make the design space extremely large. Without a modeling framework to assist with the analysis of these issues, tradeoffs cannot be effectively analyzed potentially resulting in grossly inefficient use of the resources.

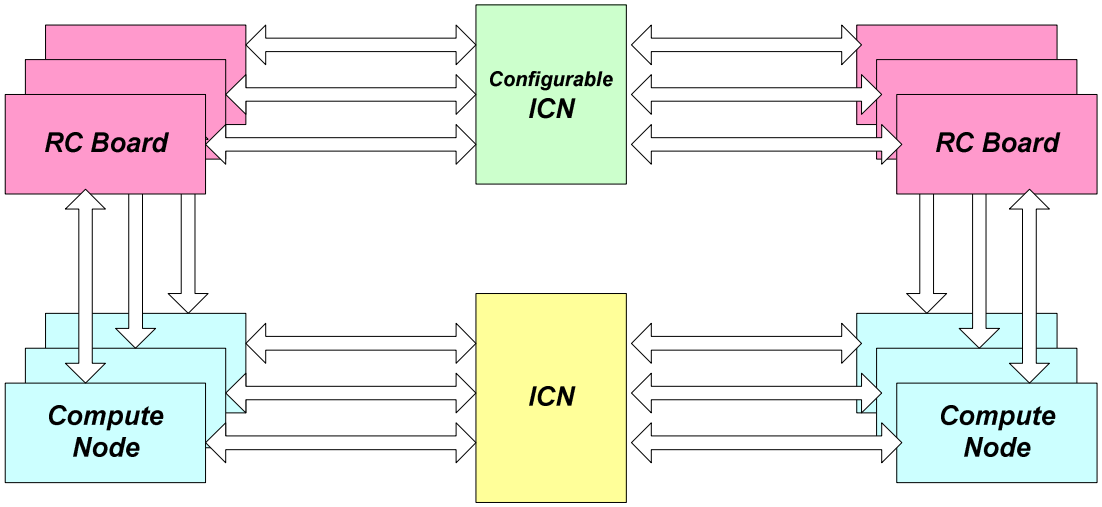


Figure 1.2 High Performance Reconfigurable Computer (HPRC) Architecture

1.2 Problem Statement

Performance analysis and architecture design for HPC and RC systems are challenging enough in their individual environments. For the proposed HPRC architecture, these issues and their interaction are potentially even more complex. Given these observations, it is evident that a mathematical modeling framework is a necessary tool for analyzing various performance metrics. Although substantial performance analysis research exists in the literature with regard to High Performance Computing (HPC) architectures [13,14,21,37,43,59,64,65,69,70,72,83] and even some with respect to Reconfigurable Computing (RC) [24,25,41,42], the analysis of these architectures working together has received little attention to date and currently there is a gap in the performance analysis research with regard to an HPRC type of architecture.

A modeling framework covering the common metrics found in HPC and RC research will fill this gap enabling performance analysis of the architecture and the analysis design tradeoffs. The development of a modeling framework for a complex architecture such as HPRC presents several challenges and questions which will need to be addressed: *modeling communication time* (node-to-node, processor-to-RC unit, and RC unit-to-RC unit), *modeling computation time* (software in processor and firmware in RC unit), *modeling setup costs* (application setup, RC unit configuration, and network configuration), and *modeling load imbalance* (application tasks or data imbalance and hardware versus software imbalance).

1.2.1 Definition of a modeling framework

A mathematical modeling framework provides for the analysis of various performance metrics such as speedup, execution time, efficiency, cost, scalability, area/volume, power, communication bandwidth, and memory capacity. The roots of metrics such as speedup, execution time, efficiency, scalability, communication bandwidth, and memory capacity can be found in high performance computing research. Other metrics such as cost, area/volume, and power are more common in the areas of embedded and reconfigurable computing. Depending on the application and environment, researchers and users will want to study many if not all of these metrics, their interactions and tradeoffs for the HPRC architecture.

As will be proposed here, the performance modeling framework consists of a set of equations used to estimate or predict performance based on a group of specified system attributes. System attributes include: number of nodes, memory size, memory distribution, processor speed, number of RC units, size of FPGA(s), configuration time for RC units, number of reconfigurations per application or iteration, etc. With these equations describing the system's characteristics, cost functions can be developed for metrics such as power, area, execution time, speedup, etc. These cost functions can then be minimized (or maximized, whatever the case) and the corresponding "best" system configuration determined.

Modeling sophisticated computing systems such as those discussed here can be very complex and cause the set of equations to become analytically unsolvable. Tradeoffs will be necessary between modeling detail and accuracy to ensure the usefulness of the model while maintaining the necessary degree of accuracy.

1.2.2 Modeling Functions and Analysis Problems

An obvious use of the proposed modeling framework is the performance evaluation of potential application mappings. With the proposed model, a variety of architectural configurations can be studied and compared without actually implementing them on the real system. The proposed model for the HPRC platform can also serve as the foundation for other task scheduling and load balancing cad tools. Another performance related use of the model, specifically the RC portion of the model, is as a method to classify the computing capability of an RC node which could be useful in the NetSolve [4] and SinRG [5] programs at the University of Tennessee.

Other interesting analysis problems possible with the proposed modeling framework are tradeoff studies of power, size, cost, network bandwidth, etc. For a given application, cost functions can be minimized for a fixed power budget or physical size for instance. The significance of this capability may be lost in traditional HPC environments, but in embedded systems where designs are often constrained by size and battery power, these issues are extremely important.

The modeling framework for the HPRC architecture is also applicable to the growing field of Systems on a Chip (SoC). Many parallels can be drawn between the issues encountered in SoC design and the architectural questions for HPRC. SoC is a self-contained electronic system residing on a single piece of silicon as shown in Figure 1.3. It is likely to contain a processor or controller, multiple-memory modules, analog or mixed signal circuitry, a system bus structure, hard and soft IP cores, and reconfigurable logic. With all of these different types of design styles to contend with, it is imperative that the chip designer have a deep understanding, not only of the functionality of the different subsystems, but also of the complex interactions between the subsystems. SoC design requires front-end planning and feedback on system performance parameters at all stages of the design cycle. A modeling framework such as proposed for the HPRC architecture could provide this performance feedback to the designer. SoC processors and memory modules are very similar to the nodes in an HPRC system and both architectures can also incorporate reconfigurable units. Also, both architectures have a communication backbone in the form of a system bus or some sort of network. They are not only architecturally similar but also share some of the same design methodologies such as hardware/software codesign. With all these similarities, successful development of a framework for the HPRC architecture would also be an advantage for the SoC community.

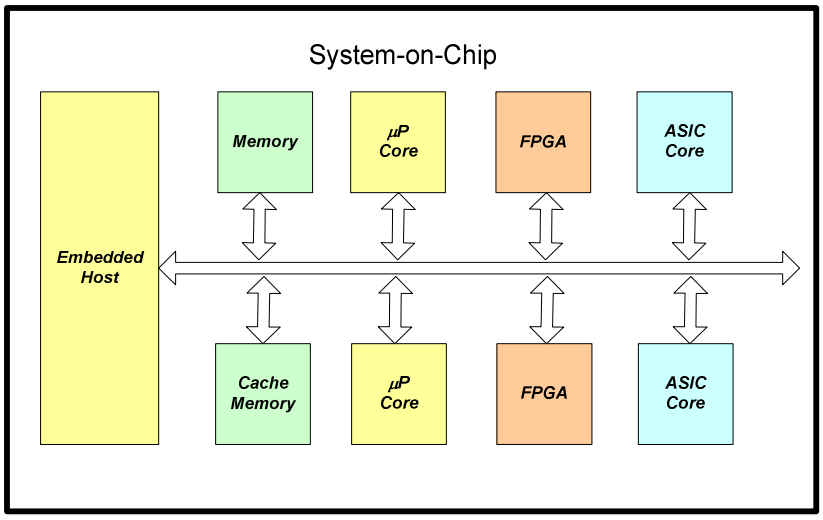


Figure 1.3 SoC Architecture Example

1.3 Goals and Expected Contributions

The main objective of this research is to develop a modeling framework which can be used in the performance analysis of the HPRC architecture and potential application mappings to the architecture. Other expected uses of this framework are as an analysis tool for other HPRC CAD tools such as automated mapping, partitioning, scheduling, etc., RC computation performance prediction, and similar contributions in the field of SoC.

2 Background

2.1 Introduction

In this section we will review some of the architectures found in High Performance Computing (HPC) and Reconfigurable Computing (RC) and introduce the High Performance Reconfigurable Computing (HPRC) platform. We will also look as some performance metrics common in these architectures and how they can be used for HPRC. Next we will look at the issues of performance evaluation and the methods we can employ to develop a modeling methodology or framework for HPRC. Finally, we will look at the development tools and environments that are available today for HPC and RC and how we can use them in the new HPRC platform.

2.2 Architecture

The architecture of a computer system often can affect the performance of the system for a given application. Issues such as dedicated and non-dedicated resources, memory size and distribution, communication busses, and instruction set all affect the performance capability of the system. In this section, we will review some of the common architectures in High Performance and Reconfigurable computing and look at how they can be used in HPRC.

2.2.1 High Performance Computing

HPC research has demonstrated dramatic computing performance advantages over single processor machines for a number of applications. Traditionally these HPC platforms have been limited to the relatively small research oriented market due to the high costs normally incurred in acquiring, developing and programming them. Specialized machines dedicated to parallel processing often consist of an array of tightly coupled homogeneous processors connected by an optimized network. The architecture is often specialized for a specific genre of applications and thus perpetuates the high costs associated with ownership while limiting the efficiency for applications other than the target application.

Using Flynn's taxonomy [30], we can classify HPC architectures in to four categories: *SISD*, *SIMD*, *MISD*, and *MIMD*. While examples of the MISD class are almost non-existent, examples of the SISD or von Neuman architecture include mainframes, early generation PCs and MACs. Examples of SIMD and MIMD architectures are given in Table 1.

Table 1 HPC Architecture Examples [44]

	SIMD	MIMD
Shared Memory	Vector Machines XMP C-90 YMP J-90	Sun HP DEC Tera
Distributed Memory	CPP CM-2 MasPar	KSR Paragon DASH SP2 BBN Butterfly NOWs nCube Clusters CM-5

Several groups have studied the performance of distributed and parallel systems of various architectures. Atallah et.al. developed a model for the performance of a distributed computing environment and use this model to optimize the scheduling problem [14]. Their algorithm attempts to balance the load based on each workstation's "duty cycle" – the ratio of cycles committed for local tasks to cycles available

for parallel tasks. Clement and Quinn [21] developed an analytical performance model for multicomputers. The assumptions and restrictions limit this model to specific architectures and applications. Another paper by Clement [22] focuses on the network performance of PVM clusters. Presented is a communication model for ATM and Ethernet networks including a factor to account for contention in the network. Other work related to the performance of PVM clusters include that by Dongarra and Casanova [18,27,28]. Nupairoj and Ni have studied the performance of MPI on some workstation clusters [66]. Zhang and Yan [89,91] have developed a performance model for non-dedicated heterogeneous Networks of Workstations (NOWs). Heterogeneity is quantified by a "computing power weight", owner workload requests are included as binomial distributions, and the network is limited to Ethernet. Peterson and Chamberlain [69], [70] study application performance in a shared, heterogeneous environment. The performance model focuses on synchronous iterative algorithms and includes the effects of load imbalance across the network and background user load. Issues unaccounted for include network communication and contention models.

The task of programming distributed architectures is complicated and time-consuming. Software libraries such as MPI [79], PVM [33], BLAS [3], and VSIPL [6], have recently made programming in HPC environments easier. Users can now write programs that are not restricted to one architecture but are rather portable to other architectures supported by the tool used. Portability has made programming for HPC environments more cost effective and thus amenable to more users. From a hardware standpoint, recent HPC research in the area of Beowolf clusters is reducing the costs associated with hardware for parallel processing. Beowolf clusters consist of relatively inexpensive commodity workstations connected by a general-purpose network [51]. Beowolf clusters are cost effective and there exists a great deal of support for setup and administration of these systems.

Another architectural alternative, *grid computing*, enables geographically distributed resources to be shared, allowing them to be used as a single, unified resource for solving large-scale computing problems. Like Beowolf clusters, grid computers offer inexpensive access to resources but irrespective of their physical location or access point. The Scalable Intra-campus Research Grid (SInRG) [5] is a grid computer with special system software that integrates high performance networks, computers and storage systems into a unified system that can provide advanced computing and information services (data staging, remote instrument control, and resource aggregation).

2.2.1.1 HPC Example

The focus of the modeling analysis for HPRC will be on systems akin to the Massively Parallel Processors (MPPs), Beowulf Clusters, and grid computers such as SinRG. The Accelerated Strategic Computing Initiative (ASCI) [7] is DoE supported research in the development of MPP platforms. The ASCI program is implemented by project leaders at Los Alamos, Sandia, and Lawrence Livermore National Laboratories, guided by the Office of Strategic Computing and Simulation under the Assistant Secretary for Defense Programs. Beowolf clusters consist of commodity workstations, which may be heterogeneous, connected by a general-purpose network. Since the network may or may not be a dedicated parallel processing environment, the workstations can be multipurpose, serving other users and tasks. The Tennessee Oak Ridge Cluster (TORC) Project [11] is a mixture of Unix and NT workstation nodes at the University of Tennessee and Oak Ridge National Laboratory. TORC currently consists of multiple clusters of up to 64 nodes and is a test bed for evaluating heterogeneous clusters, performance analysis of interconnects and protocols, and software tool development such as NetSolve [4].

2.2.1.2 Metrics for High Performance Computing

To develop a performance model and study the performance of a system, the metrics for evaluation must first be selected. The metrics selected will depend on the architecture features and the issues of interest. HPC performance is commonly measured in terms of speedup and efficiency. *Amdahl's Law* [13] for "fixed-size" speedup and *Gustafson's Law* [37] for "fixed time" speedup are common representations for the limitations to parallel efficiency. The basic definition for *speedup* is the ratio of the execution time of the best possible serial algorithm on a single processor to the parallel execution time of the parallel algorithm on an *m*-processor parallel system:

$$S(m) = \frac{R(1)}{R(m)} \quad \text{where } R \text{ represents the execution time} \qquad \textbf{(2.1)}$$

Efficiency is defined as the ratio of speedup to the number of processors, *m*:

$$Eff(m) = \frac{S(m)}{m} = \frac{R(1)}{m \cdot R(m)} \qquad \textbf{(2.2)}$$

2.2.2 Reconfigurable Computing

Reconfigurable computing (RC) is the coupling of FPGA units to general-purpose processors. The performance of reconfigurable hardware devices such as Field Programmable Gate Arrays (FPGAs) now rivals that of custom ASICs but with design flexibility not available in custom hardware. Future roles of FPGAs and reconfigurable computing include many scientific and signal processing applications [39,40]. Four possible RC topologies are shown in Figure 2.1: (a) Internal Processing Unit – most closely coupled, (b) Co-processing Unit – coupled via a memory or dedicated bus much like an ASIC co-processor, (c) Attached Unit, and (d) Standalone Unit – least coupling. Many of today's computationally intensive applications can benefit from the speed offered by application specific hardware co-processors, but for applications with multiple specialized needs, it is not feasible to have a different co-processor for every specialized function. Such diverse applications stand to benefit the most from the flexibility of RC architectures since one RC unit can provide the functionality of several ASIC co-processors. Several research groups have demonstrated successful RC architectures [17,24,34,35,41,42,45,49,55,60,61,86,90].

There are several RC options available from companies such as Annapolis Microsystems [2], Nallatech [9], Virtual Computer Corporation [12], and research organizations such as University of Southern California's Information Sciences Institute (ISI) [45], The Chinese University of Hong Kong [55], and Carnegie Mellon University [35,61]. The Wildforce [2] and Firebird [2] units from Annapolis Microsystems are both PCI-bus cards with onboard memory. The SLAAC units from ISI [45] are also PCI-bus cards. The Pilchard architecture developed by The Chinese University of Hong Kong [55] which interfaces through the memory bus is more closely coupled and best fits into the co-processor category. The PipeRench reconfigurable fabric [35] is an interconnected network of configurable logic and storage elements which uses pipeline reconfiguration to reduce overhead. We will take a closer look at some of these in a later section.

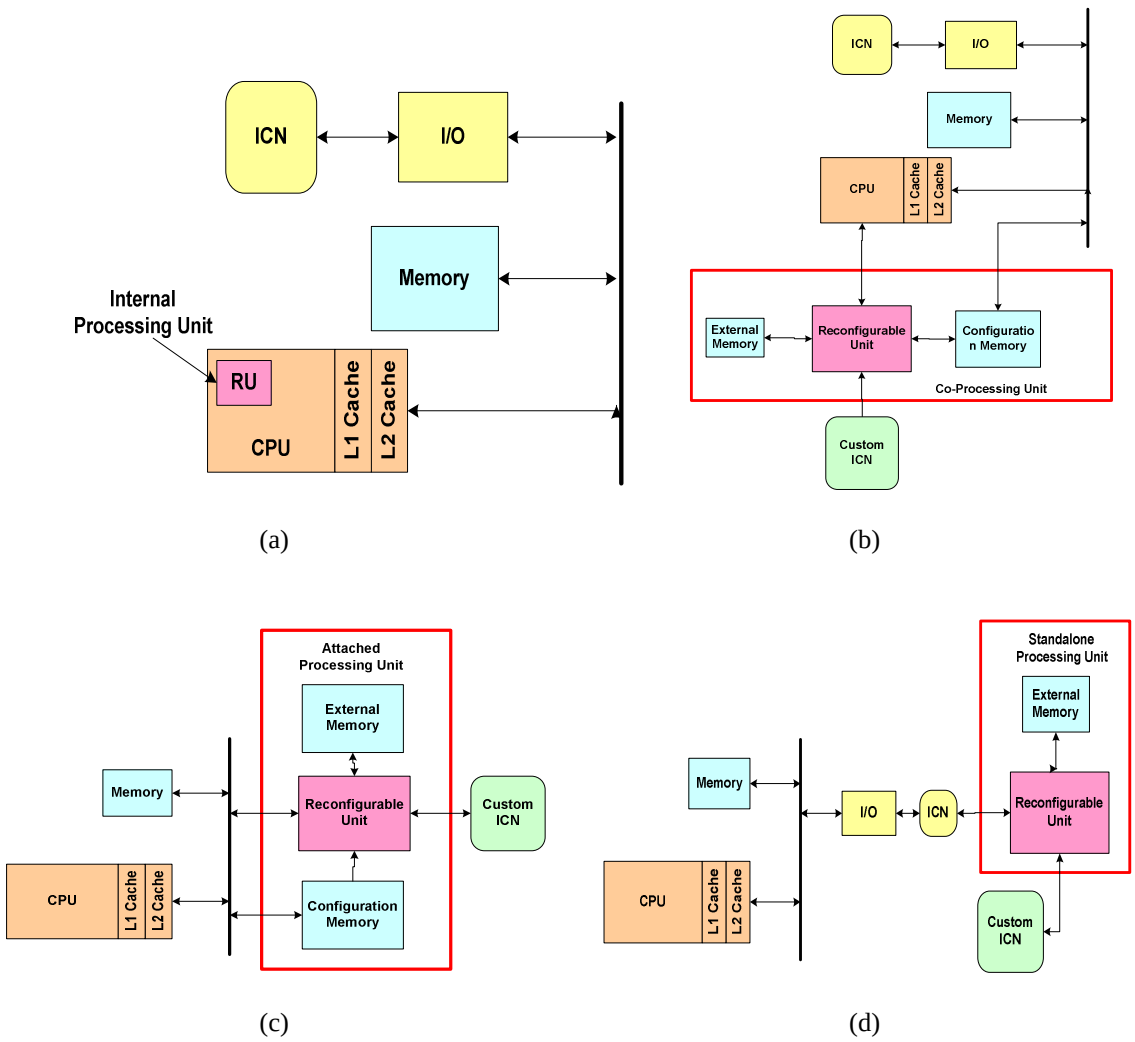


Figure 2.1 Coupling of FPGAs with general-purpose computers

Another area in which reconfigurable devices are becoming popular is Systems on a Chip (SoC). Known as SoPC (Systems on a Programmable Chip), Xilinx [87], Altera [1], and others have developed programmable devices which give the user the flexibility to include user reconfigurable area in addition to sophisticated intellectual property cores, embedded processors, memory, and other complex logic all on the same chip.

2.2.2.1 RC Coprocessor Examples

For testing and validation of the HPRC model, we have available six RC coprocessors: Pilchard [44,55], Annapolis Microsystems' Wildforce and Firebird [2], Altera's UP1 and Excalibur [1], and SLAAC [45].

The Pilchard architecture (Figure 2.2) consists of a Xilinx Virtex FPGA interfaced to the processor via the SDRAM DIMM slot [55]. The logic for the DIMM memory interface and clock generation is implemented in the FPGA. The board also provides an expansion header for interfacing to external logic, memory or analyzer. The FPGA is configured using the download and debug interface which is separate from the DIMM interface. The Pilchard architecture addresses the bandwidth bottleneck issue between the RC unit and the processor by placing the RC unit on the memory bus.

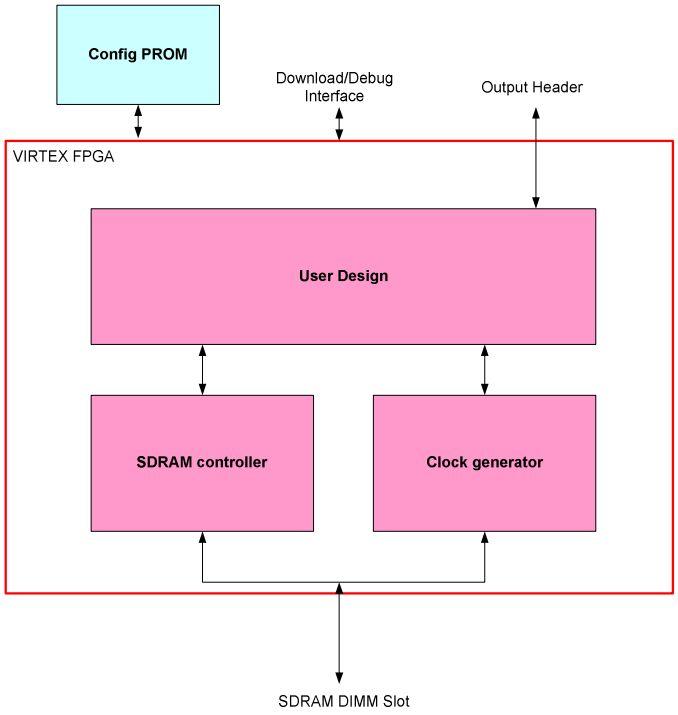


Figure 2.2 Block Diagram of the Pilchard Board

The Wildforce board from Annapolis Micro Systems [2] is a PCI-bus card, which contains four Xilinx XC4000 family chips for computational purposes and a Xilinx XC4036XL chip for communicating with the host computer. Each FPGA on the board has a small daughterboard, which allows external interfaces with the PEs. Each of the PEs on our Wildforce board has 32 KByte of 32-bit SRAM on its daughterboard. A dual-port memory controller is included on the daughterboards to allow both the PEs and the host computer access to the SRAM. A simplified block diagram of the Wildforce board is shown in Figure 2.3**.**

The Firebird board shown in Figure 2.4 consists of a single Xilinx Virtex FPGA. The board is PCI based and includes an onboard PCI controller so that valuable FPGA resources are not used for PCI functions. The device is runtime reconfigurable but not partially reconfigurable. The PCI bus interface supports 66MHz and the five 64 bit wide SRAM banks offer up to 5.4GBytes/sec memory bandwidth. The board also features an I/O connector for external interfaces.

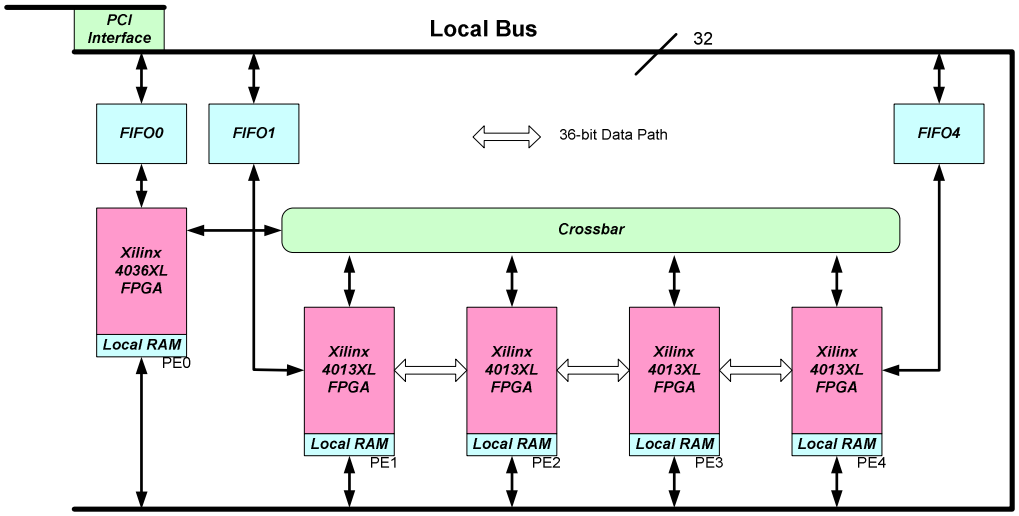


Figure 2.3 Wildforce Architecture [2]

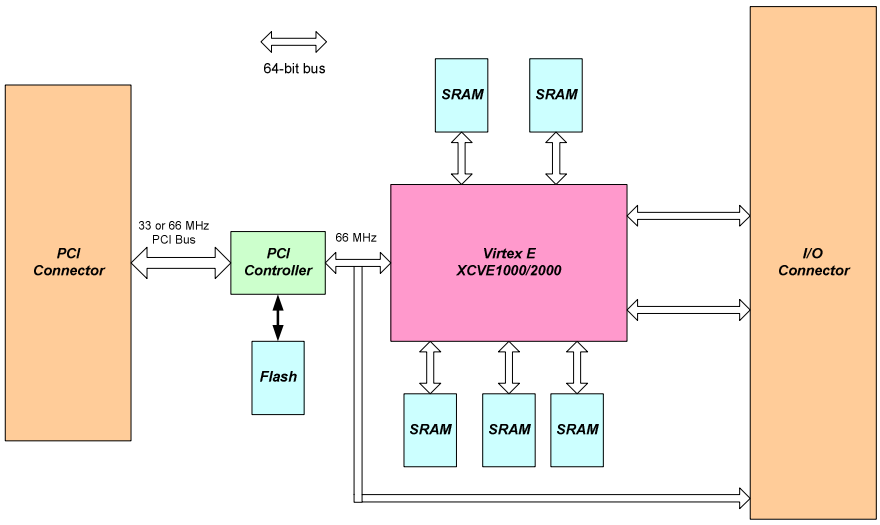


Figure 2.4 Firebird Architecture [2]

The SLAAC1 boards were developed by the University of Southern California's Information Sciences Institute under DARPA's System Level Applications of Adaptive Computing (SLAAC) project [45]. The SLAAC1 board is PCI bus card and has three Xilinx XC4000 processing elements (PEs) as shown in Figure 2.5. The two compute PEs, X1 and X2 have four independent memory ports which provide a fair amount of memory bandwidth for computing applications. X0 is the control PE. The SLAAC1-V board has a similar architecture, but is based on the Xilinx Vertex technology and supports partial reconfiguration.

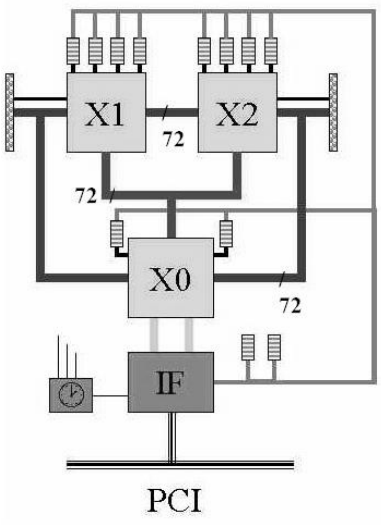


Figure 2.5 SLAAC1 Architecture [45]

2.2.2.2 Metrics for Reconfigurable Computing

In Dehon's thesis on reconfigurable architectures, he presents a high-level characterization of reconfigurable systems based on size metrics [24,25]. Based on these metrics, Dehon attempts to project the performance of the reconfigurable system.

Dehon's characterization model uses size estimates to compare the efficiency of architectures [24]. From an empirical survey, he concludes that reconfigurable devices have lower performance than dedicated hardware but higher than general purpose processors. He concludes that the performance density variation from dedicated hardware to reconfigurable devices to processors results from the increasing amount of instruction memory and distribution resources (area overhead) per active computing element. Dedicated hardware typically has very little overhead while reconfigurable devices and general purpose processors have significantly more overhead. Dehon points out that eighty to ninety percent of the area of conventional reconfigurable devices is dedicated to interconnect and associated overhead such as configuration memory. The actual area used for logic function only accounts for a few percent (Figure 2.6).

The conventional FPGA represents an interconnection overhead extreme. Another extreme occurs where the configuration memory dominates the physical area as with general purpose processors. Figure 2.7 shows the area tradeoffs for Dehon's RP-space. For conventional FPGAs, multi-context FPGAs and general purpose processors, Figure 2.7 graphically shows where processors and FPGAs are in the characterization space for (a) interconnect overhead versus configuration memory and (b) user logic versus configuration memory.

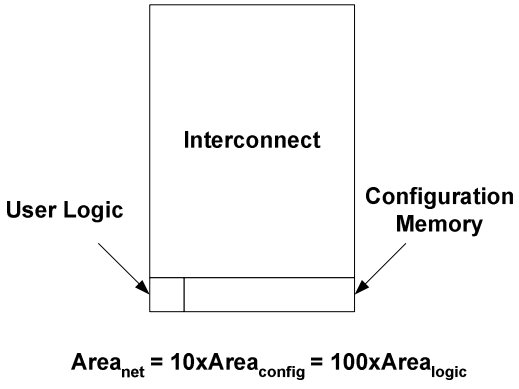


Figure 2.6 Area Density for Conventional Reconfigurable Devices [24]

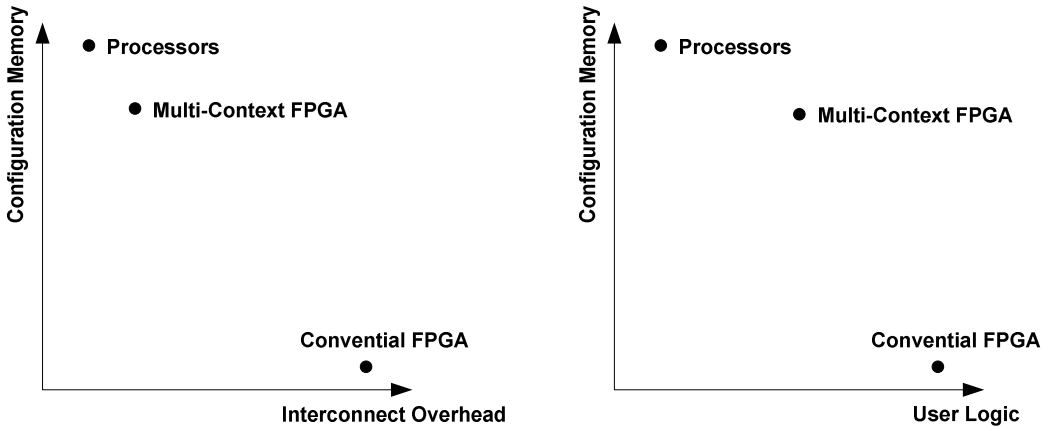


Figure 2.7 RP-Space (a) Interconnect vs. Configuration and (b) Logic vs. Configuration

Dehon's RP-space provides a good method for characterizing reconfigurable devices or FPGAs and even includes some abstraction of processors onto the RP-space. RC architectures however consist of FPGA(s) and a processor working together making their projection into the RP-space somewhat difficult. To effectively compare tradeoffs in the RC and ultimately the HPRC environment, other metrics often used in RC systems, such as cost and power, may be more practical and useful.

Developing a cost metric should be straightforward based on the processors, FPGAs, memory, and interconnect under consideration. A cost function can be developed relating execution time or speedup as determined from the proposed model to the cost of the system. Similarly a power metric cost function could be developed relating execution time determined from the proposed model to the total power used by the system.

We can also discuss *speedup* and *efficiency* for reconfigurable computing. Rather than processor nodes running software algorithms in parallel on a common problem as in HPC, in RC we have hardware and software algorithms working together on a common problem. In general, we will define *RC speedup* as the ratio of the execution time of the best possible software only algorithm on a single processor to the execution time of the RC algorithm executing in software and hardware:

$$S_{RC} = \frac{R_{software}}{R_{RC}} \quad \text{where } R \text{ represents the execution time} \qquad \textbf{(2.3)}$$

We should note that depending on the algorithm and implementation, there may be a hardware/software load imbalance. We will discuss this further in a later section but this essentially means

that the execution may have a percentage of hardware/software overlap ranging from fully parallel (both are busy 100% of the time) to no overlap where the passing from software to hardware is essentially serial. A higher percentage of overlap will result in the greatest potential for speedup leading to the idea of efficiency. The definition of *RC efficiency* will be slightly different from HPC efficiency. For HPC, we have software tasks divided among a number of processing nodes all working in parallel to produce some amount of speedup. The efficiency is a result of this speedup relative to the overhead induced from the number of nodes working in parallel. For RC, we have a software task and one or more hardware tasks, *n*. The efficiency will be the speedup achieved relative to the additional RC system overhead (setup, communication, configuration, etc.). We will assume this overhead is related to the total number of hardware and software tasks even though these tasks are not necessarily performing the same functions, resulting in our definition for *RC efficiency* as follows:

$$Eff_{RC} = \frac{S_{RC}}{n+1} = \frac{R_{software}}{(n+1) \cdot R_{RC}} \qquad \textbf{(2.4)}$$

Further analysis is required to determine if this is an appropriate estimate of RC efficiency and how the hardware and software tasks are defined. Another working model is considering the idea of a P_{dff} value for the RC unit representing the effective processing capacity of he reconfigurable hardware. These ideas will be explored during the proposed research and conclusions presented with the final results.

2.2.3 High Performance Reconfigurable Computing (HPRC)

The proposed HPRC platform is a combination of the HPC and RC architectures. HPRC consists of a system of RC nodes connected by some interconnection network (switch, hypercube, array, etc.). Each of the RC nodes may have one or more reconfigurable units or FPGAs associated with them. This architecture as stated before provides the user with the potential for more computational performance than traditional parallel computers or reconfigurable coprocessor systems alone.

The HPRC architecture has almost limitless possibilities. Starting with the roots of HPC, there are many network topologies (hypercube, switch, etc.), memory distributions (shared, distributed), and processor issues (instruction set, processing power, etc.) to choose from. These options alone make performance analysis complicated and interesting. Adding in the options available in RC such as coupling of FPGA to processor (attached unit, coprocessor, etc.), number of FPGA units, size of FPGA(s), separate

or shared memory for FPGA(s), dedicated interconnection network, among others, and the analysis problem becomes enormous. Getting a handle on these issues and their affect on the system's performance will be integral in exploiting this potentially powerful architecture. The handle is in the form of a modeling framework to analyze these issues.

2.2.3.1 HPRC Hardware

There are two HPRC clusters that will be available for developing and validation of the model. The Air Force Research Laboratory in Rome, NY is assembling a "heterogeneous HPC" which is a cluster of four Pentium nodes populated with Firebird boards. Future plans include expanding to more nodes. The HPRC cluster at UT consists of eight Pentium nodes populated with Pilchard boards. Other hardware available includes a cluster of Sun workstations as listed in Table 2.

Table 2 Sun Workstations

Hostname	Model	RAM	Disk
vlsi1	Sun 220R: dual 450MHz UltraSPARC II	2GB	2x36 GB; DVD; 218 GB array
vlsi2 - 11	Sun 220R: dual 450MHz UltraSPARC II	1GB	18 GB; DVD
microsys0	Ultra 1: 143MHz UltraSPARC I	192MB	11x2.1 GB
microsys3	SPARC 5	32MB	
microsys5	SPARC 5	128MB	
microsys6	SPARC 5	128MB	
microsys7	Ultra 1: dual 200MHz UltraSPARC I	256MB	2GB; 4x9 GB; CDROM
microsys8	Ultra 5	128MB	8GB; CDROM
microsys9	SPARC 5	32MB	

2.2.3.2 Metrics for High Performance Reconfigurable Computing (HPRC)

Again for performance metrics, the first issues normally considered are speedup and efficiency. After the hardware is assembled and testing begins, we will have a better idea of the most meaningful metrics for HPRC. For now we can discuss the metrics used in HPC and RC and later determine if these make sense for HPRC or if we will develop new metrics specifically for HPRC. As a start, our general definition for *HPRC speedup* is the ratio of the execution time of the best possible serial software only algorithm executing on a single processor to the parallel execution time of the parallel HPRC algorithm executing in software and hardware at some load balance on an *m*-node system (where a node is a workstation with or without RC units):

$$S_{HPRC}(m) = \frac{R(1)}{R_{HPRC}(m)} \quad \text{where } R \text{ represents the execution time} \qquad \textbf{(2.5)}$$

We will discuss the details of the load balance implications in a later section. Similar to the case for RC efficiency, we define *HPRC efficiency* where n is the total number of hardware tasks and k is the total number of software tasks across all nodes of the system:

$$Eff_{HPRC}(m) = \frac{S_{HPRC}(m)}{n+k} = \frac{R(1)}{(n+k)\cdot R_{HPRC}(m)} \qquad \textbf{(2.6)}$$

Other metrics such as power, cost, physical size, etc. as discussed in the RC section are important especially in embedded systems. For these metrics, we can form cost functions from the performance analysis equations and use them to maximize or minimize the metric as appropriate.

2.3 Performance Evaluation, Analysis and Modeling

2.3.1 Overview

Significant research has been conducted in performance analysis and performance modeling for HPC. Performance models will be necessary tools for understanding HPRC issues and potentially determining the best mapping of applications to HPRC resources. Techniques for performance evaluation are classified into three categories: *measurement*, *simulation*, and *analytical modeling* [43]. Each of the three techniques has several types and selection of the most suitable alternative is often difficult. One must consider the desired result or application of the model, the accuracy needed, the development stage of the computer system, and the model development cost.

Measurement techniques are based on direct measurements of the system under study; therefore, the real system must be available. Simulation and analytical modeling are both based on models constructed by abstracting essential features of the system based on assumptions. Some common abstract modeling approaches used in HPC research are queueing models, Petri nets, Markov models, general analytic models, and again simulations. In this section we will take a brief look at each of these techniques.

2.3.2 Measurement

The technique of measurement when used for performance evaluation is based on the direct measurements of the system under study using a combination of software and hardware monitors.

Measurement techniques are often used in *performance tuning* whereby the information gathered can be used later to improve performance [43]. For example, frequently used segments of the software can be found and optimized, thereby improving the performance of the whole program. Similarly, the resource utilizations of the system can also be obtained and performance bottlenecks can be identified. Other common uses of measurements is to gather data for modeling a system, characterizing workloads, or validating a system model [43]. Measurement techniques are also commonly used in other modeling techniques to calibrate and validate the models. Since there are unavoidable disturbances to the system during measurements such as loading and overhead caused by the probes or monitors, the data collected must be analyzed with statistical techniques in order to draw meaningful conclusions. Care must also be taken in selecting the output parameters to be measured, how they will be measured, and how and what inputs will be controlled [50]. Other issues that must be considered are the costs involved to instrument the system and gather data, the practicality of measuring for the desired parameters, and data perturbations from probes and monitoring devices.

2.3.3 Simulation Models

Simulation involves constructing a model for the system's behavior and driving it with an abstracted workload or trace data. It can be used to predict the performance of a system or to validate other analytical models. The major advantages of simulation are generality and flexibility [50]. Like measurement, simulation modeling requires careful attention to the experiment design, data gathering and data analysis. The level of detail of the model should be carefully considered since it is often not necessary to duplicate the complete detailed behavior of the system. Again like measurement, the amount of data can be enormous and statistical methods must be used to analyze the results.

Simulation models usually fall into one of the following categories: emulation, Monte Carlo simulation, trace-driven simulation, execution-driven simulation, and discrete-event simulation [43]. *Emulation* is simulation using hardware or firmware. *Monte Carlo simulation* is a static simulation where the simulated systems do not change their characteristics with time (computer systems are dynamic systems and do not belong to this category [43]). A *trace-driven simulation* system consists of an event or trace generator and a simulator. The event generator produces a trace of execution events for the simulator. The simulator simulates the target architecture based on the trace events to estimate the performance (this

technique is widely used to study memory). *Trace-driven simulations* are generally obtained from real program executions. They typically have the least variance but the results offer little opportunity to draw general conclusions about the system's performance on applications other than the sample program. It is also difficult to simulate parallel architectures with trace-driven simulations because the program execution path often differs depending on the architecture or even from one execution to the next [43]. *Discrete-event simulation* is used to simulate systems represented by discrete event models. It is well suited for logic simulation as well as studying queueing systems and Petri nets.

One of the major drawbacks of simulation is the performance. On sequential machines, large simulations can take enormous amounts of time. Another drawback of simulation is the inability to draw general conclusions about the system's performance from a single simulation. To understand sensitivity to parameters, multiple simulations are required.

2.3.4 Analytic Models

Analytic modeling involves constructing a mathematical model at the desired level of detail of the system and solving it [50]. The main difficulty with analytic models is obtaining a model with sufficient detail that is tractable. However analytical models have some major advantages over the previous two techniques: (a) valuable insight into the workings of the system even if the model is too difficult to solve; (b) remarkably accurate results even for simple analytic models, and (c) better predictive value from results than those obtained from measurement and simulation.

It should also be noted that simpler models have other benefits: (a) typically they are more robust, i.e., small errors in parameter values have less significant affects; (b) they are usually easier to understand, calibrate, and modify; and (c) the input parameters are often not known reliably (e.g., one input may be the average number of users). The uncertainty in a simpler model is not caused by the randomness [50].

2.3.4.1 Queueing Models

One of the classic analytical modeling techniques is queueing models [16,52]. Queueing models are attractive because they often provide the simplest mathematical representations that either have closed form solutions or very good approximation techniques such as Mean Value Analysis (MVA) [50]. However, for many systems (such as those with internal concurrency), the model is too complex and closed form solutions are not obtainable requiring the use of simulation. In these cases, queueing models fit better

in the category of simulation models rather than analytical models. In either event, the initial analysis begins from an analytical approach therefore we include them here with analytical models.

Queueing network models can be viewed as a small subset of the techniques of queueing theory and they can be solved analytically, numerically, or by simulation. Queueing systems are used to model processes in which customers arrive, wait for service, are serviced, and then depart. Characterization of systems thus requires specification of [43]: inter-arrival time probability density function (*A*), service time probability density function (*S*), number of servers (*m*), system capacity (buffers, queues) (*B*), population size (*K*), and queueing discipline (*SD*).

A common notation used in the queueing literature to specify these six parameters is – *A/S/m/B/K/SD*. In general, if there is no buffer space or population size limitations and the queueing discipline is FCFS, the notation is shortened to *A/S/m*. The most widely used distributions for *A* and *S* are: (1) *M* – Markov, exponential distribution, (2) *D* – Deterministic, all customers have the same value, and (3) *G* – General, arbitrary distribution [43].

Queueing systems are described by their states and in their paper on Analytic Queueing Network Models [83], Thomasian and Bay present a recursive algorithm to compute state probabilities for directed acyclic graphs or DAGs. The algorithm uses a hierarchical model based on a Markov chain at the higher level to compute state probabilities and an analytic solution at the lower level to compute transition rates among the states of the Markov chain.

To model parallel systems, the modeling technique must have the ability to describe synchronization between parallel processes. Two forms of synchronization can be distinguished: *mutual exclusion* and *condition synchronization* [43]. Product form queueing networks can describe mutual exclusion but not condition synchronization. Thus, other techniques such as the use of task graph models are used together with queueing models to express condition synchronization.

Many researchers have explored the use of queuing networks [16], petri net models [72], and markov models [16] in performance analysis of HPC systems. Fujimoto's research focuses on discrete simulation, synchronization issues and performance analysis with queuing theory and petri net models [31,32]. Mohapatra et.al. [58,59] use queuing models to study the performance of cluster-based multiprocessors and finite-buffered multistage interconnection networks.

2.4 Development Environment

2.4.1 HPC Development Environment

Message passing is a programming paradigm commonly used on MIMD parallel computers. Several public-domain systems have demonstrated that a message-passing system can be efficient and portable. Both Parallel Virtual Machine (PVM) and Message Passing Interface (MPI) provide a portable software API that supports message passing and are two of the more common public-domain standards.

From a historical view, PVM was developed by a research group to work on networks of workstations, *parallel virtual machine* [33]. In contrast, MPI was developed by a forum of users, developers, and manufacturers as a standard message passing specification [79]. Just the process by which they came about implies some of the differences between the two APIs. The roots of PVM being in the research committee influenced incremental development, backward compatibility, and fault tolerance. The goal of the PVM developers also leaned more toward portability and interoperability sacrificing some performance. The MPI developer's forum, having members from the parallel computer vendors and manufacturers, obviously had an eye for performance, scalability, and adaptability.

MPI is expected to be faster within a large multiprocessor. It has a large set of point to point and collective process communication functions and the ability to specify communication topologies. Unavailable in PVM, this affords the user the capability to exploit architectural advantages that map to the communication needs of the application. The communication set for MPI uses native communication functions to provide the best possible performance. PVM chose a more flexible approach which allows communication between portable applications compiled and executed on different architectures.

PVM is traditionally better for applications running on heterogeneous networks. Both MPI and PVM are portable but only PVM is truly interoperable between different hosts. Both PVM and MPI applications can be compiled and executed on different architectures without modification. However, the resulting PVM executables can also communication with each other across these architectural boundaries. For local or identical hosts, MPI and PVM both use native communication functions. For heterogeneous architectures, PVM uses standard network communication functions. PVM is also language interoperable meaning programs written in C and FORTRAN can communicate and interoperate. Interoperability costs a small amount of overhead resulting in slightly lower performance for PVM.

PVM is also capable of fault tolerant applications that can survive host or task failures. This capability is somewhat a result of PVM's dynamic process nature. The MPI standard requires no "race conditions" resulting in a more static configuration and less capability of recovering from such faults.

2.4.2 RC Development Environment

Research in the architecture, configuration, and use of RC systems is ongoing. To date, the DARPA Adaptive Computing Systems (ACS) program [6] has supported most of the efforts. RC research has primarily focused on single computing nodes with one or more RC elements. Some of the major challenges involve FPGA configuration latencies, hardware/software codesign, and the lack of suitable mapping tools. Often, the design time needed to map an application onto the RC system, or the time consumed during reconfigurations, or both outweigh any performance advantages that can be achieved by executing on the RC system. The development tools available for RC systems can be divided into two main categories: *Language-based* and *Visual-based*.

Some of the language-based tools include compiler environments for C-based applications, VHDL and Verilog tools, JHDL and JBits. Design capture is achieved through a textual description of the algorithm using a high level language or hardware description language. This type of design description enables the use of libraries containing macros or function calls. The designer must be cognizant of potential hardware conflicts, partitioning, and scheduling. Several groups have developed compiler environments for RC systems. The Nimble compiler at Berkeley [57] is a C-based environment. DEFACTO [17] uses the Stanford SUIF compiler system [38,80] and similar work at Virginia Tech [48,49] uses the BYU JHDL design environment [8,15]. Many of these compiler environments allow the designer to map high level programs into VHDL for implementation on various RC boards. JHDL (Just another HDL) allows designers to express dynamic circuit designs with an object-oriented language such as Java. JHDL supports dual hardware/software execution and runtime configuration. The circuit description serves both circuit simulation and runtime support without any redefinition of the circuit description. JBits is another Java based tool for RC runtime full and partial configuration [36,88]. JBits Java classes provide an Application Programming Interface (API) for Xilinx FPGA bitstreams. JBits is essentially a manual design tool and requires knowledge of the architecture by the designer. Modification or reconfiguration of the

circuit at the JBits level eliminates any possibility of using any analysis tools available to the circuit designer further up the tool chain, specifically the availability of timing analysis is absent in JBits.

The visual-based tools use a visual based design environment such as block diagrams and flow graphs for design capture. Many tools support development of libraries containing macros and hardware functions for easy reuse. Depending on the support provided by the infrastructure, the designer must also be aware of partitioning, scheduling, and hardware conflicts. The CHAMPION research at the University of Tennessee uses the visual programming language Khoros to assist the designer in RC hardware programming [56,62,63,68,76]. Currently CHAMPION does not support any automated migration of functionality between the CPU and RC element(s) limiting the ability to perform load balancing between nodes in a distributed system like the proposed HPRC platform. The Ptolemy project [10] uses data flow graphs to synthesize configurations for RC hardware. Other RC efforts have focused on low-level partitioning for RC systems. Researchers at Northwestern have developed libraries of MATLAB matrix and signal processing functions for parallel implementation in FPGAs [46,47]. The System Level Applications of Adaptive Computing (SLAAC) [45] effort is also a part of the DARPA ACS program. Extensions to these efforts will be needed to support partitioning into multiple FPGAs for the HPRC platform.

2.4.3 HPRC Development Environment

Currently there are no tools available which are completely suitable for the HPRC environment. We will need to take tools from both HPC and RC and grow them to form a viable toolset for HPRC. A modeling framework as proposed in this research would be an integral part of this toolset not only allowing the designer to analyze and address performance issues but also provide feedback for partitioning and scheduling tools. During the development of the demo applications described in a later section, we will be applying the modeling results in our manual partitioning, scheduling, and mapping onto the HPRC hardware. These will be simple examples in order to allow manual manipulation (partitioning and scheduling) but will nonetheless demonstrate the use of the modeling results and how it could be applied in an automated CAD tool.

3 Methodology and Initial Measurements

3.1 Introduction

To effectively use the proposed HPRC architecture, we must be able to analyze design tradeoffs and evaluate the performance of applications as they are mapped onto the architecture. Performance models are commonly used tools for analyzing and exploiting available computational resources in HPC environments. Some commonly used modeling techniques in the analysis of computing systems are analytic models, simulations, and measurement. The best suitable modeling approach depends on the required accuracy, level of complexity, and analysis goal of the model. We can employ one or more of these modeling approaches to better understand the tradeoffs in mapping applications to HPRC resources as well as the most effective ways of doing so.

To develop a representative modeling framework for HPRC we will begin by investigating and characterizing the RC architecture and expanding this model to multiple nodes representative of an HPRC platform. We will also conduct studies of the HPC environment and isolate node to node performance issues such as processor communications, network setup, and synchronization. In the RC environment, the focus will be on FPGA configuration, processor to FPGA communication, data distribution between FPGA and processor, memory access time, computation time in hardware and software, and other RC application setup costs. Next, we apply this knowledge to the multi node environment building on the earlier load balance work by Peterson [69]. We will develop an analytic modeling methodology for determining the execution time of a synchronous iterative algorithm and the potential speedup. *Synchronous iterative algorithms,* present in a large class of parallel applications, are iterative in nature and each iteration is separated from the previous and subsequent iterations by a synchronization operation. Examples of synchronous iterative algorithms include simulations and many image processing and data classification algorithms.

What follows below is the first iteration of this process using the available hardware and firmware to develop the first pass of the model. In Table 3 Symbols and Definitions used in this section are listed.

Table 3 Symbols and Definitions

Symbol	Definition		Symbol	Definition
M_k	Workstation		n	Number of hardware tasks
A	Application		t_{RC}	Time for a parallel hardware/software task to complete
$W_k(A)$	Workstation relative speed for application A		t_{RC_work}	Total work performed in hardware and software
$V_k(A)$	Speed of workstation M_k in solving application A on a dedicated system		t_{avg_task}	Average completion time of hardware or software task on RC system in a given iteration
m	Number of workstations		β	Load imbalance factor between hardware and software
$R(A,M_k)$	Execution time for computing application A on M_k		σ	Hardware acceleration factor for RC system
$T_{comm}(c)$	Communication time		R_1	Runtime on a single processor
N_C	Total number of messages per processor		r	Number of hardware tasks not requiring new configuration
τ	Message latency		d	Number of hardware tasks not requiring new data set
B_i	Size of message i		t_{config}	Time for FPGA configuration
η	Network bandwidth		t_{data}	Time for data access
γ	Network contention factor		$t_{host\ serial}$	Host serial operations
S_P	Speedup		$t_{node\ serial}$	RC node serial operations
R_P	Runtime on parallel system		$t_{host\ overhead}$	Iteration overhead operations for hosts
R_{RC}	Runtime on RC system		$t_{node\ overhead}$	Iteration overhead operations for RC nodes
I	Number of iterations		t_P	Time to complete parallel host software tasks
$t_{overhead}$	Iteration overhead operations		t_{work}	Total work performed in hardware and software on all nodes of a multi-node system
t_{SW}	Time to complete software tasks		β_k	RC node Load imbalance factor at node k in a multi-node system
t_{HW}	Time to complete hardware tasks		α	Load Imbalance factor between host nodes in a multi-node system
$t_{parallel,\ RC}$	Time to complete parallel task on multi node system		σ_k	Hardware acceleration factor for node k in muti-node system

3.2 HPC Analysis

Our analysis will begin with the HPC environment and the communication and workstations issues related to HPC performance. By starting with the HPC environment, we can isolate the node-to-node communication and workstation performance to better understand the performance related issues. The remainder of this section is a discussion of some of these HPC performance issues.

3.2.1 Workstation Relative Speed

In a network of heterogeneous workstations, each workstation may have different capabilities in terms of CPU speed, memory and I/O. The relative computing power among the set of workstations will vary depending on the application and the problem size of the application. In order to quantify the idea of a relative computing power for a given workstation M_k, a relative speed $W_k(A)$ with respect to application A is defined as [89]:

$$W_k(A) = \frac{V_k(A)}{\max_{1 \le j \le m}\{V_j(A)\}} \quad ,1 \le k \le m \qquad \textbf{(3.1)}$$

Where $V_k(A)$ is the speed of workstation M_k in solving application A on a dedicated system. As shown in equation (3.1), the relative speed of a workstation refers to its computing speed relative to the fastest workstation in the system and its value is less than or equal to 1.

Since the relative speed as defined is a ratio, it is often easier to represent it using measured execution times. If $R(A, M_k)$ represents the execution time for computing application A on M_k, the relative speed can be calculated as follows [89];

$$W_k(A) = \frac{\min_{1 \le j \le m}\{R(A, M_j)\}}{R(A, M_k)} \quad ,1 \le k \le m \qquad \textbf{(3.2)}$$

Thus, if an operation executes in t cycles on the fastest workstation, the same operation should execute in t/W_k cycles on workstation M_k. W_k implicitly accounts for (to the first order) all factors affecting a workstation's relative speed or performance such as processor speed, instruction set, memory size, and memory hierarchy. When conducting experiments and measurements for determining W_k, they should be constructed in a way such as to minimize non-architectural affects like background load.

3.2.2 Communication Delay

Communication delay between workstations in a network is affected by the network topology, communication volume, and communication patterns. Other research on network performance models report that a simple communication model that accounts for message startup time and network bandwidth is adequate [22]. For the total number of messages per processor, N_C, the message latency τ, network bandwidth η, and size of message B_i, the communication time can be modeled as [22]:

$$T_{comm}(c) = \sum_{i=1}^{N_C} (\tau + \frac{B_i}{\eta}) \quad \textbf{(3.3)}$$

Both τ and η can be approximated from measured values. It should be noted that in practice, η may not be a constant. The model represented in equation (3.3) is non-preemptive (messages are serviced one-by-one) and useful for modeling clusters connected with contention free networks. With shared-medium networks such as Ethernet, contention can significantly affect throughput. To model PVM communications over these types of networks, a contention factor, γ, is added to equation (3.3) [22]:

$$T_{comm}(c) = \sum_{i=1}^{N_C} (\tau + \frac{\gamma \cdot B_i}{\eta}) \quad \textbf{(3.4)}$$

According to [22], a contention factor of $\gamma = m$, where m is the number of workstations, is a good approximation of an Ethernet connection assuming all workstations are communicating simultaneously. However, this assumption only holds for a limited number of workstations. At some point, as the number of workstations is increased, the throughput of the network begins to drop off and there will no longer be a linear relationship between m and γ. More analysis and study will be required on this model in the remaining work.

3.2.3 Speedup

Speedup is a useful metric for evaluating parallel computing performance in a heterogeneous system [91]. Speedup in a heterogeneous system is defined as the ratio of sequential computing time of the application A on the fastest workstation in the system to the parallel computing time:

$$S_P = \frac{\min_{1 \le k \le m} \{R(A, M_k)\}}{R_P} \quad \textbf{(3.5)}$$

Where $R(A,M_k)$ is the sequential execution time for application A on workstation M_k (k = 1, 2, …, m) and R_P is the parallel algorithm execution time.

3.2.4 Scheduling and Load Balancing

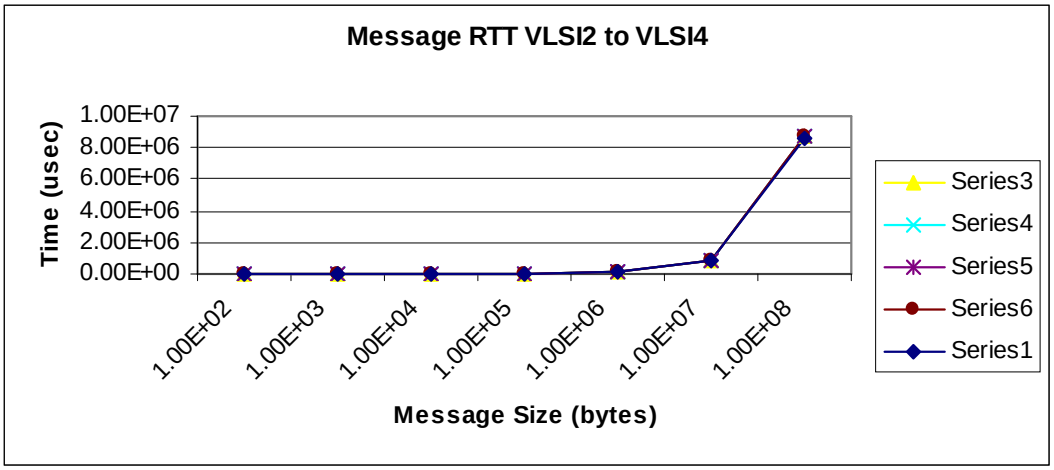
The performance of a parallel algorithm depends on the scheduling of tasks (static or dynamic) and the load balance between nodes (tasks and data). The *scheduling problem* has been well studied in the literature [19,29]. Scheduling techniques can be classified based on the availability of program task

information as deterministic (all task information and relationships are known a priori) and non-deterministic (the task graph topology representing the program, its execution and communication are unknown) [29]. Scheduling decisions can be made at compile time (*static scheduling*) or at run time (*dynamic scheduling*). In static scheduling, information regarding the task graph of a program must be estimated prior to execution and each set of tasks runs to completion on the set of processors to which it was initially allocated. Dynamic scheduling algorithms adjust the spatial and temporal allocation of processing tasks to meet the needs of the application, other users, and at the same time attempt to optimize the overall use of system resources. Dynamic scheduling is difficult because the behavior of interacting processes is hard to characterize analytically.

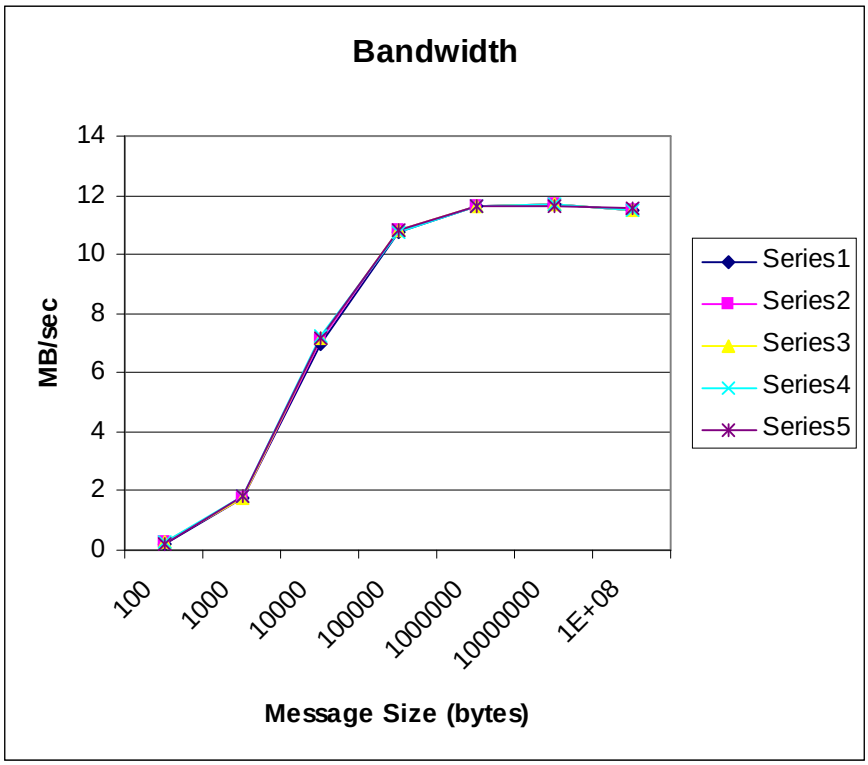
The proposed model can be used to determine execution time of a given mapping and help identify potential bottlenecks to performance such as slower processors, FPGA size limitations, too many reconfigurations, etc. From this information decisions can be made regarding scheduling of tasks and load balancing. For the scope of this proposal, all scheduling and load balancing will be static and manual. However, the process will demonstrate the usefulness of the proposed model for application in a CAD tool for automated scheduling and load balancing.

3.2.5 Initial Network Communication Measurements

The SInRG SUN GSC (grid service cluster), a network of workstations available for development tests, was measured for processor to processor communication delay. Using PVM, a simple message round trip time was measures for various message sizes. Measurements were taken overnight when the network load was light between seven of the workstations described in Table 2 (vlsi2, vlsi4, vlsi5, vlsi6, vlsi7, vlsi8 and vlsi9) with workstation vlsi2 as the master.



(a)



(b)

Figure 3.1 (a) Message Round Trip Time and (b) Network Bandwidth

3.3 RC Node Analysis

Our performance model analysis will begin with a single RC node running a synchronous iterative algorithm. These restrictions will allow us to investigate the interaction between the processor and RC unit.

First, we will assume we have a segment of an application that has I iterations and all iterations are roughly the same as shown in Figure 3.2. The RC unit has at least one FPGA (there may be other reconfigurable devices which provide control functions) and tasks can potentially execute in parallel in hardware (RC unit(s)) and in software on the processor. We should point out that the hardware and software task trees can be arbitrarily complex. Figure 3.2 shows a simple hardware/software tree structure. Additionally, hardware can be reused within a given iteration if the number of tasks or size of the task exceeds the number of available FPGAs.

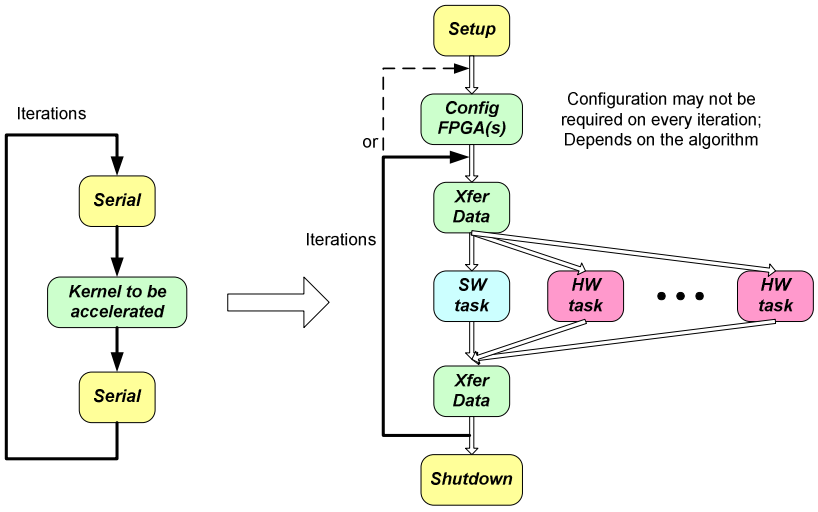


Figure 3.2 Synchronous Iterative Algorithm

For a synchronous iterative algorithm, the time to complete a given iteration is equal to the time for the last task to complete either in hardware or software. For each iteration of the algorithm, there are some operations which are not part of the kernel to be accelerated and are denoted $t_{serial,i}$. Other overhead processes that must occur such as configurations and exchange of data are denoted $t_{overhead,i}$. The time to complete the kernel tasks executing in software and hardware are $t_{SW,i}$ and $t_{HW,i}$. respectively. For I iterations of the algorithm where n is the number of hardware tasks, the runtime, R_{RC}, can be represented as [69]:

$$R_{RC} = \sum_{i=1}^{I} [t_{serial,i} + \max(t_{SW,i}, \max_{1 \le j \le n}[t_{HW,i,j}]) + t_{overhead,i}] \qquad \textbf{(3.6)}$$

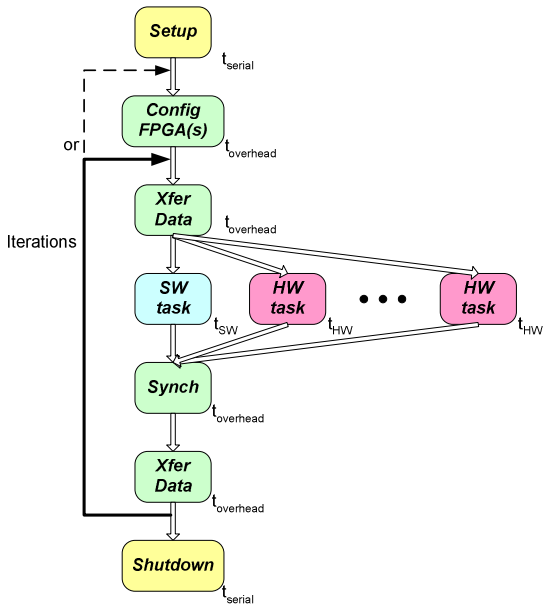


Figure 3.3 Flow of Synchronous Iterative Algorithm for RC Node

To make the math analysis cleaner, we will make a couple of reasonable assumptions. First, we will assume that each iteration requires roughly the same amount of computation allowing us to remove the reference to individual iterations in equation (3.6). Second, we will model each term as a random variable and use their expected values. Thus we define t_{serial} as the expected value of $t_{serial,i}$ and $t_{overhead}$ as the expected value of $t_{overhead,i}$. The mean time required for the completion of the parallel hardware/software tasks is represented by the expected value of the maximum (t_{SW}, t_{HW}). Finally, we will assume that each of the random variables are independent and identically distributed (iid). We can then write the run time as:

$$\begin{aligned} R_{RC} &= \sum_{i=1}^{I} (E[t_{serial,i}] + E[\max(t_{SW,i}, \max_{1 \le j \le n} t_{HW,i,j})] + E[t_{overhead,i}]) \\ &= I(t_{serial} + E[\max(t_{SW}, \max_{1 \le j \le n} t_{HW,j})] + t_{overhead}) \end{aligned} \quad \textbf{(3.7)}$$

If the iterations are not iid, we must retain the first form of equation (3.7) and the math analysis is more difficult.

We can rewrite the RC system hardware/software tasks in terms of the total work. We will assume that all hardware tasks are the same and that the time to complete a software task is the same as that for a hardware task. With these simplifying assumptions, we will define $E[t_{avg\_task}]$ as the expected average task completion time (including hardware and software tasks) within an iteration for the RC system. Therefore the total work completed on the RC system, measured in runtime is given by:

$$t_{RC\_work} = E[t_{SW} + \sum_{n} t_{HW}] = (n+1) \cdot E[t_{avg\_task}] \quad \textbf{(3.8)}$$

The division of tasks between hardware and software creates an *RC load imbalance* [69] which we will represent as β. Considering the affect of the imbalance on the task completion time, the expected value of the maximum task completion time can be expressed as the average task time within an iteration multiplied by the RC load imbalance factor:

$$E[\max(t_{SW}, \max_{1 \le j \le n}(t_{HW,j})] = \beta \cdot E[t_{RC\_system}] \quad \textbf{(3.9)}$$

Combining equations (3.8) and (3.9) we can rewrite the maximum task completion time as,

$$E[\max(t_{SW}, \max_{1 \le j \le n}(t_{HW,j})] = \frac{\beta \cdot t_{RC\_work}}{n+1} \quad \textbf{(3.10)}$$

Note that if the load is perfectly balanced β is the ideal value of 1 and the total work is divided equally among the hardware and software tasks. If the tasks are performed serially (no concurrent hardware/software operation), β is the worst case value of (n+1), where n is the number of hardware tasks. Thus as the load imbalance becomes worse, β increases ranging from a value of 1 to (n+1).

Noting that the total work measured in time for a software-only solution is not equivalent to the total work measured in time on an RC system solution, we introduce an acceleration factor σ to account for the difference. Since the goal of RC systems is to speed up an application, only tasks that would be faster in hardware are implemented in hardware. For example, an FFT in software may take longer to execute than an equivalent implementation in the hardware. Given the total work that will be completed in hardware and software on an RC system, we can represent the software only run time on a single processor as:

$$R_1 = I \cdot (t_{serial} + t_{SW} + \sigma \cdot \sum_{n} t_{HW}) \quad \textbf{(3.11)}$$

The overhead for an RC system consists of the FPGA configuration time and data transfer time. The configuration time for the FPGA(s) is (n-r) x t_{config}, where r is the number of hardware tasks not requiring a new configuration. The time to transfer data to and from the RC unit is (n-d) x t_{data}, where d is the number of hardware tasks not requiring a new data set.

The speedup, S_{RC}, is defined as the ratio of the run time on a single processor to the run time on the RC node:

$$S_{RC} = \frac{R_1}{R_{RC}} = \frac{t_{serial} + t_{SW} + \sigma \cdot \sum_n t_{HW}}{t_{serial} + \frac{\beta \cdot t_{RC\_work}}{n+1} + [(n-d) \cdot t_{data}] + [(n-r) \cdot t_{config}]} \quad \textbf{(3.12)}$$

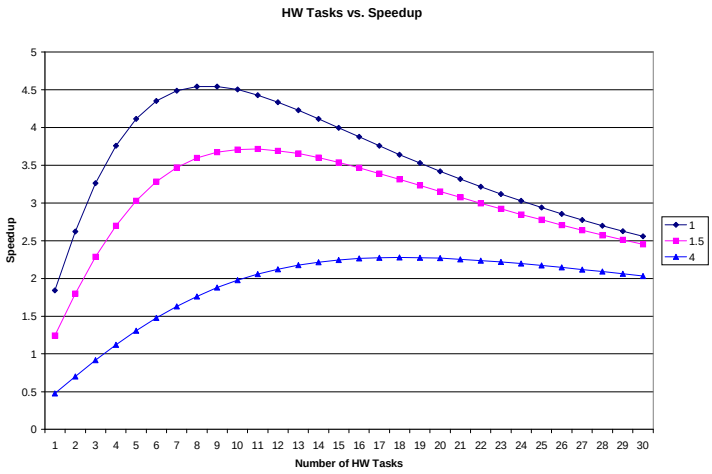
Using equation (3.12) we can investigate the impact of load imbalance and various overhead issues on the algorithm performance by varying β, t_{config}, t_{data}, t_{RC_work}, r, d, and n.

We have conducted experiments with the Wildforce and Firebird RC boards to determine the parameters t_{config} and t_{data}. Typical values are used for β, and t_{RC_work}. Figure 3.4 shows plots of equation (3.12) with d and r both equal to zero meaning that all hardware tasks require new data and new configurations. The implementation efficiency σ is fixed at 95%. Figure (a) plots speedup as the number of hardware tasks increases for three values of β and a fixed total workload of 10sec, serial overhead of 0.2usec and FPGA configuration time of 0.01usec per unit. Figure (b) plots speedup as the configuration time is varied for several β values and hardware tasks. The serial overhead and total workload values are held constant at 0.2usec and 10sec respectively. Finally, figure (c) plots speedup as the total workload increases for several β values and hardware tasks and shows that as the load imbalance increases, the obtainable speedup is reduced. The serial and configuration overhead are held constant at 0.2usec and 0.01usec respectively.

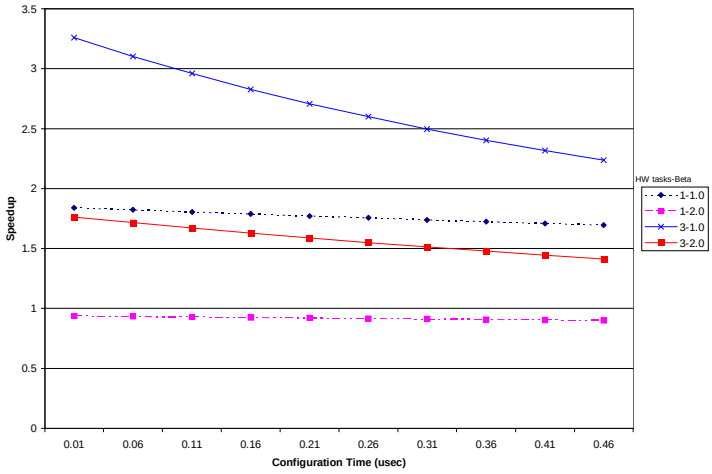
Given this model for execution time in an RC system we can begin to discuss the determination of power. To determine a rough estimate of the system power, we must determine the power used by the processor and by the RC unit(s). Assuming our processor does not operate in sleep mode (i.e. the power consumption is constant with time) we can determine the total power consumption by the processor based on the execution time from the model and power information from the processor's literature. Determining the total power consumption for the RC unit(s) is somewhat more complicated. Power consumption in FPGAs is dependent on the speed and size of the design. The FPGA literature should provide some guidelines for determining power consumption based on the number of gates or logic blocks used in the design and the speed of the design. Once these factors are known, the total power for the RC unit(s) can be

calculated using the execution time from the model. Finally, the total power consumption for the RC system is simply the sum of the total power consumed by the processor and the RC unit(s).

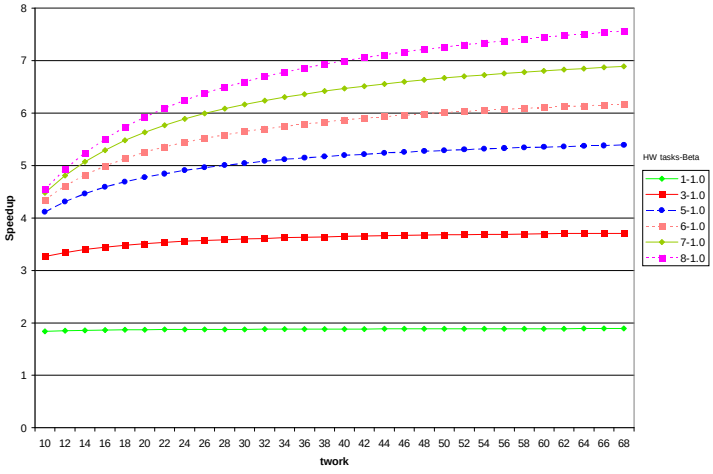
Developing a cost metric should be straightforward based on the processors, FPGAs, memory, and interconnect under consideration. A cost function can be developed relating execution time or speedup as determined from the proposed model to the cost of the system. Similarly a power metric cost function could be developed relating execution time determined from the proposed model to the total power used by the system.



(a) Vary number of RC units



(b) Vary RC configuration time



(c) Vary amount of total work

Figure 3.4 Speedup Curves: a) Vary number of RC units, curves are different β values; b) Vary configuration time; and c) Vary total work

3.4 RC Model Validation

In this section we will present the initial validation results for the RC model using the Wildforce board and some initial parameter measurements for the Firebird board.

3.4.1 Wildforce Measurements

We have made basic model parameter measurements using a sample application for the Wildforce board as a benchmark. To validate the execution time prediction of the model, the benchmark measurements are used with the developed model to predict the runtime for three of the CHAMPION demos. The details of the demo applications will be discussed in a later section. From the benchmark, we have determined the model parameters as shown in Table 4. The configuration values for CPE0 and PE1 are significantly different because they are two different Xilinx devices and we therefore account for them separately in the model calculations. For this application the only part of the algorithm considered as serial is the board configuration and setup. There is only one iteration therefore I is one. The remaining unknowns are the values for the total work and the application load imbalance.

Table 4 Model Parameters for Wildforce from Benchmark Application

	CPE0	PE1	HW	Data	Setup (tsw)	Serial
Average in usec	535275	257232.8	1250.52	33282.08	68892.34	40750.46

The total work can be determined from the amount of work completed by the software task plus the amount completed by the hardware task. This can be represented in terms of the number of events multiplied by the execution time per event:

$$t_{RC\_work} = N_e \cdot t_{hw\_exe} + t_{sw\_exe} \qquad \textbf{(3.13)}$$

Where N_e is the total number of events and t_{sw_exe} is t_{hw_exe} are the software and hardware execution times per event respectively. In this particular application, the software and hardware tasks do not overlap so the application load imbalance, β, would be the maximum worst-case value of $(n+1)$ or 2 in this case.

Using the denominator of equation (3.12), we can use the model to predict the runtime of the CHAMPION demo algorithms. The average runtime of fifty trials on the Wildforce RC system is shown in Table 5 and Figure 3.5 along with the model predictions. The number following the algorithm name indicates the input data size. The value 128 indicates an input data set of 128x128 and similarly the value 256 indicates a 256x256 input data set.

Table 5 Runtime Predictions and Measurements (time in seconds)

	model prediction	average
hipass_128	0.911342	1.313353
hipass_256	1.773769	1.907098
START_128	5.166674	4.597426
START_256	6.175542	6.121883
START20_128	6.702268	8.134971
START20_256	7.741632	8.855299

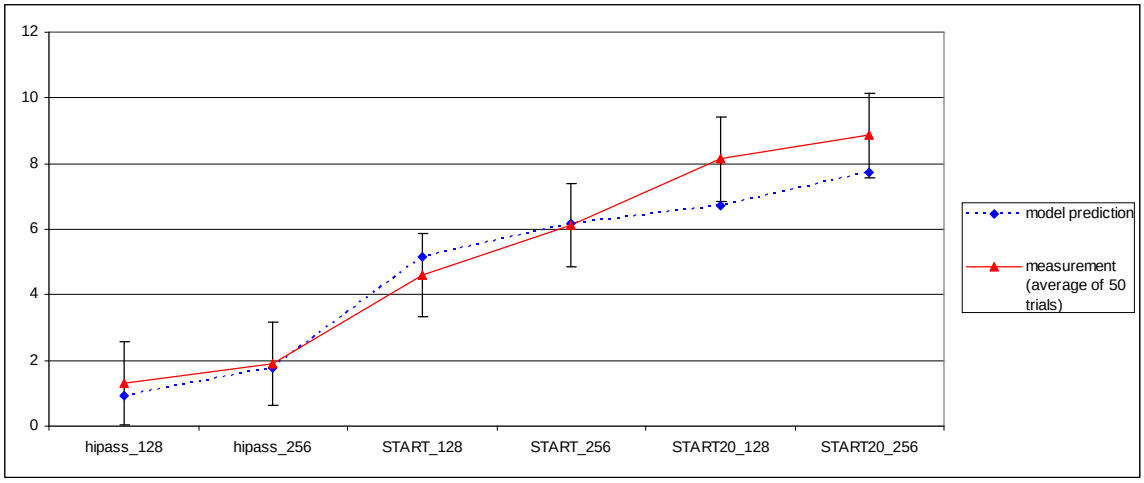


Figure 3.5 Comparison of RC Model Prediction with Measurement Results

3.4.2 Firebird Measurements

Again, using a sample application available from Annapolis Microsystems for the Firebird board some initial measurements for the model parameters were obtained. The application configures the FPGA and writes and reads to all the memory locations on the board using 64-bit data transfers. From measurements on the Firebird board, we have determined the model parameters as shown in Table 6. Since this example does not perform any computation operations, there are no measurements for t_{hw_exe} at this time. More work is needed to complete this benchmark adding tests for synchronization, load balance, and hardware task time.

Table 6 Model Parameters for Firebird from Benchmark Application

	PE0	HW	Data	Setup (tsw)	Serial
Average in usec	38160.00	unknown	499700.00	26.18	19.2

3.5 HPRC Multi-Node Analysis

Now that we have a model for a single RC node and an understanding of the basic HPC issues involved in a set of distributed nodes, we will turn our focus to expanding the model for multi-node analysis. An example of the HPRC architecture is shown in Figure 3.6. For now, we will not consider the optional configurable interconnection network between the RC units in our modeling analysis.

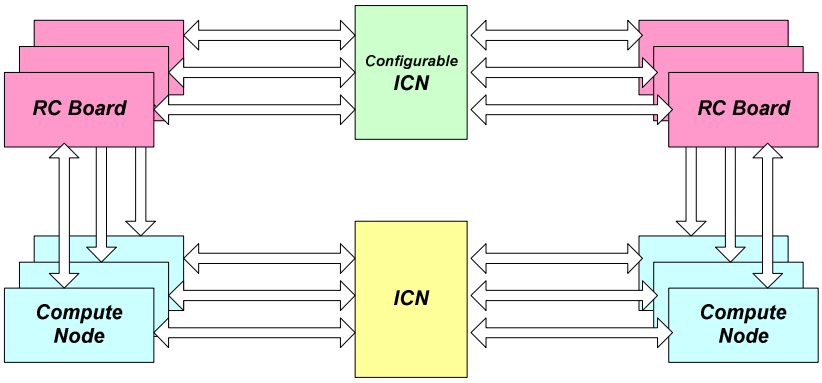


Figure 3.6 HPRC Architecture

We will again start our performance model analysis using a synchronous iterative algorithm this time running on a platform consisting of multiple RC nodes. The restriction of synchronous iterative algorithms will allow us to investigate the communication and synchronization that occurs among nodes between iterations. We will begin our model by restricting our network to a dedicated homogeneous system where there is no background load (i.e. all nodes are identical, same processor and same RC system configuration).

Again, we will assume we have a segment of an application having I iterations that will execute on parallel nodes with hardware acceleration. Additionally, we will assume that all iterations are roughly the same as is shown in Figure 3.7. Software tasks can be distributed across computing nodes in parallel and hardware tasks are distributed to the RC unit(s) at each individual node.

For a synchronous iterative algorithm, the time to complete an iteration is equal to the time for the last task to complete on the slowest node whether it be hardware or software. For each iteration of the algorithm, there are some calculations which cannot be executed in parallel or accelerated in hardware and are denoted $t_{master\_serial,i}$. There are other serial operations required by the RC hardware and they are denoted $t_{node\_serial,i}$. Other overhead processes that must occur such as synchronization and exchange of data are denoted $t_{master\_overhead,i}$ and $t_{node\_overhead,i}$ for the host and RC systems respectively. The time to complete the

tasks executing in parallel on the processor and RC unit are $t_{SW,i,k}$ and $t_{HW,i,j,k}$. respectively. For I iterations of the algorithm where n is the number of hardware tasks at node k and m is the number of processing nodes, the runtime, R_P, can be represented as [69]:

$$R_P = \sum_{i=1}^{I} [t_{master\_serial,i} + t_{node\_serial,i} + \max_{1 \le k \le m}(t_{SW,i,k}, \max_{1 \le j \le n}[t_{HW,i,k,j}]) + t_{master\_overhead,i} + t_{node\_overhead,i}]$$

$$R_P = \sum_{i=1}^{I} [t_{master\_serial,i} + t_{node\_serial,i} + \max_{1 \le k \le m}(t_{RC,i,k}) + t_{master\_overhead,i} + t_{node\_overhead,i}] \quad \textbf{(3.14)}$$

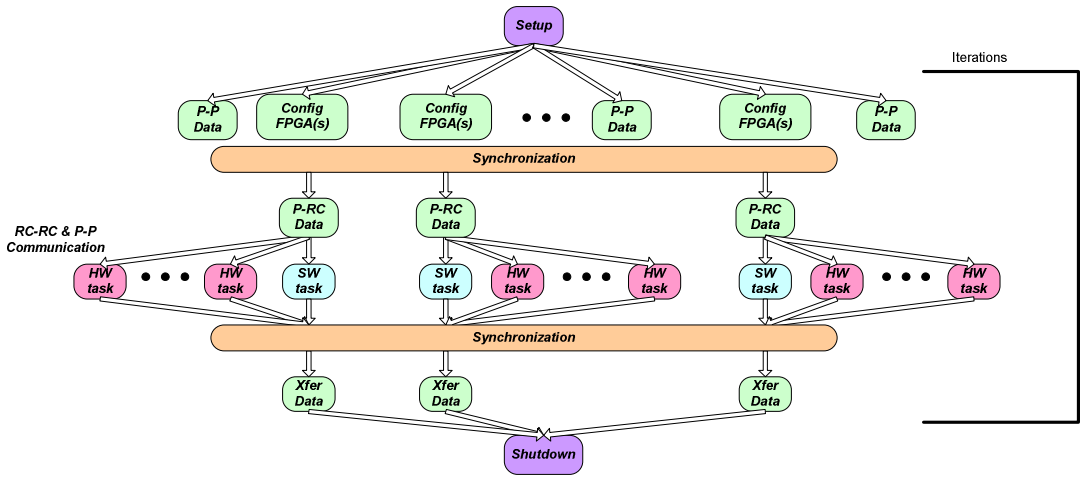


Figure 3.7 Flow of Synchronous Iterative Algorithm for Multi Node

Again, to make the math analysis cleaner, we will make a couple of reasonable assumptions. First, we will assume that each iteration requires roughly the same amount of computation thus we can remove the reference to individual iterations in equation **(3.14)**. Second, we will also assume that each node has the same hardware tasks and configuration making the configuration overhead for each node the same. Third, we will model each term as a random variable and use their expected values. Thus we define t_{master_serial} and t_{node_serial} as the expected value of $t_{master\_serial,i}$ and $t_{node\_serial,i}$. Similarly, we define $t_{master\_overhead}$ and $t_{node\_overhead}$ as the expected value of $t_{master\_overhead,i}$ and $t_{node\_overhead,i}$. The mean time required for the completion of the RC hardware/software tasks is represented by the expected value of the maximum $t_{RC,k}$ ($1 \le k \le m$). Finally, assuming that the random variables are each independent and identically distributed (iid), the run time can then be expressed as:

$$R_P = \sum_{i=1}^{I} (E[t_{master\_serial,i}] + E[t_{node\_serial,i}] + E[\max_{1\le k\le m}(t_{RC,i,k})] + E[t_{master\_overhead,i}] + E[t_{node\_overhead,i}]) \quad \textbf{(3.15)}$$

$$R_P = I(t_{master\_serial} + t_{node\_serial} + E[\max_{1\le k\le m}(t_{RC,k})] + t_{master\_overhead} + t_{node\_overhead})$$

As discussed in section 3.3, we can rewrite the total work at node *k* in terms of the average task completion time rather than the maximum (see equation (3.8)). Again assuming the random variables are iid, from equation (3.8) we can express the total work across all *m* nodes in the HPRC platform as:

$$t_{work} = \sum_{k=1}^{m} t_{RC\_work,k} = \sum_{k=1}^{m} (n+1) \cdot E[t_{RC\_system,k}] = m \cdot (n+1) \cdot E[t_{RC\_system}] \quad \textbf{(3.16)}$$

As discussed earlier for RC systems, when tasks are divided a load imbalance exists. We will represent the RC load imbalance at a particular node *k* as β_k. Additionally, there exists an application load imbalance across all of the nodes of the HPRC platform. We will represent this node-to-node load imbalance as α. We will assume the host node application load imbalance α and RC system load imbalance β_k are independent. Additionally, we will assume that the RC system load imbalance at any node is independent of the others. The completion time can then be expressed as the average task completion time within an iteration multiplied by the load imbalance factors (which together have a multiplicative effect):

$$E[\max_{1\le k\le m}(t_{RC,k})] = \alpha \cdot \beta_k \cdot E[t_{RC\_system}] \quad \textbf{(3.17)}$$

Combining equations (3.16) and (3.17) we can rewrite the maximum task completion time as,

$$E[\max_{1\le k\le m}(t_{RC,k})] = \frac{\alpha \cdot \beta_k \cdot t_{work}}{m \cdot (n+1)} \quad \textbf{(3.18)}$$

Note that if the RC load is perfectly balanced or if the algorithm runs entirely in software (*n*=0), β_k is the ideal value of 1. As the RC load imbalance becomes worse, β_k increases. Similarly, if the node-to-node load is perfectly balanced or if the algorithm runs entirely on a single node, *m*=1, α is the ideal value of 1 and the model reduces to the model for a single RC node as in equation (3.10). Finally, as the node to node imbalance becomes worse, α increases.

Noting that the total work measured in time for a software-only solution is not equivalent to the total work measured in time on an HPRC platform solution, we introduce an acceleration factor σ_k to account for the difference at each node *k*. Given the total work that will be completed in hardware and software on an HPRC platform, we can represent the software only run time on a single processor as:

$$R_1 = I \cdot [t_{master\_serial} + \sum_{k=1}^{m} (t_{SW} + \sigma_k \cdot \sum_{n} t_{HW})] \quad \textbf{(3.19)}$$

The overhead for the HPRC platform consists of the FPGA configuration and data transfers as discussed in section 3.3 and the synchronization between the nodes. We will model the time required for synchronization as a logarithmic growth with the number of nodes [69]. The speedup, S_P, for the HPRC platform is defined as the ratio of the run time on a single processor to the run time on *m* RC nodes:

$$S_P = \frac{R_1}{R_P} = \frac{t_{master\_serial} + \sum_{k=1}^{m} (t_{SW} + \sigma_k \cdot \sum_{n} t_{HW})}{t_{master\_serial} + t_{node\_serial} + \frac{\alpha \cdot \beta_k \cdot t_{work}}{m \cdot (n+1)} + [(n-d) \cdot t_{data}] + [t_{synch} \cdot \log_2[m]] + [(n-r) \cdot t_{config}]} \quad \textbf{(3.20)}$$

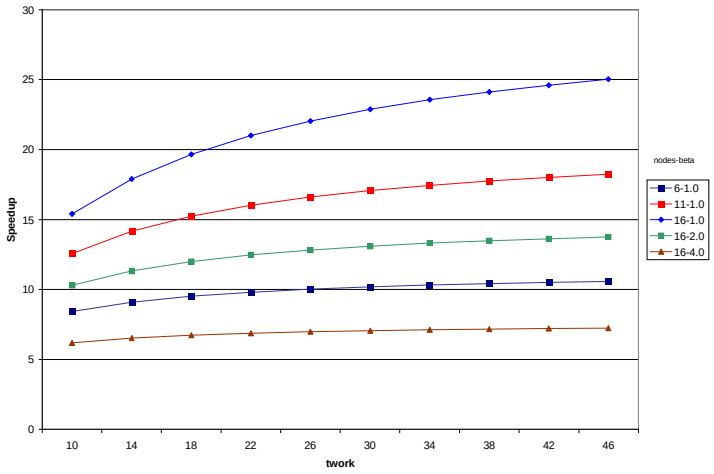
Using equation (3.20) we can investigate the impact of load imbalance and various other overhead issues on the algorithm performance by varying σ_k, α, β_k, t_{synch}, t_{config}, t_{data}, m, and n.

Again we will use the measured values from the Wildforce and Firebird experiments for the RC parameters. We also have measured values from processor to processor communication experiments for the synchronization values. Figure 3.8 (a) and (b) show the speedup curves for various β_k values and number of nodes for an increasing workload (x-axis). The serial and configuration overhead are held constant at 0.2usec and 0.01usec per FPGA respectively. In figure (a) each node has one RC unit per node, and we plot speedup for increasing work for various load imbalance and node configurations. In figure (b) each node has two RC units per node, and again we plot speedup for increasing work for various load imbalance and node configurations. As seen in both figures, the speedup improves with increasing workload for lower β_k values. As β_k increases however, the speedup obtained is impacted by the load imbalance not only between nodes but also between processors and RC hardware. Thus, for a higher β_k value, higher node systems and/or those with more FPGAs per unit are more severely impacted by the load

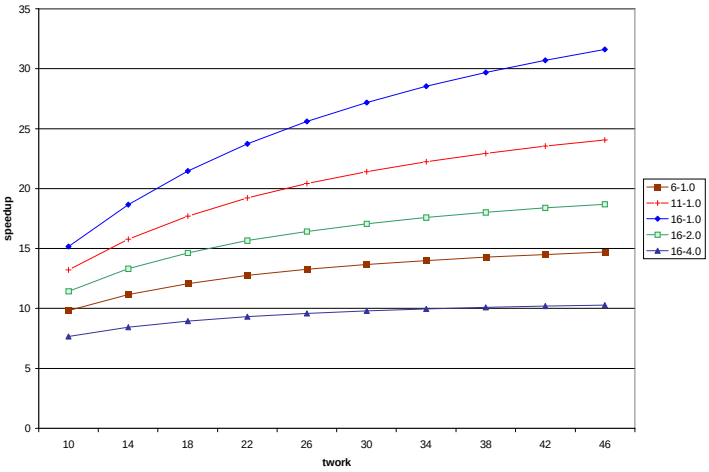
imbalance. In figure (c) we show the speedup for three load imbalance values with the amount of total work fixed as the number of nodes in the system increases. Each node has one hardware tasks.

3.6 Modeling Load Imbalance and Implementation Efficiency

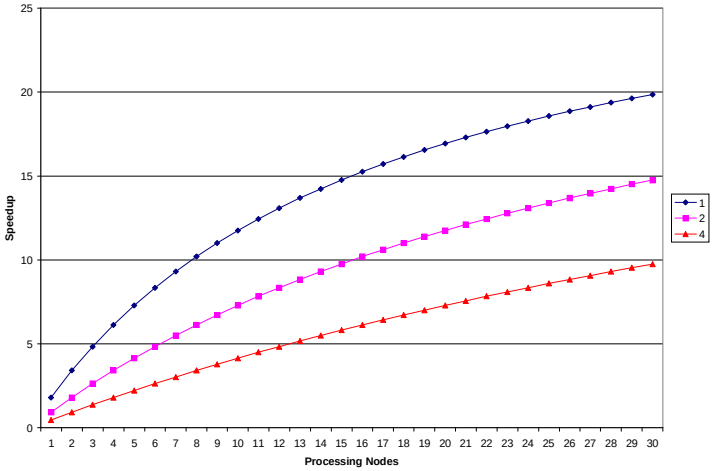
Using the previous work of Peterson as a guide [69], we will attempt to characterize the type of load imbalances in HPRC systems and develop an analytical model for predicting α and β.



(a) One RC unit per node (6, 11, and 16 nodes)



(b) Two RC units per node (6, 11, and 16 nodes)



(c) Fixed work, One RC unit per node

Figure 3.8 Speedup Curves: a) Vary work for one FPGA per RC unit and b) Vary work for two FPGAs per RC unit and c) Increasing number of nodes (one RC unit per node) work fixed

4 Dissertation Status and Future Work

To demonstrate the use of the modeling framework and the validity of the model multiple demonstration applications will be needed. The group of applications should be diverse enough to demonstrate the flexibility and accuracy of the model and how well it predicts system behavior. The goal of validation will be to identify model errors, their causes, and quantify their effects. Potential applications for the HPRC platform as discussed earlier include image processing, DSP applications, as well as various types of simulation such as logic simulation, queuing and petri net simulators, and battlefield simulations. Research into simulation applications for HPRC is underway by others at UT. Some results may be available for analysis during the proposed research but are not viewed as a critical need. Some of the image processing and DSP application that were considered include various demos from the UT CHAMPION research project, the hologram imaging research at Oak Ridge National Laboratory (ORNL), UT unsupervised learning MATLAB k-means algorithm, UT supervised learning MATLAB knn algorithm, and ORNL motion detection algorithm.

4.1 Plan of attack

Hardware/software co-design methods along with the examples and demos from CHAMPION [56,62,63,68], Annapolis Microsystems [2], and SLAAC [45] were reviewed and used for initial measurements during the first phases of the model development. To date, measurements have concentrated on analysis of the RC node and characterizing its features and performance. Once the HPRC development platform is available, these applications will be leveraged as development, measurement, and test applications for the HPRC platform. These demos alone are not complex or computationally intensive enough to utilize the potential processing power of the HPRC platform, however, they are very useful for gathering initial performance characteristics and data for analysis. Using these smaller demos as starting points helps to isolate RC environment issues while keeping the size of the problem manually manageable. There are no automated partitioning and mapping tools available for the HPRC platform so all algorithm development and deployment will be manual.

At first, the HPRC modeling framework will represent a simplified platform of homogeneous nodes with identical processors, each node having a single RC unit and no configurable interconnect between the RC units. Performance model development will begin with a single RC node. After this

single node model is validated and understood, it will be expanded to multiple nodes and the HPC issues such as node to node communication will be addressed and added to the model. Finally, the performance model will be generalized to include multiple RC elements and heterogeneous processing sets. Other interesting performance areas which could potentially be included in the performance model are FPGA reconfiguration latency, distributed versus shared memory, and non-dedicated communication networks but are not viewed as part of the proposed research.

4.2 Proposed Testing Algorithms

Test applications will be needed for measurements to validate the modeling work. Potential HPRC applications include various DSP and image processing algorithms and several types of simulation applications. The desire is to have two or three applications possessing different processing, data, and/or communication characteristics that will verify and fully test the limits/capabilities of the model. Performance results from the model will be compared with empirical measurements from the HPRC implementation and with the existing applications. The goal will be to show that the model developed provides a suitable representation of the HPRC platform for analysis of algorithm performance and HPRC architecture design tradeoffs.

4.2.1 CHAMPION Demo Algorithms

The demo algorithms selected from the CHAMPION research include a simple high pass filter application and an automatic target recognition application (START). The simplicity of the filter algorithm will allow the isolation and study of the processor to FPGA interface in addition to characterization of some of the RC elements such as configuration time and memory access and the START application will serve to further confirm these studies. Beginning with these relatively simple algorithms has several advantages: (1) they have been implemented and studied during the CHAMPION research and (2) the focus can be on the system, model, and measurements instead of debugging the algorithms.

The filter application for study is a high pass filter used to emphasize the fine details in an image corresponding to edges and other sharp details. The high pass filter attenuates or eliminates low frequency components responsible for the slowly varying characteristics in an image netting a sharper image.

The CHAMPION research [67] implemented a 3x3 high pass filter where the value of a pixel in the output image depends on the value of the pixel in the same position in the input image and the eight

pixels surrounding it. For each pixel in the input image, the pixel and its neighbors are multiplied by a mask and the output pixel value is obtained from the absolute value of the sum of all the products:

$$y(i,j) = \left| \sum_{m=i-1}^{i+1} \sum_{n=j-1}^{j=1} x(m,n) \cdot mask(m,n) \right| \qquad \textbf{(4.1)}$$

where *mask(m,n)* is the coefficient mask

A typical high pass filter uses a mask of –1/9 for neighbor pixels but to simplify the hardware implementation a mask of –1/8 will be used (division by eight is simply a 3-bit binary right shift). The resulting mask is shown below:

$$\begin{bmatrix} -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \\ -\frac{1}{8} & 1 & -\frac{1}{8} \\ -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \end{bmatrix} \qquad \textbf{(4.2)}$$

The START (Simple, Two-criterion, Automatic Recognition of Targets) algorithm [56] from CHAMPION will also be used to validate the model. This algorithm was chosen due to its availability as an existing application in addition to being more complex than the simple filter described above. The algorithm is large enough to require reconfiguration of the Wildforce board as well as use of all the available processing elements. Two versions of the algorithm were available: START and START20. START20 is an auto-mapped/partitioned version and requires reconfiguration of each processing element at each of the four stages of the algorithm. START is a manually mapped/partitioned version and reuses some of the processing element configurations across multiple stages as shown in Figure 4.1 resulting in an overall reduced overhead.

The START algorithm applies a statistical algorithm to find regions in Forward-looking InfraRed (FLIR) images where a target may exist and marks the region. Interested readers are referred to Levine's Thesis [56] for the details of the algorithm.

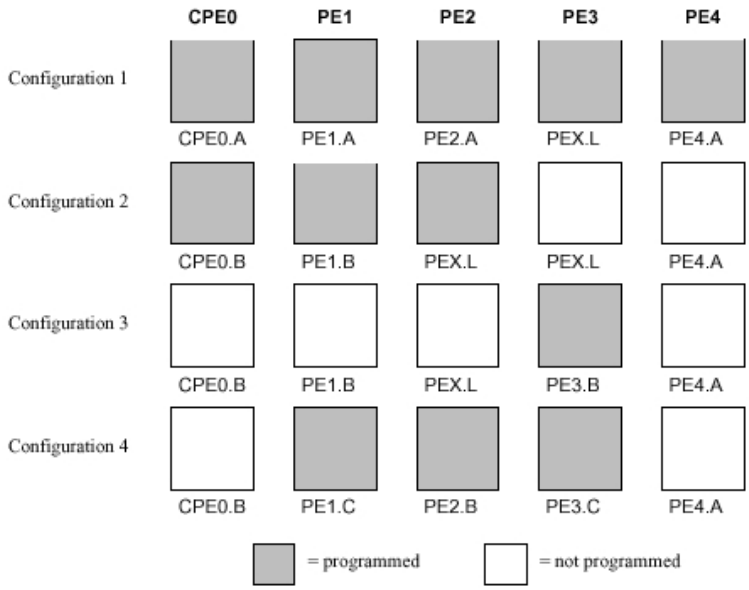


Figure 4.1 Configuration Mapping for START Algorithm [56]

The C/C++ and VHDL code generated for both algorithms during CHAMPION [67] will be recycled and adapted for implementation on the RC development hardware.

4.2.2 Classification Algorithm

The k-means clustering algorithm is a powerful technique used in many applications for data organization and analysis. To cope with the massive quantity of data generated in multispectral and hyperspectral imaging for example, data is often organized so that pixels with similar spectral content are clustered together in the same category [53] providing both compression of the data and segmentation of the image. Classic floating-point implementations of the basic algorithm in software require huge computational resources to compute the Euclidean distances and means. By fixing the precision of the computation and using alternative distance metrics, much of the iterative computations can be implemented in hardware and exploit the fine-grain parallelism [53,54,81,82]. These optimizations for hardware implementation come at the price of slightly less optimal clusters.

There are several variations on the basic algorithm, but most variants involve an iterative scheme that operates over a fixed number of clusters while attempting to maintain the class center at the mean position of all the samples in that class and each sample in the class where it is closest to the center [81]. The basic k-means algorithm is given in Figure 4.2. Traditionally, steps 1) and 2) are done iteratively and cannot be pipelined, however, within each step there is potential for parallelism and pipelining.

Initialize

Loop until termination condition is met:

For each sample, assign that sample to a class such that the distance from the sample to the center of that class is minimized

For each class, recalculate the means of the class based on the samples in that class

End loop;

Figure 4.2 k-means Basic Algorithm

There are several ways to extract parallelism from the k-means algorithm.

Divide the data among processing elements. The sample iteration can be done on any piece of data without any communication to other PEs. The data can be divided in any manner and spatial locality is not required. Communication between PEs is only necessary after all pixels have been classified in order to recalculate means.

Divide the mean table among PEs. For *m+1* PEs, give the first *m* PEs *1/m* of the mean table. Each PE communicates the class (from the classes it is assigned) to which a sample is closest to the host PE which determines the class to which the sample will be assigned. More communication is required for this scheme but only once per sample not once per classification loop.

Divide the channels among PEs. Each PE determines the distance from the spectra that it has. This is combined with the distances calculated from other PEs to determine the closest class. Note that more than the minimum distance might be required to complete this calculation. This scheme requires the most communication and should be avoided although it may be necessary for hyperspectral data [26].

The quality of results from the k-means algorithm depends greatly on the initialization. There are three primary ways to initialize:

Arbitrarily assign classes to samples: This is a good approach when the number of channels is greater than the number of classes. It works better in software than hardware because a large number of bits are needed for precision in the first few iterations.

Distribute the mean table around the sample space: This is a good approach if the number of channels is less than the number of classes.

Initialize mean values with random samples.

Theoretically, the k-means algorithm terminates when no more samples are changing classes. Running to completion may require a large number of iterations. In software, termination normally occurs when either a fixed number of iterations have occurred or fewer than *n* samples change classes.

The computation required in the k-means algorithm consists of calculating the distance between samples and the class center. The most common distance calculations are L1 or Manhattan distance and L2 or Euclidean distance. *Manhattan distance* is the absolute value of the componentwise difference between the sample and the class. This is the simplest distance to calculate and may be more robust to outliers. *Euclidean distance* is the square root of the componentwise square of the difference between the sample and the class. This requires more precision to compute. The square root can be omitted since we are only comparing the results however the squaring operation is still "expensive" in hardware. The advantage of this method is that the distance is a sphere around the centroid. The quality of a partition can be quantified by the within-class variance or the sum of squared (Euclidean) distances from each sample to that sample's cluster center.

Hardware and software implementations of the k-means algorithm will differ in their design tradeoffs. The primary goal in software is to minimize the number of iterations; in hardware, it is to simplify the underlying operations in order to exploit parallelism and speed up calculations. The calculation of the Euclidean distance is not amenable to hardware and alternative metrics must be used. As discussed in [82], the Manhattan distance is a suitable alternative and easily implemented in hardware. Also, we can fix the number of clusters *K* which is chosen by the user before hardware implementation. The initialization and termination will be determined in software allowing for more flexibility.

4.2.2.1 k-means Hardware Implementation for Multispectral and Hyperspectral Imaging

The two steps of the k-means algorithm are: (1) assign each pixel to one of the *K* clusters, and (2) re-compute the cluster centers. The hardware implementation presented in [53] includes step (1) and the accumulation portion of step (2); the cluster centers are computed in software. The design is fully

pipelined and a new pixel is read from memory and calculation begins on each clock cycle. The hardware design consists of a datapath section, accumulator, and control circuit as shown in Figure 4.3 [53].

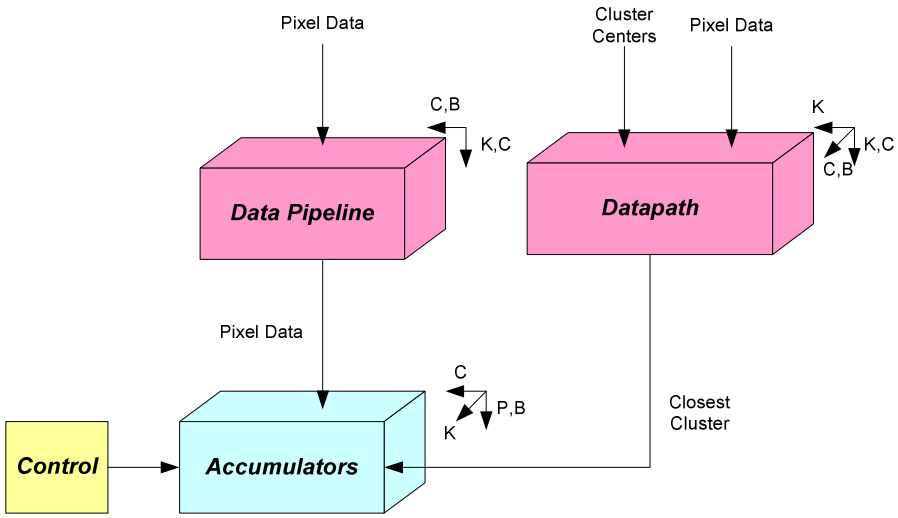


Figure 4.3 k-means Hardware Implementation

The datapath section calculates the Manhattan distance and assigns each pixel to the closest cluster. The distance to each of the *K* clusters is calculated in parallel and compared to find the minimum. The output of the datapath section is the cluster number of the closest center. Calculation of the Manhattan distance involves three serial operations: subtraction, absolute value and addition.

$$\left\|\mathbf{x}-\mathbf{c}\right\|^{p} = \sum_{i}\left|x_i - c_i\right|^{p} \quad \text{, where p=1 for Manhattan distance} \qquad \textbf{(4.3)}$$

The accumulator section adds the pixel values to the cluster to which the pixel is being assigned for recalculation of the centers at the end of the iteration. The control circuit synchronizes the pipeline and accumulator.

To adapt this implementation to HPRC, we will have to determine how to best partition the data and communicate results between nodes.

4.2.3 Digital Holography Reconstruction

The full digital reconstruction of off-axis holograms involves algorithms (FFTs) which use significant CPU time even on fast computers [85]. The live display of phase images play an important role in many of the applications. Additional processing, such as phase unwrapping or amplification, of the reconstructed amplitude and phase images is often considered part of the overall reconstruction procedure.

A well known image processing algorithm for digital holography image reconstruction will be used as a test algorithm for the modeling framework. A block diagram of the signal processing portion of the algorithm is shown in Figure 4.4.

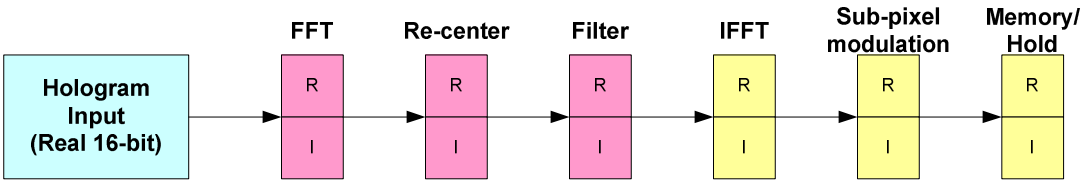


Figure 4.4 Image Processing for Hologram Reconstruction

The holographic image can be represented by equation (4.4). The first step in extracting the interesting information about the complex image wave (and from that the image amplitude and phase) from the hologram is to perform a Fourier transform [85]. The result of the FFT is the *autocorrelation* and *sidebands* (equation (4.5)). The holographic information, i.e., the information about the amplitude and phase of the image wave are present in those sidebands [85]. Note that the sidebands are complex conjugates of each other and therefore contain the same information.

$$I(x,y) = 1 + A^2 + 2\mu A\cos(2\pi(\omega_x x + \omega_y y) + \phi(x,y)) \quad \textbf{(4.4)}$$

$$\Im\{I(x,y)\} = \Im\{[1 + A^2 + 2\mu A\cos(2\pi(\omega_x x + \omega_y y) + \phi(x,y))]e^{-j2\pi(\omega_x x + \omega_y y)}\} \quad \textbf{(4.5)}$$

To extract the holographic information from equation (4.5), one of the sidebands must be isolated by centering and filtering. Several different types of apertures or filters can be used in the reconstruction process depending on the desired results and complexity [85]. *Hard apertures* with value 1 inside and 0 outside are rarely used because they tend to cause artifacts in the reconstructed images. *Soft apertures* such as the exponential filter or the Butterworth filter are more commonly used. The result of centering and filtering using a Butterworth Filter is shown in equation (4.6).

$$\Im\{[1 + A^2 + 2\mu A\cos(2\pi(\omega_x x + \omega_y y) + \phi(x,y))]e^{-j2\pi(\omega_x x + \omega_y y)}\} \cdot W_B = \Im\{2\mu A e^{j\phi(x,y)}\} \quad \textbf{(4.6)}$$

Finally an inverse FFT and a final shift are performed as shown in equation (4.7).

$$\psi(x,y) = 2\mu A(x,y)e^{j\phi(x,y)} \quad \textbf{(4.7)}$$

Table 7 shows the trace results for a reconstruction algorithm running on a dual Pentium 550. The algorithm is written in C++ and uses a precompiled FFT library. The results do not include overhead

routines since they will likely vary depending on the implementation. The trace results show a single iteration which processes one image.

Table 7 Trace results for one image

	Total Time (ms)
Real FFT	337.630
Complex Magnitude	211.738
Find Peak	5.112
Shift-x	68.608
Shift-y	84.373
Filter	111.093
Complex FFT	543.291
Nyquist Shift	126.370
Sub Pixel Shift	420.882
Magnitude/Phase Split	408.012

The first step for implementation on HPRC will be to develop a MATLAB or complete C version of the algorithm to confirm the reconstruction steps and determine the expected results or output from the input data set. A 2-D FFT implementation in VHDL will also be needed. Once we have a feel for the algorithm, we can explore how to partition the algorithm between hardware and software and ultimately across multiple nodes.

4.3 Status

We have published three papers [71,77,78] documenting our research plans and other conference and journal submissions in process or planned. The published papers describe the HPRC environment, tools needed, potential performance advantages, and an introduction to the performance modeling approach proposed here. Current and future submissions will detail modeling results and discuss the model's application to other domains such as SoC.

The method of attack as previously discussed will be to isolate as much as possible each of the interesting effects for independent study in order to quantify how well the model represents each of them. The model predictions will be compared with empirical results from the demo algorithms. Parameters such as t_{serial}, t_{config}, t_{HW}, t_{data}, etc. will be determined from empirical measurements, many of which have already been conducted for the currently available hardware. The application load imbalance for the host nodes

and RC systems still requires more analysis to characterize and complete the model. In the remainder of this section, we will discuss the phases of model development as shown in Figure 4.5 and the status of work completed in each phase. As Figure 4.5 depicts, the model development is an iterative process and may require multiple visits to each phase.

4.3.1 Phase 1 – HPC

Analysis of the parallel processing environment and network has begun with node to node communication measurements. PVM code was developed to measure the setup and communication times for message round trip time between workstations for the multiprocessor network. Results of these measurements were given earlier in section 3.2.5. In this phase, so far the focus has been on the communication between nodes of the system specifically the message travel time for different message sizes. More studies are needed to analyze the concept of workstation relative speed as discussed in section 3.2.1. Other work currently underway at UT may be useful in characterization and further refinement of the HPC parameters.

4.3.2 Phase 2 - RC

In this phase, our focus is on the elements specific to RC systems. The application demos from CHAMPION, Annapolis Microsystems, and SLAAC were used to measure and characterize a single RC node. As discussed earlier in section 3.3, a performance model based on these measurements was developed for a stand-alone RC unit. At this time, we do not have the hardware to run similar tests on the Pilchard architecture, but once the hardware is available, similar measurements will be conducted to further validate the model's representation of a single node. At that time, we will also be able to compare any potential differences in the communication interfaces. It should be noted that currently, configuration of the Pilchard boards is only available off-line and is not integrated with the application. Until we are able to integrate configuration into the running application, we will have to use an alternative representation for configuration in the model or ignore the hardware configuration portion of the model for the Pilchard boards.

4.3.3 Phase 3 - HPRC

With these initial measurements of the communication and configuration times at all levels of the architecture, we can begin looking at the algorithms and developing implementations that will run on the HPRC platform for validation of the model. In this phase, we will bring together the issues studied in phases 1 and 2 and focus on how well the model represents the entire system. The testing algorithms discussed in section 4.2 will be used to study and validate the HPRC model and determine modeling errors and if refinements to the model are necessary.

Algorithm development and system studies will begin with the simplest algorithm (CHAMPION). As confidence in the model and system are achieved, work will progress to the other two algorithms. As additional results are obtained, the HPRC model will be scrutinized with the measured values. This phase of the analysis will likely take several iterations to complete and may require revisiting of the other phases for more analysis. Once completed, final experiments can be conducted with the demo algorithms and real data sets to measure the performance metrics of the HPRC platform. These measurements can then be used to validate the model and again make any further refinements. The measurements from these experiments should also reveal the data interaction patterns for P-RC and P-P communications and help to characterize load imbalances.

Once the modeling framework is refined, it can be used to analyze performance tradeoffs in the HPRC architecture design space such as number of nodes, type of processor (relative processor speed), number and size of FPGAs, memory size and distribution, network bandwidth, and others. We can also use the model predictions to feed cost functions for analysis of metrics such as power, size, and cost.

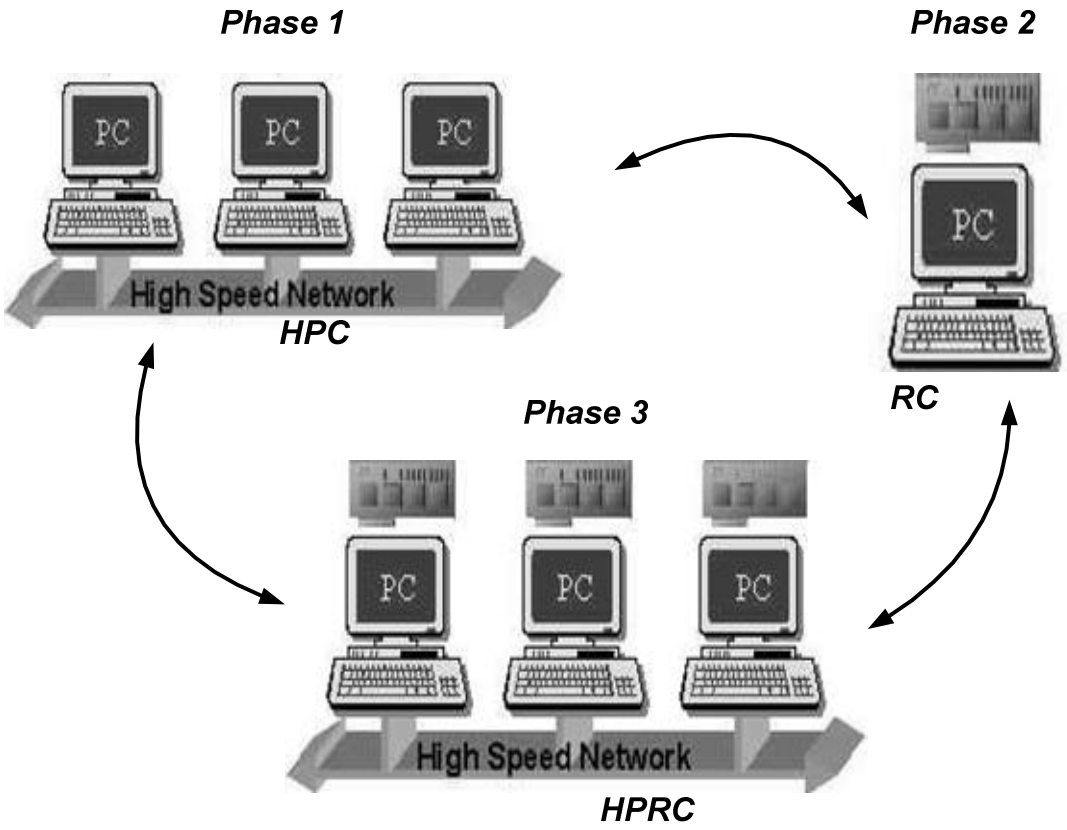


Figure 4.5 Phases of Model Development

Below in Table 8 some status checkpoints for the model and demo algorithm development are listed. The table provides easy reference for the status of each item listed from analysis and planning, model and algorithm development, measurements, and validation.

Table 8 Status Checkpoints

	Analysis & Planning	Model Development	Measurements	Validation		
Single RC Node						
P to RC Communication	✓	✓	✓			
P to RC Load Balance β	✓	✓				
RC Configuration/Reconfiguration	✓	✓	✓			
P to/from RC Data Xfer	✓	✓	✓			
Hardware Acceleration σ	✓	✓				
RC Unit Setup	✓	✓	✓			
Power	✓					
Area	✓					
Cost	✓					
HPC Study						
Setup Costs	✓	Partially Complete	✓			
Synchronization Costs	✓					
P to P communication	✓	✓	✓			
Multiple RC Nodes						
P to/from RC Data Xfer	✓	✓				
RC Unit Setup	✓	✓				
Node to Node Application Load Balance α	In Progress					
Node to Node RC Load Balance β	In Progress					
Hardware Acceleration σ	In Progress					
Synchronization Costs	✓					
Power	✓					
Area						
Cost	✓					
Demo Algorithms	Analysis & Planning	Algorithm Development	Measurements	Validation		
Algorithm 1: CHAMPION Demo	✓	✓	✓			
Algorithm 2: Classification Algorithm k-means	✓	✓				
Algorithm 3: Digital Holography Reconstruction	✓	✓				

4.4 Remaining Work

Initial analysis and measurements have been conducted on the available hardware and a first pass of the RC and HPRC models have been developed. The RC platform benchmarks need to be expanded to include tests for determining the synchronization and load balancing parameters. More measurements are needed for the unavailable hardware (Pilchard board) and to determine the effects of synchronization in a multi node environment and to determine load balance factors. Modeling to predict the two load imbalance factors is currently under development using Peterson's results as a guide [69]. Also, cost functions for power and total cost of the system will need to be developed. More studies are needed for the HPC communication model to determine if the current representation for contention will be sufficient. It is

expected that results from other research efforts underway at UT will be useful in the HPC communication study. As the demo algorithms are developed and studied, application of the model to scheduling and load balancing will be demonstrated as a manual exercise.

Efficiency is a common metric for HPC performance but is not easily transformed into the RC domain. We gave a preliminary analysis of the concept of efficiency in RC systems but more investigation and thought will be given to this concept as well as applying it to the HPRC platform. One idea currently being explored is the concept of an *effective processor* quantity which can be applied to the FPGA. If we can determine the effective processing capability or P_{eff} for a given FPGA in an RC system, the efficiency concept from the HPC domain will more directly apply. A review of the work by Dehon [24] may be helpful in developing this quantity.

Four demo algorithms have been selected and described in the previous sections. The algorithms from CHAMPION have existing software and firmware but modifications are necessary for implementation in our HPRC environment. The firmware will have to be implemented for the Pilchard architecture and other modifications to both firmware and software will be necessary to convert the existing algorithms into parallel ones. Once this is complete, model validation measurements will be conducted. The main features of interest in these algorithms will be all of the RC system overhead issues (serial setup, synchronization, communication, RC configuration, memory access, etc.). The relative simplicity of the algorithms will permit easier analysis of these issues without unnecessary distractions from the algorithm itself.

The selected classification algorithm has been studied by other researchers for implementation in RC hardware [26]. The firmware code is adaptable for our RC hardware and modifications for parallel operation will be needed. Software for the parallel algorithm will also need to be developed. Again, once the application is running, measurements will be conducted to validate the model. This algorithm will provide a more detailed study into the load balance issues since it should be large enough (at least in data sets) to benefit from a parallel algorithm implementation.

Finally, the Holography algorithm has the least amount of preliminary work completed thus far. The only implementation immediately available is a C/C++ algorithm running on WinNT; however the FFT functions are from a precompiled library. Consequently, a MATLAB and C version are in

development so that we have the complete algorithm in a language-based environment that can be used to target the RC system. A parallel version with hardware and software components suitable for HPRC will also need to be developed. Again, measurements on the running application will be conducted and used to validate the model. This algorithm should also provide a good test of the HPRC capabilities and enable us to study all of the model parameters.

Table 9 Planned Experiments and Goals, lists some of the planned tests and what they are designed to achieve.

Table 9 Planned Experiments and Goals

Experiment	Goals and Objective
CHAMPION Demo	Mostly RC System Analysis
Existing Wildforce Implementation	Measurements for serial setup, RC configuration, synchronization, memory access, and initial analysis of implementation efficiency using CHAMPION results
Direct port to Pilchard	Comparison of measurements for serial setup, RC configuration, synchronization, and memory access
Parallel version	Elementary parallel application to study data distribution and synchronization/communication parameters
Classification Demo	Basic HPRC System Analysis
Single node	Study of RC parameters
Multi-node w/ different parallelization techniques	Study load imbalance
Mult-inode with and without hardware acceleration	Study implementation efficiency
Holography Demo	Further HPRC Validation
Software only	Baseline for implementation efficiency and speedup
HPRC	Study complete model

5 References

[1] Altera: Systems on a Programmable Chip, http://www.altera.com, 2001.

[2] Annapolis Microsystems, http://www.annapmicro.com, 2001.

[3] BLAS: Basic Library of Algebraic Subroutines, http://www.netlib.org/blas/index.html, 2001.

[4] NetSolve, http://icl.cs.utk.edu/netsolve/, 2001.

[5] SInRG: Scalable Intracampus Research Grid, http://www.cs.utk.edu/sinrg/index.html, 2001.

[6] Vector Signal Image Processing Library (VSIPL), http://www.vsipl.org, 2001.

[7] Accelerated Strategic Computing Initiative, http://www.llnl.gov/asci/, 2002.

[8] BYU Configurable Computing Laboratory, http://www.jhdl.org, 2002.

[9] Nallatech FPGA-Centric Systems & Design Services, http://www.nallatech.com/, 2002.

[10] Ptolemy for Adaptive Computing Systems, http://ptolemy.eecs.berkeley.edu/, 2002.

[11] Tennessee Oak Ridge Cluster (TORC) Project, http://www.csm.ornl.gov/torc/, 2002.

[12] Virtual Computer Corporation, http://www.vcc.com/index.html, 2002.

[13] Amdahl, G. M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities", *In AFIPS Conference Proceedings*, pp. 483-485, 1967, Reston, VA.

[14] Atallah, M. J., Black, C. L., Marinescu, D. C., Segel, H. J., and Casavant, T. L., "Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations," *Journal of Parallel and Distributed Computing*, vol. 16 pp. 319-327, 1992.

[15] Bellows, P. and Hutchings, B. L., "JHDL - An HDL for Reconfigurable Systems," Pocek, K. and Arnold, J., *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 175-184, 1998, Napa, CA, IEEE Computer Society.

[16] Bolch, G., Greiner, S., de Meer, H., and Trivedi, K. S., *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications* New York: John Wiley and Sons, Inc., 1998.

[17] Bondalapati, K., Dinz, P., Duncan, P., Granacki, J., Hall, M., Jain, R., and Ziegler, H., "DEFACTO: A Design Environment for Adaptive Computing Technology," *Proceedings of the 6th Reconfigurable Architectures Workshop (RAW99)*, 1999, Springer-Verlag.

[18] Casanova, H., Dongarra, J., and Jiang, W., "The Performance of PVM on MPP Systems," CS-95-301, pp. -19, 7-19-1995, Knoxville, TN, University of Tennessee.

[19] Casavant, T. L., "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *IEEE Transactions on Software Engineering*, vol. SE-14, no. 2, pp. 141-154, Feb.1988.

[20] Chamberlain, R. D., "Parallel Logic Simulation of VLSI Systems," *Proc. of 32nd Design Automation Conf.*, pp. 139-143, 1995.

[21] Clement, M. J. and Quinn, M. J., "Analytical Performance Prediction on Multicomputers," *Proceedings of Supercomputing '93*, 1993.

[22] Clement, M. J., Steed, M. R., and Crandall, P. E., "Network Performance Modeling for PVM Clusters," *Proceedings of Supercomputing '96*, 1996.

[23] Compton, K. and Hauck, S., “Configurable Computing: A Survey of Systems and Software,” Northwestern University, Dept. of ECE Technical Report, 1999, Northwestern University.

[24] DeHon, Andre, "Reconfigurable Architectures for General-Purpose Computing," Ph.D. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1996.

[25] DeHon, A., "Comparing Computing Machines," *Proceedings of SPIE*, vol. 3526, no. Configurable Computing: Technology and Applications, pp. 124, Nov.1998.

[26] Dept.ECE Northeastern University, *K-Means Clustering on Reconfigurable Hardware*, http://www.ece.neu.edu/groups/rpl/kmeans/index.html, 2001.

[27] Dongarra, J. and Dunigan, T., “Message-Passing Performance of Various Computers,” pp. -16, 1-21-1997.

[28] Dongarra, J., Goel, P. S., Marinescu, D., and Jiang, W., “Using PVM 3.0 to Run Grand Challenge Applications on a Heterogeneous Network of Parallel Computers,” Sincovec, R. and et al., *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, pp. 873-877, 1993, Philadelphia, SIAM Publications.

[29] El-Rewini, H. and Lewis, T. G., *Task Scheduling in Parallel Distributed Systems* Prentice Hall, 1994.

[30] Flynn, M. J., "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948-960, Sept.972.

[31] Fujimoto, R. M., “Parallel and Distributed Discrete Event Simulation: Algorithms and Applications,” *Proceedings of the 1993 Winter Simulation Conference*, pp. 106-114, 1993,

[32] Fujimoto, R. M., “Parallel and Distributed Simulation”, P*roceedings of the 1999 Winter Simulation Conference*, pp. 122-131, 1999.

[33] Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., and Sundarem, V., *PVM: A User's Guide and Tutorial for Networked Parallel Computing* MIT Press, 1994.

[34] Gokhale, M., Homes, W., Kopser, A., Lucas, S., Minnich, R., Sweely, D., and Lopresti, D., "Building and Using a Highly Parallel Programmable Logic Array," *IEEE Computer*, vol. 24, no. 1, pp. 81-89, Jan.1991.

[35] Goldstein, S. C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., and Taylor, R. R., "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer*, pp. 70-77, Apr.2000.

[36] Guccione, S. A., Levi, D., and Sundararajan, P., “JBits: A Java-Based Interface for Reconfigurable Computing,” *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999, Laurel, MD.

[37] Gustafson, J. L., "Reevaluating Amdahl's Law," *Communications of the ACM*, vol. 31, no. 5, pp. 532-533, May1988.

[38] Hall, M., Anderson, J. M., Amarasinghe, S. P., Murphy, B. R., Liao, S.-W., Bugnion, E., and Lam Monica S., "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer*, vol. 29, no. 12, pp. 84-89, Dec.1996.

[39] Hauck, S., "The Roles of FPGAs in Reprogrammable Systems," *Proceedings of the IEEE*, vol. 86, no. 4, pp. 615-638, Apr.1998.

[40] Hauck, S., “The Future of Reconfigurable Systems, Keynote Address,” *5th Canadian Conference on Field Programmable Devices*, 1998, Montreal.

[41] Hauck, S., Fry, T. W., Hosler, M. M., and Kao, J. P., "The Chimaera Reconfigurable Functional Unit," *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. -10, 1997.

[42] Hauser, J. R. and Wawrzynek, J., "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997.

[43] Hu, L. and Gorton, I., "Performance Evaluation for Parallel Systems: A Survey," UNSW-CSE-TR-9707, pp. -56, 1997, Sydney, Australia, University of NSW, School of Computer Science and Engineering.

[44] Hwang, K., *Advanced Computer Architecture: Parallelism, Scalability, and Programmability*, First ed. New York: McGraw-Hill, Inc., 1993, pp. -771.

[45] I.S.I.East, SLAAC: System-Level Applications of Adaptive Computing, http://, 1998.

[46] Jones, A., "Matrix and Signal Processing Libraries Based on Intrinsic MATLAB Functions for FPGAs," Master Computer Engineering, Northwestern, 2001.

[47] Jones, A., Nayak, A., and Banerjee, P., "Parallel Implementation of Matrix and Signal Processing Libraries on FPGAs," *PDCS*, 2001.

[48] Jones, M., Scharf, L., Scott, J., Twaddle, C., Yaconis, M., Yao, K., Athanas, P., and Schott, B., "Implementing an API for Distributed Adaptive Computing Systems," *FCCM Conference*, 1999.

[49] Jones, M. T., Langston, M. A., and Raghavan, P., "Tools for mapping applications to CCMs," *In SPIE Photonics East '98*, 1998.

[50] Kant, K., *Introduction to Computer System Performance Evaluation* New York: McGraw-Hill, Inc., 1992.

[51] Katz, D. S., Cwik, T., Kwan, B. H., Lou, J. Z., Springer, P. L., Sterling, T. L., and Wang, P., "An assessment of a Beowulf system for a wide class of analysis and design software," *Advances in Engineering Software*, vol. 29, no. 3-6, pp. 451-461, 1998.

[52] Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C., *Quantitative System Performance: Computer System Analysis Using Queueing Network Models* Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1984, pp. -417.

[53] Leeser, M., Belanovic, P., Estlick, M., Gokhale, M., Szymanski, J., and Theiler, J., "Applying Reconfigurable Hardware to the Analysis of Multispectral and Hyperspectral Imagery," *Proc.SPIE*, vol. 4480 2001.

[54] Leeser, M., Kitaryeva, N., and Crisman, J., "Spatial and Color Clustering on an FPGA-based Computer System," *Proc.SPIE*, vol. 3526 pp. 25-33, 1998.

[55] Leong, P. H. W., Leong, M. P., Cheung, O. Y. H., Tung, T., Kwok, C. M., Wong, M. Y., and Lee, K. H., "Pilchard - A Reconfigurable Computing Platform With Memory Slot Interface," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2001, California USA, IEEE.

[56] Levine, Ben, "A Systematic Implementation of Image Processing Algorithms on Configurable Computing Hardware," Master of Science Electrical Engineering, The University of Tennessee, 1999.

[57] Li, Y., Callahan, T., Darnell, E., Harr, R., Kurkure, U., and Stockwood, J., "Hardware-Software Co-Design of Embedded Reconfigurable Architectures," *Design Automation Conference DAC 2000*, pp. 507-512, 2000, Los Angeles, California.

[58] Mohapatra, P. and Das, C. R., "Performance Analysis of Finite-Buffered Asynchronous Multistage Interconnection Networks," *Transactions on Parallel and Distributed Systems*, pp. 18-25, Jan.1996.

[59] Mohapatra, P., Das, C. R., and Feng, T., "Performance Analysis of Cluster-Based Multiprocessors," *IEEE Transactions on Computers*, pp. 109-115, 1994.

[60] Moll, L., Vuillemin, J., and Boucard, P., “High-Energy Physics on DECPeRLe-1 Programmable Active Memory,” *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 47-52, 1995.

[61] Myers, M., Jaget, K., Cadambi, S., Weener, J., Moe, M., Schmit, H., Goldstein, S. C., and Bowersox, D., *PipeRench Manual*, pp. -41, 1998, Carnegie Mellon University.

[62] Natarajan, Senthil, "Development and Verification of Library Cells for Reconfigurable Logic," Master of Science Electrical Engineering, The University of Tennessee, 1999.

[63] Natarajan, S., Levine, B., Tan, C., Newport, D., and Bouldin, D., “Automatic Mapping of Khoros-Based Applications to Adaptive Computing Systems,” *MAPLD-99*, 1999, Laurel, MD.

[64] Noble, B. L. and Chamberlain, R. D., “Performance Model for Speculative Simulation Using Predictive Optimism,” *Proceedings of the 32nd Hawaii International Conference on System Sciences*, pp. 1-8, 1999.

[65] Noble, B. L. and Chamberlain, R. D., “Analytic Performance Model for Speculative, Synchronous, Discrete-Event Simulation,” *Proc. of 14th Workshop on Parallel and Distributed Simulation*, 2000.

[66] Nupairoj, N. and Ni, L. M., “Performance Evaluation of Some MPI Implementations on Workstation Clusters,” *Proceedings of the 1994 Scalable Parallel Libraries Conference*, pp. 98-105, 1994, IEEE Computer Society.

[67] Ong, Sze-Wei, "Automatic Mapping of Graphical Programming Applications to Microelectronic Technologies," Doctor of Philosophy Electrical Engineering, University of Tennessee, 2001.

[68] Ong, S.-W., Kerkiz, N., Srijanto, B., Tan, C., Langston, M., Newport, D., and Bouldin, D., “Automatic Mapping of Multiple Applications to Multiple Adaptive Computing Systems,” *FCCM Conference 2001*, 2001.

[69] Peterson, Gregory D., "Parallel Application Performance on Shared, Heterogeneous Workstations." Doctor of Science Washington University Sever Institute of Technology, Saint Louis, Missouri, 1994.

[70] Peterson, G. D. and Chamberlain, R. D., "Parallel application performance in a shared resource environment," *Distributed Systems Engineering*, vol. 3 pp. 9-19, 1996.

[71] Peterson, G. D. and Smith, M. C., “Programming High Performance Reconfigurable Computers,” *SSGRR 2001*, 2001, Rome, Italy.

[72] Peterson, J. L., *Petri Net Theory and the Modeling of Systems* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[73] Reynolds, P. F., Jr. and Pancerella, C. M., “Hardware Support for Parallel Discrete Event Simulations,” TR-92-08, 1992, Computer Science Dept.

[74] Reynolds, P. F., Jr., Pancerella, C. M., and Srinivasan, S., “Making Parallel Simulations Go Fast,” *1992 ACM Winter Simulation Conference*, 1992.

[75] Reynolds, P. F., Jr., Pancerella, C. M., and Srinivasan, S., "Design and Performance Analysis of Hardware Support for Parallel Simulations," *Journal of Parallel and Distributed Computing*, Aug.1993.

[76] Shetters, Carl Wayne, "Scheduling Task Chains on an Array of Reconfigurable FPGAs", Master of Science University of Tennessee, 1999.

[77] Smith, M. C., Drager, S. L., Pochet, Lt. L., and Peterson, G. D., “High Performance Reconfigurable Computing Systems,” *Proceedings of 2001 IEEE Midwest Symposium on Circuits and Systems*, 2001.

[78] Smith, M. C. and Peterson, G. D., “Programming High Performance Reconfigurable Computers (HPRC),” *SPIE International Symposium ITCom 2001*, 8-19-2001, Denver, CO, SPIE.

[79] Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J., *MPI: The Complete Reference*, 2nd ed. MIT Press, 1998.

[80] SUIF, The Stanford SUIF Compilation System: Public Domain Software and Documentation, http://suif.stanford.edu, 2001.

[81] Theiler, J. and Gisler, G., "A contiguity-enhanced k-means clustering algorithm for unsupervised multispectral image segmentation," *Proc. SPIE 3159*, pp. 108-118, 1997.

[82] Theiler, J., Leeser, M., Estlick, M., and Szymanski, J., "Design Issues for Hardware Implementation of an Algorithm for Segmenting Hyperspectral Imagery," *Proc. SPIE*, vol. 4132 pp. 99-106, 2000.

[83] Thomasian, A. and Bay, P. F., "Analytic Queueing Network Models for Parallel Processing of Task Systems," *IEEE Transactions on Computers*, vol. C-35, no. 12, pp. 1045-1054, Dec.1986.

[84] Underwood, K. D., Sass, R. R., and Ligon, W. B., III, "A Reconfigurable Extension to the Network Interface of Beowulf Clusters," *Proc. of the 2001 IEEE International Conference on Cluster Computing*, pp. -10, 2001, IEEE Computer Society,

[85] Volkl, E., Allard, L. F., and Joy, D. C., *Introduction to Electron Holography* New York, NY: Kluwer Academic, 1999, pp. -353.

[86] Vuillemin, J., Bertin, P., Roncin, D., Shand, M., Touati, H., and Boucard, P., "Programmable Active Memories: Reconfigurable Systems Come of Age," *IEEE Transactions on VLSI Systems*, vol. 4, no. 1, pp. 56-69, Mar.1996.

[87] Xilinx, Virtex Series FPGAs, http://www.xilinx.com, 2001.

[88] Xilinx, JBits SDK, http://www.xilinx.com/products/jbits/index.htm, 2002.

[89] Yan, Y., Zhang, X., and Song, Y., "An Effective and Practical Performance Prediction Model for Parallel Computing on Non-dedicated Heterogeneous NOW," *Journal of Parallel and Distributed Computing*, vol. 38 pp. 63-80, 1996.

[90] Ye, Z. A., Moshovos, A., Hauck, S., and Banerjee, P., "CHIMAERA: A High-Performance Architecture with a Tightly-Coupled Reconfigurable Functional Unit," *In Proc. Of International Symposium on Computer Architecture*, 1998.

[91] Zhang, X. and Yan, Y., "Modeling and Characterizing Parallel Computing Performance on Heterogeneous Networks of Workstations," *Proceeding of the 7th IEEE Symposium on Parallel and Distributed Processing*, pp. 25-34, 1995.