12-2003

Analytical Modeling of High Performance Reconfigurable Computers: Prediction and Analysis of System Performance.

Melissa C. Smith
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a dissertation written by Melissa C. Smith entitled "Analytical Modeling of High Performance Reconfigurable Computers: Prediction and Analysis of System Performance.." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Electrical Engineering.

Gregory D. Peterson, Major Professor

We have read this dissertation and recommend its acceptance:

Don W. Bouldin, Hairong Qi, Michael Langston, Lynne Parker

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a dissertation written by Melissa C. Smith entitled “Analytical Modeling of High Performance Reconfigurable Computers: Prediction and Analysis of System Performance.” I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Electrical Engineering.

Gregory D. Peterson

Major Professor

We have read this dissertation
and recommend its acceptance:

Don W. Bouldin

Hairong Qi

Michael Langston

Lynne Parker

Acceptance for the Council:

Anne Mayhew

Vice Provost and Dean of Graduate Studies

(Original signatures are on file with official student records.)

Analytical Modeling of High Performance Reconfigurable Computers: Prediction and Analysis of System Performance

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Melissa C. Smith

December 2003

DEDICATION

This dissertation is dedicated to my husband, Harrison, for always believing in me, my girls, Allison and Courtney, for being the joy of my life, and to the rest of my family, for their love and support, inspiring me and encouraging me to reach higher and strive harder.

ACKNOWLEDGEMENTS

I would like to send a big thank you to everyone who stood by me and helped me complete this long journey. I have been very fortunate to have had the support of numerous people over the years as I struggled through this degree. First, I would like to thank my family who sacrificed plenty and suffered the brunt of my frustrations and stress much of the time. My parents and my brother who have always been there to provide support and encouragement when I had second thoughts and doubted myself. I am truly blessed to have them in my life. A very special thanks goes out to my husband, Harrison, who has always been there for me no matter what the circumstances. Without his love and patience, none of this would have been possible. He kept me centered, grounded, and always focused on the end goal. I love him dearly. And most importantly, my girls, Allison and Courtney. Although you will not remember all the hours mommy spent working so diligently on her research (because you were only six months old when she finished), you helped her everyday and provided inspiration and motivation. I always knew, no matter how difficult the day had been, I could always count on both of you for your unconditional love. Your smiles and hugs carried me through many tough days. You are my inspiration, my joy, my life, and I love both of you more than words can say.

My advisor, Greg Peterson was the force that kept me going. His guidance and advice were key in molding this dissertation. Somehow he managed to keep me focused, extracting my best, even when I doubted myself. In addition, I would like to thank the other members of my committee who also played key roles in my success, Don Bouldin, Michael Langston, Hairong Qi, and Lynne Parker. Their comments and encouragement greatly improved my dissertation and I am indebted to them for their service. Thanks to all of you.

Many others have played seemingly small roles but without them, I could not have finished. Gary Alley, for standing by me and fighting for my best interest. I hope to make you proud. Bill Holmes for his telecommuting advice and expertise. I would also like to extend my appreciation to all my colleagues at the Oak Ridge National Laboratory for being so supportive during my trials and tribulations. Also, thanks to everyone involved in our HPRC research group (Mahesh, Chandra, Adam, Kirk, Joe, Bahnu, Ashwin, Venky, Jason) you each played an integral part in my success. My friends in Hong Kong who helped with the Pilchard machines especially Hok, I could not have done it without you.

Many gracious thanks to those who supported my research with funding and equipment. The HPRC computers and other equipment provided through the SInRG project at UT and funded by the National Science Foundation (contracts NSF 0075792 and NSF 9972889). For their financial support, I would also like to thank the Engineering Science and Technology Division at the Oak Ridge National Laboratory and the Air Force Research Laboratory (AFRL contract F30602-00-D-0221). The work contained in this dissertation was developed in part based upon technology licensed and developed by SRC Computers, Inc.

Last but not least, many thanks to all my friends who helped me “reset” on occasions and many other good deeds. These include Renee, Joely, Gail, Gayle, Janice, Missy, TJ, Trudi, Judy, Kelly, and many others who I have no doubtedly forgot, I hope they can forgive me! There have been so many people that have helped me enjoy (and survive) this time. Thanks to all of you. Finally, thanks to God for giving me the knowledge, patience, perseverance, and placing these individuals in my life to make it all happen.

ABSTRACT

The use of a network of shared, heterogeneous workstations each harboring a Reconfigurable Computing (RC) system offers high performance users an inexpensive platform for a wide range of computationally demanding problems. However, effectively using the full potential of these systems can be challenging without the knowledge of the system's performance characteristics. While some performance models exist for shared, heterogeneous workstations, none thus far account for the addition of Reconfigurable Computing systems. This dissertation develops and validates an analytic performance modeling methodology for a class of fork-join algorithms executing on a High Performance Reconfigurable Computing (HPRC) platform. The model includes the effects of the reconfigurable device, application load imbalance, background user load, basic message passing communication, and processor heterogeneity. Three fork-join class of applications, a Boolean Satisfiability Solver, a Matrix-Vector Multiplication algorithm, and an Advanced Encryption Standard algorithm are used to validate the model with homogeneous and simulated heterogeneous workstations. A synthetic load is used to validate the model under various loading conditions including simulating heterogeneity by making some workstations appear slower than others by the use of background loading. The performance modeling methodology proves to be accurate in characterizing the effects of reconfigurable devices, application load imbalance, background user load and heterogeneity for applications running on shared, homogeneous and heterogeneous HPRC resources. The model error in all cases was found to be less than five percent for application runtimes greater than thirty seconds and less than fifteen percent for runtimes less than thirty seconds.

The performance modeling methodology enables us to characterize applications running on shared HPRC resources. Cost functions are used to impose system usage policies and the results of

the modeling methodology are utilized to find the optimal (or near-optimal) set of workstations to use for a given application. The usage policies investigated include determining the computational costs for the workstations and balancing the priority of the background user load with the parallel application. The applications studied fall within the Master-Worker paradigm and are well suited for a grid computing approach. A method for using NetSolve, a grid middleware, with the model and cost functions is introduced whereby users can produce optimal workstation sets and schedules for Master-Worker applications running on shared HPRC resources.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivation

Integrating the architecture and techniques from parallel processing or High Performance Computing (HPC) with those of Reconfigurable Computing (RC) systems offers high performance users with the potential of increased performance and flexibility for a wide range of computationaly demanding problems. HPC architectures and RC systems have independently demonstrated performance advantages for many applications such as digital signal processing, circuit simulation, and pattern recognition among others. By exploiting the near "hardware specific" speed of RC systems in a distributed network of workstations there is promise for additional performance advantages over other software-only or uniprocessor solutions.

1.1.1 What is HPC?

High Performance Computing 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 classic taxonomy for classifying computer architectures as defined by Flynn in 1972 [59] is shown in Figure 1.1. Flynn's taxonomy 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 struc-

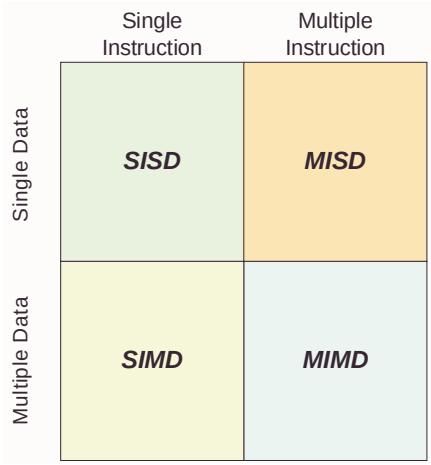


FIGURE 1.1 Flynn's Taxonomy

ture. 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 our modeling on distributed memory MIMD architectures.

1.1.2 What is RC?

Reconfigurable Computing is the combination of reconfigurable logic (most commonly in the form of Field Programmable Gate Arrays or FPGAs) with a general-purpose microprocessor. The architectural intention is to achieve higher performance than normally available from software-only solutions while at the same time providing flexibility not available with Application Specific Integrated Circuits (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 [43] to achieve the greatest performance advantage. We will discuss more details regarding RC systems in a later section including some representative architectures.

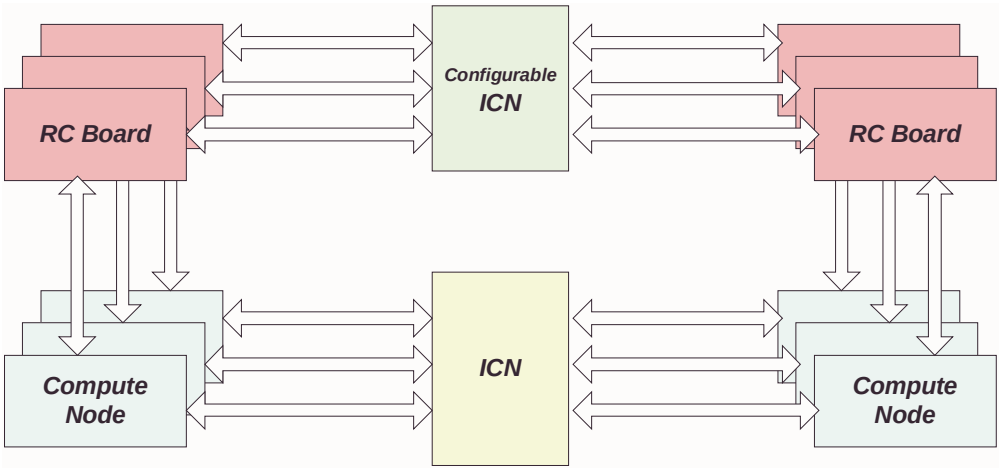


FIGURE 1.2 High Performance Reconfigurable Computer (HPRC) Architecture

1.1.3 What is HPRC?

High Performance Reconfigurable Computing or HPRC is the proposed combination of High Performance Computing and Reconfigurable Computing. The proposed HPRC platform shown in Figure 1.2 consists of a number of distributed computing nodes connected by some interconnection network (ICN) (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 coprocessing performance of the RC element. Many computationally intensive applications stand to benefit from this architecture: image and signal processing, simulations, among others. The focus of our modeling will be on algorithms and applications that fit into the *fork-join* class and more specifically *synchronous iterative algorithms*. More discussion on this class of algorithms and limitations follows in a later section.

An additional configurable network between RC elements may offer some applications such as discrete event simulation even more performance advantages by providing a less inhibited

route for synchronization, data exchange, and other communications necessary between processing nodes. Research by Chamberlain indicates the potential performance improvement from a dedicated synchronization network for synchronous, discrete-event simulations [38, 97, 98]. The *parallel reduction* network (PRN) proposed by Reynolds, et al., demonstrated the performance advantages from dedicated hardware to support synchronization in parallel simulations [111, 112, 113]. 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 between nodes. Other research by Underwood, et al. [123, 124, 125], confirms the performance benefits of a specialized configurable network interface card in a Beowulf cluster. A Beowulf cluster is an approach to building a supercompter as a cluster of commodity off-the-shelf personal computers, interconnected with an ICN. The idea is to build a cost effective, high performance computer. The Intelligent Network Interface Card (INIC) developed by Underwood, et al., uses reconfigurable computing to assist with both network communications and computational operations enhancing both the network and processor capabilities of the cluster. The results presented for a 2-D Fast Fourier Transform (FFT) and an integer sorting algorithm show significant performance benefit for both applications as the cluster size increases. The resulting performance for the two test applications is significantly better than on a Beowulf cluster with commodity network interface cards.

HPC and RC individually are challenging to program and utilize effectively. Combining these powerful domains will require new analysis tools to aid us in understanding and exploiting the design space to its full potential. 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 system performance is affected by architecture variables such as number of nodes, number of FPGAs, FPGA type and 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.

1.2 General Problem Statement

Networks or clusters of workstations can provide significant computational capabilities if effectively utilized. Adding to this powerful architecture the capabilities of RC systems introduces challenging problems in efficient utilization of resources. 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. Although substantial performance analysis research exists in the literature with regard to High Performance Computing (HPC) architectures [22, 23, 41, 67, 74, 92, 97, 98, 102, 106, 108, 122] and even some with respect to Reconfigurable Computing (RC) [47, 48, 72, 73], 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. To evaluate the tradoffs associated with this architecture, we need an accurate understanding of the computational system, including the workstations, the RC units, the distributed applications, and the effect of competing users of the resources. As part of this dissertation, we develop an accurate performance modeling methodology for synchronous iterative algorithms, a sub-class of fork-join algorithms, running on shared, heterogeneous resources.

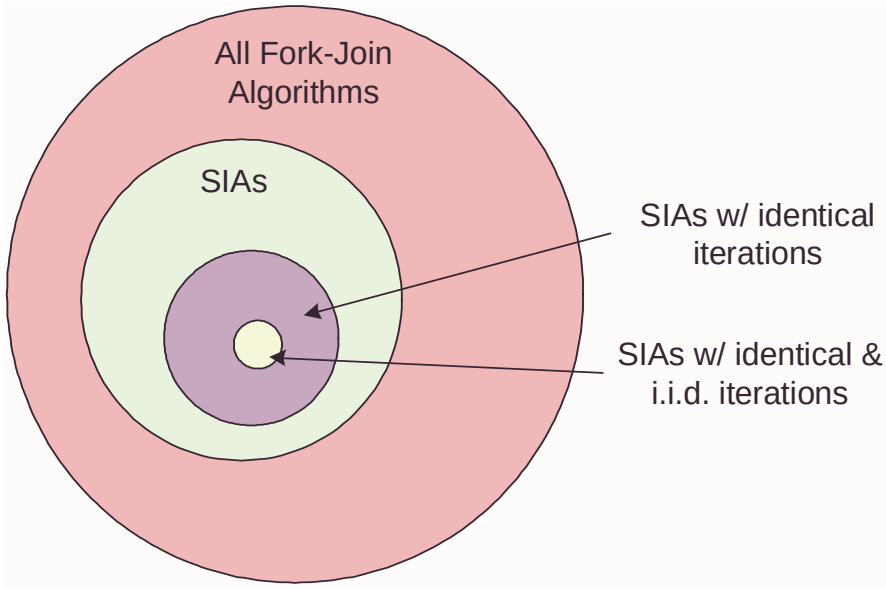


FIGURE 1.3 Fork-Join Class of Algorithms

1.2.1 Fork-Join and Synchronous Iterative Algorithms

As the name implies, *Synchronous Iterative Algorithms (SIAs)*, also known as *multi-phase algorithms* in the literature, are iterative in nature with each processor performing some portion of the required computation each iteration. Many computationaly intensive parallel applications fall into this large class of algorithms. SIAs are a sub-class of a much broader set of algorithms known as fork-join algorithms (see Figure 1.3). Also within the class of SIAs we find those with identical and independent iterations where the iterations of the algorithm are independent from one another and identical. In fork-join algorithms, a main process or thread forks off some number of other processes or threads that then continue in parallel to accomplish some portion of the overall work before rejoining the main process or thread. Numeric computations such as discrete-event simulation (excluding Time Warp), numeric optimizations, Gaussian elimination, FFTs, Encryption (Data Encryption Standard - DES and Advanced Encryption Standard - AES), sorting algorithms, solu-

tions of partial differential equations, and others are all members of this class of algorithms. The modeling methodology developed here is applicable to members of the fork-join class.

We develop a performance model for fork-join type algorithms that takes into account division of computation between the workstation processor and the reconfigurable unit, variance in the computational requirements of the application across the set of workstations, background loading (due to shared resources), and workstation heterogeneity. The development of an analytic performance model is a significant contribution that facilitates resource management optimization. By helping users understand the performance tradoffs in the computer architecture, one can quickly determine the optimium application mapping for given constraints or identify the best set of workstations for optimium runtime.

As an application of the performance model, we consider the scheduling algorithms employed in distributed systems. Many of the scheduling algorithms do not account for the performance implications of design choices [102]. We use the modeling results of this dissertation to improve the scheduling of applications and achieve better performance. We explore the impact of load conditions, workstation make-up, and other constraints using our modeling results as input to a scheduler.

We assume for our HPRC platform shown in Figure 1.2 that we have a network of shared heterogeneous workstations to which we will map our distributed applications. According to Flynn's taxonomy, we have a loosely-coupled distributed memory, MIMD processing environment. To facilitate distributed processing with these workstations, the Message Passing Interface (MPI) [118] system is used. MPI supports message-passing and the control of processes on heterogeneous distributed computers.

In the next chapter, a brief survey is given which discusses important related literature. Following this survey, we consider three fork-join type of applications which are used throughout the dissertation to compare our analytic modeling results to the empirical measurements. We then develop a performance modeling methodology for describing the performance of these algorithms executing on a cluster of shared heterogeneous workstations. Next, using the empirical results from the test applications, we validate the performance model. With the performance model validated, we then apply the model to optimizing the resource management of the cluster. Finally, we present conclusions of this dissertation and discuss future work to extend the applicability of the model and optimization results.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Introduction

In this chapter 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 at some performance metrics common in these architectures and how they relate to 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 Building the HPRC Architecture

The architecture of a computer system affects the performance, whether good or bad, of a given application. Issues such as dedicated and non-dedicated resources, memory size and distribution, communication structure, 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 and Networks of Workstations

The results found in High Performance Computing research confirm performance advantages over single processor machines for many computationally intensive applications. However, many HPC platforms have been limited to the relatively small research oriented market due to the

TABLE 2.1 HPC Architecture Examples [75, 17]

	SIMD	**MIMD**	
Shared Memory	Hitachi S3600 series Vector Machines XMP	KSR1 Cray C90, J-90, SV1ex YMP DASH	HP/Convex C4 series Tera MTA Hitachi S3800 series
Distributed Memory	CPP CM-1, CM-2 DEC MPP MasPar MP1, MP2 AMT DAP	BBN nCube CM-5 Intel iPSC Cray T3E Hitachi SR8000 Series	Sun Fire 3800-15K Paragon SP2 NOWs Clusters

high costs normally incurred in acquiring, developing and programming them. Although trends indicate a changing market [17, 50, 52] as venders migrate toward clusters and networks of workstations the cost of ownership still often precludes mass market use. These specialized machines dedicated to parallel processing often consist of an array of tightly coupled homogeneous processors connected by an optimized network. Furthermore, their architecture is often specialized for a specific genre of applications thus perpetuating the high costs associated with ownership and limiting the efficiency for applications other than the target application.

As discussed in Chap. 1, computer architectures can be divided into four main categories as described in Flynn's taxonomy [59]: *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 2.1.

By taking advantage of cheaper microprocessor technology and ever improving interconnection networks, today massively parallel systems can achieve higher performance for less cost. The task of harnessing the full potential of these systems is difficult at best but the results can be quite dramatic, with speedups many times that of a serial processor [67].

From a hardware standpoint, the recent HPC research in the area of Beowolf clusters is confirming high performance at reduced cost. Beowolf clusters consist of relatively inexpensive commodity workstations connected by a general-purpose network [82]. These systems, although they may not be recognized as Beowolf clusters, currently exist throughout industry, academia, and government. It has been shown that many of these workstations are often idle up to ninety-five percent of the time [33, 84] leaving most of their computing power unused. A means of harnessing the unused computing cycles of these workstations can provide a cost-effective and powerful HPC platform.

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) at the University of Tennessee [16] 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).

2.2.2 Reconfigurable Computing

Reconfigurable computing (RC) is the coupling of reconfigurable units (often in the form of FPGAs) to general-purpose processors. The performance advantage of reconfigurable hardware

devices such as Field Programmable Gate Arrays (FPGAs) now rivals that of custom ASICs but with the added run-time design flexibility not available in custom hardware. The role of FPGAs and reconfigurable computing in the present and near future include improving the performance of many scientific and signal processing applications [70, 71]. 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 potentially provide the functionality of several ASIC co-processors. Several research groups have demonstrated successful RC architectures [30, 47, 63, 64, 72, 73, 11, 80, 87, 93, 94, 126, 131].

There are several RC options available from companies such as Annapolis Microsystems [5], Nallatech [13], Virtual Computer Corporation (VCC) [20], and research organizations such as University of Southern California's Information Sciences Institute (ISI) [11], The Chinese University of Hong Kong [87], and Carnegie Mellon University [64, 94]. The Wildforce [5] and Firebird [5] units from Annapolis Microsystems are both PCI-bus cards with onboard memory. The BenNUEY RC system from Nallatech [13], the H.O.T. I and H.O.T. II systems from VCC [20], and the SLAAC units from ISI [11] are all PCI-bus cards. The Pilchard architecture developed by The Chinese University of Hong Kong [87] interfaces through the memory bus and is more closely coupled to the processor. The PipeRench reconfigurable fabric from Carnegie Mellon [64] is an interconnected network of configurable logic and storage elements which uses pipeline reconfiguration to reduce overhead.

Another research area in which the use of reconfigurable devices is becoming popular is Systems on a Chip (SoC). Known as SoPC (Systems on a Programmable Chip), Xilinx [128],

Altera [4], Atmel [6] 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.

RC Hardware. For testing and validation of the HPRC model, we will use three RC coprocessors: Pilchard [87], Annapolis Microsystems' Wildforce and Firebird [5].

The Pilchard architecture (Figure 2.1) consists of a Xilinx Virtex 1000E FPGA interfaced to the processor via the SDRAM DIMM slot [87]. 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 and as such requires a separate host program to configure the FPGA. The Pilchard architecture addresses the bandwidth bottleneck of the PCI bus between the RC unit and the processor by placing the RC unit on the memory bus. However, this interface may be limited since it is less flexible than a PCI bus.

The Wildforce board from Annapolis Micro Systems [5] is a PCI-bus card, which contains four Xilinx XC4013XL 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 programmable elements (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 2.2**.**

The Firebird board shown in Figure 2.3 consists of a single Xilinx Virtex 1000E FPGA. The board is PCI based and includes an onboard PCI controller so that valuable FPGA resources

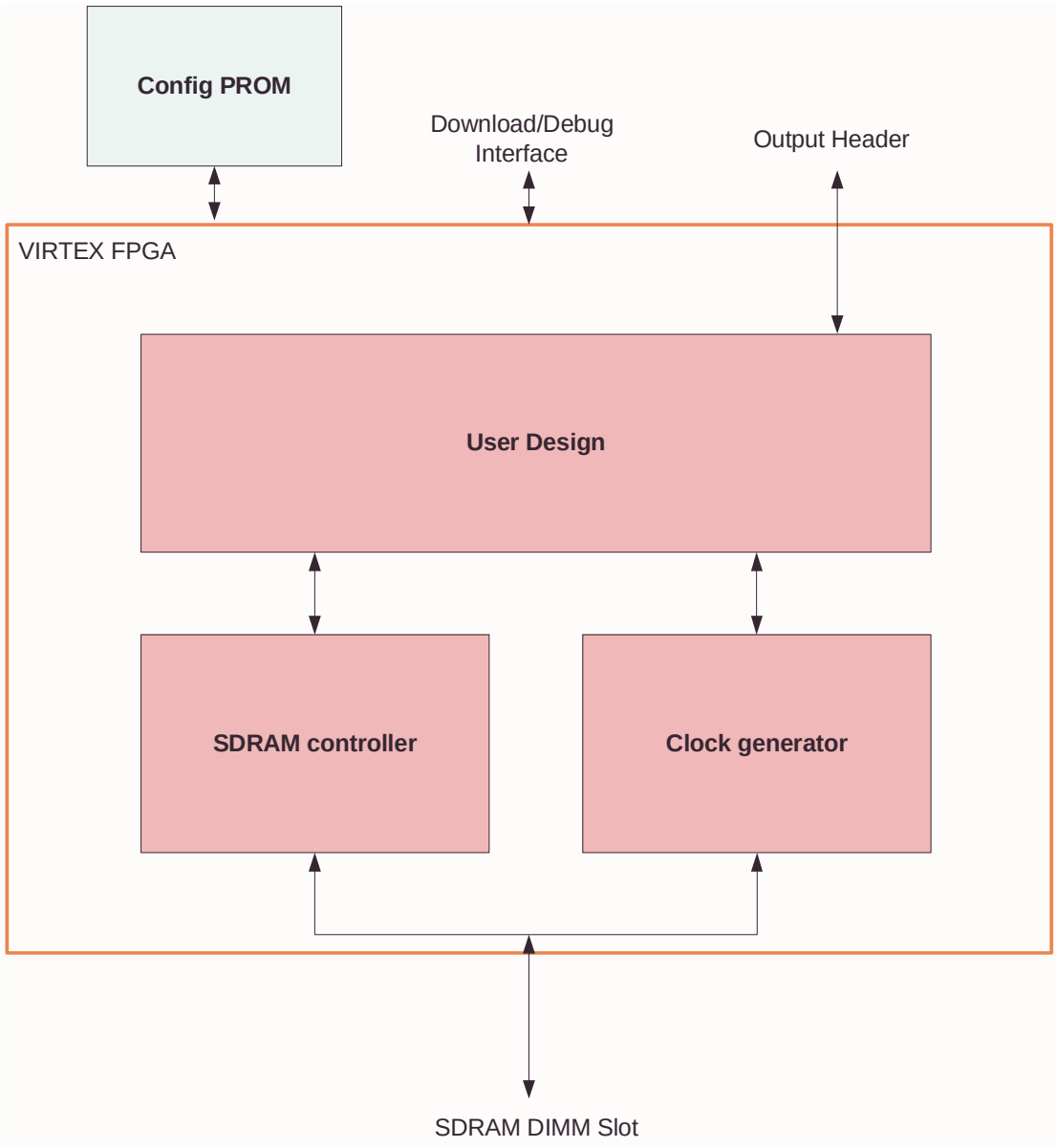


FIGURE 2.1 Block Diagram of the Pilchard Board

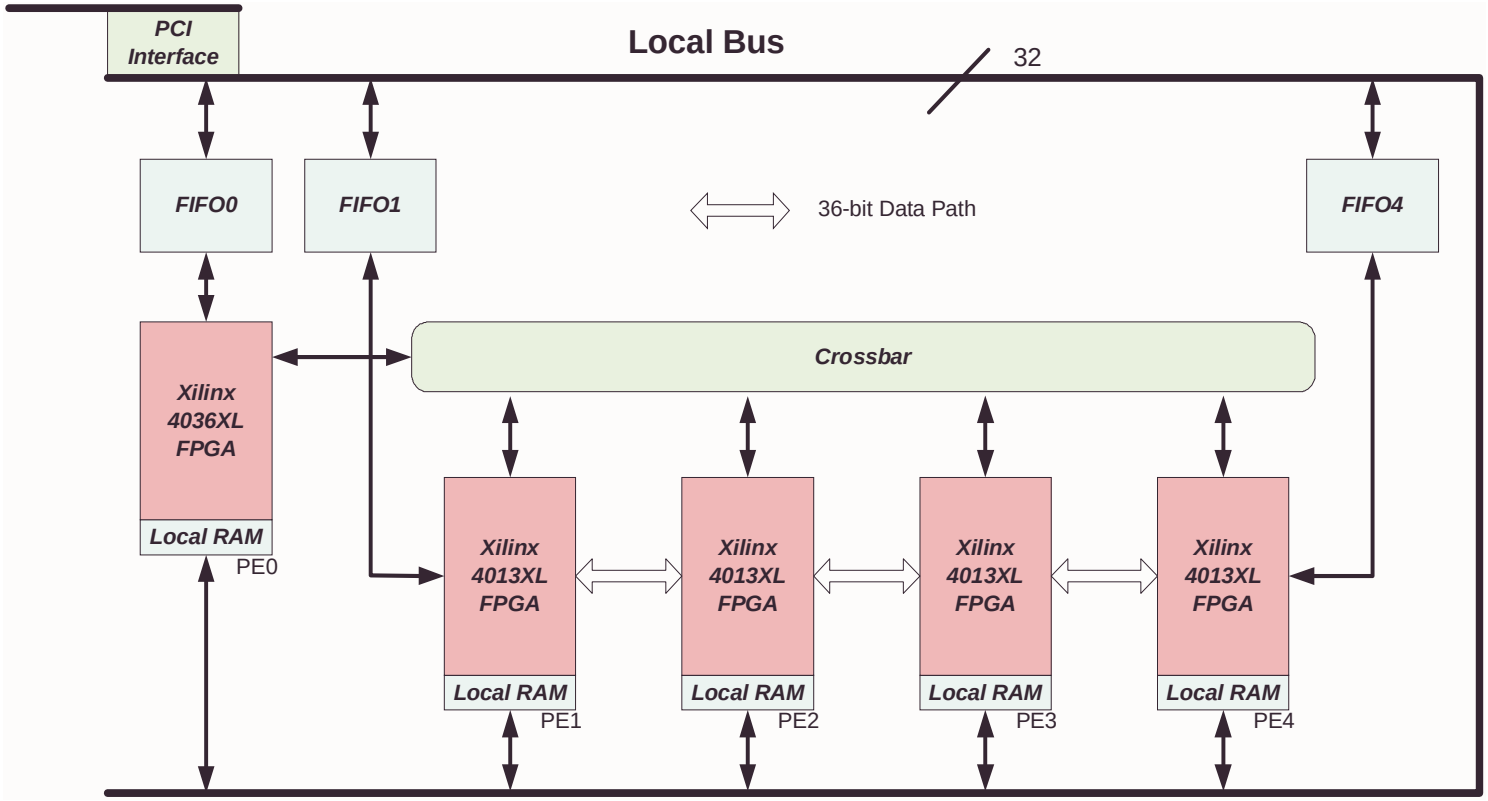


FIGURE 2.2 Wildforce Architecture [5]

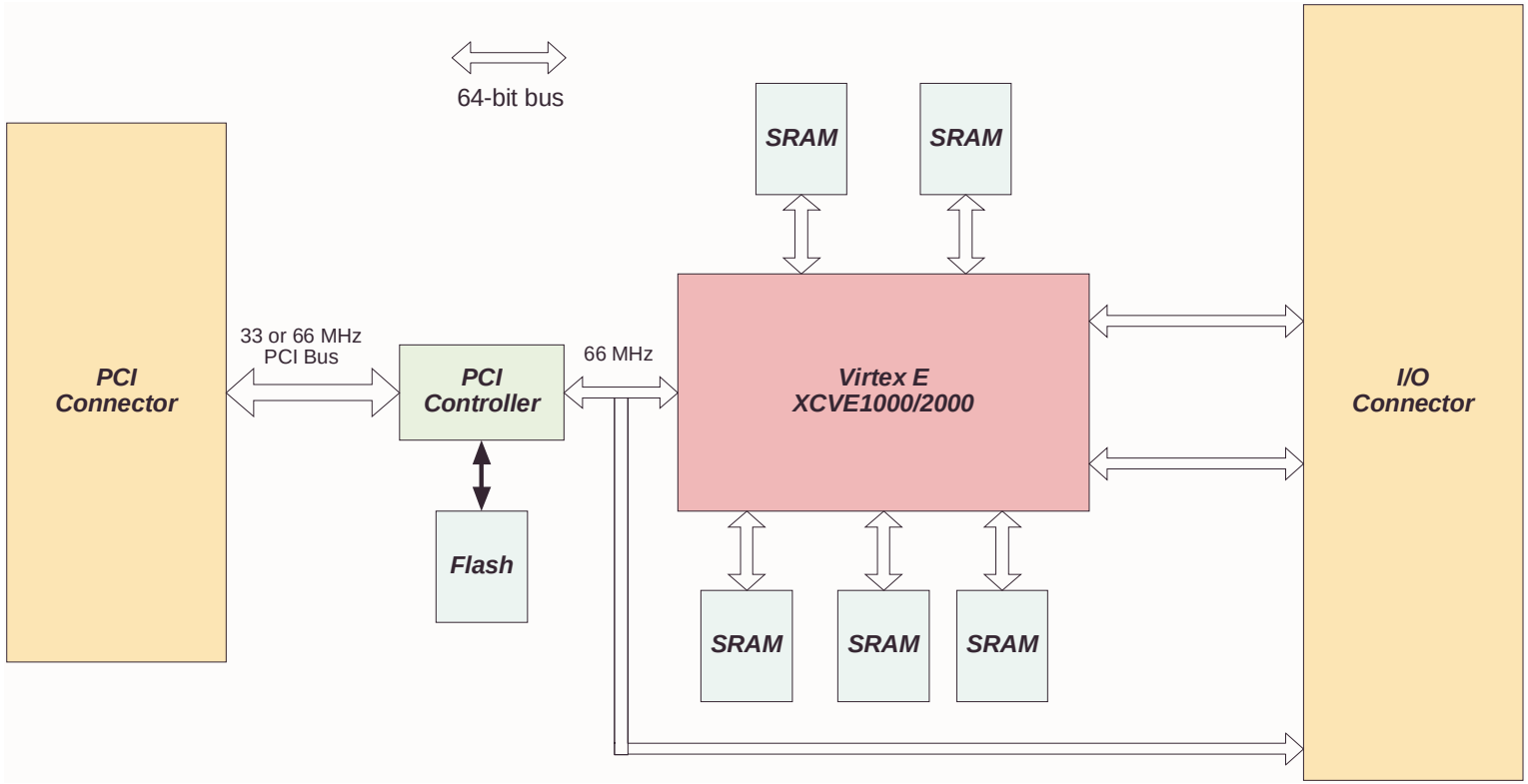


FIGURE 2.3 Firebird Architecture [5]

are not used for PCI functions. The device is runtime reconfigurable through the host interface 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.

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 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 consider. 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 with the use of a modeling framework will be integral in exploiting this potentially powerful architecture.

HPRC Hardware. Two HPRC clusters were available for developing and validating the model. The Air Force Research Laboratory in Rome, NY has assembled a two chassis, 48 node Heterogeneous HPC from HPTi, which marries the benefits of Beowolf cluster computing with the recon-

TABLE 2.2 HPRC Development Platforms

Platform	Number of Nodes	Number of FPGAs/ Node	FPGA Type	Gates per Node	I/O Technology	Type of RC Card
Wildforce	1	5	XC4013 (4) XC4036 (1)	88K	PCI Bus	Wildforce
Firebird	1	1	XCVE1000	1M	PCI Bus	Firebird
Heterogeneous HPC	48	2	XC2V6000	12M	PCI Bus	Firebird
Pilchard Cluster	8	1	XCVE1000	1M	Memory Bus	Pilchard

figurability of FPGAs. Each node harbors an Annapolis Micro Systems [5] Wildstar II which is populated with two Xilinx XC2V6000 FPGAs. The system is designed for a throughput of 845 GFLOPS (SP) or 422 GFLOPS (DP), from the Beowolf cluster, and 34 FIR TeraOPS from the FPGAs [3]. The HPRC cluster at UT consists of eight Pentium nodes populated with Pilchard boards. Listed in Table 2.2 are the HPRC platforms used to develop our modeling framework.

We will work with a MIMD, distributed memory, Beowolf cluster of eight Pentium nodes populated with one Pilchard reconfigurable card per node. Each of the Pilchard cards has on-board FPGA memory and no dedicated network between RC units.

2.3 Performance Evaluation, Analysis and Modeling

2.3.1 Overview

Development of a model for studying system performance requires selection of a modeling tool or technique and definition of metrics for evaluation. The metrics selected will depend on the architecture features and the issues of interest. Other issues of interest include effective

resource management and usage cost determination. We will now look at some performance evaluation techniques followed by some of the performance modeling research on fork-join and SIAs found in the literature.

2.3.2 Performance Evaluation Techniques

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. There are three broad classes of performance evaluation techniques: *measurement*, *simulation*, and *analytical models* [74]. Each technique has variations and selection of the most suitable alternative is often difficult. Issues to consider are the desired result or application of the model, the accuracy needed, the development stage of the computer system, and the model development cost. With these issues in mind we now look at the three techniques and determine the best approach for our needs.

Measurement. The technique of measurement when used for performance evaluation is based on the direct measurements of the system under study using software and/or hardware monitors. This technique provides the most accurate representation but requires that the system be available. Measurement techniques are very often used in *performance tuning* whereby the information gathered can be used later to improve performance [74]. For example, frequently used segments of the software can be optimized with performance tuning thereby improving the performance of the whole program. Similarly, the resource utilizations of the system can also be obtained and performance bottlenecks identified. Other common uses of measurements is gathering data for other models (parameterization and calibration), characterizing workloads, or validating a system model [74]. Since there are unavoidable disturbances to the system during measurements such as loading and other overhead caused by the probes or monitors, the data collected must be analyzed and

scrutinized 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 to avoid corrupting results with invalid stimuli [81]. 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 performance perturbations from probes and monitoring devices.

The use of measurement as a performance evaluation tool has a number of drawbacks. First, measurements must be meaningful, accurate, and within the capabilities of available monitors. Second, monitors and probes are characteristically intrusive and can perturb system performance resulting in corrupted data. Finally, the real system must be implemented and available which does not allow for prediction of system performance and/or analysis of different system configurations.

Simulation Models. Simulation involves constructing a model for the system's behavior and driving it with an abstracted workload or trace data. It is often used to predict the performance of a system or to validate other analytical models and it is not necessary that the system exist enabling the examination of a larger set of options than with measurement techniques alone. The major advantages of simulation are generality, visibility, controllability, and flexibility [81] and the collection of data does not modify the system behavior as in measurement. However, like measurement, simulation modeling has some disadvantages. Simulation modeling also requires careful attention to the experiment design, data gathering process, and subsequent data analysis since the end results will only be as accurate as the model. 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. As the detail and complexity of the model increases, typically so do the model runtime and

development costs. Again like measurement, the amount of data can be enormous and statistical methods must be used to analyze the results.

One of the major drawbacks of simulation is the performance; on sequential machines, large simulations can take enormous amounts of time and the simulation runs must be long enough that startup transient effects do not impact results. Other problems with simulations are difficulty in model validation and balancing the model's fidelity. Completely validating simulation models is impractical since all possibilities cannot conceivably be tested. Additionally, highly detailed models may reflect microscopic behavior while at the expense of the ability to examine the macroscopic behavior. High fidelity models also require more coding and debugging time. Another drawback of simulation is the inability to draw general conclusions about the system's performance from a single simulation since multiple simulations are required to understand sensitivity to parameters. Despite the limitations, simulation provides valuable information in cases where measurement is restricted by physical constraints or analytical modeling is limited by mathematical intractability.

Analytic Models. Analytic models are widely used in performance evaluation due to their power and flexibility [81]. Analytic modeling involves constructing a mathematical model at the desired level of detail of the system and solving it [81]. The main advantage of analytic models is that they allow for exploration of system performance when it is impractical to build a system prior to construction. 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, (c) better predictive value from results than those obtained from measurement and simulation and (d) insight into perfor-

mance variations with individual parameters. Hence, analytic models can often be used to optimize a system with respect to a set of parameters such as the number of processors or work load distribution.

Analytic models also have their disadvantages. Models must be evaluated for correctness against the real system and any simplifying assumptions made during analysis to maintain tractability must be carefully validated. Even when an accurate model cannot be developed due to tractability or other limitations, analytic models are often useful for determining performance trends or for comparing the performance of different algorithms.

One of the classic analytical modeling techniques is queueing models [29, 85]. 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) [81]. 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 [74]: 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 are no buffer space or population size limitations and the queueing discipline is FCFS (First Come First Serve), 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 [74].

Several researchers have explored the use of queuing networks [29], petri net models [108], and markov models [29] in the performance evaluation and analysis of HPC systems. Mohapatra et.al. [91, 92] use queuing models to study the performance of cluster-based multiprocessors with multistage interconnection networks. The performance model developed is for a shared-memory cluster of multiprocessors. A queueing network model is developed for the complete system using hierarchical decomposition resulting in a two-level model. Significant changes to the model would be required to represent our distributed memory system.

We will use analytic modeling in our model development and employ queueing theory in the analysis and development of the background load portion of the model.

2.3.3 Performance Modeling

The performance of algorithms are often described by their asymptotic behavior as the problem size varies. With this approach, one must be careful to provide an accurate representation of the runtime performance. Often, scale factors and lower-order polynomial terms can have a dramatic performance impact, but are not reflected in asymptotic models [102]. We will focus our attention on *fork-join* types of algorithms and more specifically *synchronous iterative* (or *multiphase*) algorithms. In the remainder of this section we review some performance results for this class of problems which include optimization techniques, simulation, and many numerical methods.

The performance of synchronous iterative algorithms is dramatically impacted by the random effects in the per iteration runtime of each processor. Differences in processor runtime caused by load imbalances affect the completion time of the algorithm and degrade the overall performance. Modeling these affects analytically is difficult due to the mathematical complexity [102]. Dubois and Briggs describe the performance of these algorithms on shared-memory multiprocessors [54]. To model load imbalance, they represent the amount of work that each processor completes by a sampled random variable and apply *order statistics* [44] to describe the expected runtime of the last processor to complete and thus the overall runtime of the algorithm. Govindan and Franklin address the dynamic load imbalance across iterations of a synchronous iterative algorithm [60, 65]. Their work differs from other models in that they do not assume that the task distribution at any iteration is independent of the distribution at previous iterations.

For synchronous discrete-event simulation, Chamberlain and Franklin develop a performance model that predicts the execution time assuming a known load imbalance scale factor [37]. Peterson and Chamberlain validate this model for simulation of queueing networks and investigate the impact of load imbalance [103, 104, 105, 102]. Agrawal and Chakradhar use Bernoulli trials to determine the number of events at each processor allowing for an analytic solution of the application load imbalance of the performance model [21]. Accurate performance models exist for these algorithms running on *dedicated processors* where the load imbalance results entirely from the uneven distribution of the application workload. Modeling load imbalance on *shared machines* requires the use of order statistics and other techniques to model the randomness of the load imbalance due to other users.

For shared resources, the randomness of the load imbalance complicates performance evaluation. Two factors contribute to the load imbalance: *application load imbalance* and *back-*

ground load imbalance. Application load imbalance is the result of an unequal distribution of the workload among processors while background load imbalance is the result of computing resources being shared with other tasks. Much of the work regarding load imbalance modeling for dedicated resources can be applied to the application load imbalance modeling for shared resources.

Several groups have studied the performance of distributed and parallel systems of various architectures and focused on the performance impact of background load imbalance. Atallah et.al. developed a performance model for compute-intensive tasks running simultaneously on a shared network of workstations [23]. They include the performance impact of other users of the shared resources by the "duty cycle" η_i. The duty cycle is defined as the ratio of clock cycles that a workstation *i* commits to local or other user tasks to the number of clock cycles available to the distributed application. They do not consider application load imbalance or discuss how to find the duty cycle values for each workstation. They also use this model to develop algorithms to optimize the scheduling problem for shared resources. Efe and Schaar extended this work to find the optimal mean response time for executing multiple batch jobs on the networked workstations [55]. In [56], Efe optimizes the scheduling algorithms to reduce their runtime for SPMD (Single Program Multiple Data) applications.

Peterson and Chamberlain [102, 103, 104, 105, 106] study application performance in a shared, heterogeneous environment. Their analytic performance model focuses on synchronous iterative algorithms and includes the effects of application and background load imbalance. Issues unaccounted for include network communication and contention models. Their work however provides a thorough investigation of performance evaluation and specifically the background and application load models used in this dissertation.

Clement and Quinn [41] developed an analytical performance model for multicomputers however the assumptions and restrictions limit this model to specific architectures and applications. Another paper by Clement [42] focuses on the network performance of PVM (Parallel Virtual Machine, a discussion of PVM is included later in this chapter) clusters. A communication model for ATM and Ethernet networks is developed 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 [34, 51, 53]. Nupairoj and Ni have studied the performance of MPI on some workstation clusters [99]. Zhang and Yan [130, 132] have developed a performance model for non-dedicated heterogeneous Networks of Workstations (NOWs) where heterogeneity is quantified by a "computing power weight", owner workload requests are included as binomial distributions, and the network is limited to Ethernet.

We shall combine the results of the research on background load imbalance modeling on shared resources, application load imbalance modeling on dedicated resources, and communication modeling for shared resources to model distributed applications running on shared resources. Our contribution will be to modify these models for use in HPRC and more specifically, account for the new contributions of the RC elements.

2.4 Performance Metrics

HPC performance is commonly measured in terms of speedup and efficiency. *Amdahl's Law* [22] for "fixed-size" speedup and *Gustafson's Law* [67] for "fixed time" speedup are common representations for the limitations to parallel efficiency.

Metrics for reconfigurable computing are limited in the literature. In Dehon's thesis on reconfigurable architectures, he presents a high-level characterization of reconfigurable systems using size metrics [47, 48] to project the performance of a reconfigurable system.

Dehon's characterization model uses size estimates to compare the efficiency of different architectures [47]. From an empirical survey, he concludes that reconfigurable devices have lower performance than dedicated hardware but higher than general purpose processors. He also 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 2.4).

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 2.5 shows the area trade-offs for Dehon's Reconfigurable Processing or RP-space. For conventional FPGAs, multi-context FPGAs and general purpose processors, Figure 2.5 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.

Dehon's RP-space provides a method for characterizing reconfigurable devices or FPGAs and even includes some abstraction of processors onto the RP-space. RC architectures however

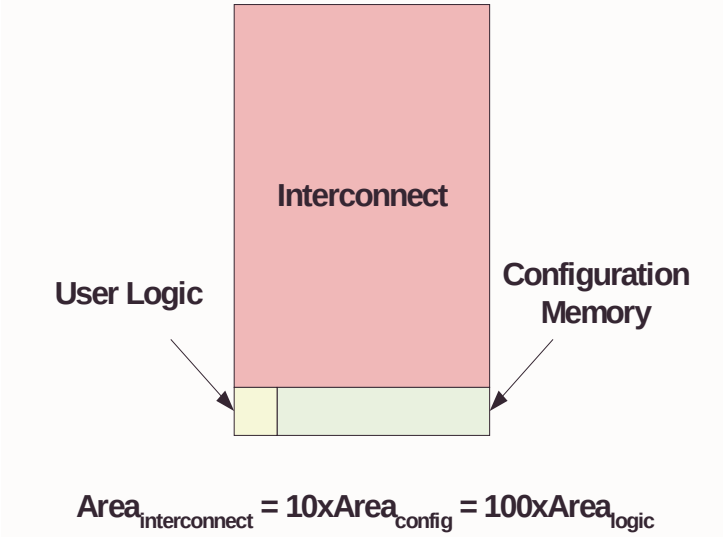
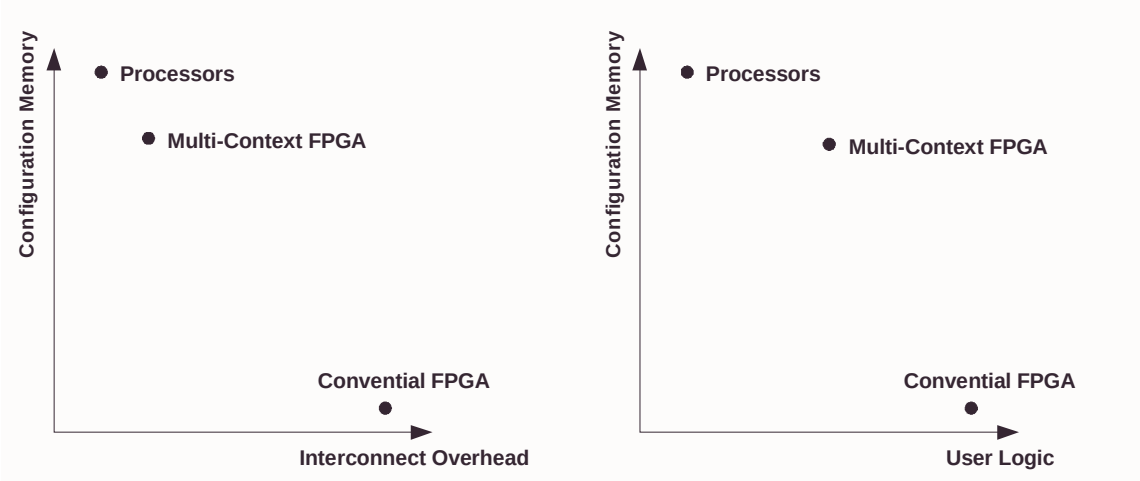


FIGURE 2.4 Area Density for Conventional Reconfigurable Devices [47]



(a) **(b)**

FIGURE 2.5 RP-Space (a) Interconnect vs. Configuration and (b) Logic vs. Configuration

consist of FPGA(s) and a processor working together making their projection into the RP-space somewhat difficult because we must consider how the RC system is constructed. We have to account for the FPGA(s) (or reconfigurable unit(s)), the workstation (or microprocessor), the communication interface between them, and any external memories connected to the reconfigurable unit(s). With the plethora of RC architectures available, each architecture would effectively map differently into the RP-space. To effectively compare trade-offs 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 model proposed in this dissertation to the cost of the system. Similarly a power metric cost function can be developed relating execution time determined from the proposed model to the total power used by the system. The metrics of interest will depend on the issue to be optimized (runtime, system utilization, system cost, etc.). We will touch on this analysis more in Chap. 6.

2.5 Resource Allocation, Scheduling, and Load Balancing

To effectively exploit computational resources, access to these resources must be somehow managed. Operating systems are normally tasked with the management and allocation of resources among competing service requests. The allocation policy must also preserve fairness among users allowing resource access to all, prevent deadlock to ensure work can be performed and maintain security [120]. Handling these service requests for distributed applications on networks of workstations can be significantly more complicated than managing a single workstation.

For parallel or distributed applications, it is necessary to partition the work into subproblems and map them onto a set of processors. Data parallelism is often exploited to create subproblems that perform computations with minimum interaction between subproblems. These subproblems are then mapped onto a set of processors where the goal is to minimize the amount of communication between processes and maximize processor utilization [110]. Mapping subproblems onto a distributed system can be viewed as a restricted case of the general scheduling problem. The parallel scheduling problem is NP-Hard [61], so finding optimal scheduling solutions for general problems is not feasible. Research has as a result focused on techniques for restricted cases or near-optimal solutions. These techniques are referred to as *static* if the scheduling decisions are predetermined, or *dynamic* if the scheduling decisions are made at execution time [36].

Although parallel scheduling is known to be difficult, load balancing techniques have been used to improved the performance of parallel applications [33]. Many researchers have investigated both static and dynamic load balancing approaches. Wang and Morris propose a simple load distribution classification method based on whether the load distribution is source or receiver initiated and the level of information dependency [127]. Information dependency refers to the level at which a resource has information about the current state or workload of other resources. Since most current distribution techniques are sender initiated, the coarseness of this classification method provides little distinction between members of this class and it is not extensive enough. Casavant and Kuhl provide a pivotal taxonomy that includes both local and global scheduling of load balancing algorithms [36]. Their classification, designed for distributed computing systems, is based on strategic design choices such as static or non-static and distributed versus non-distributed. The complete taxonomy consists of a hierarchical and flat classification. The hierarchical classification is used to show where some characteristics are exclusive and the flat classification gives definitions of attributes that are not exclusive. Again this classification method does not pro-

vide a detailed comparison of algorithms within each class. Baumgartner and Wah attempt to address both deterministic and stochastic problems with a classification scheme based on three categories of input components to the scheduling problem: *Events*, *Environment* (or surroundings), and *Requirements* [26]. The ESR classification is high level, meaning that attributes can be specified with varying degrees of completeness. This is helpful in the case of stochastic scheduling problems since unlike deterministic cases, they are not enumerable; it is impossible to list every problem. The scheme is limited in scope for describing the granularity of the load distribution algorithm under R (requirements) [31]. Jacqmot and Milgrom took a different approach and used the order in which load distribution decisions were made to classify the load distribution [76]. It fails however to distinguish between initial placement and process migration and also cannot classify symmetrically initiated policies. Bubendorfer and Hine improve on Casavant and Kuhl's work by providing clear separation of policy and mechanism and providing a basis for the comparison of the major components of a load distribution system [31]. Their approach to classification is based on three policy decisions (participation, location, and candidate selection) and three mechanism choices (transfer, load metric, and load communication). With so many approaches available, Kremien and Kramer developed a useful framework for comparing methods quantitatively based on simulation or empirical measurements [83]. Feitelson and Rudolph also developed metrics and benchmarks for comparing parallel job schedulers using a standard workload [58].

For our studies we will focus primarily on static load balancing. Static approaches typically use graphic theoretic techniques, queueing theoretic techniques, state-space searching, or heuristics [102]. The graph theoretic techniques exploit the regularity of some applications to form optimal schedules. Bokhari considers problems with regular topologies such as trees, chains, and pipelines and has found an efficient algorithm for finding the optimal assignment for these

restricted cases [28]. Chou and Abraham consider task graphs that fork and join probabilistically [40].

Others have studied the use of queueing networks and similar models for finding the best static assignment. De Souza e Silva and Gerla consider scheduling for distributed systems that can be described by queueing models with product form solutions [49]. Tantawi and Towsley statically schedule tasks on heterogeneous machines to minimize the mean response time and formulate the optimal schedule as a nonlinear optimization derived from a queueing model [121].

Other static scheduling techniques include state searching and heuristics. Shen and Tsai find weak homomorphisms which map from the problem topology to the processor topology by searching using branch and bound techniques [114]. Ma et al. develop a cost function for assigning tasks to processors and the resultant communications costs. They use 0 - 1 integer programming to minimize the cost function although it lacks computational efficiency [90].

Leland and Ott suggest some simple heuristics for two load balancing schemes based on the behavior of Unix processes: initial placement and process migration [86]. They note errors in the exponential load distribution model for the CPU requirements of individual processes. They observed that there are many short jobs and a few long jobs, and the variance of the distribution is greater than that of an exponential distribution. They conclude that with a sufficiently intelligent local scheduling policy, dynamic assignment will significantly improve response times for the most demanding processes without adversely affecting the other processes. Harchol-Balter and Downey extend the work of Leland and Ott using the distribution model to derive a policy for pre-emptive migration where running processes may be suspended, moved to a remote host, and restarted [69]. The motivation is to reduce the average completion time of processes and improve

the utilization of all the processors. Included are empirical observations about the workload on a network of UNIX workstations including distribution of process lifetimes and arrival rates.

In Chap. 6 we will use the results of our modeling methodology to improve the effectiveness of a static scheduler.

2.6 Development Environment

2.6.1 HPC Development Environment and Available Software Tools

The task of programming distributed architectures is complicated and time-consuming. A popular approach to using networked resources is to use a message passing environment to provide the software infrastructure across the resources. A number of environments and libraries such as MPI [118], PVM [62], BLAS [7], and VSIPL [19], allow the user to program distributed or networked resources easier. Users can 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 practical and cost effective and thus amenable to more users.

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 *parallel virtual machine* [62], was developed by a research group to work on networks of workstations. In contrast, MPI was developed by a forum of users, developers, and manufacturers as a standard message passing specification [118]. Just the process by which they came about implies some of the differences between the two Application Program-

ming Interfaces (APIs). The roots of PVM being in the research committee influenced incremental development, backward compatibility, and fault tolerance. The goals of the PVM developers also leaned more toward portability and interoperability by 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 which is unavailable in PVM. This enables the user to exploit architectural advantages that map to the communication needs of the application. Additionally the communication set for MPI uses native communication functions to provide the best possible performance. PVM's developers chose a more flexible approach allowing communication between portable applications compiled and executed on different architectures.

Both MPI and PVM are portable but only PVM is truly interoperable between different hosts and is traditionally better for applications running on heterogeneous networks. Both PVM and MPI applications can be compiled and executed on different architectures without modification, however, only in PVM can the resulting executables also communicate with each other across these architectural boundaries. For local or identical hosts, MPI and PVM both use native communication functions. For heterogeneous architectures on the other hand, PVM uses standard network communication functions. PVM is also language interoperable meaning programs written in C and FORTRAN can communicate and interoperate. This 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. For our applications, we will use MPI.

2.6.2 RC Development Environment and Available Software Tools

Research in the architecture, configuration, and use of RC systems is ongoing. To date, most RC research has focused on single computing nodes with one or more RC elements with a few exceptions the Adaptable Computing Cluster at Clemson [2], the System Level Applications of Adaptive Computing (SLAAC) [11] project at USC, the Configurable Computing Lab at Virginia Tech [10], the Heterogeneous HPC at AFRL/IF [3], High-performance Computing and Simulations Research Lab RC Group at the University of Florida [18], and the High Performance Reconfigurable Computing research at University of Tennessee [116, 117]. 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. Improvements to both of these areas will open the market on RC systems.

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, Handel-C for hardware design, VHDL and Verilog tools, JHDL and JBits, and MATLAB and MATCH. 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 encouraging design reuse and hierarchical implementation. When using these tools, the designer must be cognizant of

potential hardware conflicts, partitioning, and scheduling since the tools do not provide that level of support.

Several groups have developed compiler environments for RC systems. The Nimble compiler at Berkeley [89] is a C-based environment. Handel-C by Celoxica is a C-based language for describing functionality with symbolic debugging and libraries of predefined functions [9]. Handel-C is a subset of ANSI-C with the necessary constructs added for hardware design. The level of design abstraction is above RTL but below behavioral (VHDL). DEFACTO [30] uses the Stanford SUIF compiler system [68, 119] and similar work at Virginia Tech [79, 80] uses the BYU JHDL design environment [8, 27]. 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 [66, 129]. 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 CHAM-

PION research at the University of Tennessee uses the visual programming language Khoros to assist the designer in RC hardware programming [88, 95, 96, 101, 115]. 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 [15] 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 [77, 78]. They extended this work into MATCH: A MATLAB Compilation Environment for Distributed Heterogeneous Adaptive Computing Systems [1, 12]. MATCH is a mechanism for parsing MATLAB programs into intermediate representations, building a data and control dependence graph. It automatically identifies areas for parallelism and maps operations to multiple FPGAs, embedded processors and DSP processors.

Developing applications for the proposed HPRC platform will require learning from these research efforts and extending their work to support the needs of a parallel, heterogeneous RC platform.

2.6.3 HPRC Development Environment and Available Software Tools

Currently there are no tools available which are completely suitable for the HPRC environment. The work by Banerjee and others at Northwestern [24] on a MATLAB compiler for distributed, heterogeneous, reconfigurable computing systems begins to address the issues of HPRC development tools. The MATLAB compiler, also known as MATCH (MATlab Compiler for distributed Heterogeneous computing systems), takes MATLAB descriptions of embedded systems applications and automatically maps them onto a heterogeneous computing environment. The computing environment they have addressed consists of FPGAs, embedded processors, and DSPs

connected via a VME backplane. The elements of the computing environment are commercial off-the-shelf (COTS) components, each plugged directly into the VME backplane which serves as the interconnection network. This differs from our HPRC platform where the reconfigurable elements are part of workstations which are then connected in a cluster by some arbitrary network. With the HPRC platform, each element of the computing environment poses a software/hardware design problem that must be addressed by the development tool.

To address the lack of a toolset for HPRC, one approach is to take existing tools from both HPC and RC (such as MATCH) 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 demonstration applications described in a later section, we apply the modeling results in our manual partitioning, scheduling, and mapping onto the HPRC hardware. These are 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.

We have discussed the makeup of the HPRC platform and reviewed the literature for performance evaluation techniques. The HPC research has proven that by using performance evaluation techniques, parallel applications can be optimized and system bottlenecks identified. We reviewed performance modeling results for both shared and dedicated systems and reviewed some performance metrics and how they might be applied to HPRC. We also investigated the issue of scheduling and resource management noting that by applying performance evaluation techniques to these problems we can achieve better optimization and exploitation of the idle workstations.

Finally we review the development environment including software tools for RC systems. The next chapter discusses the parallel applications that will be used to verify the model.

CHAPTER 3

PARALLEL APPLICATIONS

3.1 Introduction

To validate the modeling methodology developed later, representative test applications will be needed. The fork-join class encompasses many important scientific and computing algorithms that are potentially well suited for the HPRC platform. Included within this class is a subset of algorithms known as Synchronous Iterative Algorithms (SIAs) as discussed earlier in Chap. 1. SIAs include many interesting scientific and computationally intensive algorithms such as optimization, discrete-event simulation, solutions to partial differential equations, gaussian elimination and matrix operations, FFTs, boolean satisfiability, and many others. As discussed earlier, SIAs repeatedly execute a computation and exchange data at the end of each computation or iteration via synchronization of the tasks. Each processor reaches a barrier synchronization after every iteration and awaits the arrival of the other processors before continuing. The timing of an SIA running on four processors is shown in Figure 3.1. As shown, each processor performs computations

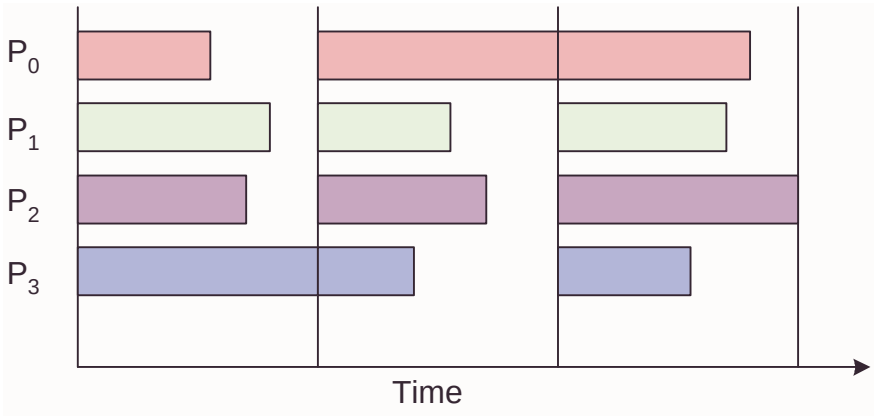


FIGURE 3.1 Synchronous Iterative Algorithm running on 4 processors

for the amount of time represented by the horizontal bars. The time required for the last processor to reach the synchronization point determines the time for that iteration. After the barrier synchronization, the processors repeat the cycle.

In this chapter, we investigate the characteristics of three applications we will use to validate the modeling methodology.

3.2 Boolean Satisfiability

The *Boolean satisfiability problem* (SAT) is a fundamental problem in mathematical logic and computing theory with many practical applications in areas such as computer-aided design of digital systems, automated reasoning, and machine vision. In computer-aided design, tools for synthesis, optimization, verification, timing analysis, and test pattern generation use variants of SAT solvers as core algorithms. Given a boolean formula, the goal is either to find an assignment of 0 and 1 values to the variables such that the formula evaluates to 1, or to establish that no such assignment exists. The SAT problem is commonly defined as follows [109]: Given

- a set of n Boolean variables $x1, x2, ... , xn$,
- a set of literals, where a literal is a variable xi or the complement of a variable $\bar{xi}$, and
- a set of m distinctive clauses $C1, C2, ... , Cm$, where each clause consists of literals combined by the logical *or* connective,

determine, whether there exists an assignment of truth values to the variables that makes the Conjunctive Normal Form (CNF)

$$C_1 \wedge C_2 \wedge .... \wedge C_m \qquad \text{(EQ 3.1)}$$

true, where $\wedge$ denotes the logical *and* connective.

The boolean formula is typically expressed in *Conjunctive Normal Form* (CNF), also called the product-of-sums form. Each sum term, or *clause*, in the CNF is a sum of single *literals*, where a literal is a variable or its negations. In order for the entire formula to evaluate to 1, each clause must be satisfied (i.e., at least one of its literals should be 1).

Since the general SAT problem is NP-complete, exact methods to solve SAT problems show an exponential worst-case runtime complexity. This limits the applicability of exact SAT solvers in many areas. Heuristics can be used to find solutions faster, but they may fail to prove satisfiability.

A straight-forward approach to solving the SAT problem exactly is to enumerate all possible truth value assignments and check if one satisfies the CNF. The solution time of such an approach grows exponentially with the size of the problem. Improved techniques based on heuristics have been developed to shorten the solution time and some have been implemented in RC hardware [133, 134]. The two basic methods are *splitting* and *resolution*. Resolution was implemented in the original Davis-Putnam (DP) algorithm [45]. Splitting was used first in Loveland's modification to DP, the DPL algorithm [46].

The goal in using the SAT problem in the validation of the HPRC model is not to develop the most efficient SAT solver, but to validate and demonstrate the use of the HPRC performance modeling methodology. Hence, the straight-forward approach will be employed for the HPRC implementation. Utilizing this approach offers the advantage of predictable problem complexity during the model validations as well as an easily parallelized solution method.

The object of solving SAT problems on the HPRC platform will be to speed up the exact SAT solver by exploiting parallelism. Since a new hardware implementation is needed for each

new problem instance to reflect the particular structure of the CNF, the architecture is considered *instance-specific*. The advantage of instance-specific SAT solvers in hardware is that the deduction step can be implemented very fast by exploiting fine-grained parallelism. The disadvantage of instance-specific hardware is that reconfigurable computing machines still require relatively long compilation times resulting in advantages for only the hard SAT problems where software algorithms show a long runtime.

3.2.1 Boolean Satisfiability Implementation

The HPRC SAT solver methodology will be a straight forward evaluation of all possible variable combinations. The basic SAT solver core as shown in Figure 3.2, will consist of the hardware implementation of the CNF boolean formula and an $(v\text{-}2^m\text{-}2^n)$-bit counter where v is the number of variables, m is the number of copies in the FPGA, and n is the number of nodes in the HPRC platform. Parallelism will be exploited at the hardware level by tiling multiple copies of the SAT solver core in the FPGA. The number of copies m, and thus the degree of hardware parallelism will depend on the complexity of the CNF problem and the size of the FPGA. The counter can be loaded with a seed value so that SAT solvers running on multiple workstations in the HPRC platform can start solving in different areas of the search space. A broadcast message is sent by the master host to all workstations to start the search algorithm. When a solution is found, a signal indicates success to the host and interrupts processing. The solver engine is stopped on that workstation and the correct solution is passed to the host who broadcasts a message to all workstations stopping the search algorithm. The counter at each workstation has a terminal count (TC) and when that value is reached, it is determined that a solution is not possible. In this case, a flag is set for the host indicating that no solution has been found. If all nodes reach terminal count, then no solution is possible.

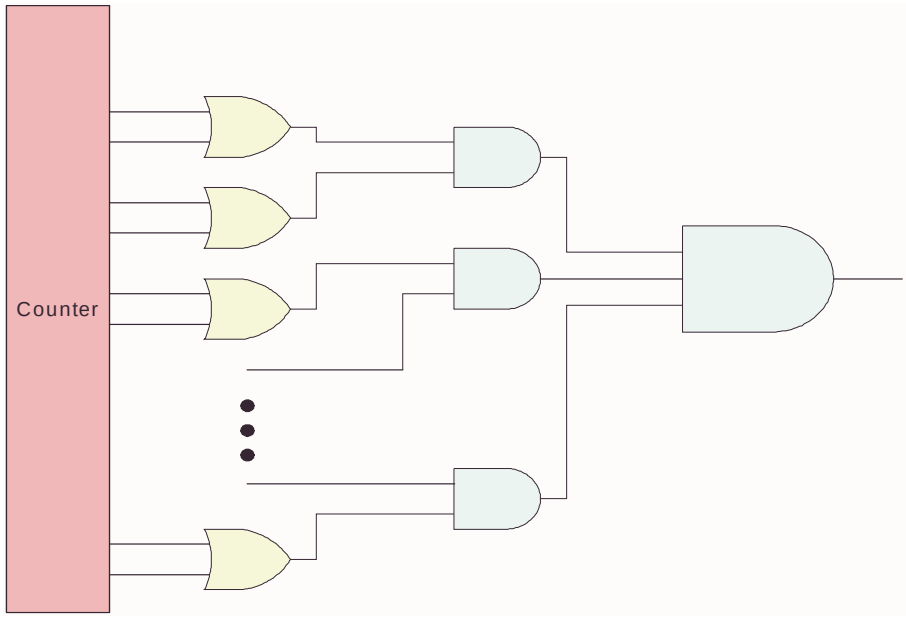


FIGURE 3.2 SAT Solver Core

The maximum search time for a problem on the HPRC platform will be:

$$t_{runtime} = 2^{v-2^m-2^n} \quad \text{(EQ 3.2)}$$

The SAT solver will be used to verify all aspects of the model independently. Search space modifications will be used to generate an application imbalance, synthetic background load will be added to simulate other users while the SAT solver is running, and various problem sizes will be implemented to test the robustness of the model over a span of runtimes. All of these variations of the SAT solver application will be discussed further in Chap. 5.

3.3 Matrix-Vector Multiplication

A simple algorithm for floating-point matrix-vector multiplication will be used to further test the model and also focus on the communication aspects of the model. The algorithm will take

an arbitrary size matrix *A* and multiply it by the vector *X* whose size equals the number of columns in *A*. The result as shown in Eqn. 3.3 below is the vector *Y*.

$$\begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} X_0 \\ X_1 \\ X_2 \end{bmatrix} = \begin{bmatrix} Y_0 \\ Y_1 \\ Y_2 \end{bmatrix} \quad \text{(EQ 3.3)}$$

The matrix-vector multiplication involves both floating-point multiplication and addition which is implemented in the hardware of the RC unit. Eqn. 3.4 shows the process of calculating Y_0.

$$A_{00} \times X_0 + A_{01} \times X_1 + A_{02} \times X_2 = Y_0 \quad \text{(EQ 3.4)}$$

3.3.1 Matrix-Vector Multiplication Implementation

The 32-bit data elements are passed in pairs to the hardware memory using the 64-bit data bus which allows a row and corresponding column element to be sent to the hardware concurrently. The hardware consists of an eight stage pipeline floating-point multiplier and a six stage pipeline floating-point adder. Details of these core designs can be found in [39]. Upon receiving the start signal, the hardware retrieves data from the memory and every six clock cycles, sends the data to the multiplier and starts the multiply. When the multiplication is complete, the result is sent to the adder for accumulation. Once the entire row-column is completed, the accumulated result is sent back to the hardware memory for retrieval by the software. The matrix-vector multiplication is performed by iteratively applying the inner-product operations on the data. For each iteration, one row of the matrix and the column vector is sent to the hardware.

The matrix vector multiplication algorithm will be used to further verify all the load imbalance conditions of the model (no-load imbalance, application imbalance, background load imbalance, and combined background and applications load imbalance). The multiplying algorithm will also be used to verify some of the basic file access and communication aspects of the model. Application imbalance will come from a matrix size which is not divisible by the number of processors available, synthetic background load will be added to simulate other users while the application is running, and various matrix sizes will be used to investigate the communication and file access portions. All of these variations of the application will be discussed further in Chap. 5.

3.4 Encryption Using AES

The Advanced Encryption Standard (AES) is a block cipher with a block size of 128 bits and key sizes of 128, 192, and 256 bits. The two inputs to the block cipher are the “plaintext block” and the “key”. The output is called the “ciphertext block”. The purpose of a block cipher is to make it as difficult as possible (even with a large number of computers) to find the key from the plaintext block and the ciphertext block. However, if one has the key, it should be easy (for a computer) to calculate the ciphertext block from the plaintext block and vice versa.

3.4.1 AES Implementation

The AES RC code accepts data in the form of test vectors which include the key, plaintext and ciphertext. The plaintext and key are written to the RC hardware and the signal is given to start the calculations. The AES algorithm is allowed to run for 10,000 iterations using the same key and taking the output from one iteration as the input for the next iteration. After 10k iterations, the ciphertext is output and compared to the expected ciphertext value. The result of the comparison is returned to the host computer and the RC hardware is ready for the next set of vectors.

Implementation in a multi-node environment consists of a central processor which orchestrates the participation of the other processors. The job of the central processor is to read in the test vectors and disseminate them to the other processors, keeping them busy by using non-blocking sends and receives. Message latency is avoided by allowing two test vectors at any given time to be queued at each processor. Once a processor receives a test vector, the plaintext and key are written to the RC hardware as described above. The result of the RC hardware computations is sent back to the central processor for tabulation and the host issues the next set of vectors to the RC hardware.

For validation of the HPRC modeling methodology, the AES algorithm will be used to further verify the no-load and background load conditions as well as the communication and file access aspects. Again a synthetic background load will be added to simulate other users while the application is running and different numbers of processors will be employed to test the communication and file access portions. All of these variations of the application will be discussed further in Chap. 5.

3.5 CHAMPION Demo Algorithms

The demo algorithms selected from the CHAMPION research [100, 101, 88] 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 [100] 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:

$$y(i,j) = \left| \sum_{m=j-1}^{j+1} \sum_{n=j-1}^{j+1} x(m,n) \cdot mask(m,n) \right| \qquad \text{(EQ 3.5)}$$

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:

$$\begin{bmatrix} -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \\ -\frac{1}{8} & 1 & -\frac{1}{8} \\ -\frac{1}{8} & -\frac{1}{8} & -\frac{1}{8} \end{bmatrix} \qquad \text{(EQ 3.6)}$$

The START (Simple, Two-criterion, Automatic Recognition of Targets) algorithm [88] 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 reconfigurations 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 reconfigurations 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 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 [88] for the details of the algorithm. The C/C++ and VHDL code generated for both algorithms during CHAMPION [100] were recycled and used for validation of the single node RC model.

In the next chapter we develop a performance modeling methodology for fork-join and SIAs running on a shared, heterogeneous HPRC platform. The applications described in this chapter are then used to validate and evaluate the performance modeling methodology for both a single node RC system and the HPRC platform.

CHAPTER 4

MODEL DEVELOPMENT

4.1 Introduction

In the last chapter we looked at some example fork-join and SIA applications. To effectively use the proposed HPRC architecture, we must be able to analyze design trade-offs and evaluate the performance of applications as they are mapped onto the architecture. In this chapter we will develop an analytic performance model for fork-join and SIA algorithms running on an HPRC platform with distributed memory. The model describes the execution time of an application in terms of its parameters (e.g., required computation within an iteration, number of iterations), architectural parameters (e.g., number of processors, processor speed, number of RC units, RC configuration time, message communication time, file access time), and background load parameters (e.g., arrival processes, service distribution).

The literature discussed in Chap. 2 describes several performance models for SIAs running on dedicated, homogeneous resources and the model developed by Peterson for SIAs on shared, heterogeneous resources [102]. We are interested in expanding these results to include RC units and better account for processor communications.

4.2 General Model Description

In fork-join algorithms and SIAs, the time required to complete an iteration is equal to the time required for the last processor to complete its tasks as discussed in Chap. 3. The join processes in fork-join algorithms or barrier synchronizations in SIAs ensure that all processors start an

iteration together and that processors who complete their tasks early sit idle until the end of the iteration.

Within an iteration, a parallel algorithm includes some serial calculations (operations that cannot be parallelized), the parallelized operations, and some additional overhead. For the applications used in this dissertation, each iteration requires roughly the same amount of computation making iterations similar enough that we consider the computations required for a “typical” iteration. This is not a necessary limitation but is used here to make the mathematics of the model less complex. References to individual iterations could be maintained at the expense of a more complex model in the end.

In the following sections we will develop a representative analytical model for the HPRC platform. 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 [102]. We will develop an analytic modeling methodology for determining the execution time of a synchronous iterative algorithm and the potential speedup. The symbols used in this chapter and their definitions are listed below in Table 4.1.

TABLE 4.1 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
ω	Time per computational element on baseline workstation		t_{work}	Total work performed on all nodes of a multi-node system
ρ_k	Time per computational element on workstation M_k		σ	Hardware acceleration factor for RC system
t_e	Time per event processed		R_1	Runtime on a single processor
m	Number of workstations		r	Number of hardware tasks not requiring new configuration
$R(A,M_k)$	Execution time for computing application A on M_k		d	Number of hardware tasks not requiring new data set
$T_{comm}(c)$	Communication time		t_{config}	Time for FPGA configuration
N_C	Total number of messages per processor		t_{data}	Time for data access
τ	Message latency		$t_{mserial}$	Host serial operations
B_i	Size of message i		$t_{nserial}$	RC node serial operations
π	Network bandwidth		t_{movhd}	Iteration overhead operations for hosts
υ	Network contention factor		t_{novhd}	Iteration overhead operations for RC nodes
S_P	Speedup		t_P	Time to complete parallel host software tasks
R_P	Runtime on parallel system		γ	Background load imbalance
R_{RC}	Runtime on RC system		β	Application Load imbalance
I	Number of iterations		η	Load imbalance factor
$t_{overhead}$	Iteration overhead operations		σ_k	Hardware acceleration factor for node k in multi-node system
t_{SW}	Time to complete software tasks		μ	Service rate
t_{HW}	Time to complete hardware tasks		λ	Arrival rate

4.3 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.

4.3.1 Workstation Relative Speed

In a network of heterogeneous resources, each processing node may have different capabilities in terms of CPU speed, memory and I/O. The relative computing power among the set of nodes 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 [130]:

$$W_k(A) = \frac{V_k(A)}{\max_{1 \le j \le m} \{V_j(A)\}}, 1 \le k \le m \qquad \text{(EQ 4.1)}$$

Where $V_k(A)$ is the speed of workstation M_k in solving application A on a dedicated system. As shown in Eqn. 4.1, 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 [130]:

$$W_k(A) = \frac{\min_{1 \le j \le m} \{R(A, M_j)\}}{R(A, M_k)}, 1 \le k \le m \quad \text{(EQ 4.2)}$$

Finally, we can also represent the ratio in terms of the execution time per computation element on workstation j, $t_{e,j}$. A computation element is defined on a per application basis and represents a single iterative task to be computed. The execution time per computation element on the baseline workstation is ω and ρ_k represents the time per computation element on the workstation of interest.

$$W_k = \frac{\min_{1 \le j \le m} t_{e,j}}{t_{e,k}} = \frac{\omega}{\rho_k}, 1 \le k \le m \quad \text{(EQ 4.3)}$$

W_k implicitly accounts for (to the first order) all factors affecting a workstation's relative speed or performance such as processor speed, I/O, and memory. 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.

4.3.2 Communication Between Processors

Communication delay between processors 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 [42]. For the total number of messages per processor, N_C, the message latency τ, network bandwidth π, and size of the message B_i, the communication time can be modeled as [42]:

$$T_{comm}(c) = \sum_{i=1}^{Nc} \left(\tau + \frac{B_i}{\pi} \right) \quad \text{(EQ 4.4)}$$

Both τ and π can be approximated from measured values. It should be noted that in practice, π may not be a constant. The model represented in Eqn. 4.4 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 communications over these types of networks, a contention factor, υ, is added to Eqn. 4.4 [42]:

$$T_{comm}(c) = \sum_{i=1}^{Nc} \left(\tau + \frac{\upsilon \cdot B_i}{\pi} \right) \quad \text{(EQ 4.5)}$$

According to [42], a contention factor of $\upsilon = m$, where m is the number of nodes, is a good approximation of an Ethernet connection assuming all nodes are communicating simultaneously. However, this assumption only holds for a limited number of nodes. At some point, as the number of nodes is increased, the throughput of the network begins to drop off and there will no longer be a linear relationship between m and υ.

4.3.3 Speedup and Efficiency as Performance Metrics in HPC

Amdahl's Law [22] for *"fixed-size" speedup* and *Gustafson's Law* [67] for *"fixed time" speedup* are useful metrics for evaluating parallel computing performance in a heterogeneous system [132]. The basic definition of 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 system. For a heterogeneous system, we define speedup as the ratio of sequential

computing time of the application *A* on the fastest workstation in the system to the parallel computing time:

$$S_P = \frac{\min_{1 \le k \le m} \{R(A, M_k)\}}{R_P} \quad \text{(EQ 4.6)}$$

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.

Another common metric for representing the limitations in parallel computing performance is *efficiency.* Efficiency is defined as the ratio of speedup to the number of processors, *m*:

$$Eff_P = \frac{S_P}{m} = \frac{\min_{1 \le k \le m} \{R(A, M_k)\}}{m \cdot R_P} \quad \text{(EQ 4.7)}$$

These performance metrics will be used in the following sections as we develop our modeling methodology.

4.4 RC Node Analysis

Our performance model analysis will begin with a single RC node executing a fork-join or SIA. Restricting the analysis to a single node will allow us to investigate the interaction between the processor and RC unit before expanding our analysis to multiple nodes.

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 4.1. 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 the RC hardware and in software on the host processor. Additionally, hardware can be

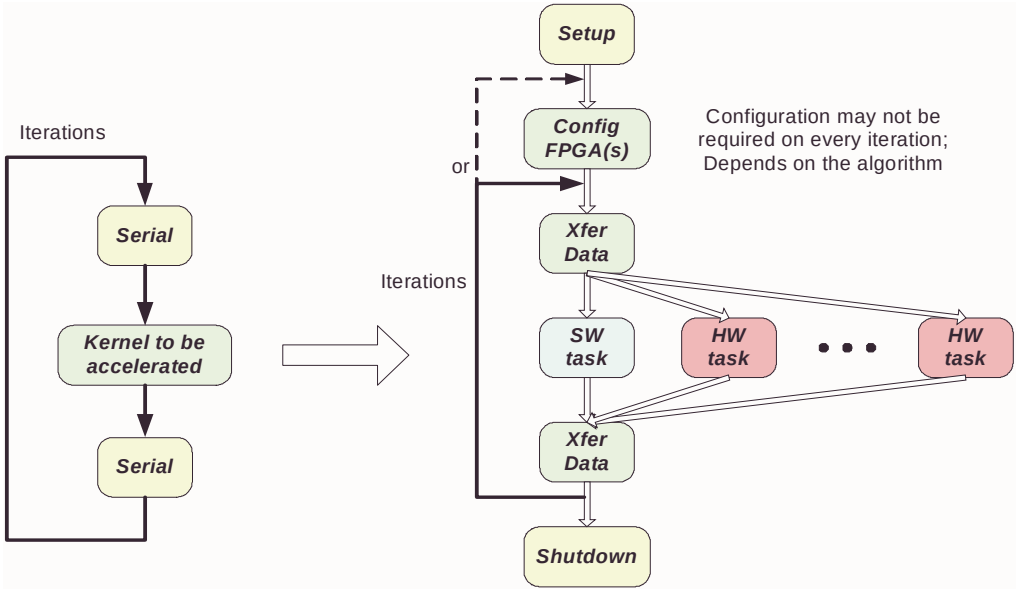


FIGURE 4.1 Synchronous Iterative Algorithm

reused within a given iteration if the number of tasks or size of the task exceeds the number of available FPGAs.

For an SIA (Figure 4.2), the time to complete a given iteration is equal to the time for the last task to complete whether it be in hardware or software. For each iteration of the algorithm, there are some operations which are not part of the accelerated kernel 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 parallel hardware tasks, the runtime, R_{RC}, can be represented as:

$$R_{RC} = \sum_{i=1}^{I} \left[t_{serial,\,i} + max\left(t_{SW,\,i}, \max_{1 \le j \le n} [t_{HW,\,i,\,j}] \right) + t_{overhead,\,i} \right] \qquad \text{(EQ 4.8)}$$

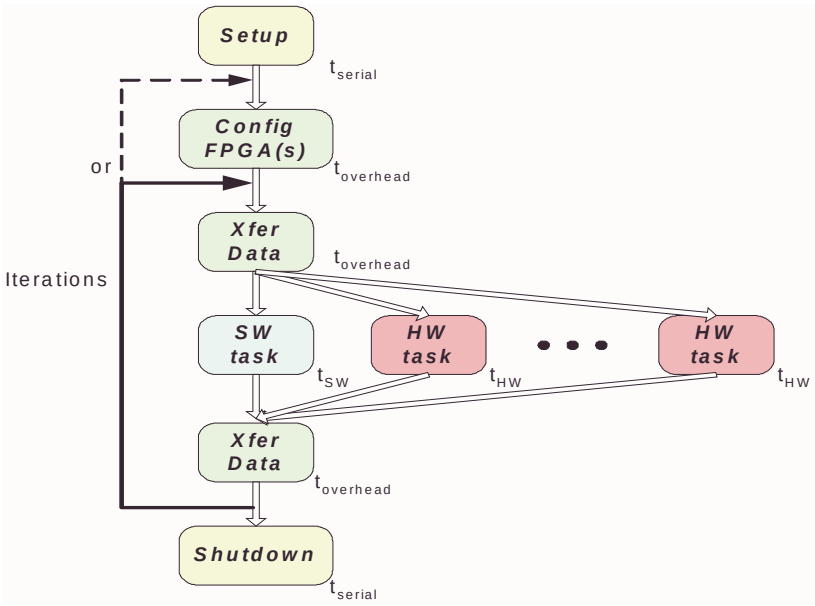


FIGURE 4.2 Flow of Synchronous Iterative Algorithm for RC Node

To simplify the math analysis, we will make a couple of reasonable assumptions. First, we will assume that each iteration requires roughly the same amount of computation. Focusing on a "typical" iteration allows us to remove the reference to individual iterations in Eqn. 4.8 making the mathematical analysis easier. 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). Again, this assumption is not necessary but does make the math analysis easier without limiting our scope of representative algorithms too severely. For the applications addressed in this dissertation, this is a reasonable and valid assumption (see Sec. 1.2.1). If the iterations are not iid, we must retain the first form of Eqn. 4.9 which includes the summation over all iterations I and the math analysis is more difficult. We can then write the run time as:

$$R_{RC} = \sum_{i=1}^{I} \left(E[t_{serial,i}] + E\left[max\left(t_{SW,i}, \max_{1 \le j \le n} [t_{HW,i,j}] \right) \right] + E[t_{overhead,i}] \right)$$
$$= I\left(t_{serial} + E\left[max\left(t_{SW}, \max_{1 \le j \le n} [t_{HW,j}] \right) \right] + t_{overhead} \right) \quad \text{(EQ 4.9)}$$

The execution time for hardware tasks should be deterministic (i.e. there are no decision loops such as *if* or *while* loops) and related to the clock frequency of the hardware. In the applications we are considering, only code suitable for acceleration in hardware are implemented in hardware, meaning decision loops will likely remain in software where they are more efficiently implemented. Thus, the execution time of the hardware can be estimated based on clock frequency and/or simulations. We will assume that all concurrent or parallel hardware tasks are the same. Also, we will initially assume that the hardware and software tasks do not overlap. This assumption is valid for the applications addressed in this dissertation but is not a limiting factor of the model. Although the math again would be more complex, the term could be carried throughout the analysis as the maximum value of concurrently running software and hardware. Since our tasks do not overlap however, we can represent the expected hardware execution time with the mean value, t_{HW}, and simplify the equation. The execution time of the software tasks will depend not only on the speed of the processor but also on the background load of the system. We will represent this background load as γ and represent the expected completion time of the software on a dedicated processor as t_{SW}. The expected execution or runtime on the RC system becomes:

$$R_{RC} = I(t_{serial} + (\gamma \cdot t_{SW}) + (t_{HW}) + t_{overhead}) \quad \text{(EQ 4.10)}$$

If there is no background load on the processor, $\gamma = 1$, and the equation reduces to a dedicated system. If there are multiple sequential hardware tasks, g, the expected value becomes:

$$R_{RC} = I\left(t_{serial} + (\gamma \cdot t_{SW}) + \sum_{j=1}^{g} t_{HW,j} + t_{overhead}\right) \quad \text{(EQ 4.11)}$$

The time spent on serial operations is also affected by the background load and thus must be multiplied by the background load factor γ. The runtime for an application running on a single RC node is then

$$R_{RC} = \gamma \cdot t_{serial} + (\gamma \cdot t_{SW}) + (t_{HW}) + [(n-d) \cdot t_{data} + (n-r) \cdot t_{config}] \quad \text{(EQ 4.12)}$$

where *n* is the number of hardware tasks, *d* is the number of hardware tasks not requiring a new data set, and *r* is the number of hardware tasks not requiring a new configuration of the RC unit.

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:

$$R_1 = I\left(t_{serial} + t_{SW} + \sigma \cdot \sum_{n} t_{HW}\right) \quad \text{(EQ 4.13)}$$

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.

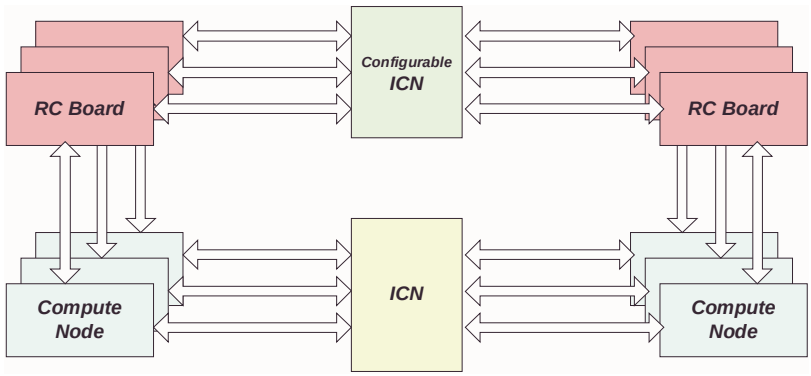


FIGURE 4.3 HPRC Architecture

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

$$S_{RC} = \frac{R_1}{R_{RC}} = \frac{t_{serial} + t_{SW} + \sigma \cdot \sum_{n} t_{HW}}{\gamma \cdot t_{serial} + (\gamma \cdot t_{SW}) + (t_{HW}) + [(n-d) \cdot t_{data} + (n-r) \cdot t_{config}]} \quad \text{(EQ 4.14)}$$

In the next section we will take our single node analysis and apply it in a multi-node environment of shared, heterogeneous resources.

4.5 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 4.3. For now, we will not consider the optional configurable interconnection network between the RC units in our modeling analysis.

Again, we will assume we have a segment of an application having I iterations that will execute on parallel nodes with RC hardware acceleration and as before in the single node analysis we will assume that all iterations are roughly the same as is shown in Figure 4.4. 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 an SIA, we know the time to complete an iteration is equal to the time for the last task, which could be hardware or software, to complete on the slowest node. For each iteration of the algorithm, there are some calculations which cannot be executed in parallel or accelerated in hardware and are denoted $t_{mserial,i}$. There are other serial operations required by the RC hardware and they are denoted $t_{nserial,i}$. Other overhead processes that must occur such as synchronization and exchange of data are denoted $t_{movhd,i}$ and $t_{novhd,i}$ 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 parallel runtime, R_P, can be represented as:

$$R_P = \sum_{i=1}^{I} \left[t_{mserial,i} + t_{nserial,i} + \max_{1 \le k \le m} \left(t_{SW,i,k}, \max_{1 \le j \le n} [t_{HW,i,k,j}] \right) + t_{movhd,i} + t_{novhd,i} \right] \quad \text{(EQ 4.15)}$$

$$= \sum_{i=1}^{I} \left[t_{mserial,i} + t_{nserial,i} + \max_{1 \le k \le m} (t_{RC,i,k}) + t_{movhd,i} + t_{novhd,i} \right]$$

Again, to simplify the math analysis, 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 Eqn. 4.15 and focus on the computations for a

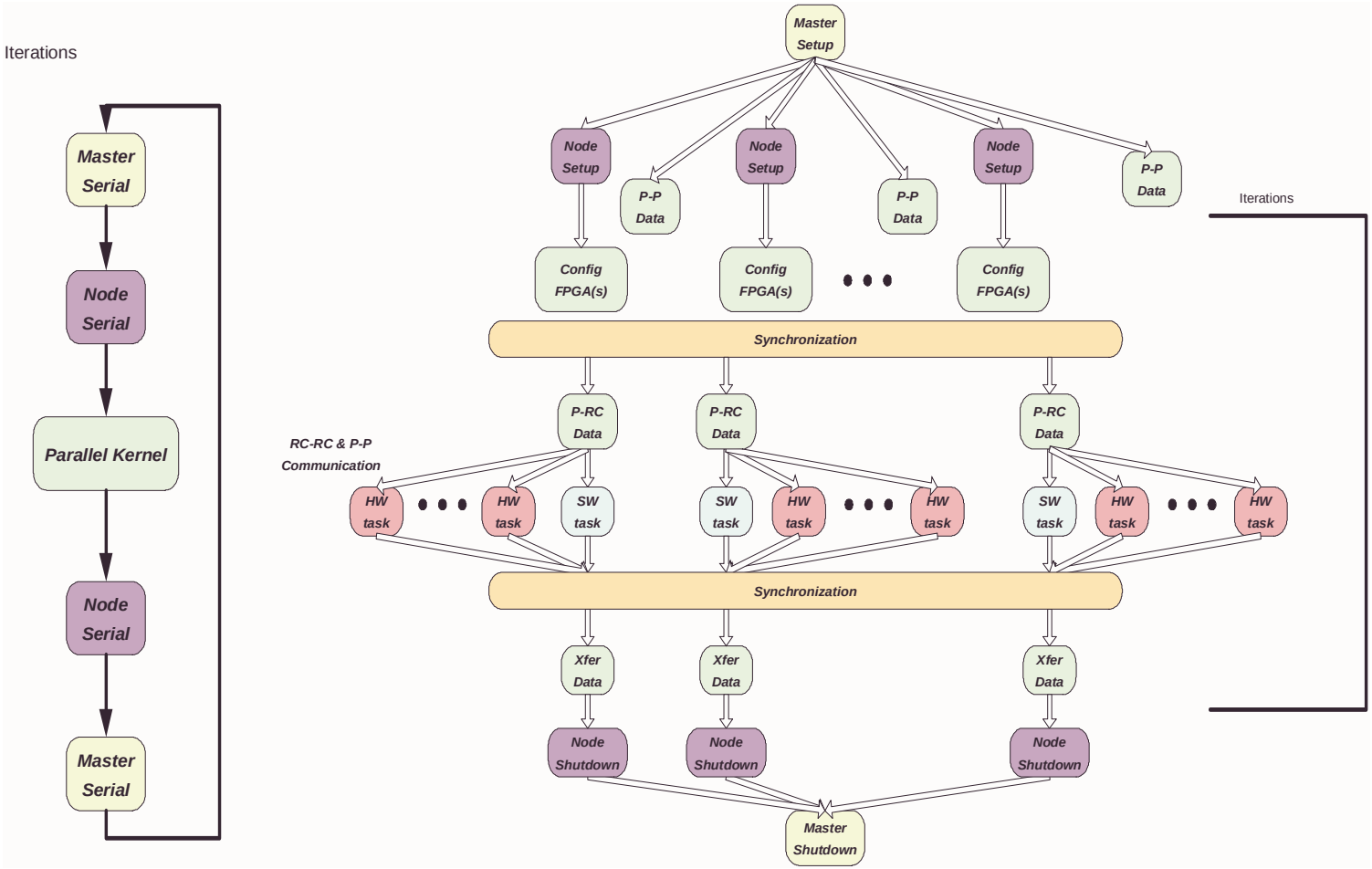


FIGURE 4.4 Flow of Synchronous Iterative Algorithm for Multi Node

"typical" iteration. Second, we will also assume that each node has the same hardware tasks and configuration making the configuration overhead for each node the same. This assumption is not necessary but removes a level of complication from the analysis not needed for the type of applications considered in this dissertation since all nodes are processing the same type of data. Third, we will model each term as a random variable and use their expected values. Thus we define $t_{mserial}$ and $t_{nserial}$ as the expected value of $t_{mserial,i}$ and $t_{nserial,i}$. Similarly, we define t_{movhd} and t_{novhd} as the expected value of $t_{movhd,i}$ and $t_{novhd,i}$. 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, as before in the single node analysis, we assume that the random variables are each independent and identically distributed (iid), thus the run time can then be expressed as:

$$R_P = \sum_{i=1}^{I} \left[E[t_{mserial,i}] + E\left[\max_{1 \le k \le m} \{ t_{RC,i,k} + t_{nserial,i} + t_{novhd,i} \} \right] + E[t_{movhd,i}] \right] \quad \text{(EQ 4.16)}$$

$$= I\left(t_{mserial} + E\left[\max_{1 \le k \le m} \{ t_{RC,k} + t_{nserial} + t_{novhd} \} \right] + t_{movhd} \right)$$

As discussed earlier in the section on the single node model, the execution time for hardware tasks will be deterministic and available from calculations or simulations and are the same tasks at each node. Also, for the applications we are currently working with, the hardware and software tasks do not overlap. Therefore we can represent the expected hardware execution time with the mean value, t_{HW}, and simplify the equation. The execution time of the software tasks, t_{SW}, will depend not only on the speed of the processor but also will be affected by the background load of the system and the hardware execution time will be affected by the application load imbalance. Rewriting Eqn. 4.16:

$$R_P = I\left(t_{mserial} + \beta \cdot t_{HW} + E\left[\max_{1 \le k \le m}\{t_{SW,k} + t_{nserial} + t_{novhd}\}\right] + t_{movhd}\right)$$
$$= I\left(t_{mserial} + \beta \cdot t_{HW} + E\left[\max_{1 \le k \le m}\{t_{node,k}\}\right] + t_{movhd}\right) \quad \text{(EQ 4.17)}$$

We can rewrite the total processor work at node *k* in terms of the average task completion time rather than the maximum and later multiply by an imbalance factor to account for application and background load imbalances. Again assuming the random variables are iid, we can express the total work across all *m* nodes in the HPRC platform as:

$$t_{work} = m \cdot E[t_{node,k}] \quad \text{(EQ 4.18)}$$

When tasks are divided across the nodes, a load imbalance due to application workload distribution, background users, or network heterogeneity exists. We will represent this load imbalance as η. 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 factor:

$$E\left[\max_{1 \le k \le m}(t_{node,k})\right] = \eta \cdot E[t_{node,k}] \quad \text{(EQ 4.19)}$$

Combining Eqn. 4.18 and Eqn. 4.19 we can rewrite the maximum task completion time as,

$$E\left[\max_{1 \le k \le m}(t_{node,k})\right] = \frac{\eta \cdot t_{work}}{m} \quad \text{(EQ 4.20)}$$

Note that if the load is perfectly balanced, η is the ideal value of 1. As the load imbalance becomes worse, η increases. If the algorithm runs entirely on a single node, *m*=1, η is the ideal value of 1 and the model reduces to that for a single processor.

The time spent on serial operations is also affected by the background load and thus must be multiplied by the background load factor γ. The runtime for an application running on a shared, heterogeneous HPRC platform is then

$$R_P = \gamma \cdot t_{mserial} + \beta \cdot t_{HW} + \frac{\eta \cdot t_{work}}{m} + [t_{synch} \cdot \log m] + T_{comm}(c) \qquad \text{(EQ 4.21)}$$

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:

$$R_1 = I \cdot \left[t_{mserial} + \sum_{k=1}^{m} \left(t_{SW} + \sigma_k \cdot \sum_{n} t_{HW} \right) \right] \qquad \text{(EQ 4.22)}$$

The overhead for the HPRC platform consists of the communication and synchronization between the nodes. We will initially model the time required for synchronization as a logarithmic growth with the number of nodes [102]. The communication between nodes can be modeled using Eqn. 4.4 or Eqn. 4.5 from Sec. 4.3.2.

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:

$$S_P = \frac{R_1}{R_P} = \frac{t_{mserial} + \sum_{k=1}^{m} \left(t_{SW} + \sigma_k \cdot \sum_{n} t_{HW} \right)}{\gamma \cdot t_{mserial} + \beta \cdot t_{HW} + \frac{\eta \cdot t_{work}}{m} + [t_{synch} \cdot \log m] + T_{comm}(c)} \qquad \text{(EQ 4.23)}$$

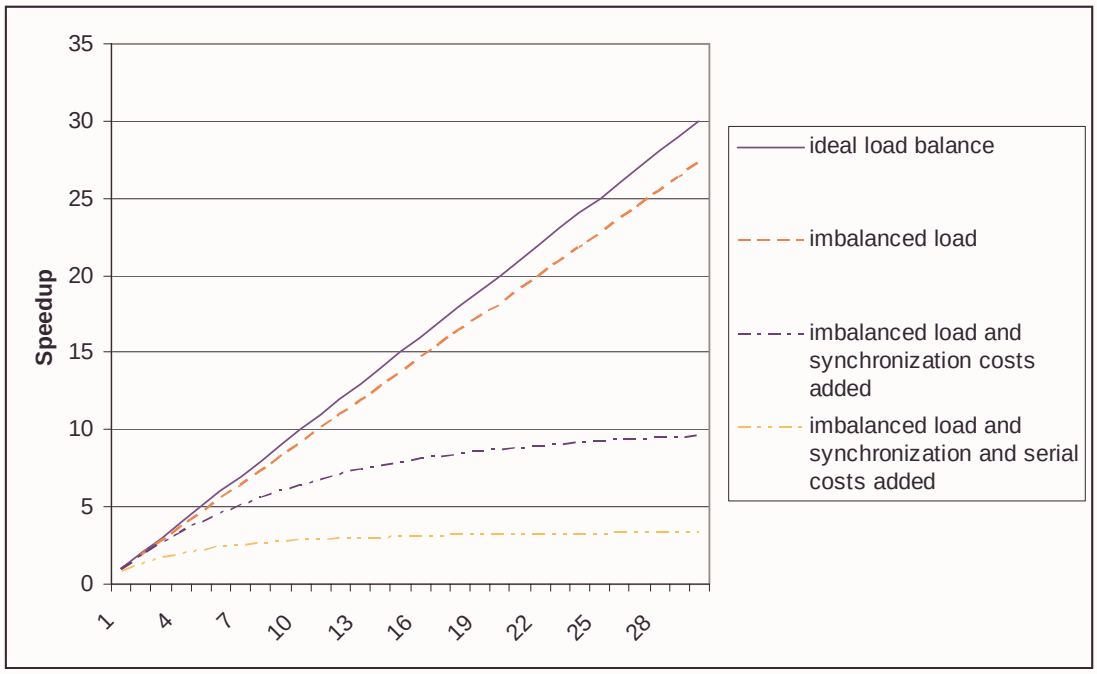


FIGURE 4.5 Speedup Curves

Using Eqn. 4.23, we can investigate the impact of load imbalance, synchronization, and communications on algorithms performance by varying η, t_{synch}, and the $T_{comm}(c)$ parameters. Consider the cases illustrated in Figure 4.5. The first case is the ideal situation where the load is perfectly balanced and there is no background load (η is 1), t_{synch} is zero for negligible synchronization costs, $t_{mserial}$ is zero for no serial computations, and so the speedup is equal to m, the number of nodes. The second case includes the impact of load imbalance by increasing η and retaining serial and synchronization costs at zero. Here the speedup still increases linearly with m but the slope is less than unity ($1/\eta$). Therefore, as m gets larger, the performance continues to degrade. In the third curve, synchronization costs are added while serial computation costs are held at zero. As expected, the speedup drops off logarithmically as m increases according to Eqn. 4.23. In the

fourth curve, the serial computation costs are added further reducing the speedup as suggested by Amdahl's Law [22].

This model is widely applicable to a number of fork-join and SIA applications. In this dissertation we will use this model to describe the performance of the applications discussed in Chap. 3. In the next section we will develop the model for load imbalance.

4.6 Load Imbalance Model

4.6.1 Introduction

Having developed a model framework for predicting performance, we will now look at the model details for predicting the computational load on the processing nodes. The load induced by the distributed application is referred to as the *application load* and the load from shared resources (other users and operating system tasks) is referred to as the *background load.* Both types of computation load can cause load imbalance and degrade the performance of a distributed application. As stated earlier, we will assume the work performed by the reconfigurable unit is deterministic, i.e. there are no decision loops such as *if* or *while* loops, and therefore the execution time for the hardware will be known *a priori* from simulations or calculations. Since the execution time is deterministic, the hardware execution will not contribute to the load imbalance issues and is not affected by background user load. Thus we can use the load imbalance results from Peterson's work [102] by modifying where necessary to fit our applications and system architecture.

The specific characteristics of the load imbalance will depend on both the application and the architecture under study. Thus as we develop these models, we will need to tune them to fit the characteristics of the applications (for application load imbalance) and the system architecture for the background load.

4.6.2 General Load Imbalance Model

In [102], Peterson develops a model for the load imbalance on shared, heterogeneous workstations. He noted that the total load imbalance on each processor *j* can be modeled with a scale factor $\eta_j = \gamma_j\beta_j$, where the factors γ_j and β_j represent *background* and *application load imbalance* respectively. This scale factor has a multiplicative effect on the execution time. The total load imbalance for the system will be the maximum η_j for $1 \leq j \leq m$ where *m* is the total number of processing nodes.

Assuming each task gets an equal fraction of the CPU time (processor sharing model) and the background load on a processor *j* is *l* tasks, then the application will take *l* + 1 times as long to complete as on an idle processor. Therefore, the background load imbalance scale factor is $\gamma_j = l + 1$ in this case [102]. Defining γ_j to be a discrete, positive integer-valued random variable and β_j to be a positive integer-valued random variable representing the amount of work done on processor *j*, the probability distribution for η_j can be expressed as follows [102]:

$$Prob\{\eta_j = \beta_j\gamma_j = k\} = \sum_{\alpha = 1}^{k} Prob\{\beta_j = \alpha\} Prob\left\{\gamma_j = \frac{k}{\alpha}\right\} \qquad \text{(EQ 4.24)}$$

where $Prob\{\beta_j = k/\alpha\} = 0$ if k/α is not a positive integer.

To model heterogeneous processors, the scaling factors δ_j representing the processing time per unit work of processor *j*, and ω the time per unit work of the baseline processor are introduced. (For homogeneous resources, $\delta_j = \omega = 1$.) Finally, *B* represents the average work for a processing node and β_j/B is the application load imbalance scale factor for processor *j*. Therefore, for heterogeneous processors, η is defined as:

$$\eta = \frac{1}{B} E\left[\max_{1 \le j \le m} \left(\frac{\eta_j}{W_j} \right) \right] = \frac{1}{\omega B} E\left[\max_{1 \le j \le m} (\beta_j \gamma_j \delta_j) \right] \quad \text{(EQ 4.25)}$$

Also from Peterson [102], assuming the application and background load imbalances are independent of each other and that the application and background load imbalances of any processor are independent of the other processors, the distribution, cumulative distribution, probability distribution, and expectation are as follows:

Distribution Function:

$$Prob\{\eta_j = k\} = \sum_{\alpha = 1}^{k} Prob\{\beta_j = \alpha\} Prob\left\{ \gamma_j = \frac{k}{b\delta_j} \right\} \quad \text{(EQ 4.26)}$$

$$Prob\{\eta_j \le k\} = \sum_{\alpha = 1}^{k} Prob\{\beta_j = \alpha\} Prob\left\{ \gamma_j \le \left\lfloor \frac{k}{\alpha\delta_j} \right\rfloor \right\}$$

Cumulative Distribution Function:

$$Prob\left\{ \max_{1 \le j \le m} (\eta_j) \le k \right\} = \prod_{j = 1}^{m} \sum_{\alpha = 1}^{k} Prob\{\beta_j = \alpha\} Prob\left\{ \gamma_j \le \left\lfloor \frac{k}{\alpha\delta_j} \right\rfloor \right\} \quad \text{(EQ 4.27)}$$

Probability Distribution:

$$Prob\left\{ \max_{1 \le j \le m} (\eta_j) = k \right\} = Prob\left\{ \max_{1 \le j \le m} (\eta_j) \le k \right\} \quad \text{(EQ 4.28)}$$

$$- Prob\left\{ \max_{1 \le j \le m} (\eta_j) \le k - 1 \right\}$$

Expected Value:

$$\eta = \frac{1}{B\omega} \cdot \sum_{k=1}^{\infty} k \left(\prod_{j=1}^{m} \sum_{\alpha=1}^{k} Prob\{\beta_j = \alpha\} Prob\left\{ \gamma_j \le \left\lfloor \frac{k}{\alpha\delta_j} \right\rfloor \right\} \right. \quad \text{(EQ 4.29)}$$

$$\left. - \prod_{j=1}^{m} \sum_{\alpha=1}^{k-1} Prob\{\beta_j = \alpha\} Prob\left\{ \gamma_j \le \left\lfloor \frac{k-1}{\alpha\delta_j} \right\rfloor \right\} \right)$$

The application and background load imbalances are modeled with discrete-valued scale factors since the loads consist of some number of processes or tasks (a non-negative integer).

Now we will investigate application and background load imbalance independently and develop models for each. First we will look at the impact of application load imbalance and then consider the effects of background load imbalance and heterogeneity. Finally, we consider the interaction of application and background load imbalance for shared, heterogeneous resources.

4.6.3 Application Load Imbalance Model

First, we will isolate the application load imbalance by considering the applications running on a dedicated set of homogeneous workstations ($\gamma_j = 1$ and $\delta_j = \omega = 1$). The application load imbalance can be a result of unbalanced problem partitioning or can appear over time as the workload distribution changes from iteration to iteration. This change can result from the migration of processes in some applications or as in the case of simulations, not every node has an event to process for every iteration. So to determine the application load imbalance, we must first assess the cause and then characterize what happens during each iteration of the application and determine if the workload distribution changes and how it changes over time.

SAT Solver Application Imbalance . For the SAT Solver application, if the problem is initially balanced, the amount of computation remains constant across iterations and nodes thus there

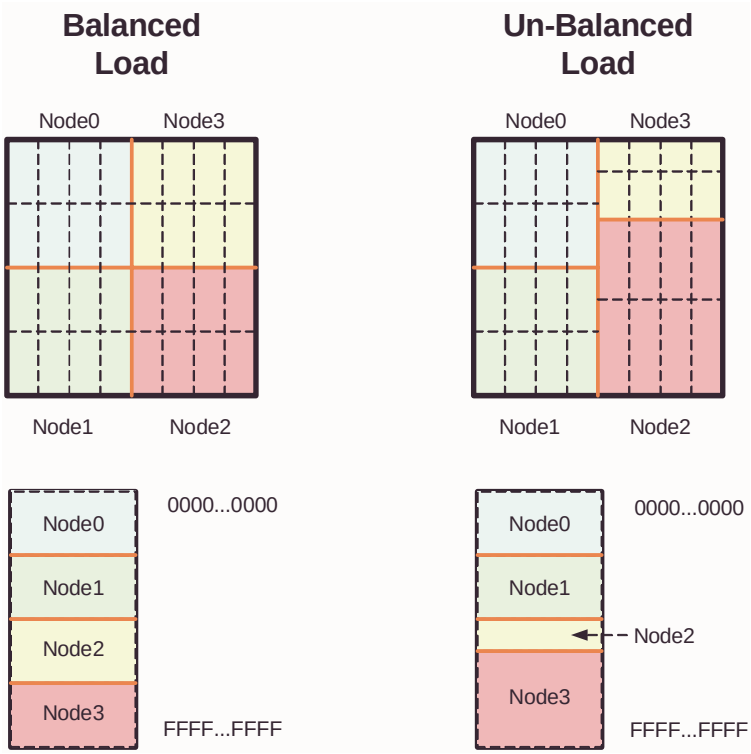


FIGURE 4.6 Application Load Distribution for SAT Solver Application

is no application load imbalance. Therefore, if the application is divided equally across nodes at the start of the execution, then it will remain balanced as far as computation load at each subsequent iteration and we can set $\beta_j = B = 1$. However, if we induce an application load imbalance a priori by modifying the search space assigned to each workstation as shown in Figure 4.6, we can study the effects of an application load imbalance on the SAT Solver application.

The application load imbalance can be modeled as the ratio of the amount of work done on the heaviest loaded processor β_{max} to the average amount of work for a processor B. With no background load ($\gamma_j = 1$) the load imbalance from Eqn. 4.25 becomes

$$\eta = \frac{\beta_{max}}{B} \qquad \text{(EQ 4.30)}$$

To determine the load imbalance for the case where the application load or search space is unevenly distributed, we must determine the average load for each processor, *B*, and the amount of work for the most heavily loaded processor, β_{max}. Since we will prescribe the distribution of the search space by issuing the seed values, we can easily find the values for β_{max} and *B*.

Matrix Vector Multiplication Algorithm Application Imbalance. For the matrix vector multiplication algorithm, an application load imbalance exists when the size of the matrix is not integer divisible by the number of nodes. In this situation, some of the processing nodes will receive more rows of the matrix than others. By monitoring these cases, we can study the effects of application load imbalance on the matrix vector algorithm.

Again, the application load imbalance can be modeled as the ratio of the amount of work done on the heaviest loaded processor β_{max} to the average amount of work for a processor *B*. To determine the load imbalance for the case where a processing node receives more rows of the matrix than other nodes, we must determine the average load for each processor, *B*, and the amount of work for the most heavily loaded processor, β_{max}. The average load for each processor will simply be:

$$B = \frac{rows}{nodes} \quad \text{(EQ 4.31)}$$

and the work for the most heavily loaded processor is:

$$\beta_{max} = \left\lfloor \frac{rows}{nodes} \right\rfloor \quad \text{(EQ 4.32)}$$

Other Applications. For other applications such as simulations, this is not the case. The work by Peterson [102] presents a study of the application load imbalance model for such a class of applications. In his work he isolates the application load imbalance for a detailed study by assum-

ing a dedicated, homogeneous set of nodes and uses analytical modeling with known probability distributions and order statistics to develop models for the application load imbalance.

Now we will look at the effects of background load imbalance and heterogeneity.

4.6.4 Background Load Imbalance Model

We will now focus on the imbalance due to background load on shared resources. First a few assumptions: 1) the application and background load imbalances are independent, 2) the background load on each processor is independent and identically distributed (i.i.d.), 3) all tasks on a processor have the same priority, and 4) processors are homogeneous. To isolate the background imbalance, we will assume no application load imbalance. Substituting $\beta_j = B = 1$ and $\delta_j = \omega = 1$ into Eqn. 4.29 yields:

$$\eta = \sum_{k=1}^{\infty} k \left(\prod_{j=1}^{m} Prob\{\gamma_j \le k\} - \prod_{j=1}^{m} Prob\{\gamma_j \le k-1\} \right) \quad \text{(EQ 4.33)}$$

As Peterson noted, if the background load on the processor is *k* tasks and each task (including the application) gets an equal fraction of the CPU time, the application will take *k+1* times longer than if it were run on a dedicated processor. The probability of *k* background tasks on processor *j* can be represented as:

$$Prob\{\gamma_j = k+1\} = Q_{j,k} \quad \text{(EQ 4.34)}$$

In [102], Peterson derives the background load distribution $Q_{j,k}$ using two background load queueing models: processor sharing or M/M/1 and FCFS M/G/1 model and concluded that

the former is an adequate representation of the UNIX environment. It will be determined through our experiments if this is also an adequate model for the Linux environments.

In a Processor Sharing (PS) model, the server is equally shared among tasks and can be viewed as the limiting case of the round-robin discipline where the time slice approaches zero [81]. We will assume the background jobs arrive following a Poison process with a given rate λ_j and that the service distribution is Coxian [81, 102]. The queue length distribution can be derived from queueing theory as $Q_{j,k}= (1 - \rho_j)\rho_j^k$, where ρ_j is the ratio of the arrival rate to the service rate ($\rho_j = \lambda_j / \mu_j$). Finding the queue length distribution for general service distributions of a PS model is impractical however, the results for a PS model with Coxian distributions are the same as those from M/M/1 queueing theory (queues with exponential arrival and service distributions) [102].

Assuming PS on shared, homogeneous resources with no application load imbalance, the load imbalance is given by [102]:

$$\eta = \sum_{k=1}^{\infty} k \left[\prod_{j=1}^{P} (1 - \rho_j^k) - \prod_{j=1}^{P} (1 - \rho_j^{k-1}) \right] \qquad \text{(EQ 4.35)}$$

where k is the number of tasks.

To extend the model to heterogeneous resources modeled with processor sharing, we substitute $\beta_j = B = 1$ into Eqn. 4.29.

$$\eta = \frac{1}{\omega} \cdot \sum_{k=1}^{\infty} k \left(\prod_{j=1}^{m} Prob\left\{ \gamma_j \le \left\lfloor \frac{k}{\delta_j} \right\rfloor \right\} - \prod_{j=1}^{m} Prob\left\{ \gamma_j \le \left\lfloor \frac{k-1}{\delta_j} \right\rfloor \right\} \right) \qquad \text{(EQ 4.36)}$$

Heterogeneity also impacts service rate μ for background tasks. By defining the service rate $\mu_j = \mu\omega/\delta_j$, where μ_j is simply the baseline service rate μ scaled to reflect the relative processing power

of node *j*, we can apply the impact of heterogeneity [102]. The load imbalance then becomes (with no application load imbalance):

$$\eta = \frac{1}{\omega} \cdot \sum_{k=1}^{\infty} k \left[\prod_{j=1}^{m} \left(1 - \left(\frac{\delta_j \lambda_j}{\mu \omega} \right)^{\lfloor k/\delta_j \rfloor} \right) - \prod_{j=1}^{m} \left(1 - \left(\frac{\delta_j \lambda_j}{\mu \omega} \right)^{\lfloor (k-1)/\delta_j \rfloor} \right) \right] \quad \text{(EQ 4.37)}$$

We have developed an analytic model to describe the background load on heterogeneous resources with Coxian service distributions and processor sharing. Next we integrate our model results for application and background load imbalance.

4.6.5 Complete Load Imbalance Model

Now that we have developed models for the application and background load imbalances independently, we will now combine these models to form a complete model and examine them for applications running on shared resources.

First, we consider homogeneous resources. From Eqn. 4.29 we model the load imbalance as

$$\eta = \frac{1}{B} \cdot \sum_{k=1}^{\infty} k \left(\prod_{j=1}^{m} \sum_{\alpha=1}^{k} Prob\{\beta_j = \alpha\} Prob\left\{ \gamma_j \le \left\lfloor \frac{k}{\alpha} \right\rfloor \right\} \right. \quad \text{(EQ 4.38)}$$
$$\left. - \prod_{j=1}^{m} \sum_{\alpha=1}^{k-1} Prob\{\beta_j = \alpha\} Prob\left\{ \gamma_j \le \left\lfloor \frac{k-1}{\alpha} \right\rfloor \right\} \right)$$

For the SAT Solver application and matrix vector algorithm, we use the results found in Sec. 4.6.3 and Sec. 4.6.4 to find the load imbalance is

$$\eta = \frac{\beta_{max}}{B} \cdot \sum_{k=1}^{\infty} k \left[\prod_{j=1}^{m} \left(1 - \left(\frac{\lambda_j}{\mu} \right)^{\lfloor k \rfloor} \right) - \prod_{j=1}^{m} \left(1 - \left(\frac{\lambda_j}{\mu} \right)^{\lfloor k-1 \rfloor} \right) \right] \quad \text{(EQ 4.39)}$$

Now considering the case of heterogeneous resources, the load imbalance is

$$\eta = \frac{\beta_{max}}{B} \cdot \frac{1}{\omega} \cdot \sum_{k=1}^{\infty} k \left[\prod_{j=1}^{m} \left(1 - \left(\frac{\delta_j \lambda_j}{\mu \omega} \right)^{\lfloor k/\delta_j \rfloor} \right) - \prod_{j=1}^{m} \left(1 - \left(\frac{\delta_j \lambda_j}{\mu \omega} \right)^{\lfloor (k-1)/\delta_j \rfloor} \right) \right] \quad \text{(EQ 4.40)}$$

Using this equation and the performance model given in Eqn. 4.21, we can describe the performance of our applications running on shared, heterogeneous HPRC resources.

In this chapter we have developed an analytic modeling methodology for predicting the performance of fork-join and SIAs running on shared, heterogeneous HPRC resources. The modeling methodology characterizes the performance effects of application load imbalance, background load imbalance, heterogeneity of processors, and the interaction of these combined effects. In the next chapter, we will validate this modeling methodology with the applications discussed in Chap. 3.

CHAPTER 5

MODEL VALIDATION

In the last chapter we developed a modeling methodology for algorithms running on shared, heterogeneous HPRC resources and now we will focus on validating that methodology with the use on sample applications. We will look at the accuracy of the performance model and how well it predicts the system behavior as well as identifying the causes of modeling errors. The method of model validation will be to isolate main components of the model (single node, application load imbalance, background load imbalance, communication, etc.) for independent study in order to quantify how well the model represents each of them before finally looking at the complete model for accuracy. In this chapter, we first discuss the method used to verify the model (including the HPRC configurations, the applications used, and the experiments performed). We then investigate the model step by step covering each of the main components to determine the accuracy before finally considering the combined effects and overall model. Figure 5.1 shows the phases of our model development and depicts the iterative process involved.

5.1 Validation Methodology

To verify the performance model presented in Chap. 4, we now focus on the main components of the model to evaluate how accurately their performance implications are characterized. By focusing on each component of the model individually, we can quantify how well the model describes the system and identify errors. Such an approach will provide a better and more complete understanding of the model and its limitations.

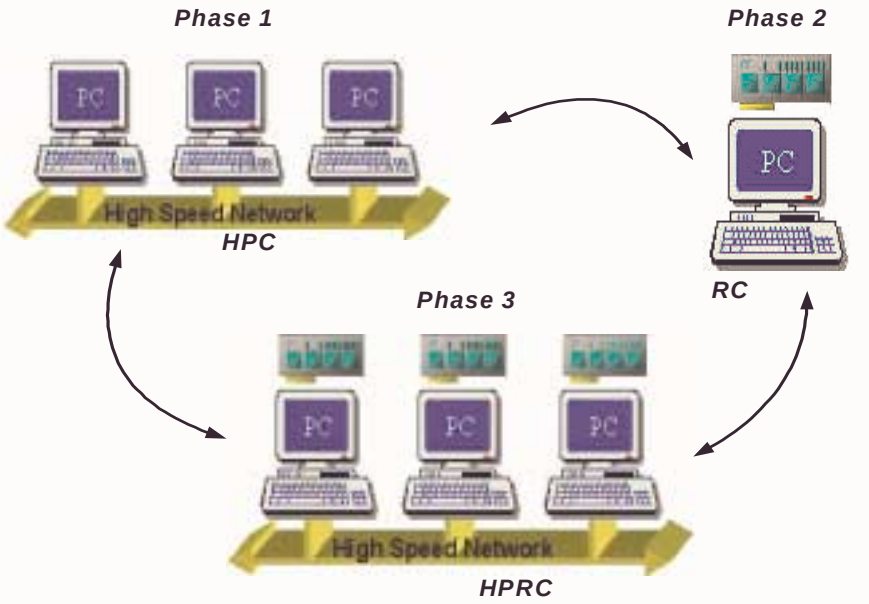


FIGURE 5.1 Phases of Model Development

To evaluate the accuracy of the model, we compare model predictions with empirical results from several test applications. Potential "good fit" HPRC applications include various DSP and image processing algorithms, simulation applications, and any master/slave applications where part of the algorithm can be accelerated by hardware. The desire is to have two or three candidate applications possessing different processing, data, and/or communication characteristics that will verify and fully test the limits/capabilities of the model. We will use a simple boolean satisfiability algorithm (SAT Solver), a matrix vector multiplication algorithm, and an encryption algorithm to evaluate the performance model. These algorithms were discussed in detail in Chap. 3 and represent the types of computationally intensive scientific algorithms that can be accelerated by implementation on the HPRC platform. The experiments we will use to validate the model are given in Table 5.1.

TABLE 5.1 Validation Experiments and Goals

Experiment Focus	Resource(s)	Application	Section Discussed
t_{comm}	UNIX LINUX	PVM/MPI Measurement Code	Sec. 5.2
RC model	Wildforce	Champion Demos	Sec. 5.3.1
	Firebird	SAT Solver	Sec. 5.3.4
	Pilchard	SAT Solver	Sec. 5.3.4
no-load HPRC	Pilchard Cluster	SAT Solver	Sec. 5.4.1
Application Load	Pilchard Cluster	SAT Solver	Sec. 5.4.2
Background Load	Pilchard Cluster	SAT Solver	Sec. 5.4.3
Application and Background Load	Pilchard Cluster	SAT Solver	Sec. 5.4.4
no-load HPRC and communication	Pilchard Cluster	Matrix Vector Multiplication	Sec. 5.4.1
Application and Background Load, communications and file access	Pilchard Cluster	Matrix Vector Multiplication	Sec. 5.4.4
no-load HPRC and communication	Pilchard Cluster	Advanced Encryption Standard Implementation	Sec. 5.4.1
Background Load, communications and file access	Pilchard Cluster	Advanced Encryption Standard Implementation	Sec. 5.4.3
Heterogeneity	Pilchard Cluster	SAT Solver, Matrix Vector Multiplication, and Advanced Encryption Standard Implementation	Sec. 5.4.5

Before exercising the multinode HPRC model, we look at the communication model using PVM and MPI running on a cluster of SPARC workstations running UNIX and a cluster of Pentium workstations running LINUX. We also use these clusters to obtain parameter measurements for the synthetic background load experiments. Background loading on shared resources degrades application performance due to competition for processor cycles. Since we cannot control the background loading on shared workstation networks where the load is the result of other users, we generate a synthetic load on the workstations such that it dominates the background load. The synthetic load is used in experiments to measure the accuracy of the model in the presence of background loads.

Next we look at the single-node RC model and evaluate its accuracy on several platforms. Using some simple algorithms we compare the model predictions with empirical results from three different RC platforms: Wildforce, Firebird, and Pilchard. These platforms were discussed in detail in Chap. 2.

The modeling methodology developed in Chap. 4 is capable of characterizing the performance on dedicated or shared as well as homogeneous or heterogeneous HPRC resources. The only resources we have available at this time are shared, homogeneous resources but with careful experiment planning, we can simulate the other conditions. To simulate a dedicated HPRC platform, experiments were planned and conducted during off hours. Studies were conducted to determine when the resources were not in use so that experiments could be planned. To simulate a heterogeneous HPRC platform, we varied the synthetic background load across the nodes.

Using our performance model we can characterize the effects of various application load, background load, and heterogeneous conditions. A different instantiation of the model is required for each combination of an application and computational resources condition. The experiments

conducted are given in Table 5.1 and each row reflects a different instantiation of the model. We find the parameters for each of the models by using empirical measurements of application runtime parameters such as $t_{mserial}$, t_{SW}, etc.

To determine the arrival rate of background tasks, we use UNIX scripts and the *ps* command to monitor process arrivals. The number of new processes within a fixed time period is recorded, with samples being collected to create histograms for the arrival distribution. These scripts were run for several weeks on a number of machines to determine arrival statistics. The arrival rates are higher during the week and during the day, however, for multiple hour time frames, the mean number of arrivals is nearly constant. These conditions allow us to model the arrival rates with a stationary Poisson process.

Likewise, statistics for the service distribution were collected using UNIX scripts and the *lastcomm* command. Data was again collected for several weeks on a number of machines to determine the computational requirements of the processes. These results were used to determine the service distribution parameters for the performance model.

Next, we begin the process of validating the modeling methodology by first looking at how to model the communication time between processors.

5.2 Accuracy of Modeling Communication Times

PVM and MPI code was developed to measure the setup and communication times for message round trip times between workstations for the multiprocessor network. In these experiments the focus is on the communication between nodes of the system, specifically the message travel time for different message sizes.

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 measured for various message sizes. Measurements were taken overnight when the network load was light between seven of the workstations. The measurement results are shown in Figure 5.2. Similar measurements were conducted using MPI on the Pilchard cluster to verify the model parameters on those machines.

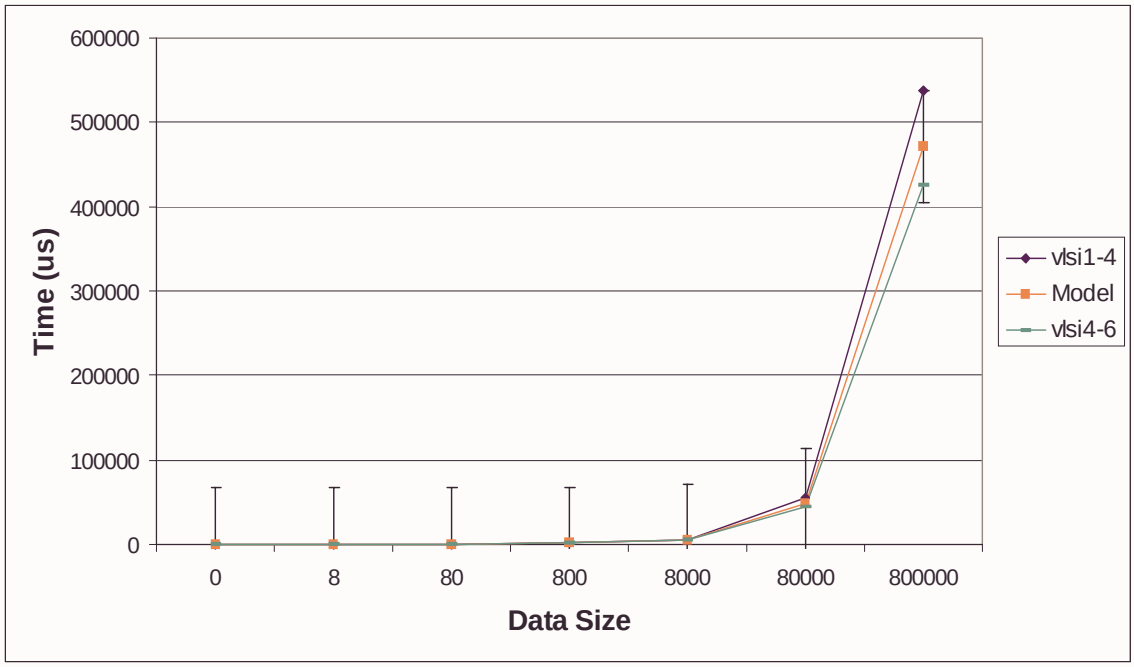
5.3 Accuracy of Single Node RC Model

In this section, our focus is on the elements specific to RC systems. Due to the architectural differences between RC systems, there are some slight variations in parameter definitions necessary. For example, the Wildforce requires the loading of multiple devices while the Firebird and Pilchard have only one FPGA device. Additionally, the Pilchard currently must be loaded by a separate executable. Additionally, different RC systems have different communication interfaces, programming interfaces, etc. all of which require consideration with developing the model for that particular system. The overall structure or framework of the RC model remains the same and only those differences need to be customized for the particular RC system. As discussed earlier in Sec. 4.4, a performance model was developed for a stand-alone RC unit.

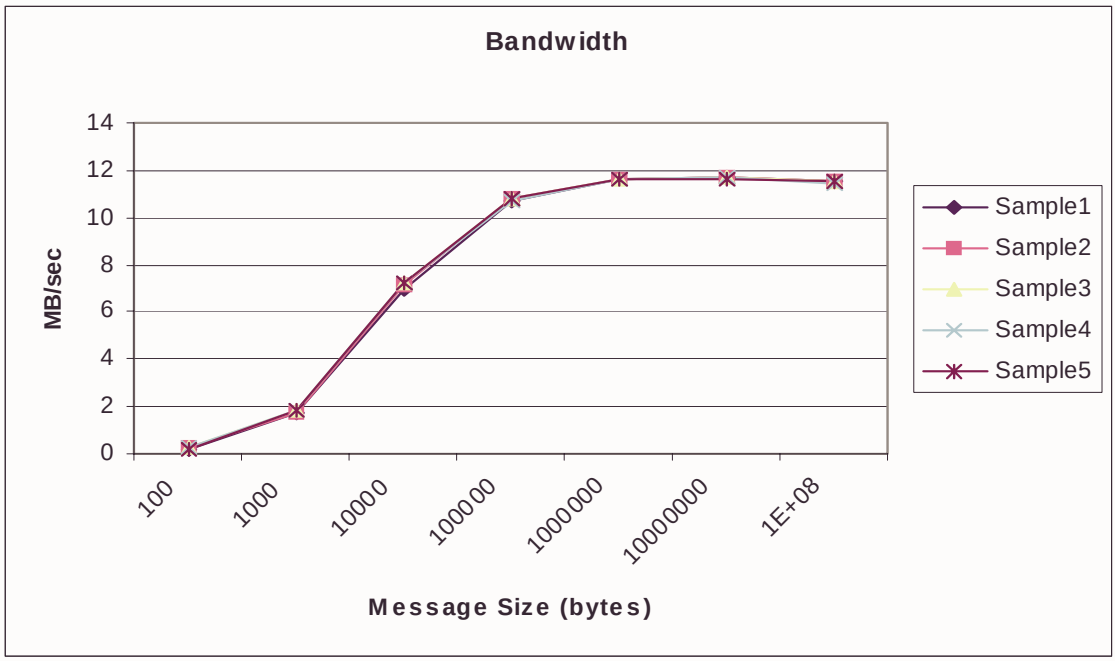
In this section we will present the validation results for the RC model using the Wildforce, Firebird, and Pilchard RC systems. Each of these systems were discussed in detail in Chap. 2.

5.3.1 Wildforce Measurements

Basic model parameter measurements were conducted using a benchmark program for the Wildforce board as shown in Table 5.2. To validate the execution time prediction of the model, the



(a)



(b)

FIGURE 5.2 Communication Measurement Results
(a) Message Round Trip Time and (b) Network Bandwidth

TABLE 5.2 Model Parameters for Wildforce from Benchmark Application

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

benchmark measurements are used with the developed model to predict the runtime for three of the CHAMPION demos. The details of the demo applications were discussed in Chap. 3. 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 and there is only one iteration (I =1). The remaining unknowns are the values for the total hardware and software work which is dependent on the application.

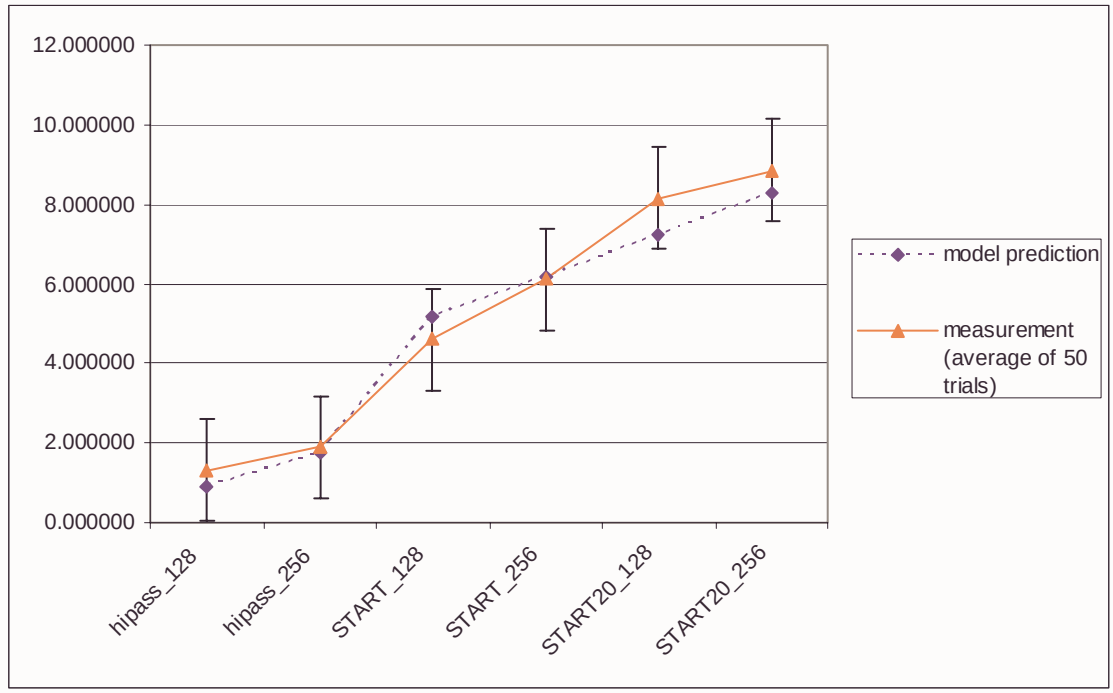
From Eqn. 4.12 we can find the runtime as:

$$R_{RC} = t_{serial} + (\gamma \cdot t_{SW}) + (t_{HW} \cdot N_e) + [(n-d) \cdot t_{data} + (n-r) \cdot t_{config}] \qquad \text{(EQ 5.1)}$$

Where N_e is the total number of events and t_{SW} and t_{HW} are the software and hardware execution times per event respectively. We can use the model to predict the runtime of the CHAMPION demo algorithms [107]. The average runtime of fifty trials on the Wildforce RC system is shown in Table 5.3 and Figure 5.3 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. Other tests were conducted on an application which passes a matrix of data between the processing elements of the Wildforce board to determine the communication delays between elements and a computation is performed on the

TABLE 5.3 Runtime Predictions and Measurements (time in seconds)

	Model Prediction	Measured Mean Value	Model Error (%)
hipass_128	0.911342	1.313353	30.61
hipass_256	1.773769	1.907098	6.997
START_128	5.166674	4.597426	12.38
START_256	6.175542	6.121883	0.869
START20_128	6.702268	8.134971	10.838
START20_256	7.741632	8.855299	6.357
MatrixA	3.188342	4.1	22.24
MatrixB	3.188345	4.1	22.24
MatrixAHW	0.002627	0.002644	0.633
MatrixBHW	0.002630	0.002641	0.3998



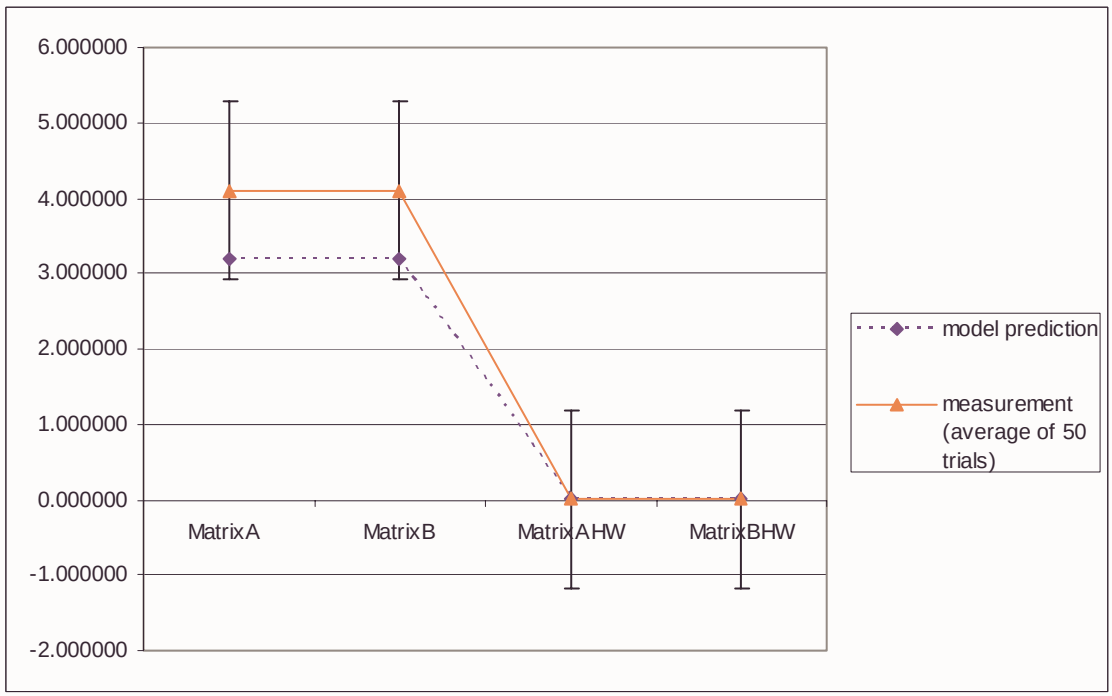


FIGURE 5.3 Comparison of RC Model Prediction with Measurement Results on Wildforce

TABLE 5.4 Model Parameters for Firebird from Benchmark Application

	Setup Clocks & Program PE0	Open/ Shutdown Board (Each)	Read Data File	Setup (tsw)	Interrupt Processing	Serial
Mean Value (msec) for 30 MHz clk	94	15	499.7	0.02618	0.0366	0.0192
Mean Value (msec) for 70 MHz clk	47	15	499.7	0.02618	0.0266	0.0192

data. Those tests are listed below as MatrixA and MatrixB where the computation is done in software and MatrixAHW and MatrixBHW where the computation is done in the RC unit.

5.3.2 Firebird Measurements

Again, using a sample benchmark application available from Annapolis Microsystems for the Firebird board some initial measurements for the model parameters were obtained. The application configures the FPGA, resets the board, and the software polls and waits for an interrupt from the hardware. This application is suitable for a benchmark because the Boolean SAT application will depend on interrupts to determine when the RC hardware has finished processing. Another benchmark application which reads and writes to the memory was used to determine memory access times on the Firebird board. From measurements on the Firebird board, we have determined the model parameters as shown in Table 5.4.

TABLE 5.5 Model Parameters for Pilchard from Benchmark Applications

	Open Pilchard	Close Pilchard	Write Value to Pilchard	Read Value from Pilchard
Mean Value (msec)	0.5	0.5	0.1	0.1

5.3.3 Pilchard Measurements

Again, using a sample benchmark application for the Pilchard board some initial measurements for the model parameters were obtained. The application writes a table of values to the Pilchard, signals the hardware to perform an FFT, and reads back the result. From measurements on the Pilchard board, we have determined the model parameters as shown in Table 5.5.

5.3.4 Single Node Boolean SAT Solver Comparisons

A Boolean SAT solver was developed for both the Firebird and Pilchard RC systems and in this section we will compare the results from both implementations and with a software-only version. The software-only version was executed on the VLSI UNIX machines and on the PILCHARD LINUX machines and results are given in Table 5.6. The RC solver engine on the Firebird machine runs at 35 MHz and there are 8 copies of the SAT core running on the FPGA. The Pilchard version runs at 33.25 MHz for problem sizes up to 38-bits and 26.6 MHz for problem sizes of 41, 43, and 44-bits. Again there are 8 copies of the SAT core running on the FPGA. The Pilchard version again uses a separate program to configure the FPGA and results do not include the configuration time. Figure 5.5 and Figure 5.4 show the measured versus model results for the Firebird and Pilchard systems respectively. Table 5.7, Table 5.8, and Table 5.9 show the numerical results of these tests including synthesis and implementation costs and modeling error.

TABLE 5.6 SAT Software-Only Runtime Comparisons (time in seconds)

Bits	SW VLSI2[a]	SW Pilchard[b]	Result
32	11471.990	4515.744	B0000060
36	11518.712	4521.625	D90000060
38	11511.338	4530.306	2D90000060
41	11521.457	4529.441	12D90000060
43	11547.886	4549.187	72D90000060
44	11566.326	4552.614	F2D90000060

a. Sun 220R: dual 450 MHz, UltraSPARC II, 1 GB RAM

b. 800 MHz Pentium III, Linux, 133 MHz SDRAM

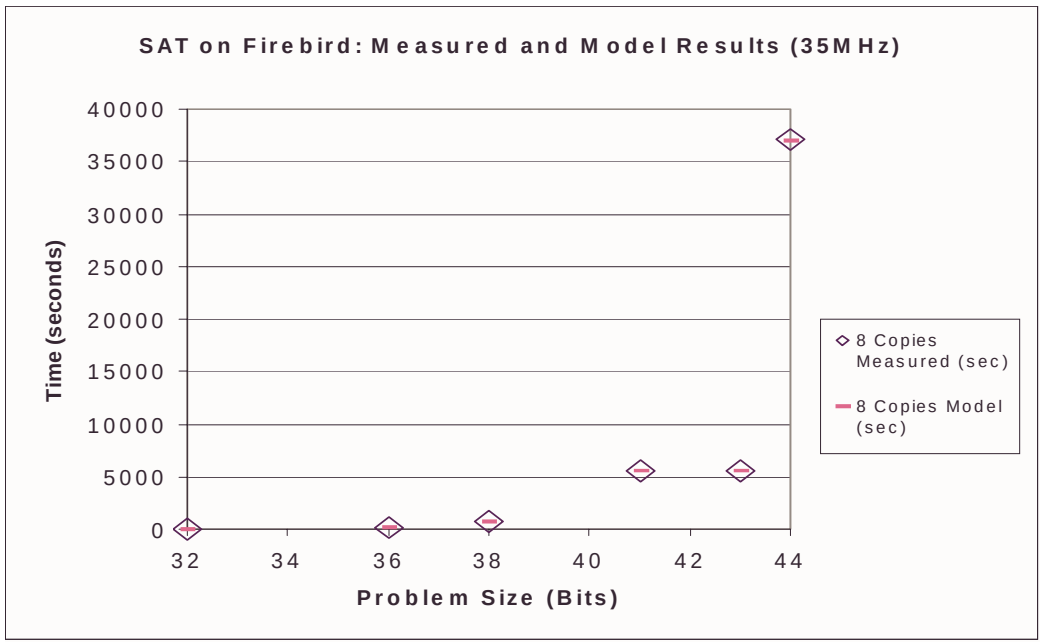
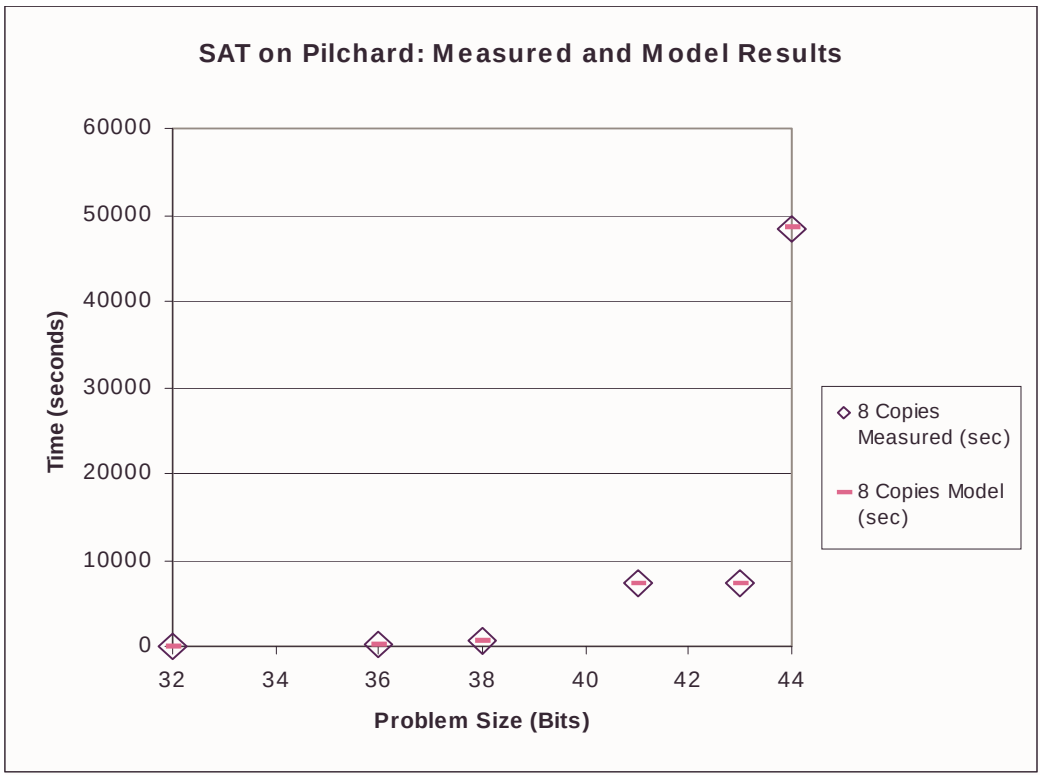


FIGURE 5.4 Boolean SAT Measured and Model Results on Firebird Running at 35MHz



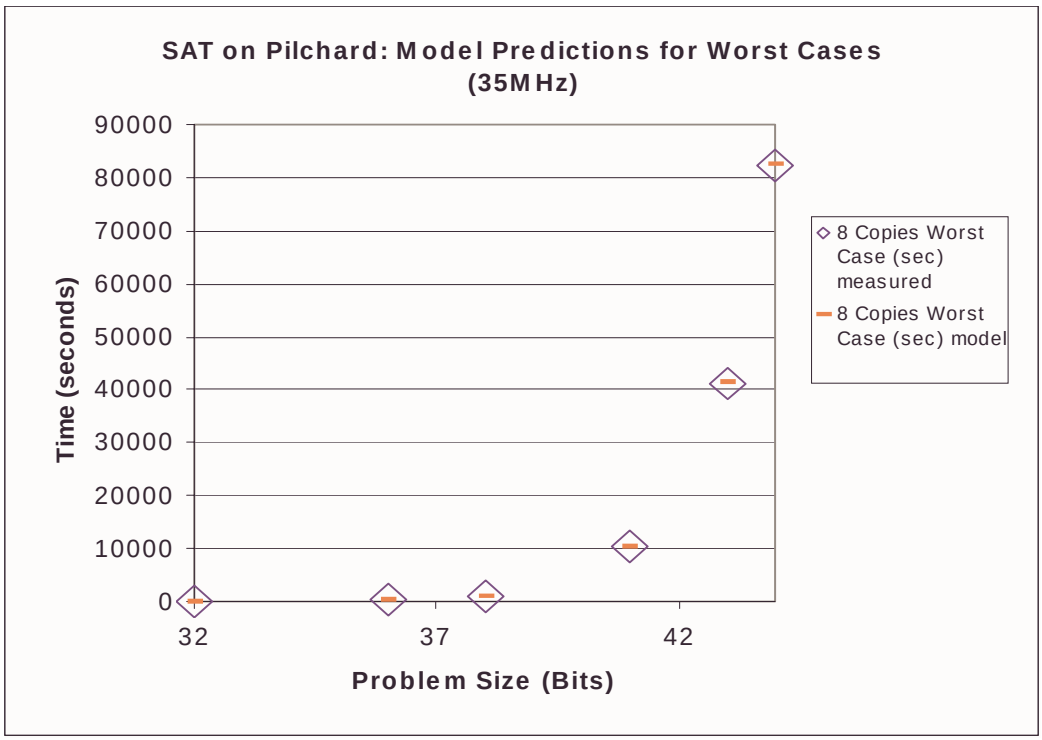


FIGURE 5.5 Boolean SAT Measured and Model Results on Pilchard Running at 33.25 MHz for 32, 36, and 38-bits and 26.6 MHz for 41, 43, and 44-bits

TABLE 5.7 SAT RC Node Runtime Comparisons (time in seconds)

Bits	Firebird Measured[a]	Firebird Model	Error (%)	Pilchard Measured[b]	Pilchard Model	Error (%)	Result
32	7.953	7.795	1.987	8.035004	8.07525	0.5	B0000060
36	191.859	191.865	0.003	200.867360	201.8332	0.48	D90000060
38	682.828	682.718	0.016	715.086893	718.521	0.48	2D90000060
41	5592.453	5591.252	0.021	7322	7356.748	0.47	12D90000060
43	5592.438	5591.252	0.021	7321	7356.748	0.49	72D90000060
44	37014.062	37005.87	0.022	48459	48691.77	0.48	F2D90000060

a. 866 MHz Pentium III, 1 GB RAM, WinNT, 66 MHz PCI card

b. 800 MHz Pentium III, Linux, 133 MHz SDRAM

TABLE 5.8 Overhead for RC systems (time in minutes)

	Firebird[a]		Pilchard[b]	
Bits	Synthesis	Place & Route	Synthesis	Place & Route
32	1	50	2	29
36	1	50	2	29
38	1	52	2	35
41	1	52	2	30
43	1	54	2	27
44	1	54	2	31

a. 1.4 GHz Pentium 4, 256 MB RAM, Windows 2000

b. Sun 220R: dual 450 MHz, UltraSPARC II, 1 GB RAM

TABLE 5.9 Worst Case Values (time in seconds)

Bits	Model Firebird	Measured Pilchard	Model Pilchard	Error (%)
32	15.464	16.0697	16.14849	0.49
36	245.552	257.11011	258.3459	0.48
38	981.832	1028.4394	1033.378	0.48
41	7853.780	10285	10333.76	0.47
43	31414.74	41137	41335.03	0.48
44	62829.36	82275	82670.05	0.48

5.4 HPRC Model Validation

Having validated the communication model and the single-node mode, we will now look at the applications running on the HPRC platform and validate the complete model. We will bring together the issues studied during the previous validation steps and focus on how well the model represents the entire system. The testing applications discussed in Chap. 3 will be used to study and validate the HPRC model and identify modeling errors and limitations.

5.4.1 No-Load Imbalance Results

First, we will look at the model under ideal conditions (no application load imbalance and no background load) on homogeneous HPRC resources. We will look at all three applications under these conditions starting with the SAT Solver.

SAT Solver. At the start of the SAT Solver program, each node is passed a starting seed by the master node which equally divides the problem space between the workstation nodes, m (in this case m=4). Next, the command to start the search is given with a broadcast command from the master node. The master node also searches part of the problem space while awaiting results from the other nodes. Once a node finds a solution, a message is sent back to the master node containing

TABLE 5.10 MPI SAT Solver No-Load Imbalance Results

Problem Size (bits)	Measured Mean (sec)	Model Prediction (sec)	Model Error (%)	Standard Deviation σ	Error of Estimation (90% Confidence)
32	4.9073	4.852047	1.126	0.1943	0.0452
36	65.213	65.4014	0.289	0.1851	0.0431
38	322.33	323.7453	0.439	0.1906	0.0572
41	2572	2584.254	0.476	0.2161	0.0795
43	10286	10334.57	0.472	0.3145	0.1636
44	20571	20668.33	0.473	0.10096	0.0959

the solution vector and indicating the solution was found. The master node then broadcasts the stop signal to the other nodes. Table 5.10 and Figure 5.6 show the results of these tests. The model error is very low with small estimation error indicating the model predicts the measured results very accurately.

AES Algorithm. The AES algorithm is also equally divided across the available nodes since the test vectors to be encrypted are sent to available workstations, keeping all workstations busy. The master in this scenario does not participate in the computations, rather operates as an orchestrator of the algorithm. Up to two test vectors are sent to each node at any given time to reduce the impact of message latency. Once a node receives a test vector, work begins immediately and the result, when found, is sent back to the master node to be recorded. Table 5.11 and Figure 5.7 show the results of these tests. The model error is very low with small estimation error indicating the model predicts the measured results very accurately.

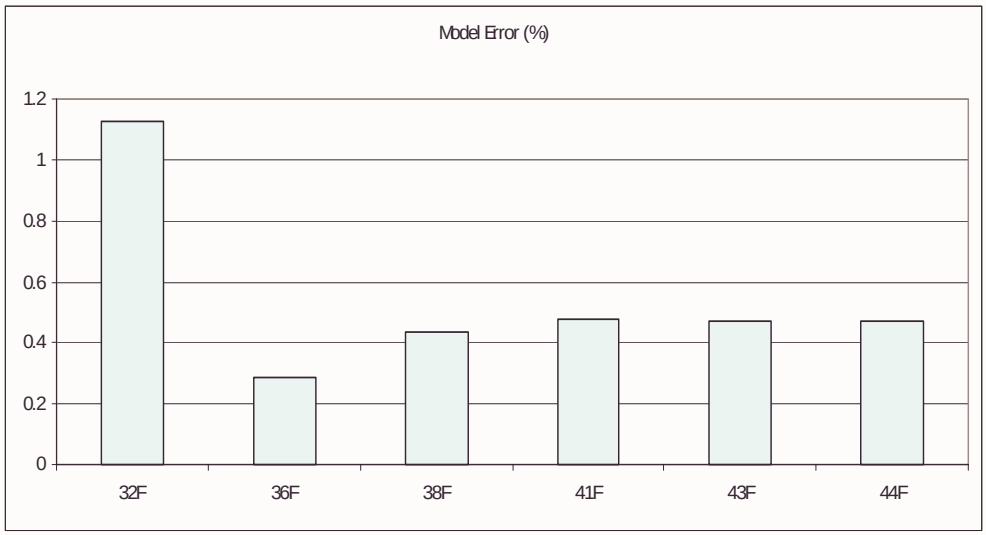
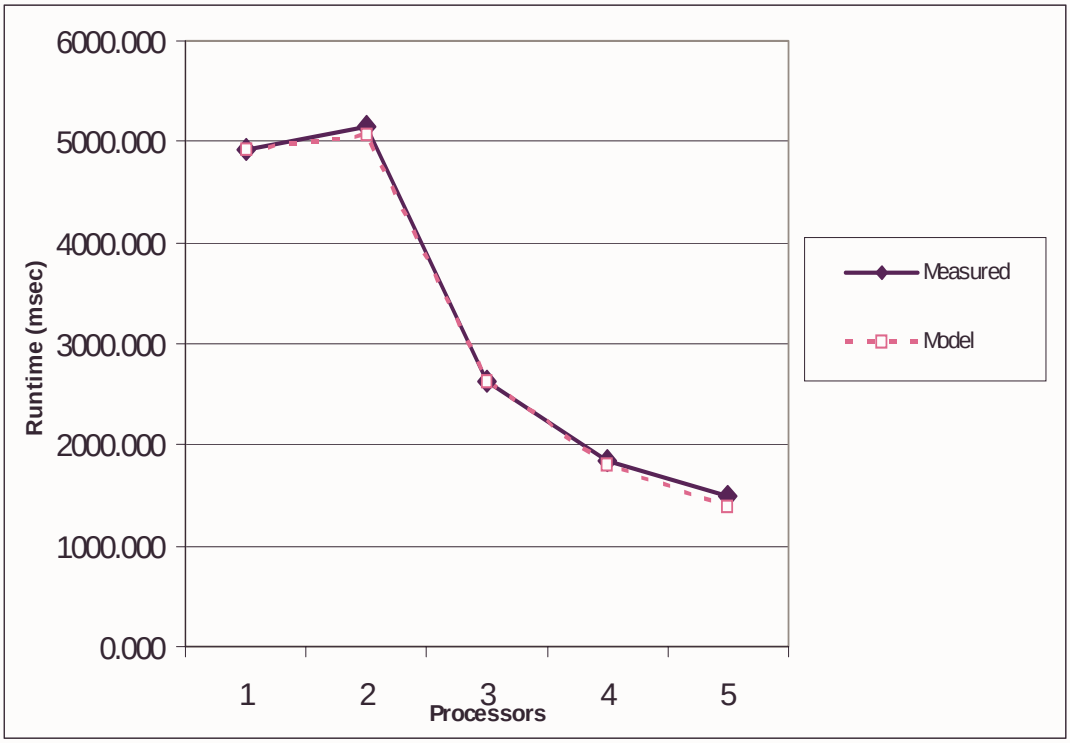


FIGURE 5.6 SAT Solver No-Load Imbalance Results

TABLE 5.11 MPI AES Algorithm No-load Imbalance Results

Processors	Measured Mean (msec)	Model Prediction (msec)	Model Error (%)	Standard Deviation σ	Error of Estimation (90% Confidence)
1	4932.578	4933.236	0.0132	8.9809	1.9071
2	5144.15	5075.157	1.3412	11.44897	2.4312
3	2629.1864	2620.52	0.3296	14.9268	3.1697
4	1849.672	1802.314	2.5603	28.6469	6.0832
5	1486.638	1393.217	6.284	26.5573	5.6394



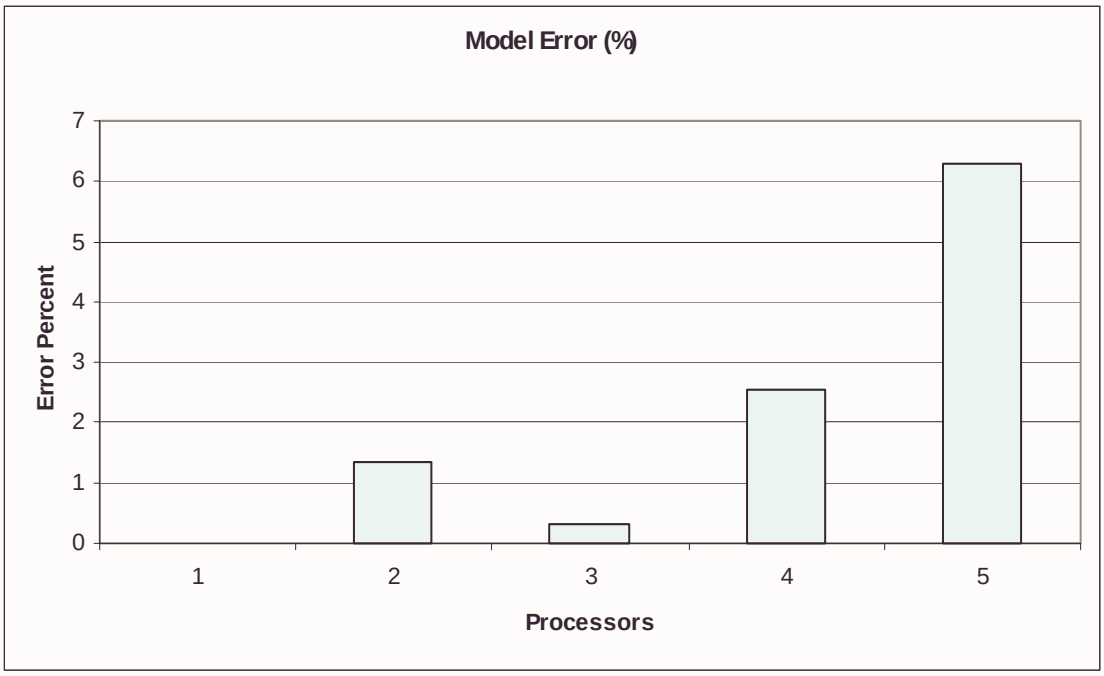


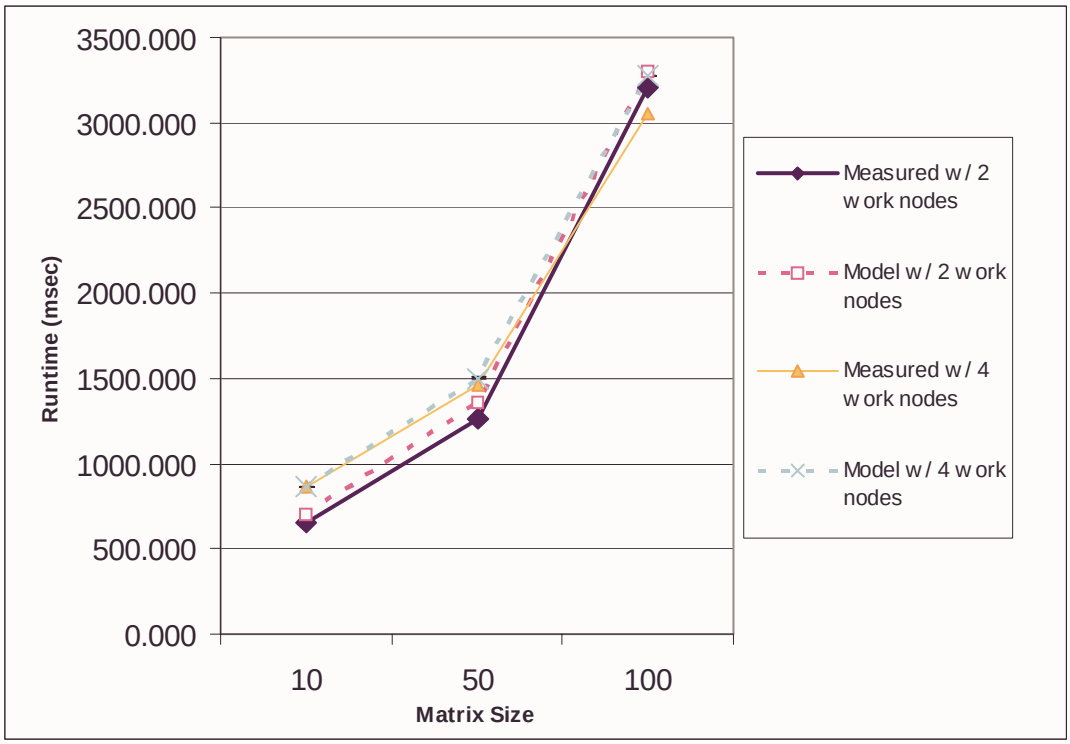
FIGURE 5.7 AES Algorithm No-Load Imbalance Results

TABLE 5.12 MPI Matrix Vector Multiplication Algorithm No-load Imbalance Results

Matrix Size (2 work nodes)	Measured Mean (msec)	Model Prediction (msec)	Model Error (%)	Standard Deviation σ	Error of Estimation (90% Confidence)
10	653.983	704.13	7.6679	169.5792	36.01
50	1261.883	1352.34	7.1684	37.3427	7.9297
100	3211.867	3299.18	2.7184	152.9048	32.4693
Matrix Size (4 work nodes)					
100	3058.283	3273.59	7.0401	129.4665	27.4922

Matrix Vector Multiplication. The Matrix Vector Multiplication algorithm has inherent application load imbalance when the number of rows is not integer divisible by the number of available machines. With this in mind, we must carefully consider our test cases to include only those with truly no application load imbalance. Again the master node does not participate in the algorithm calculation but acts as the orchestrator, deciding how many and which rows to send to each node. The rows of the matrix are divided and sent to the node and the command to start is given. The master node then waits the return of the resultant vector and if any rows are left, sends them back to the node. Once all the resultant vectors are collected the application ends unless more iterations are requested. Table 5.12 and Figure 5.8 show the results of these tests. Again, the model error is very low with small estimation error indicating the model predicts the measured results very accurately.

Now that we have confidence the model predicts the ideal case accurately, less than one percent for all cases over ten seconds and less than two percent for the cases less than ten seconds, we will focus on the addition of application load imbalance and test the model accuracy under these conditions.



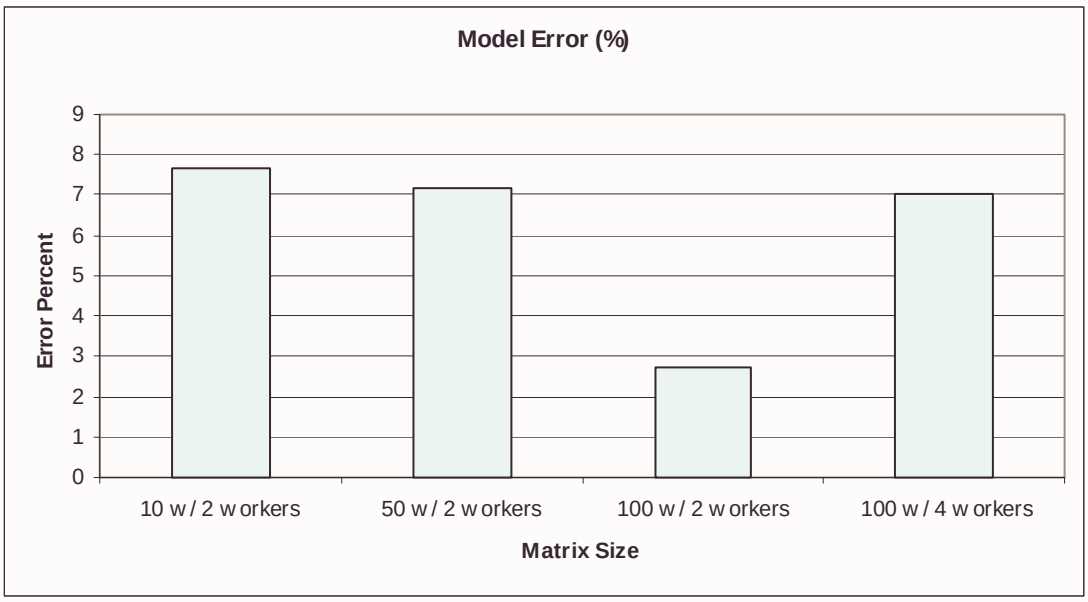


FIGURE 5.8 Matrix Vector Algorithm No-Load Imbalance Results

5.4.2 Application Load Imbalance Results

To validate the modeling methodology for application load imbalance, we look at the results from the SAT Solver application and the Matrix Vector Multiplication algorithm under application imbalance conditions.

SAT Solver. To demonstrate the model's validity and measure its accuracy for application load imbalance we have designed an experiment with the SAT solver where the seeds or starting points for the counters are not equally distributed in the design space causing an uneven distribution of the problem across the workstations, *m*. For this experiment we are looking for the worst case solution (i.e. the solution is all F's) and we purposely assign the node in the upper portion of the search space a larger portion of the problem as shown in Figure 5.9. This prescribing of the load makes the load distribution deterministic and thus easy to model while allowing use to test the model's accuracy.

For this case, the application load imbalance can be modeled as the ratio of the amount of work done on the heaviest loaded processor β_{max} to the average amount of work for a processor *B*. With no background load ($\gamma_j = 1$) the load imbalance from Eqn. 4.25 becomes

$$\eta = \frac{\beta_{max}}{B} \qquad \text{(EQ 5.2)}$$

To determine the load imbalance for the case where the application load or search space is unevenly distributed, we must determine the average load for each processor, *B*, and the amount of work for the most heavily loaded processor, β_{max}. Since for this experiment, we have prescribed

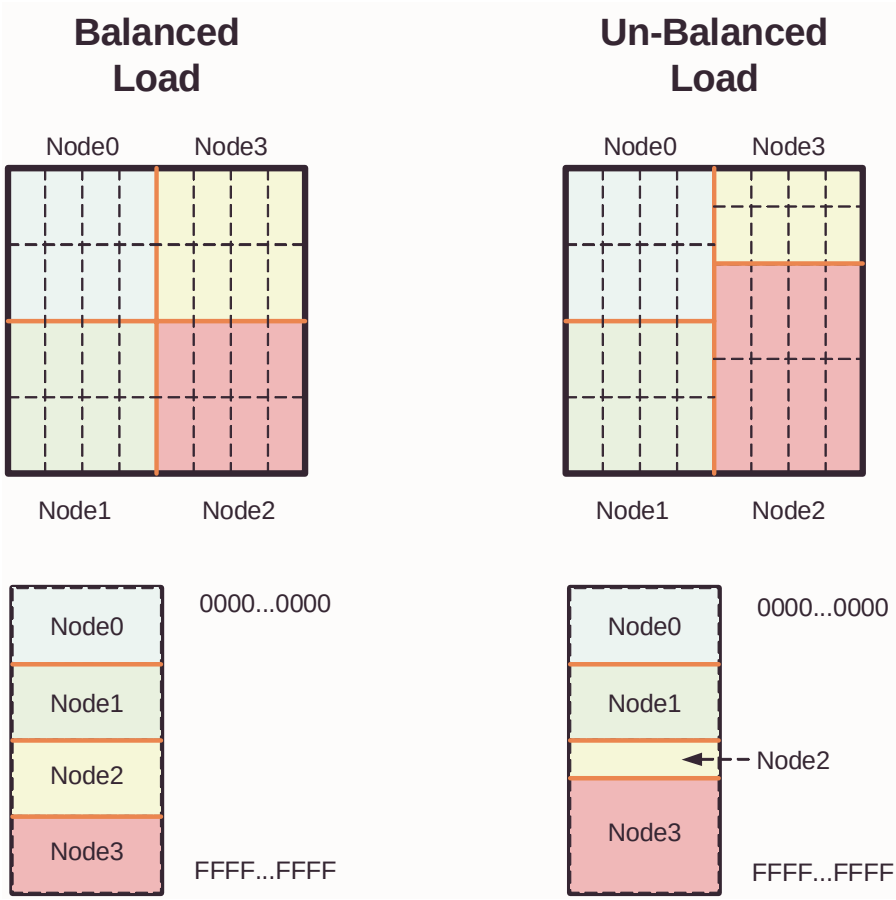


FIGURE 5.9 Application Load Distribution for SAT Solver Application

the distribution of the search space by issuing the seed values, we can easily find the values for β_{max} and B.

To find the average load for a processor B, we need to know the number of processors and the number of hardware engines per processor. In this case we have four processors (2-bits) and eight hardware engines per processor (3-bits). Thus we have a 5-bit seed. So the average work on a processor, B, would be the worst case value or 0xFFF...F minus the maximum seed value which for a 5-bit seed is 0xF80...0.

$$B = 0xFF\ddot{F}\ddot{.}\ldots F - 0xF80\ldots 0 = 0x07F\ldots F \quad \text{(EQ 5.3)}$$

To find the amount of work for the most heavily loaded processor, we also need the seed value for that processor. In this case, the seed is 0xBC0...0. Similar to finding the average load, we can now find β_{max}:

$$\beta_{max} = 0xFFF\ldots F - 0xBC0\ldots 0 = 0x43F\ldots F \quad \text{(EQ 5.4)}$$

Finally, the application load imbalance is simply the ratio of these values. Since there is no background load in this experiment, the load imbalance factor becomes:

$$\eta = \frac{\beta_{max}}{B} = \frac{0x43F\ldots F}{0x07F\ldots F} = 8.5 \quad \text{(EQ 5.5)}$$

The results of the application load imbalance only experiments are given in Table 5.13 and Figure 5.10. The model error is very low with small estimation error indicating the model predicts the measured results very accurately.

TABLE 5.13 MPI SAT Solver Application Load Imbalance Results

Problem Size (bits)	Measured Mean (sec)	Model Prediction (sec)	Model Error (%)	Standard Deviation σ	Error of Estimation (90% Confidence)
32	34.367	35.12672	2.21	0.0259	0.0062
36	546.58	549.7962	0.588	0.0156	0.0036
38	2732.023	2745.719	0.501	0.0294	0.0117
41	21855.31	21960.05	0.479	0.8952	0.6585
43	87417.6384	87837.74	0.4806	0.0602	0.0443

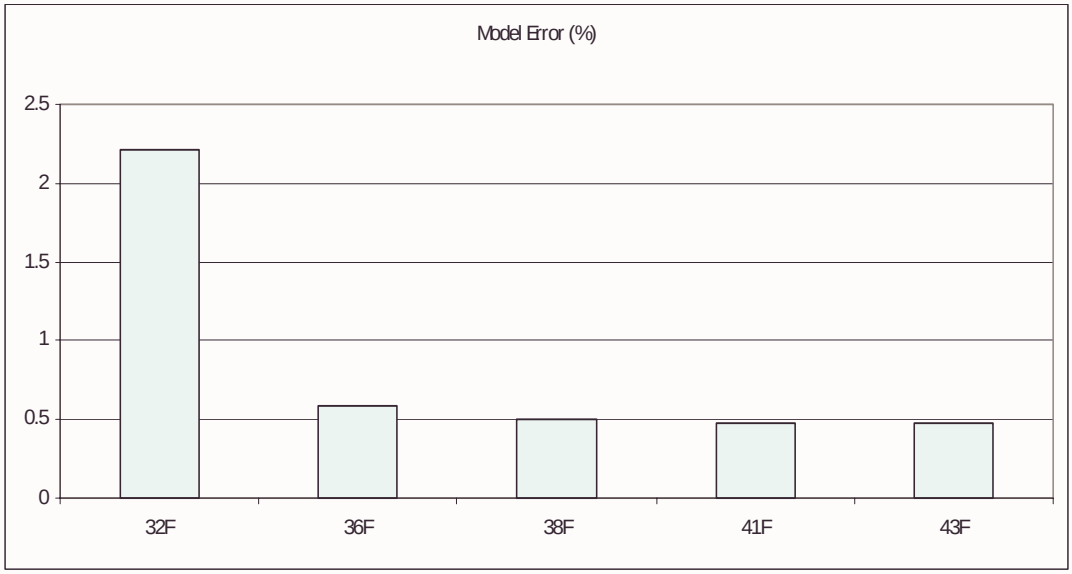


FIGURE 5.10 SAT Solver Application Load Imbalance Results

Matrix Vector Multiplication. The Matrix Vector Multiplication algorithm has inherent application load imbalance when the number of rows in not integer divisible by the number of processing nodes. To validate the application load imbalance modeling methodology, we now look at these cases.

Again, the application load imbalance can be modeled as the ratio of the amount of work done on the heaviest loaded processor β_{max} to the average amount of work for a processor *B*. With no background load ($\gamma_j = 1$) the load imbalance from Eqn. 4.25 becomes

$$\eta = \frac{\beta_{max}}{B} \quad \text{(EQ 5.6)}$$

To determine the load imbalance for the case where the application load or number of rows is unevenly distributed, we must determine the average load for each processor, *B*, and the amount of work for the most heavily loaded processor, β_{max}.

To find the average load for a processor *B*, we need to know the number of processors and the total number of rows in the matrix. The average work on a processor, *B*, is simply:

$$B = \frac{rows}{nodes} \quad \text{(EQ 5.7)}$$

To find the amount of work for the most heavily loaded processor, we need the number of rows that will be sent to the most heavily loaded processor. This is simply:

$$\beta_{max} = \left\lfloor \frac{rows}{nodes} \right\rfloor \quad \text{(EQ 5.8)}$$

TABLE 5.14 MPI Matrix Vector Algorithm Application Load Imbalance Results

Matrix Size (4 work nodes)	Measured Average (msec)	Model Prediction (msec)	Model Error (%)	Standard Deviation σ	Error of Estimation (90% Confidence)
10	871.4	862.465	1.0254	32.9736	7.0019
50	1459.183	1503.617	3.0451	765.2082	162.4917

Finally, the application load imbalance is simply the ratio of these values. The results of the application load imbalance only experiments are given in Table 5.14 and Figure 5.11. The model error is very low with small estimation error indicating the model predicts the measured results very accurately.

We therefore conclude that the performance modeling methodology accurately describes the performance of fork-join and more specifically, SIAs that suffer from application load imbalance with less than one percent error for all cases whose runtime is longer than thirty seconds and less than five percent for the cases less than thirty seconds. We now turn our focus to the background load imbalance effects that arise from shared use of our resources.

5.4.3 Background Load Imbalance Results

To demonstrate the model's accuracy for predicting the performance in the presence of background load, we compare the predicted performance with the empirical results of the application running with a synthetic background load on the homogeneous set of pilchard workstations. By adding synthetic background load to the workstations, we can investigate the model accuracy under various loading conditions and types of background loading. The workstations are generally idle ($\gamma = 1$) enabling us to predictably control the background loading with a synthetic load and investigate the effects of various background loading conditions under controlled circumstances.

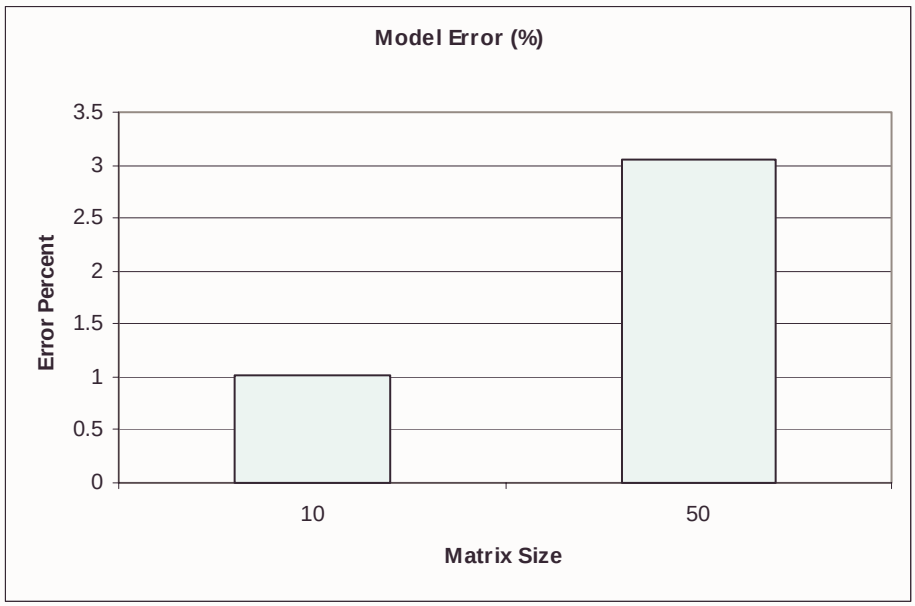


FIGURE 5.11 Matrix Vector Algorithm Application Load Imbalance Results

TABLE 5.15 Typical factors for moderately Loaded Homogeneous Nodes Using PS Model

Arrival Rate	Number of Processors	Background Load Imbalance Factor
λ	**P**	γ
0.1	2	1.16
	3	1.23
	4	1.30
	5	1.36
0.2	2	1.34
	3	1.47
	4	1.59
	5	1.70
0.4	2	1.78
	3	2.04
	4	2.25
	5	2.42

In Sec. 5.4.1 we considered the model for the case of no background loading and are now confident the model is accurate for the no-load case (no background or application load). Now we will consider the impact of background loading using a simple case. As discussed in Chap. 4 and in detail in [102], the background load imbalance can be modeled using Processor Sharing (PS) servers or a FCFS M/G/1 queuing model to allow general service distributions (we will concentrate on the PS server model). When computing the η terms with Eqn. 4.29, the summation includes an infinite number of terms. To compute η, we assume the expression converges to a solution which is true for the PS case.

To find the arrival rate of background tasks, we measure the arrivals when the applications are run (with a synthetic load we know a priori the arrival rate). For various arrival rates, the performance model predictions for γ are summarized in Table 5.15.

SAT Solver. With synthetic background load running on the shared, homogeneous HPRC resources, experiments were conducted with the SAT Solver application. Table 5.16, Table 5.17, and Figure 5.12 show the results of those experiments. The model error is very low for the cases where the runtime is long. For the shorter runtime cases, the variance in overhead is a large percentage of the error. Still, we have a small estimation error, indicating the model remains a suitable predictor of the system behavior.

Matrix Vector Multiplication. Again with a synthetic background load running on the shared, homogeneous HPRC resources, we look at experiments running the matrix vector multiplication algorithm. Table 5.18 and Figure 5.13 show the results of those experiments. In these results we see the model error still low in most cases but the standard deviation is significantly higher. The two sources for these errors are overhead variation dominating the runtime since these runtimes are very short and variations in the transfer of the matrix data from the file into memory. The latter is evident where for a given gamma, we see that the error tends to increase as the matrix size (or data size) gets larger. These theories will be discussed further in the conclusions chapter.

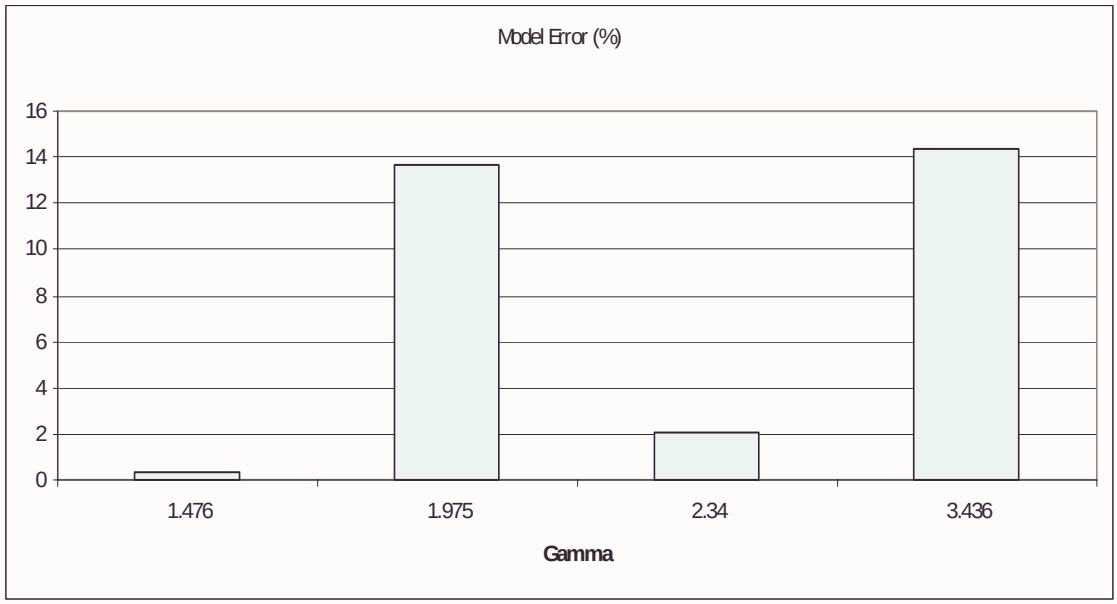
AES Algorithm. Finally, with a synthetic background load running on the shared, homogeneous HPRC resources, we look at experiments running the AES encryption algorithm. Table 5.19 and Figure 5.14 show the results of those experiments. In these results we see the model error low in with a reasonable standard deviation. The AES algorithm has short runtimes and is affected by the overhead variation dominating the runtime. The other cause for error pointed out in the previous discussions is not as significant a factor for this algorithm since the amount of data transfer is much smaller. Again, we will discuss these theories further in the conclusions chapter.

TABLE 5.16 MPI SAT Solver Background Load Imbalance Results (Problem Size 32F)

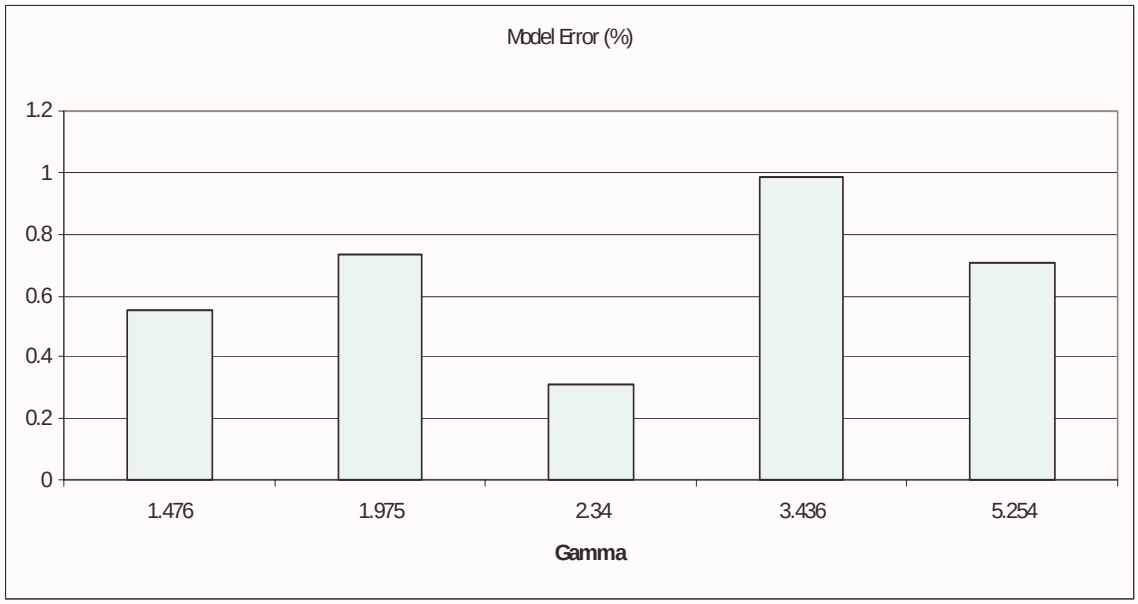
γ	Measured Mean (sec)	Model Prediction (sec)	Model Error (%)	Standard Deviation σ	Error of Estimation (90% Confidence)
1.476	4.36	4.34	0.3058	0.1821	0.0424
1.975	5.158	4.453	13.671	0.6122	0.1869
2.34	4.623	4.5286	2.0454	0.0972	0.0302
3.436	5.555	4.7597	14.3181	0.6476	0.1945
5.254	5.9129	5.141	13.050	0.3457	0.1075

TABLE 5.17 MPI SAT Solver Background Load Imbalance Results (Problem Size 36F)

γ	Measured Mean (sec)	Model Prediction (sec)	Model Error (%)	Standard Deviation σ	Error of Estimation (90% Confidence)
1.476	64.541	64.897	0.5514	0.240	0.0475
1.975	65.484	65.022	0.7358	0.5804	0.1773
2.34	64.877	65.077	0.3095	0.143	0.0444
3.436	65.960	65.309	0.9871	0.6863	0.2060
5.254	66.1586	65.6906	0.7073	0.272	0.0846



Problem Size 32F



Problem Size 36F

FIGURE 5.12 SAT Solver Background Load Imbalance Results

TABLE 5.18 MPI Matrix Vector Algorithm Background Load Imbalance Results

γ	Matrix Size	Measured Mean (msec)	Model Prediction (msec)	Model Error (%)	Standard Deviation σ	Error of Estimation (90% Confidence)
1.265	10	911.867	953.897	4.609	615.72	130.748
	50	1520	1649.178	8.499	385.1265	81.781
	100	3741.35	3741.677	0.00874	37.859	8.0394
1.588	10	1247.883	1235.229	1.014	1149.542	244.105
	50	2159.3	1983.53	8.1401	1523.801	323.5785
	100	4472.483	4240.096	5.1959	2405.007	510.7025
1.836	10	1338.817	1443.656	7.8307	824.6266	175.1092
	50	2267.85	2231.237	1.6144	1241.049	263.5363
	100	4561.117	4609.353	1.0576	1990.197	422.6177
2.618	10	2126.717	2085.594	1.9336	1958.295	415.843
	50	3128.7	2994.154	4.3	2528.062	536.833
	100	6077.8	5746.635	5.4488	3831.283	813.572
3.923	10	3227.65	3146.918	2.5012	1018.402	216.2573
	50	4776.533	4255.493	10.9083	1630.588	346.2547
	100	8612.25	7626.916	11.44108	2702.773	573.9329

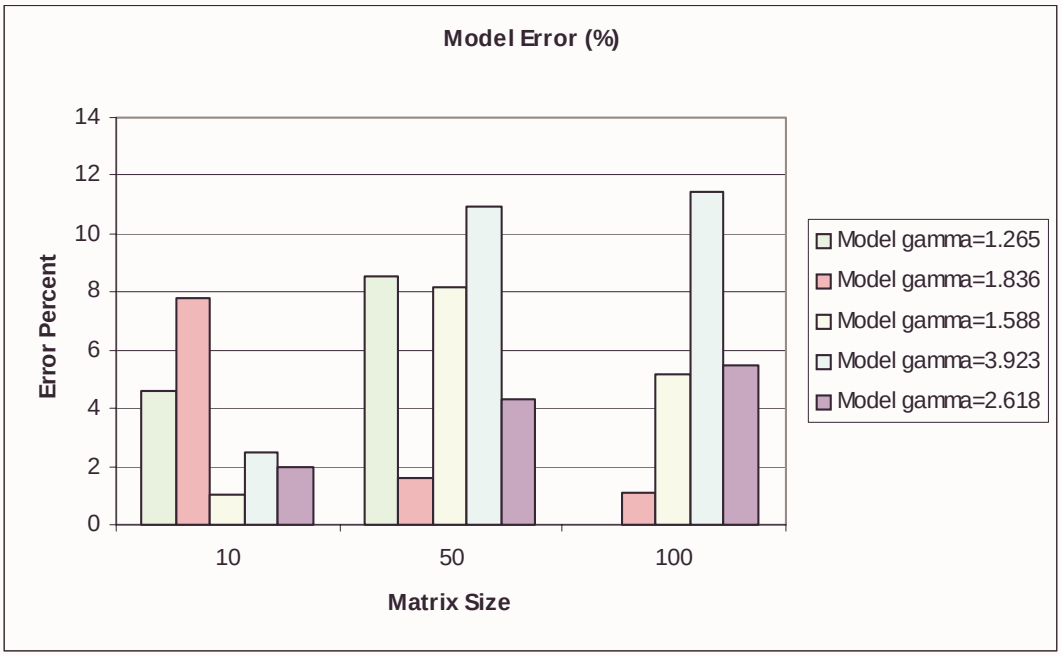


FIGURE 5.13 Matrix Vector Algorithm Background Load Imbalance Results

TABLE 5.19 MPI AES Encryption Algorithm Background Load Imbalance Results

λ	γ	Nodes	Measured Mean (msec)	Model Prediction (msec)	Model Error (%)	Standard Deviation σ	Error of Estimation (90% Confidence)
0.1	1.16	2	5136.763	5171.825	0.6826	73.0905	15.5207
	1.23	3	2637.051	2709.629	2.7522	32.653	6.934
	1.3	4	1868.220	1895.23	1.4457	94.676	20.104
	1.36	5	1487.593	1492.262	0.3138	21.09	4.4785
0.1	1.06	2	5128.6167	5110.994	0.3436	64.1132	13.6144
	1.09	3	2639.6	2654.268	0.5557	36.584	7.7687
	1.12	4	1859.983	1838.246	1.1687	41.085	8.7243
	1.14	5	1490.367	1432.308	3.8956	24.818	5.27
0.2	1.34	2	5176.607	5279.189	1.982	100.308	21.3
	1.47	3	2647	2802.957	5.8918	46.53	9.881
	1.59	4	1862.633	1987.194	6.687	36.441	7.738
	1.70	5	1503.047	1585.11	5.4598	24.738	5.253
0.2	1.12	2	5138.833	5148.045	0.1783	113.12	21.021
	1.18	3	2651.983	2688.246	1.367	68.958	14.643
	1.23	4	1858.35	1873.48	0.8142	46.03	9.774
	1.28	5	1502.55	1469.646	2.1899	65.2	13.845

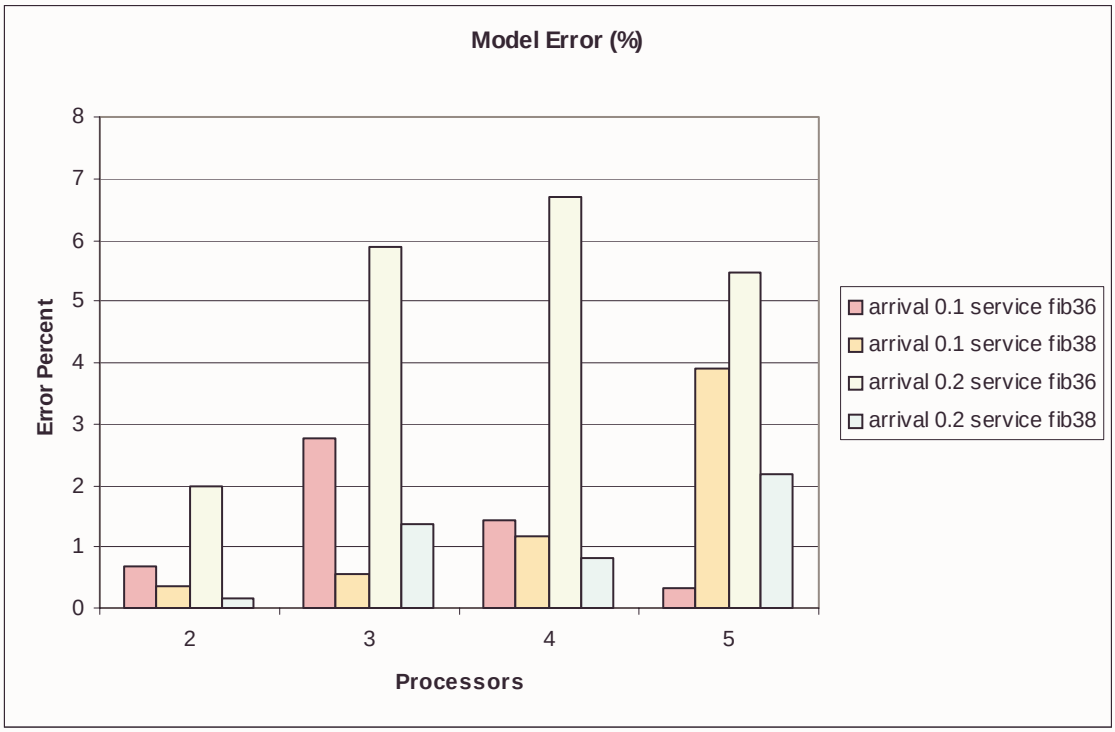


FIGURE 5.14 AES Algorithm Background Load Imbalance Results

TABLE 5.20 MPI SAT Solver Application and Background Load Imbalance Results

Problem Size 32 Bits					
γ	**Measured Mean (sec)**	**Model Prediction (sec)**	**Model Error (%)**	**Standard Deviation σ**	**Error of Estimation (90% Confidence)**
1.476	34.4703	34.6227	0.442	0.2	0.0466
1.975	35.545	34.7275	2.3	0.269	0.868
2.336	34.6727	34.803	0.3764	0.1436	0.0542
3.436	35.274	35.034	0.68	0.4088	0.1249
5.254	35.6406	35.416	0.6301	0.3438	0.0709

By using our three applications running on homogeneous HPRC resources, we conclude the performance modeling methodology adequately represents the impact of background load on the runtime performance with less than one percent error for all tested cases over thirty seconds and less than fifteen percent for the cases less than thirty seconds. Experiments with these applications in conjunction with synthetic background load validate the modeling methodology and indicate the processor sharing model is adequate for predicting background load under various loading conditions.

5.4.4 Application and Background Load Imbalance Results

Having validated the accuracy of the application and background load imbalance models independently, we now investigate their interaction. We use the SAT Solver application and the Matrix Vector Multiplication Algorithm to run validation experiments on a homogeneous set of HPRC resources.

SAT Solver. We conducted experiments on the homogeneous HPRC cluster running the SAT Solver application with application load imbalance and synthetic background load. Table 5.20,

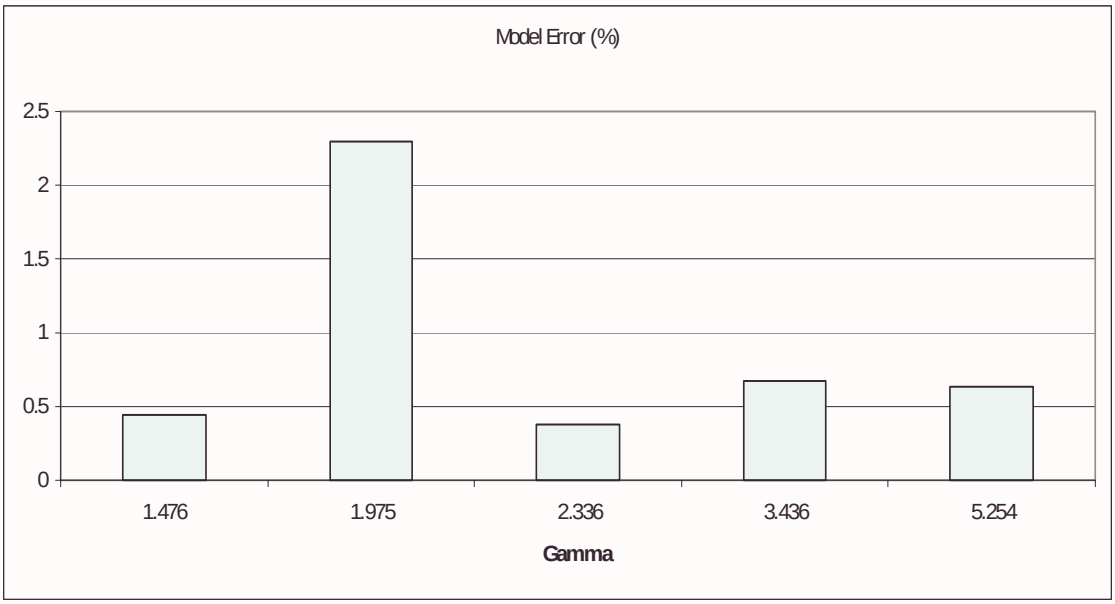
TABLE 5.21 MPI SAT Solver Application and Background Load Imbalance Results

Problem Size 36 Bits					
γ	**Measured Mean (sec)**	**Model Prediction (sec)**	**Model Error (%)**	**Standard Deviation σ**	**Error of Estimation (90% Confidence)**
1.476	546.671	549.2922	0.4794	0.1781	0.0414
1.975	547.495	549.397	0.3473	0.572	0.718
2.336	546.8605	549.4727	0.4777	0.143	0.054
3.436	547.95	549.704	0.32	0.51	0.156
5.254	548.192	550.0855	0.3454	0.4107	0.135

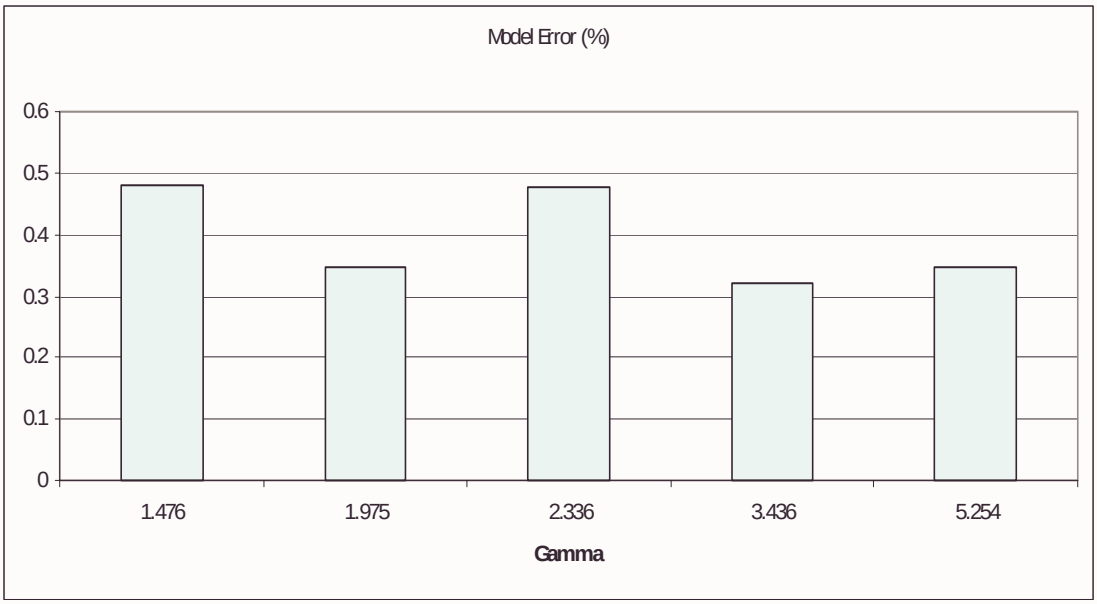
Table 5.21, and Figure 5.15 show the results of those experiments. The model error is very low with small estimation error indicating the model predicts the measured results very accurately.

Matrix Vector Multiplication. The results of experiments conducted on the homogeneous HPRC cluster running the Matrix Vector Multiplication Algorithm with application load imbalance and synthetic background load are shown in Table 5.22 and Figure 5.16. As with previous results with this application, we see the model error low in most cases but the standard deviation is high. The two sources for these errors are overhead variation dominating the runtime since these runtimes are very short and variations in the transfer of the matrix data from the file into memory. These theories will be discussed further in the conclusions chapter.

By using our three applications running on homogeneous HPRC resources, we conclude the performance modeling methodology adequately represents the impact of the combination of application and background load imbalances on the runtime performance with less than five percent error for all cases over thirty seconds and less than fifteen percent for the case less than thirty seconds.



Problem Size 32F



Problem Size 36F

FIGURE 5.15 SAT Solver Application and Background Load Imbalance Results

TABLE 5.22 MPI Matrix Vector Algorithm Application and Background Load Imbalance Results

γ	Matrix Size	Measured Mean (msec)	Model Prediction (msec)	Model Error (%)	Standard Deviation σ	Error of Estimation (90% Confidence)
1.476	10	1250.917	1320.065	5.528	816.565	173.397
	50	1751.167	1998.039	14.0976	804.3814	170.810
1.975	10	1889.367	1769.945	6.321	1608.169	341.494
	50	2530.383	2484.12	1.828	1831.339	388.884
2.336	10	1894.783	2089.602	10.2818	1191.69	253.055
	50	2510.733	2829.499	12.696	1274.953	270.736
3.436	10	3097.783	3064.163	1.085	2641.673	560.958
	50	3829.967	3882.481	1.371	2752.885	584.574
5.254	10	5070.3	4676.042	7.776	964.4059	204.791
	50	6003.9	5624.064	6.326	1022.148	217.053

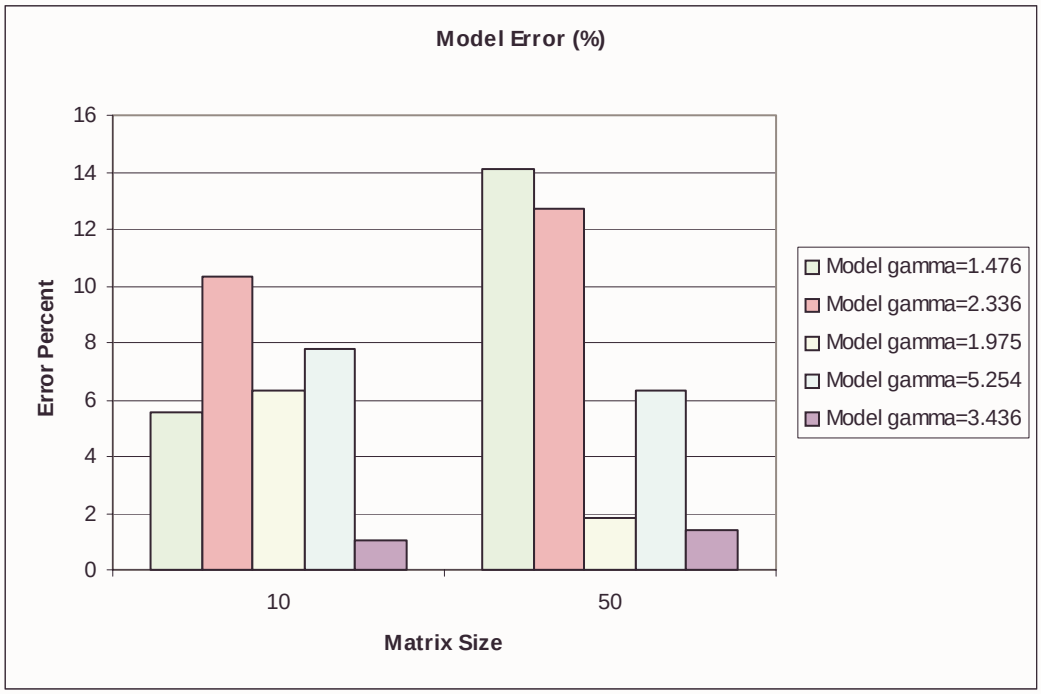


FIGURE 5.16 Matrix Vector Algorithm Application and Background Load Imbalance Results

TABLE 5.23 MPI SAT Solver Heterogeneous Resources Results

Problem Size	Measured Mean (sec)	Model Prediction (sec)	Model Error (%)	Standard Deviation σ	Error of Estimation (90% Confidence)
32F	4.674	4.5335	3.006	1.387	0.41
36F	65.4674	65.0829	0.5874	0.416	0.093

5.4.5 Heterogeneity Results

Having validated the model for various application and background loading conditions we now look at the effect of heterogeneous resources on the model. Since a cluster of heterogeneous HPRC nodes is not available at this time, we will simulate the effect of different processor speeds by varying the synthetic background load across the available nodes by varying the arrival rate of background processes. These experiments were conducted for each of our three applications.

SAT Solver. For the SAT Solver application, we varied the arrival rate of background tasks across the four nodes and there was no application load imbalance. The results of these test are show in Table 5.23 and Figure 5.17.

Matrix Vector Multiplication. For the Matrix Vector Multiplication algorithm, we again varied the arrival rate of background tasks across the nodes in both cases (2 workers and 4 workers). There was application load imbalance for matrix sizes 10x10 and 50x50 for 4 worker nodes. The results of these test are show in Table 5.24 and Figure 5.18.

AES Algorithm. For the SAT Solver application, we varied the arrival rate of background tasks across the four nodes and there was no application load imbalance. The results of these test are show in Table 5.25 and Figure 5.19.

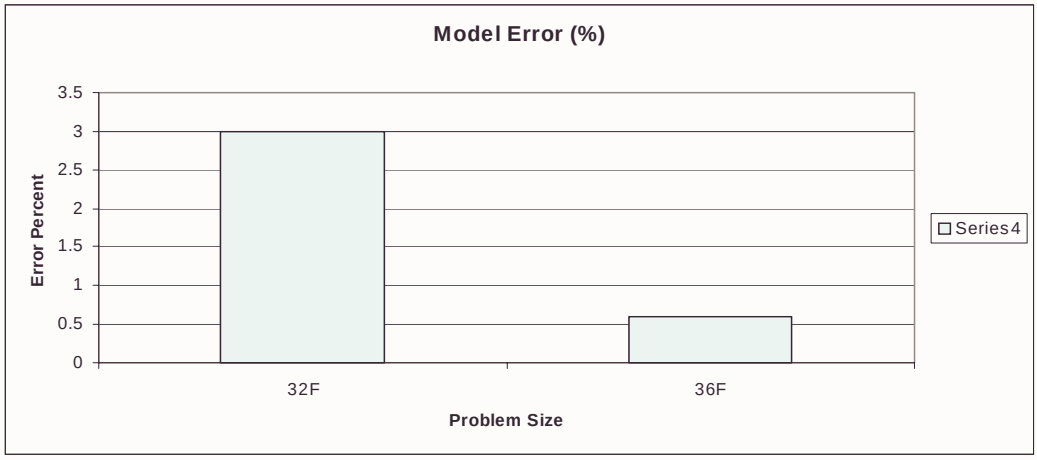


FIGURE 5.17 SAT Solver Heterogeneous Resources Results

TABLE 5.24 MPI Matrix Vector Algorithm Heterogeneous Resources Results

Matrix Size	Measured Average (msec)	Model Prediction (msec)	Model Error (%)	Standard Deviation σ	Error of Estimation (90% Confidence)
10x10 on 2 nodes	1565.333	1511.286	3.453	1104.1	234.455
50x50 on 2 nodes	2091.16	2311.612	10.542	1077.049	250.540
100x100 on 2 nodes	4653.05	4729.168	1.6359	1849.306	392.699
10x10 on 4 nodes	1561.4	1637.545	4.8767	824.251	214.366
50x50 on 4 nodes	2426.55	2370.063	2.3279	1118.85	290.984
100x100 on 4 nodes	4314.325	4407.567	2.1612	1256.559	326.799

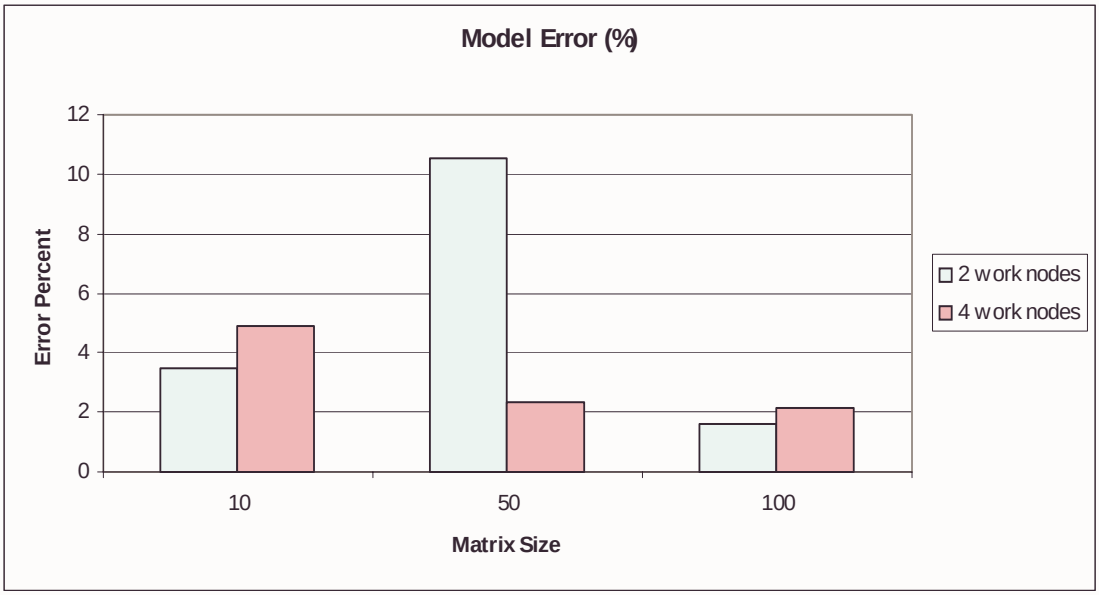


FIGURE 5.18 Matrix Vector Algorithm Heterogeneous Resources Results

TABLE 5.25 MPI AES Encryption Algorithm Heterogeneous Resources Results

Processors	Measured Average (msec)	Model Prediction (msec)	Model Error (%)	Standard Deviation σ	Error of Estimation (90% Confidence)
2	5249.8	5339.106	1.701	120.016	31.213
3	3180.58	3090.333	2.8374	1024.054	238.213
4	2521.275	2227.373	11.657	666.88	173.438
5	1889.875	1780.229	5.802	431.658	112.263

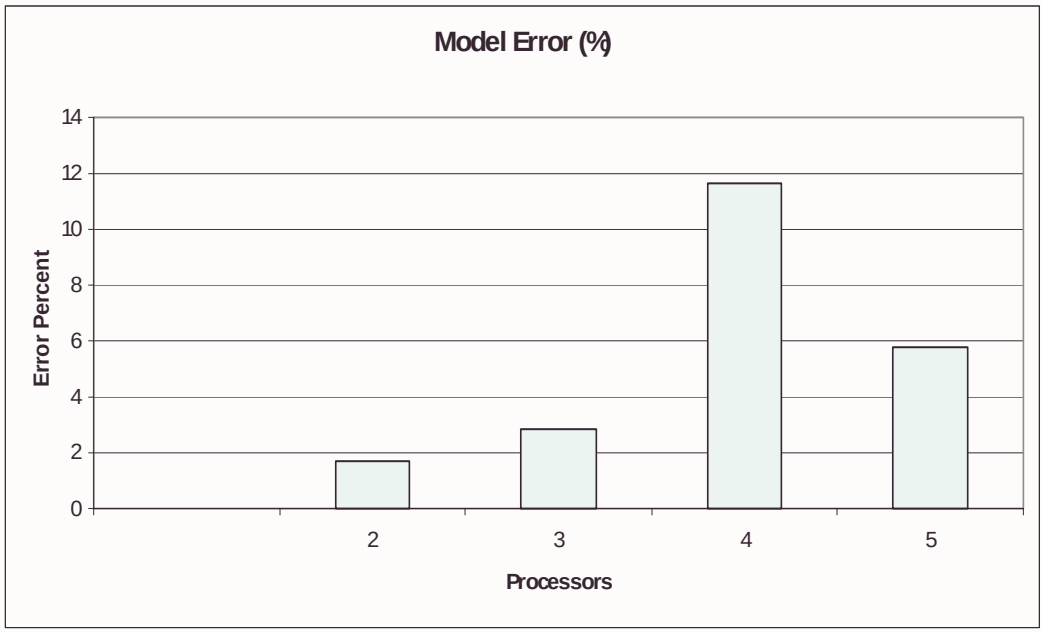


FIGURE 5.19 AES Encryption Algorithm Heterogeneous Resources Results

The results from the heterogeneous resources experiments are consistent with our other validation results. The errors are low (less than one percent for runtimes greater than thirty seconds and less than fifteen percent for runtimes less than thirty seconds) with some high standard deviation values on the Matrix Vector Multiplication and AES Encryption algorithms. These errors are consistent with those found earlier and can be attributed to the setup overhead variations and file access.

In this chapter we have investigated the accuracy of the modeling methodology in characterizing the performance of applications running on shared, homogeneous and heterogeneous HPRC resources and found the model error in all cases to be less than five percent for application runtimes greater than thirty seconds and less than fifteen percent for runtimes less than thirty seconds. To better understand and completely validate the model, each component of the modeling

methodology has been investigated separately, followed by the validation of the interaction of the modeling components. Beginning with all three applications, we validated the model under no load imbalance conditions on homogeneous HPRC resources. Next, we used the SAT Solver application and the Matrix Vector Multiplication algorithm to show the accuracy of the methodology in modeling application load imbalance. We then showed the accuracy of the modeling methodology for shared HPRC resources with synthetic background loads using all three applications. Finally, we validated the modeling methodology for the interaction of application and background load imbalance on both homogeneous and heterogeneous HPRC resources. Therefore, we conclude the performance modeling methodology is accurate for SIAs running on shared, heterogeneous HPRC resources.

CHAPTER 6

APPLICATION OF MODEL

Having developed and validated a performance modeling methodology for algorithms running on shared HPRC resources we now look at potential applications of the model. By applying the modeling methodology to problems such as optimizing resource usage, maximizing efficiency, minimizing runtime, and characterizing trade-offs we can make resource decisions based on quantitative analysis of the different approaches. In this chapter we consider some of the optimization problems that can be addressed with our modeling methodology.

6.1 Application Scheduling

The performance of a parallel algorithm depends not only on the performance characteristics of the resources but also on the scheduling of tasks and the load balance between nodes. The *scheduling problem* has been well studied in the literature [36, 57]. 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) [57]. 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.

In this work, we focus on static scheduling of applications that follow a generalized Master-Worker paradigm (also called *task farming*). The generalized Master-Worker paradigm is very easy to program and is especially attractive for grid platforms. In grid platforms, the availability of resources is typically in a state of flux and worker tasks in a Master-Worker paradigm can be easily re-assigned as needed when resource availability changes. Furthermore, many scientific and computationally intensive problems can be mapped naturally to this paradigm: N-body simulations [65], genetic algorithms [32], Monte Carlo simulations [25], parameter-space searches, as well as the three applications used in the validation of our model. In scheduling Master-Worker applications, there are two main challenges: 1) How may workers should be allocated to the application? and 2) How to assign tasks to the workers? We would like to make the most efficient use of our resources while at the same time minimizing our runtime.

In order to effectively utilize our shared resources, we will employ a usage policy with a measurable goal or goals. Specifically, our objective is to choose an appropriate set of workstations on which to execute the given parallel application that best meets our usage policy goals. The scheduling decisions must account for the individual processor performance characteristics as well as the existing background load. The desired usage policy is based on the relative importance of parallel applications to the existing background load (priority) and individual workstation characteristics such as processing power, type of or lack of reconfigurable hardware, current workload, or other factors.

To implement the desired usage policy into a schedule for the parallel application, we build a cost function that represents the policy goals. Using optimization techniques to minimize

that cost function, we find a schedule for the parallel application that will most effectively utilize the available resources. It is well known that the optimization of a general cost function is an NP-hard problem [61], therefore true optimization is limited to restricted classes of problems which have efficient solutions or the use of heuristics to fine near-optimal solutions.

6.1.1 Minimizing Runtime

To determine the schedule with the minimum runtime for homogeneous HPRC resources, we use the algorithms shown in Figure 6.1 [102]. First the workstations are sorted based on the arrival rate of background jobs. Then the set of processors for the schedule is selected to contain only the workstation with the smallest background task arrival rate. The runtime for the application is calculated for this processor set and the next workstation is added in sorted order, repeating the runtime calculation until the added workstation does not improve the runtime. This algorithm [102], similar to the one described in [23], gives the optimal solution for minimum runtime only when there is a minimum runtime. This is always true for SIAs because such algorithms require a barrier synchronization after each iteration. Proof of the optimal solution is given in [23].

```
Sort available processors S by background job arrival rate λ
P ← {j}, where j is processor in S with lowest arrival rate
while not done do
    P' ← P ∪ {k}, where k is next processor in S
    if R_P' < R_P
        P ← P'
    else
        done ← TRUE
endwhile
return optimal set P
```

FIGURE 6.1 Algorithm for Minimum Runtime on Homogeneous Resources [102]

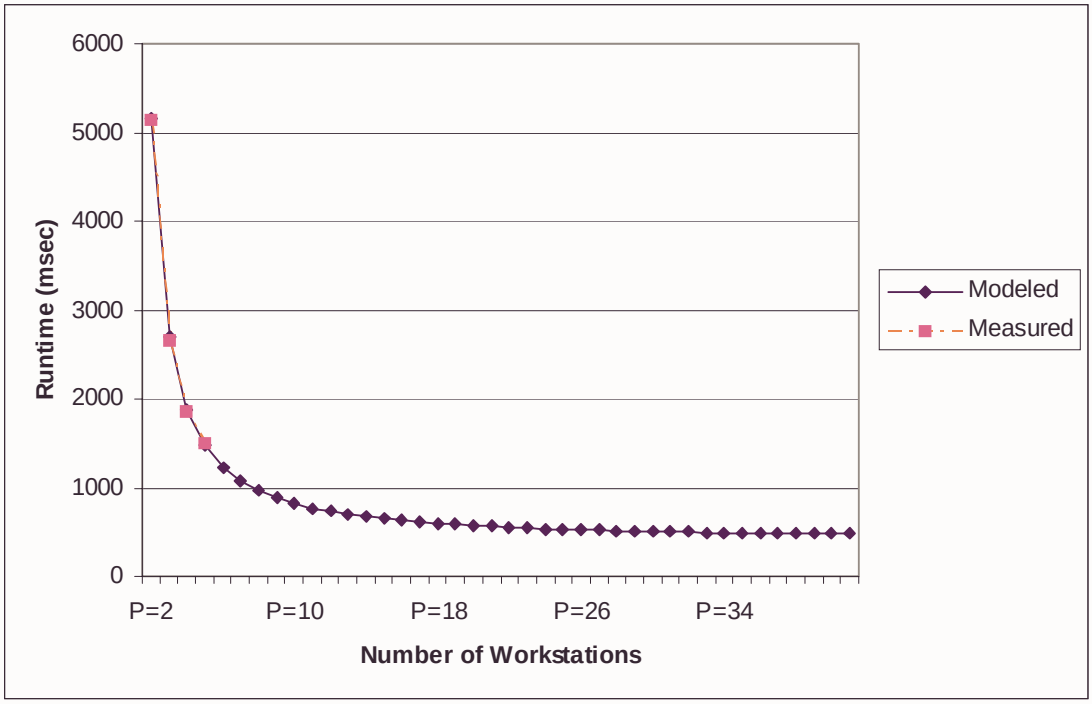


FIGURE 6.2 Optimum Set of Homogeneous Resources for AES Algorithm

Homogeneous Resources. For the AES algorithm, we see in Figure 6.2 that the runtime continues to improve from adding homogeneous workstations until we are using 25 processors (in this case, we assume the same background loading model at each workstation, with $\lambda = 0.2$). As more nodes are added, the additional overheads and background loads increase the total execution time and we get diminishing returns as we continue to add nodes to the workstation set. Our measured results are limited by the size of our development platform which was 6 nodes.

Again using the AES algorithm, we study the results of a non-sorted set of workstations versus three cases where the workstations were sorted based on arrival rate λ. In the non-sorted set of workstations, the arrival rates for the five homogeneous workstations range from 0.2 to 0.8. In Figure 6.3, we see that if the five workstations are sorted based on background arrival rates, and we choose a set with the lowest arrival rates, the runtime is significantly better than the case that

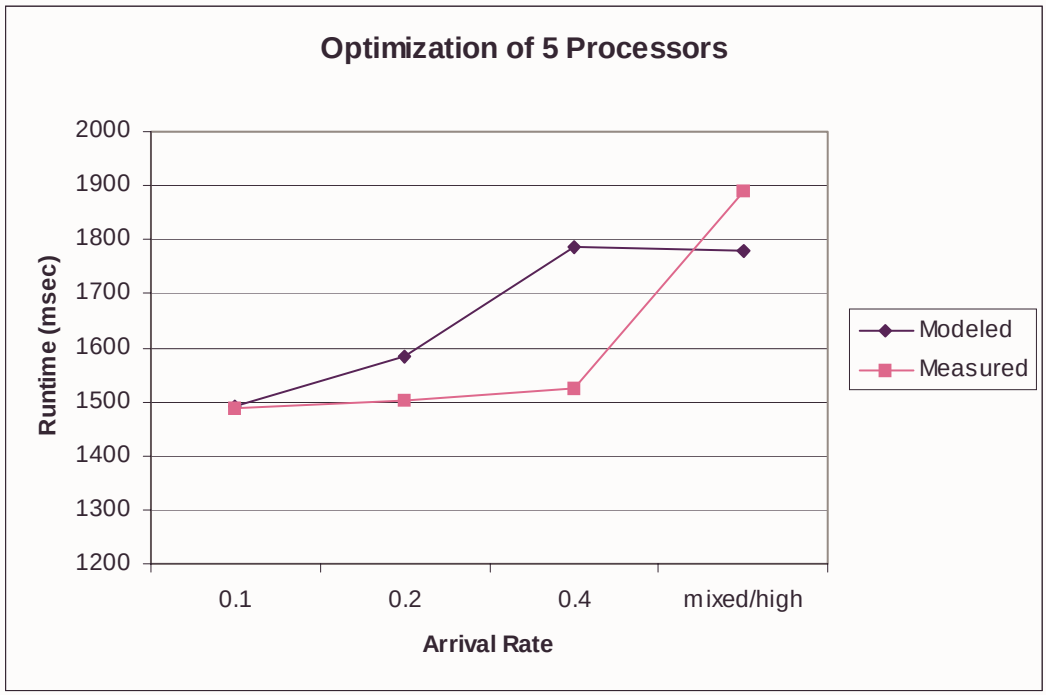


FIGURE 6.3 Optimum Set of 5 Homogeneous Resources, Mixed Background Arrival Rates

includes the higher arrival rates. As shown in the figure, the runtime steadily increases with the corresponding increase in background arrival rate from 0.1 to 0.4 (all workstations in the set having the same arrival rate) confirming that the optimum set of workstations consists of those with the lightest background load. For the case where our background loads are mixed (0.1, 0.2, and 0.4), we see that the runtime for the application is affected and dominated by the workstation with the heaviest load.

Next we look at the SAT Solver Application and investigate how changing the number of hardware engines per workstation and speed of the hardware will affect the application runtime. As shown in Figure 6.4, as one would expect, increasing the number of hardware engines copies per workstation reduces the application runtime. However, as the total number of workstations increases, the three cases (4, 8, and 16 copies) approach one another and the performance advan-

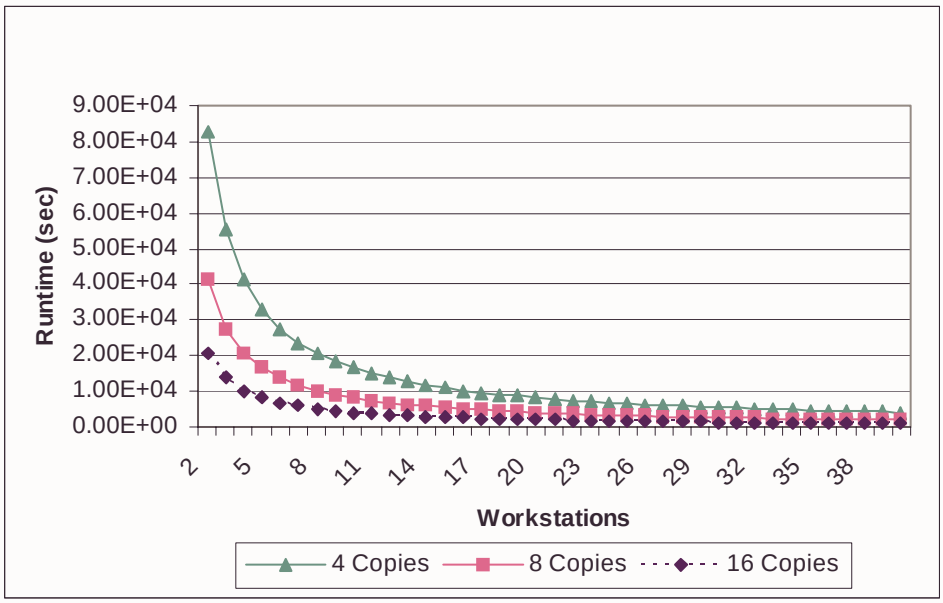


FIGURE 6.4 Optimum Set of Homogeneous Resources for SAT Solver: Compare number of Hardware Copies

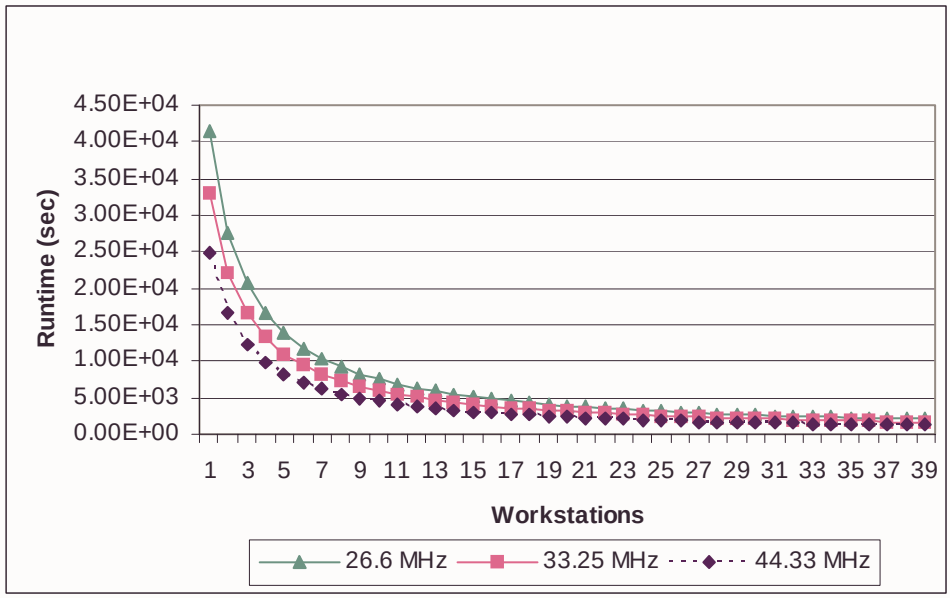


FIGURE 6.5 Optimum Set of Homogeneous Resources for SAT Solver: Compare Hardware Speed

tage of extra hardware copies is not as significant. In Figure 6.5, we again see that increasing the speed of the hardware engines reduces the application runtime. However, as the total number of workstations increases, the three cases approach one another and the performance advantage of faster hardware is not as significant.

Heterogeneous Resources. For applications running on heterogeneous resources, the performance model and algorithm are more complex and we must resort to a greedy heuristic to find a near optimal solution since an efficient optimal solution does not exist [102]. Greedy heuristics make the best choice for the current state even though it may not be the best global choice. While greedy heuristics can get stuck in local extrema, they are widely used for their speed and ease of implementation.

Sort available processors S by $\delta_j/[\omega(1 - \rho_j)]$, the scale factor for processor j
$P \leftarrow \{j\}$, where j is first processor in S
$P \leftarrow \{j+1\}$, where j is first processor in S
while not done **do**
    $P' \leftarrow P \cup \{k\}$, where k is next processor in S
    if $R_{P'} < R_P$
        $P \leftarrow P'$
    else
        done ← TRUE
endwhile

FIGURE 6.6 Greedy Heuristic for Minimum Runtime on Heterogeneous Resources [102]

If we assume that the parallel application is divided equally among the processors, then we only need to consider the background load and heterogeneity when determining the optimum set of processors. From the model development in Chap. 4, the background load is modeled as a processor sharing queueing model, so the expected number of background tasks at processor j is $\rho_j/(1 - \rho_j)$, assuming that the service distribution of background tasks is Coxian [81, 102]. Adding the parallel application to the workstation load, the expected number of background tasks at processor j becomes $1/(1 - \rho_j)$. Since the processors are heterogeneous, from the model in Chap. 4 and Eqn. 4.25, we scale the runtime by δ_j/ω, where δ_j represents the processing time per unit work of processor j, and ω the time per unit work of the baseline processor. Multiplying by the background load and heterogeneity scale factors, we find that processor j is expected to take $\delta_j/[\omega(1 - \rho_j)]$ times as long as if the application was run on a dedicated baseline workstation. Using the heuristic algorithm, we sort the workstations based on the values of this scale factor rather than simply the background arrival rate as shown in Figure 6.6. We start with at least two workstations in the set and test the addition of more workstations to find a near-optimal set of workstations P.

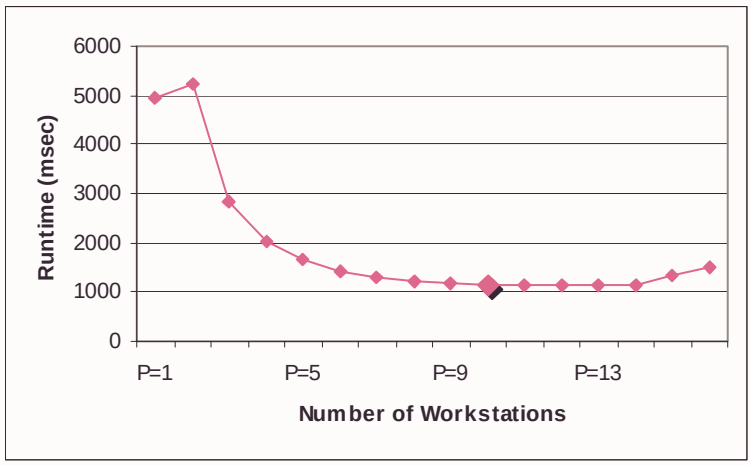
In our first example, we model the AES algorithm running on eight heterogeneous nodes with four different speeds as denoted by the value δ_j for workstation j. Two of the workstations have $\delta_j = 1$, two have $\delta_j = 2$, two have $\delta_j = 4$, and two have $\delta_j = 6$. In Figure 6.7 (a), we see that for eight heterogeneous nodes, the runtime is at a minimum at six nodes. In the second example we have a set of sixteen heterogeneous nodes with four different speeds. Two of the workstations have $\delta_j = 1$, four have $\delta_j = 2$, four have $\delta_j = 3$, four have $\delta_j = 4$, and two have $\delta_j = 6$. In Figure 6.7 (b) we see that for sixteen heterogeneous nodes, the runtime begins is at a minimum at ten nodes.

The algorithms we have discussed thus far ignore the impact of the parallel application on other users. By minimizing the runtime of the parallel application, without considering the impact on other users, the cost function essentially gives the parallel application priority over other users. In the next section, we will look at some cost functions to balance the runtime of the parallel application with the impact on other workstation users.

6.1.2 Minimizing Impact to Other Users

A more general scheduling and optimization problem is to minimize the runtime while simultaneously minimizing the impact on other users. We first develop a cost function which represents the relation of runtime to the cost of running the application on the shared resources. First, we assume there is some cost per unit time, x, which reflects the goal of minimizing the runtime of the parallel application. Second, we assume that each workstation, j, has some cost per unit time for use by the parallel application c_j. Finally, we derive a cost function which represents the trade-off between maximizing the application performance and the impact to other users [102]:

(a)



(b)

FIGURE 6.7 Near-Optimum Set of Heterogeneous Resources for AES Algorithm: (a) 8 nodes and (b) 16 nodes

$$C_P = xR_P + \sum_{j=1}^{P} c_j R_P = R_P\left(x + \sum_{j=1}^{P} c_j\right) \quad \text{(EQ 6.1)}$$

The cost function reflects usage policies and which workstations are encourage for use with parallel application based on the assignment of the c_j values. As the c_j values increase, the cost of using additional workstations increases, making their use less desirable. The use of the x term reflects the importance of minimizing the execution time of the parallel application regardless of the number of processors. By changing the relative values of x and c_j terms, we can study a spectrum of policies describing the application performance relative to its impact on background users.

Identical Usage Costs: Homogeneous Resources. First, we will consider a case with homogeneous resources and identical costs for the use of each workstation (the cost function for this case will be $C_P = R_P(x + cP)$). If we assume that the parallel application efficiency S_P/P is less than one and decreases as P increases (which is true due to the overheads), the cost of using the workstations, cPR_P, exceeds the cost of using a single workstation cR_1 and is monotonically increasing. As discussed earlier, the runtime R_P has some minimum or approaches a minimum as the number of processors increases and thus the cost function C_P will also have a minimum for a finite P. Again, we use the algorithm in Figure 6.1 replacing R_P with C_P. Since the algorithm orders the processors and adds them in sorted order, knowing the number of processors in P is sufficient to describe the set P. In Figure 6.8 we show the optimum number of processors to use for the AES algorithm running on a homogeneous set of resources. In this figure, we vary the ratio of

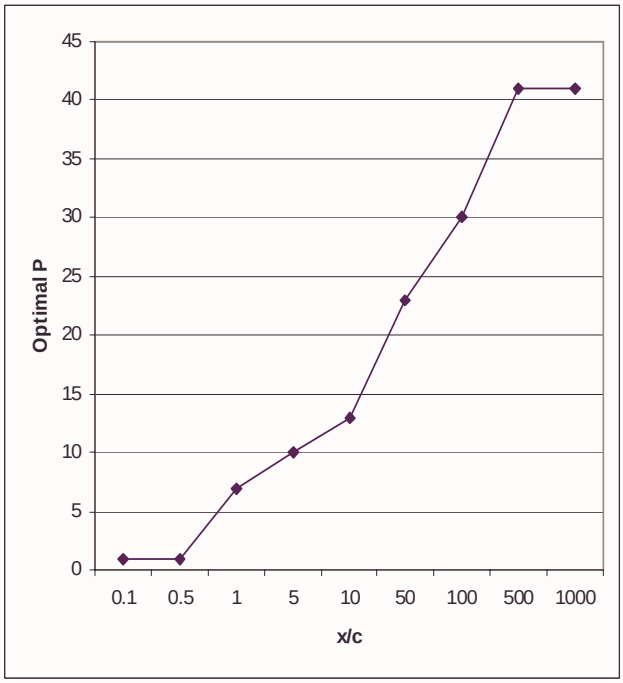


FIGURE 6.8 Optimal Set of Homogeneous Processors for AES Algorithm

x/*c* to reflect a variation in the relative priority between the background user load and the parallel application. With higher priority given to the background user load (low *x*/*c*), the parallel application is assigned to only one processor as would be expected. If higher priority is given to the parallel application (high *x*/*c*), the application runtime is minimized by using all available workstations and we see that there is a maximum benefit achieved at forty-one workstations for this application. Between these two extremes, the number of workstations assigned to the parallel application reflects the relative priority between the background users and the parallel application. For example, one may chose to make the cost of using workstations somewhat expensive for parallel applications by setting a low *x*/*c* value such that parallelism is only exploited when there is potential for significant performance improvements. In Figure 6.9 we enact such a policy to show the optimal set of workstations for the values $x = 1$ and $c = 1$. In this example, we see that using seven worksta-

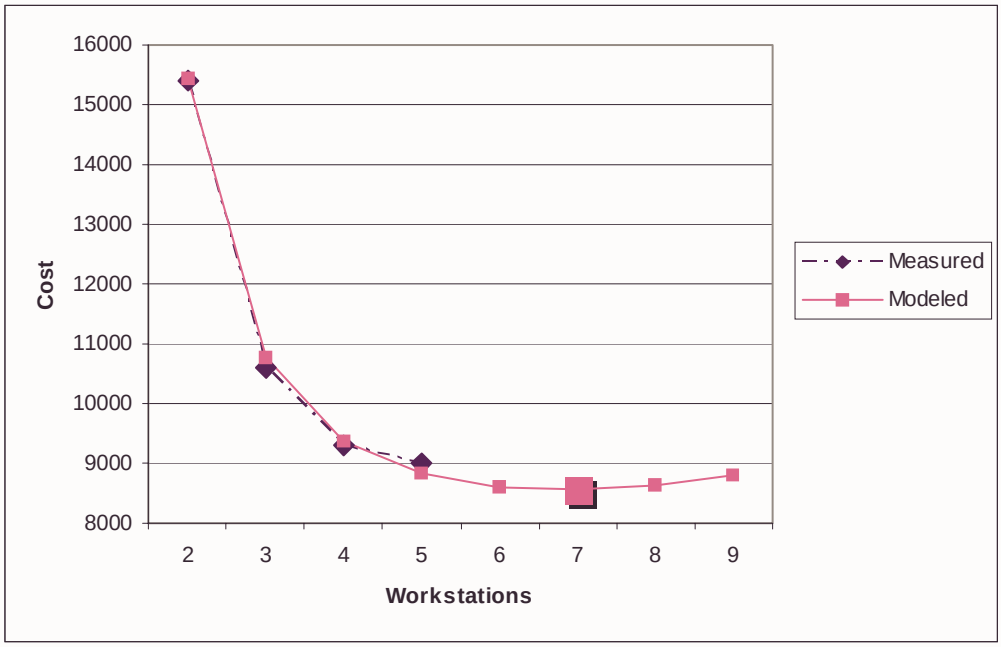


FIGURE 6.9 Cost for AES Algorithm on Homogeneous Processors (x=1, c=1)

tions results in the minimum cost. Our measured results for this usage policy is limited by the size of our validation platform. Note the cost found from the model predicts the measured cost reasonably, thus validating the predicted optimal set.

Identical Usage Costs: Heterogeneous Resources. When running the application on heterogeneous resources, we must use a heuristic as before (Figure 6.6), again replacing R_P with C_P. In this example, the heterogeneous network consists of eight workstations of four different speeds, as denoted by the value δ_j for workstation j. Two of the workstations have $\delta_j = 1$, two have $\delta_j = 2$, two have $\delta_j = 4$, and two have $\delta_j = 6$. The performance of the serial case is taken from the fastest processor while unloaded and the arrival rate of the background jobs at each workstation is identically distributed for each.

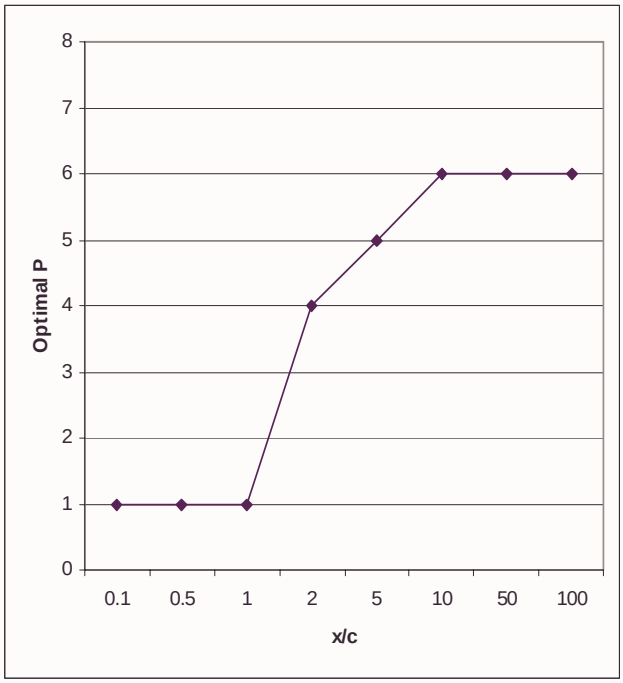


FIGURE 6.10 Optimal Set of Heterogeneous Processors for AES Algorithm

The near-optimal number of processors to use for the AES application running on heterogeneous resources is shown in Figure 6.10. As in the homogeneous case, we vary the relative importance of the parallel application and the background user load (x/c). Again when the parallel application is given low priority (low x/c), it is assigned to only one workstation (in this case the fastest available workstation with the lowest background load). When the parallel application is give higher priority (high x/c), a larger set of workstations is used to minimize the cost. In Figure 6.10 we see that the optimum set of workstations plateaus at six. In this case, adding the slower processors does not lower the cost and we get the best relative performance from six workstations.

To ensure that the optimization algorithm yields a local minima of the runtime, we compare the near-optimal solution to the costs of surrounding states. Since the near-optimal solution

TABLE 6.1 Modeled Costs of Surrounding States of Near-Optimal Solution for AES on 5 Heterogeneous Workstations (x=5, c=1)

Workstations	Added/Subtracted	Cost
near-optimal set		17802.29
1	Subtracted	20046.36
2	Subtracted	20046.36
3	Subtracted	19796.51
4	Subtracted	19796.51
5	Subtracted	18331.43
6	Added	17878.4
7	Added	18019.26
8	Added	18019.26

includes a subset of the available workstations, the surrounding states are comprised of all the subsets formed by adding or removing workstations from the near-optimal set. As shown in Table 6.1 for a typical AES example, this optimization algorithm does find a local minima of the runtime (all surrounding states have a higher cost). In this example, the optimum runtime is achieved with five workstations. When an available workstation is added to the set or one is removed from the near-optimal set, the runtime increases.

Varying Usage Costs. If we assign different values of c_j, or usage costs, to different workstations, many different policies can be studied. One simple use of this policy would be to assign higher c_j values to some workstations and lower c_j values to others to encourage the use of workstations with the lower values. This could be used for several reasons, one for example is to discourage the use of a particular workstation or set of workstations whose load is highly varying and thus whose background load arrival rate is not a good predictor of the processing power actually available to the parallel application. Another example is by using a low c_j value on one or a small set of workstations to encourage the parallel application to only branch out to other workstations if

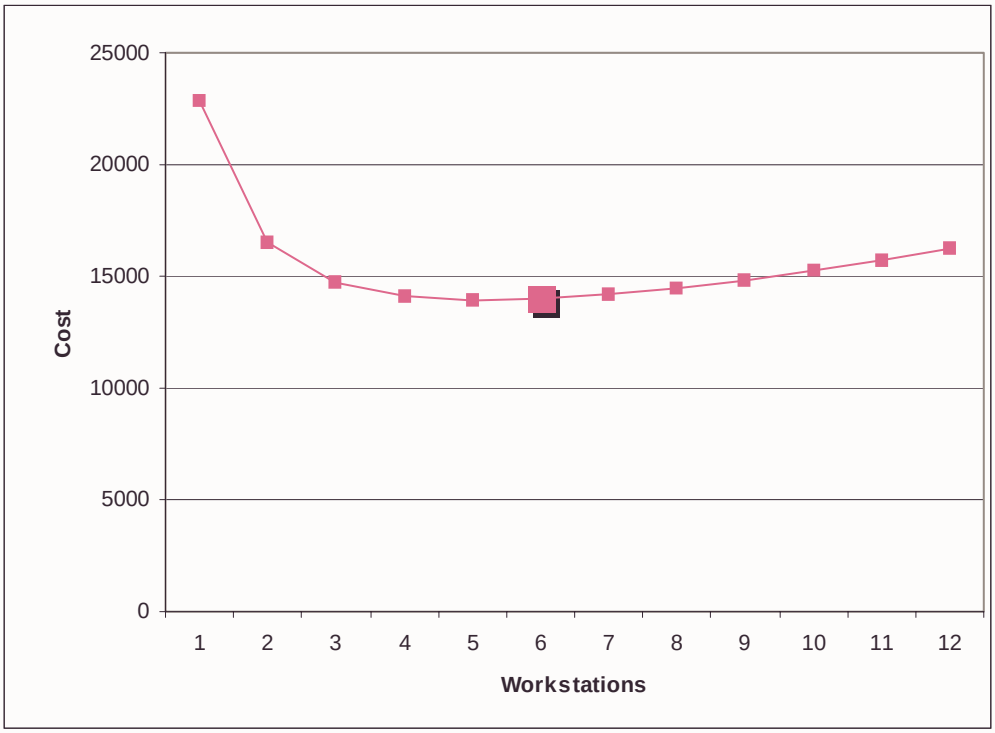


FIGURE 6.11 Cost based on load for AES Algorithm on Homogeneous Processors

there is a potential for significant performance improvements. We can also make the c_j values more sophisticated by making them functions of the time of day, the current load, the reconfigurable unit capabilities, etc. All of these enhancements to the cost function can be used to steer the parallel application toward a desired policy, whether it be toward a specific group of workstations or to avoid heavily loaded ones during peak user hours.

If we want to adopt a policy to avoid workstations with current heavy background load, we can base the c_j values on the background load by setting $c_j = \rho_j$ and $x = 1$. Figure 6.11 shows the cost of the AES algorithm on a homogeneous set of resources with uniformly distributed values of ρ_j. From the figure, we see that the best cost is achieved when six workstations are used. Again, we used the algorithm from Figure 6.1 to find the best set of processors.

TABLE 6.2 Constrained Runtime

Runtime Constraint (sec)	Runtime (sec)	Number of Workstations	Cost
2.0	1.9	5	1134
3.0	2.5	4	1009
4.0	3.2	3	636
5.5	5.3	2	525

Thus far our policy restrictions have been based solely on varying the priorities of the parallel application relative to the background load. We can also consider a policy that restricts the execution time of the parallel application. For example, we could stipulate that the parallel application be assigned to a set of processors so that it is expected to complete in a set amount of time while minimizing the impact on other users. We can formulate this policy as an optimization problem with an execution time goal G_R to minimize [102]:

$$\text{minimize} \sum_{j=1}^{P} c_j R_P \text{ under the constraint, } R_P \leq G_R \qquad \text{(EQ 6.2)}$$

Such a policy can be used to restrict the time in which a parallel application runs and occupies resources. Using our homogeneous set of resources and the cost function $x = 0$, $c_j = \lambda_j$, we find the best set of workstations to optimize our cost function and complete execution within the specified runtime as shown in Table 6.2. Also shown in Table 6.2, as the time constraint is relaxed, we can use fewer workstations, reducing the cost.

Similarly, we can choose to specify a policy bounding the resource utilization by the parallel application while minimizing the execution time. The cost function is formulated with a resource utilization bound G_C, while minimizing the runtime [102]:

TABLE 6.3 Constrained Cost

Cost Constraint	Runtime (sec)	Number of Workstations	Cost
550	5.3	2	525
650	3.2	3	636
1200	1.9	5	1134

$$\text{minimize } R_P \text{ under the constraint, } \sum_{j=1}^{P} c_j R_P \leq G_C \qquad \text{(EQ 6.3)}$$

An example of this policy usage would be a case where we assume a workstation usage cost per unit time, and we impose a maximum cost limit while still trying to minimize the runtime of the parallel application. Again we turn to our AES algorithm running on our homogeneous set of resources with a cost function $x = 0$ and $c_j = \lambda_j$. As shown in Table 6.3, we have different solutions depending on the maximum cost imposed.

In the two examples above we are constraining either the resources or the runtime while trying to optimize the other. As the cost functions become more complex (by varying the c_j values or imposing more constraints), the optimization problem becomes increasingly difficult to solve and as a result, we must resort to more general optimization techniques as the simple algorithms given in Figure 6.1 and Figure 6.6 are no longer sufficient. Two commonly used general optimization techniques are simulated annealing and genetic algorithms. These techniques are generally effective for finding near-optimal solutions to the more general cost function but can require a significant amount of computing power.

6.1.3 Analyzing Optimization Space

To illustrate the power of the modeling methodology when used with cost function analysis, we will now analyze the optimization space using a fixed cost function and vary other parameters in the model. In our example, we will look at the SAT Solver Application running on homogeneous resources with identical costs and the same cost function used in Sec. 6.1.2. First, we will vary the number of hardware copies of the SAT Solver engine present at each node and determine how that affects both the runtime and the cost. Figure 6.12 shows the optimal set of homogeneous resources for the SAT Solver application when we vary the number of hardware copies at each workstation from 4, 8, and 16 copies. In Figure 6.13 we select the values of x and c such that $x/c = 0.0001$ to enact a usage policy that gives more priority to the background users relative to the parallel application. From the three plots we see that by changing the number of hardware copies at each workstation we reduce the overall runtime as expected but the change also affects the cost analysis results. As the number of hardware copies increases, the number of workstations in the optimum set decreases; although our runtime would be shorted with a larger set of workstations, the optimum set for the enforced user policy consists of fewer workstations as the number of hardware copies increases.

As another example, we will vary the speed of the hardware units and use the modeling methodology and cost function analysis to determine the optimum set of workstations. Again, we will look at the SAT Solver Application running on homogeneous resources with identical costs and the same cost function used in Sec. 6.1.2. As we vary the speed of the hardware units from 26.6 MHz to 44.33 MHz, Figure 6.14 shows the optimal set of homogeneous resources for the SAT Solver application. In Figure 6.15 we again select the values of x and c such that $x/c = 0.0001$

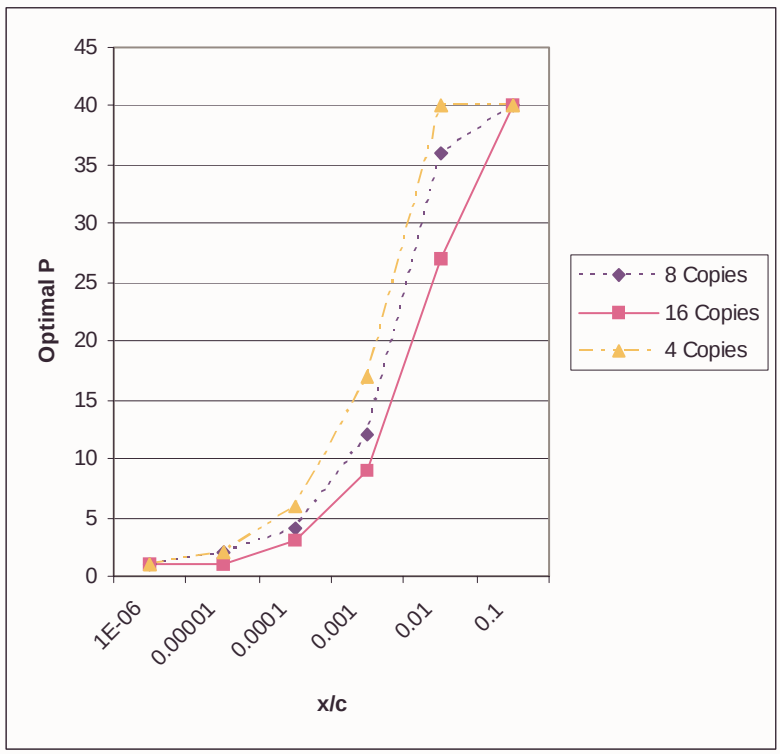


FIGURE 6.12 Optimal Set of Homogeneous Resources for SAT Solver, Varying the Number of Hardware Copies

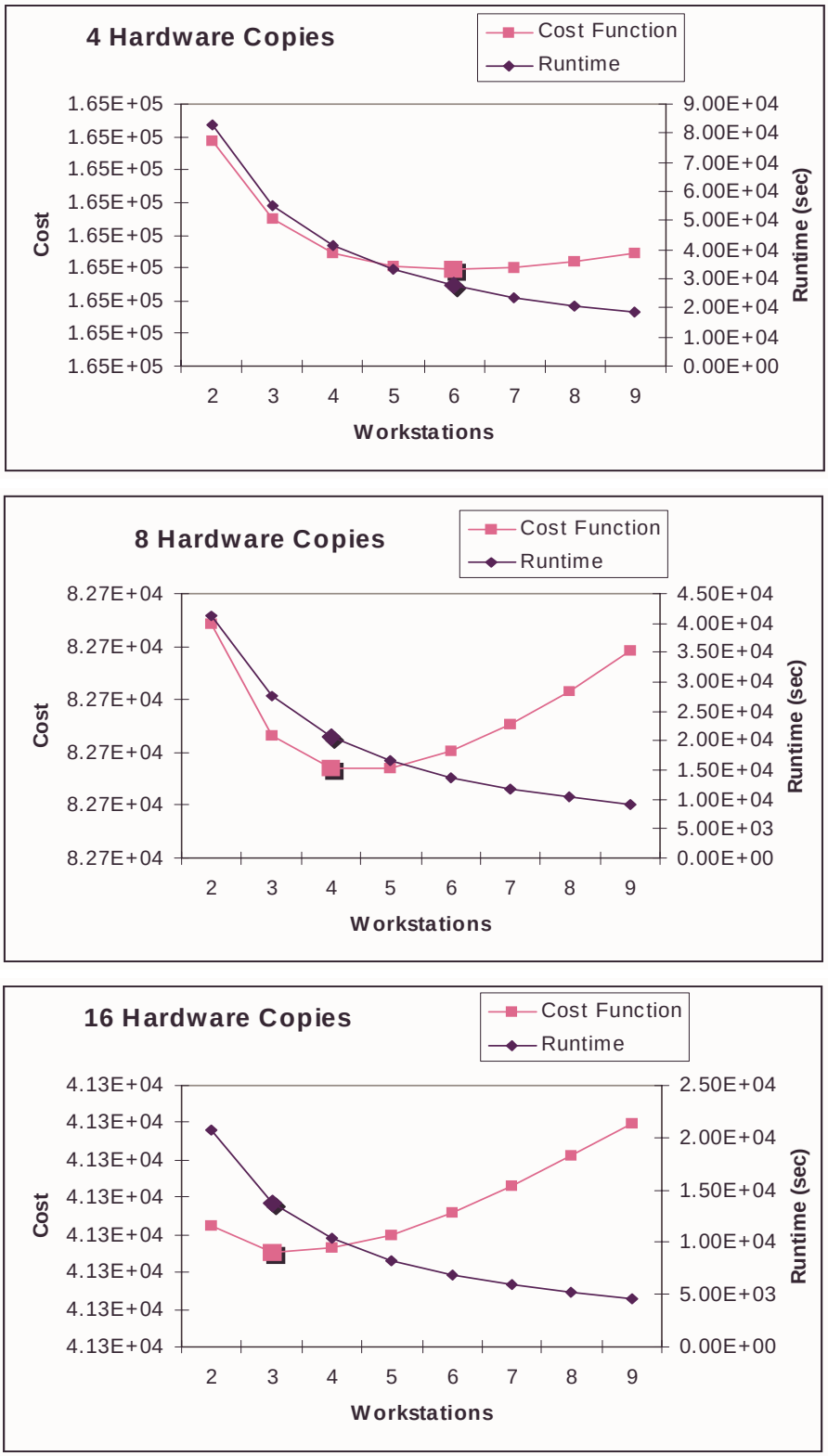


FIGURE 6.13 Cost Function Analysis for SAT Solver, Varying the Number of Hardware Copies (x/c = 0.0001)

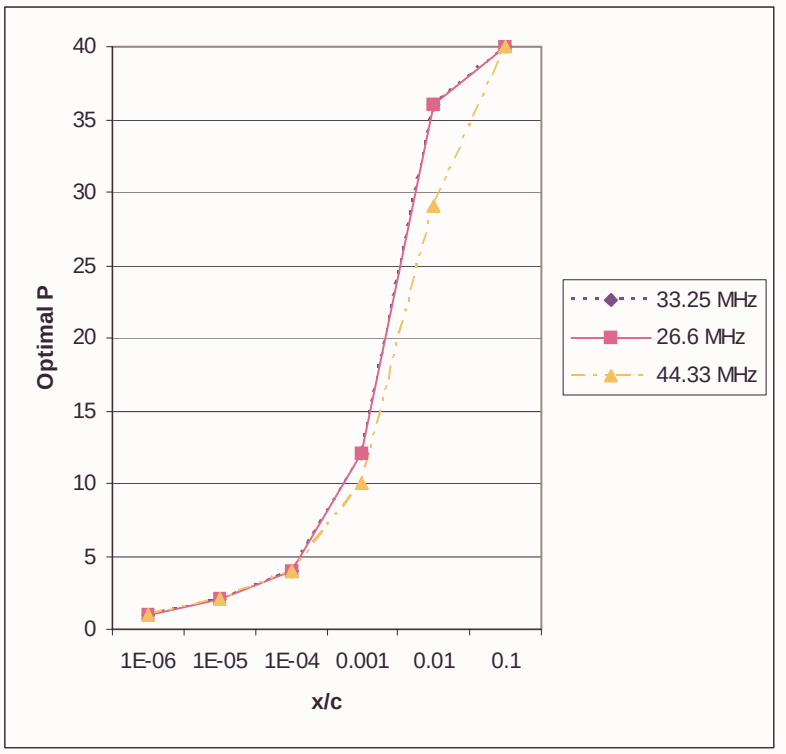


FIGURE 6.14 Optimal Set of Homogeneous Resources for SAT Solver, Varying the Hardware Speed

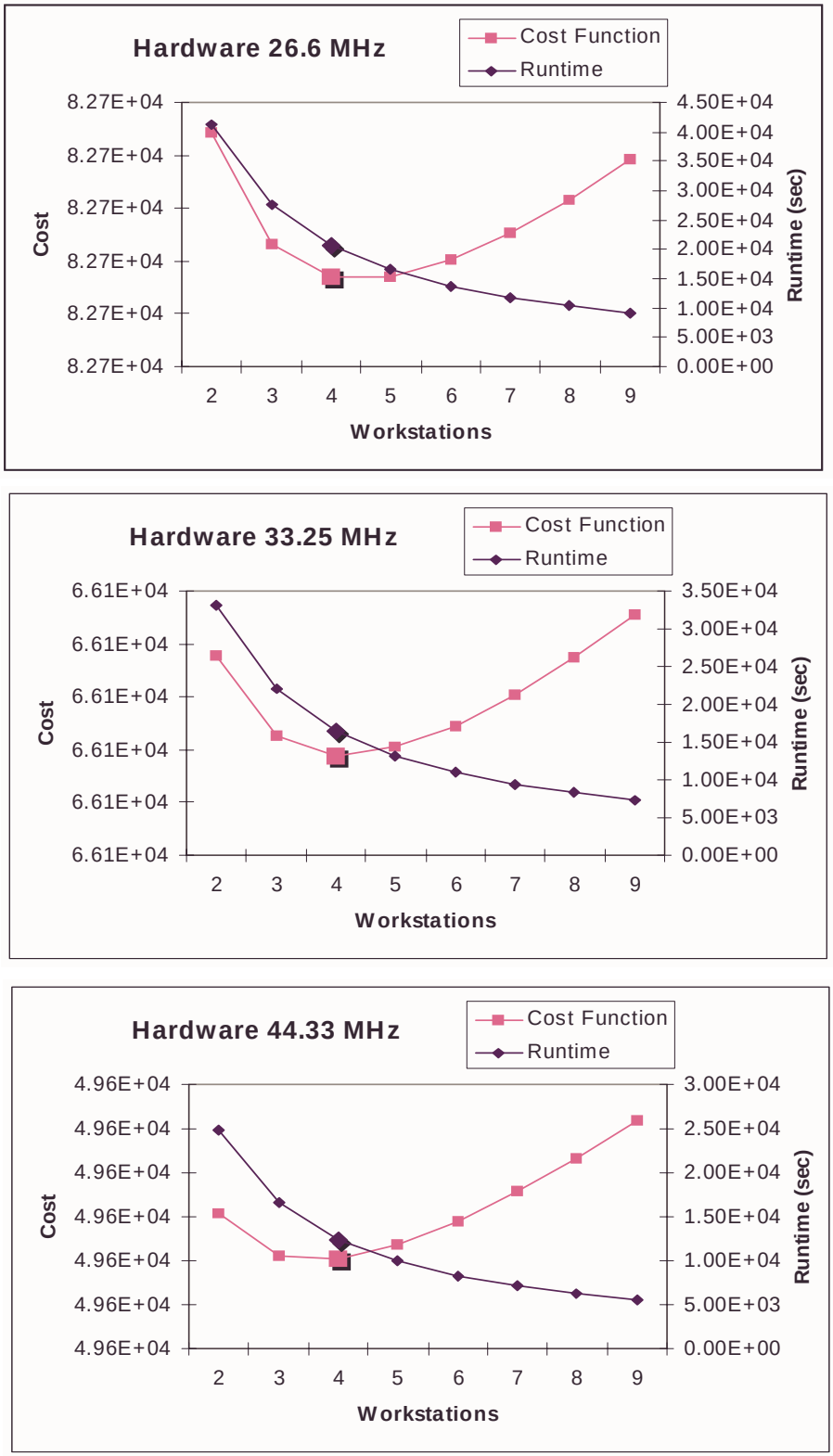


FIGURE 6.15 Cost Function Analysis for SAT Solver, Varying the Hardware Speed (x/c = 0.0001)

to enact a usage policy that gives more priority to the background users relative to the parallel application. From the three plots we see that by changing the speed of the hardware at each workstation we reduce the overall runtime as expected but the speed change does not affect the cost analysis results in the same way as the previous example. As the speed of the hardware increases, the number of workstations in the optimum set stays the same.

Finally we will vary the application load imbalance β and use the modeling methodology and cost function analysis to determine the optimum set of workstations. Again, we will look at the SAT Solver Application running on homogeneous resources with identical costs and the same cost function used in Sec. 6.1.2. As we vary the application load imbalance from $\beta = 6$, $\beta = 8.5$ to $\beta = 12$, Figure 6.16 shows the optimal set of homogeneous resources for the SAT Solver application. In Figure 6.17 we again select the values of x and c such that $x/c = 0.0001$ to enact a usage policy that gives more priority to the background users relative to the parallel application. From the three plots we see that by changing the application load imbalance at each workstation we affect the overall runtime (an increase in β produces an increase in the runtime) as expected. The load imbalance change also affects the cost analysis results in an unanticipated way. As the load imbalance factor increases, the size of the optimum set for a given cost function does not always increase. This condition would be difficult to predict without the use of our model and cost function analysis.

6.1.4 Other Optimization Problems

In the policies discussed above, we have shown that many interesting scheduling optimization problems can be studied quantitatively with our performance modeling methodology for

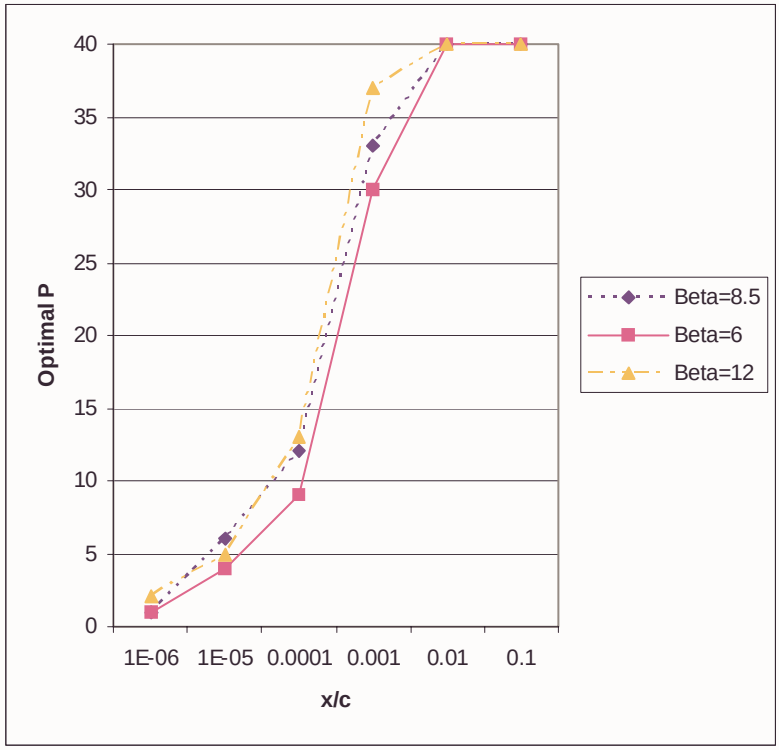


FIGURE 6.16 Optimal Set of Homogeneous Resources for SAT Solver, Varying the Application Load Imbalance

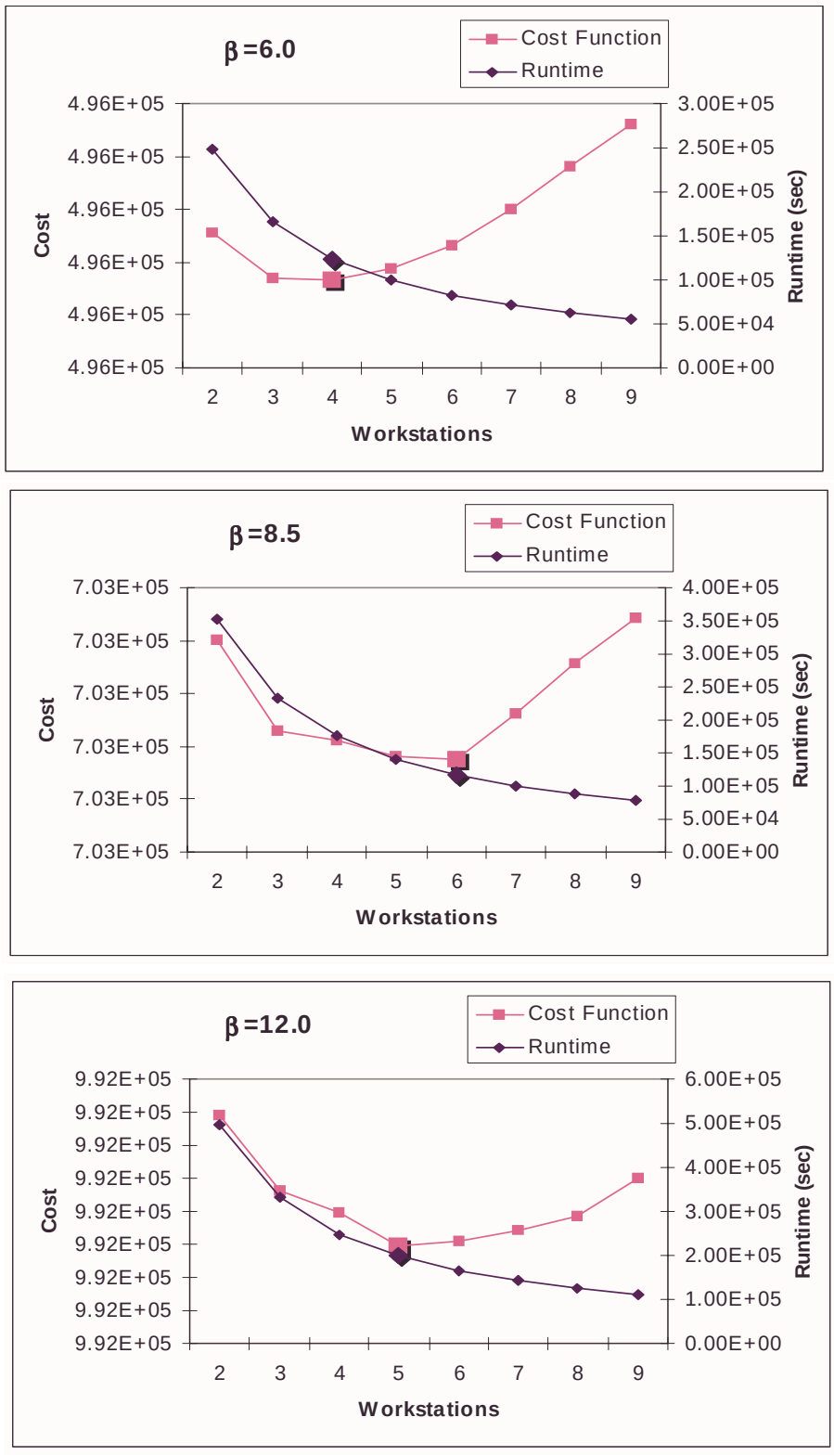


FIGURE 6.17 Cost Function Analysis for SAT Solver, Varying the Application Load Imbalance (x/c = 0.0001)

shared HPRC resources. We will now consider other usage policies and parallel application scheduling questions and how our modeling methodology can be used to investigate them.

Optimization of Other Applications. An obvious use of the modeling methodology in optimization and scheduling is to predict the viability of applications executing on our shared HPRC resources. We have investigated the characteristics of a particular application under various workload conditions and policies, but we could also use the model to evaluate the performance impact of varying the problem size, network size, and other effects. The modeling methodology provides for easy investigation of these issues by simply changing the values for such things as the problem size, overhead, communication time, etc. We can easily consider the addition of more workstations, faster workstations, more RC units, etc. and using cost functions, determine their cost-effectiveness. Hence the modeling methodology is a powerful tool for investigating the performance of different application and the impact of the hardware, background loading, and policy choices.

Static Load Balancing. Thus far in our performance and cost case studies, we have assumed a fixed partitioning of the application. While this is an interesting problem and finding the optimal or near-optimal set of workstations for a given partitioning is beneficial, significant performance gains are possible by partitioning the workload based on some cost function, workstation performance, and/or background load. Given the complexity of such a problem, finding a near-optimal set of workstations and partitioning will require the use of general optimization techniques such as simulated annealing or genetic algorithms as mentioned before. Even considering the computational requirements of such algorithms, the potential performance gains often outweigh these computing costs.

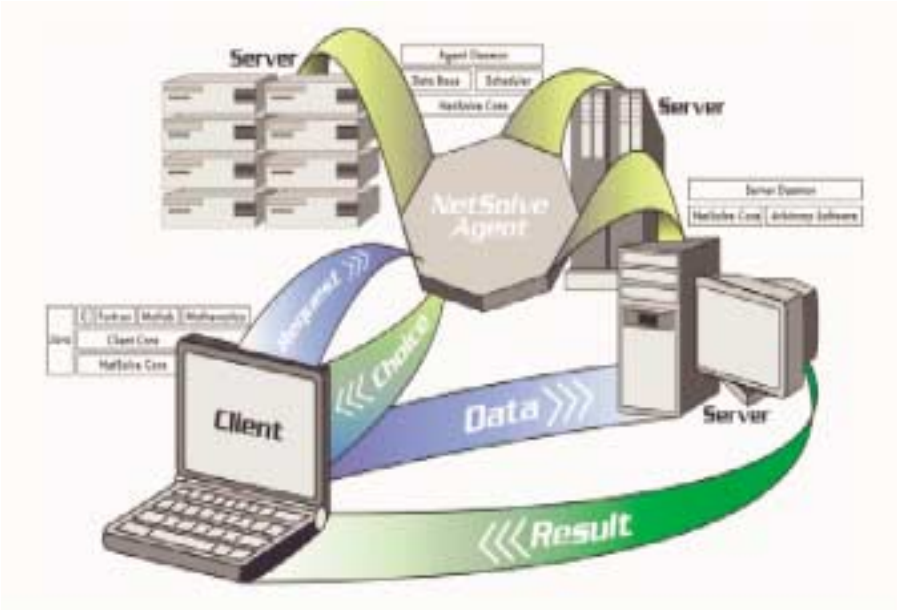


FIGURE 6.18 The NetSolve System [35]

6.2 Scheduling in a NetSolve Environment

The NetSolve project underway at the University of Tennessee and Oak Ridge National Laboratory provides the user easy access to software and hardware resources across multiple platforms without concern for the location or type of resources [14]. NetSolve also provides fault-tolerance and attempts to minimize overall response time with scheduling algorithms. As shown in Figure 6.18, the NetSolve hierarchy has three components: a *client*, an *agent*, and a *server*. The server exists on the hardware resource (single workstation, workstation cluster, MPP, etc.) and provides access to the software installed on that resource. The agent is instrumental in providing the scheduling and mapping decisions since it maintains a database of the statuses and capabilities of servers. The agent is also the primary participant in the fault-tolerant mechanisms. The client provides the user access to NetSolve. Here the user can submit requests to the system and retrieve results.

In the NetSolve system, independent requests or tasks that may be serviced simultaneously are submitted with a farming API [35]. The task farming jobs fall into the category of "embarrassingly parallel" programs and are easily partitioned for parallel programming environments. For scheduling in the NetSolve system, the user manages a ready queue by submitting the farming requests to the system. The agent then manages this queue based on resource availability. Here, our modeling methodology could be used to assist in determining and predicting resource availability and selecting the appropriate set of workstations for the submitted request. Cost functions such as those presented earlier in this chapter could be used to enforce a usage policy. Optimization of the cost function via simple algorithms where appropriate or heuristics when necessary, provides the agent with the optimal set of workstations on which to schedule the worker tasks to meet the prescribed usage policy goals.

In this chapter we used the performance modeling methodology to optimize the usage of our shared HPRC resources. A sampling of different cost functions were considered to reflect different policy goals and priorities. We discussed how the model can be used to analyze different applications and their viability on the shared HPRC resources. We also discussed how the model may be used for load balancing. Finally, we looked at how the model would fit into the NETSOLVE environment and contribute to the optimization and scheduling of applications. We have only scratched the surface of the wide applicability and use of the model for problems in optimization, scheduling, and performance evaluation.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

High Performance Reconfigurable Computing (HPRC) platforms offer the potential for cost-effective performance improvements for many computationally intensive applications provided the resources are used efficiently. In this dissertation, we have developed a performance modeling methodology for fork-join and more specifically Synchronous Iterative Algorithms (SIAs) running on shared HPRC resources. We also considered how to exploit those resources by optimizing scheduling of the parallel applications using the results of our modeling methodology. We now draw conclusions from our results and look forward to extensions of this dissertation and how they can be achieved in future work.

7.1 Conclusions

In this dissertation, we have developed an analytic modeling methodology for characterizing the performance of applications running on HPRC resources. This methodology includes the impact of RC hardware, communication, application load imbalance, background load, and heterogeneous resources. These models were validated using three HPRC applications: Boolean SAT Solver, Matrix Vector Multiplication Algorithm, and AES Encryption Algorithm. The validation efforts indicate the model is accurate for longer runtimes but applications with short runtimes are affected by two issues: variance in the setup overhead costs and variance in file access costs. For applications with short runtimes, these variances can dominate the overall runtime and cause a high standard deviation as seen in our results.

RC systems are becoming widely used to accelerate scientific applications. The use of reconfigurable hardware provides the user performance improvements akin to dedicated co-processors with the dynamic runtime flexibility of software. We analyze applications running on an RC node and develop an analytic model to characterize their performance. Validation experiments show we can accurately model the RC system and their performance. This model was then extended to a multi-node model representative of the HPRC platform. Again, validation experiments were conducted under ideal conditions which show the model accurately characterizes the performance of applications running on dedicated, homogeneous HPRC resources.

In parallel applications, load imbalance across processors, the *application load imbalance*, causes execution times to vary and degrades the overall performance. We model this load imbalance analytically and illustrate its effect with the SAT Solver and Matrix Vector Multiplication applications. Our results show that we can accurately model the effects of application load imbalance of algorithms running on dedicated HPRC resources.

In networks or clusters of workstations, resources are often shared with other users, the *background load*, impacting performance. We use a model based on processor sharing queues to characterize this background load. Using all three HPRC applications with synthetic background load, we show how the model accurately characterizes the impact of other users under various loading conditions. Having validated both the application and background loading models, we use the SAT Solver and Matrix Vector Multiplication applications to validate the methodology under the interaction of application and background loading on homogeneous HPRC resources.

Networks and clusters often consist of workstations from various manufactures, models, and performance capabilities so we include the effect heterogeneous resources in the modeling methodology. This component of the model allows users to quantify the impact of adding or drop-

ping various workstations from a set of heterogeneous computational resources. Since our development HPRC platform consists of homogeneous resources, we simulate the effect of heterogeneous resources by varying the arrival rate of background processes across the workstation set. This effectively simulates a workstation to workstation performance variance and thus a heterogeneous set of resources. Again, using the three HPRC applications, we validate the model's accuracy for predicting the performance of these applications and found the model error to be less than five percent for application runtimes longer than thirty seconds and less than fifteen percent for application runtimes less than thirty seconds. Therefore, we have the capability to characterize the performance of applications running on an HPRC platform under these conditions: when subject to application load imbalance, the effects of other users, and/or a heterogeneous workstation set.

After developing and validating our modeling methodology, we look to apply the model to improve the scheduling of resources and their usage efficiency. We developed several cost functions reflecting prescribed usage policies and employed our modeling results to optimize the use of our shared HPRC resources. We presented examples of the implementation of these cost functions for optimization of the AES algorithm on homogeneous, shared HPRC resources. The implementation results were limited by the number of workstation nodes available and heterogeneity. We also discussed how the modeling methodology is useful in determining the viability of an application for the HPRC platform and how varying the characteristics of the platform can affect the performance. Finally we discussed how the model fits within the NetSolve system and can provide performance data for improving scheduling and performance.

The performance modeling methodology developed in this dissertation and its application to scheduling and other problems are important contributions that will help us to effectively and

efficiently exploit the full potential of processing power available in a HPRC platform. A few possible extensions to this work are explored in the next section.

7.2 Future Work

The performance modeling methodology has been developed and validated using parallel applications which fall under the master/slave paradigm. To better understand the applicability of the model, it would be interesting to consider other parallel applications that include more communication between nodes. Additional applications such as simulations and various DSP algorithms, which have made use of RC systems thus far [70, 71], would further test the robustness of the modeling methodology. Also to be considered are applications with concurrent software and hardware tasks and extensive node to node communication.

We need to further validate the model on a true set of heterogeneous resources and even a larger cluster of resources would allow for bigger and more interesting problems. Increasing both the size of the cluster and the heterogeneity would permit us to determine the model accuracy over a more diverse processor population. Another factor not yet considered is the possibility of heterogeneous RC boards either within a single node or across the cluster. This would require only minor if any changes to the model but could yield some interesting performance results in cases where there is significant application load imbalance.

While we scratched the surface on scheduling, there are many interesting issues left to explore. The current development system consist of homogeneous nodes and it would be interesting to investigate the model's effectiveness in optimization of schedules for heterogeneous resources. Also, we would like to explore the co-scheduling of multiple parallel applications on HPRC resource. This introduces the issue of multiple tasks programming the RC units and the

associated overhead involved. Can we schedule multiple parallel applications simultaneously, all vying for the RC resources, and context switch between applications while still providing reasonable throughput for the user?

While we have made no mention of reliability and fault tolerance thus far, in large networks of workstations and/or for long runtime applications, these issues take on significant importance. These modeling results could be incorporated within a methodology for maintaining a fault tolerant network to assist in selecting the optimum set of available workstations.

Another important extension of this work would be to relax some of our assumptions regarding the class of algorithms modeled. It would be interesting to consider more general fork-join algorithms as well as other classes.

While we have primarily focused on the processor runtime of applications, we could change our focus to include other issues or resources for potential optimization such as cost, power, size, minimizing communications, memory usage, FPGA size, etc.

Achieving these goals of future work would extend the applicability and contributions of this work. However, this dissertation is an important step towards a better understanding of the analytical performance modeling of parallel applications running on shared HPRC resources. Use of the model will provide users with a means to better exploit available HPRC resources and understand the implications of various loading conditions on such resources.

BIBLIOGRAPHY

[1] AccelChip, http://www.accelchip.com/, 2003.

[2] Adaptable Computing Cluster, http://www.parl.clemson.edu/acc/, 2003.

[3] AFRL/IF, http://www.if.afrl.af.mil/tech/facilities/HPC/hpcf.html, 2003.

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

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

[6] Atmel, http://www.atmel.com, 2001.

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

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

[9] Celoxica, http://www.celoxica.com, 2003.

[10] Configurable Computing Lab, http://www.ccm.ece.vt.edu/, 2003

[11] I.S.I.East, SLAAC: System-Level Applications of Adaptive Computing, http://slaac.east.isi.edu/, 2003.

[12] MATCH, http://www.ece.northwestern.edu/cpdc/Match/

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

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

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

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

[17] Top 500 Supercomputer Sites: Overview of Recent Supercomputers, Aad J. van der Steen and Jack J. Dongarra, http://www.top500.org/ORSC/, 2002.

[18] University of Florida High-performance Computing and Simulation Research Lab, http://www.hcs.ufl.edu/prj/rcgroup/teamHome.php, 2003

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

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

[21] Agrawal, Vishwani, and Chakradhar, Srimat T., “Performance Analysis of Synchronized Iterative Algorithms on Multiprocessor Systems,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 6, pp. 739-746, November 1992.

[22] 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.

[23] 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.

[24] Banerjee, P., N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden, D. Zaretsky, “A MATLAB Compiler for Distributed, Heterogeneous, Reconfigurable Computing Systems,” *Proc. of the 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM-00)*, 2000, IEEE Computer Society.

[25] Basney, J., Raman, B., and Livny, M., “High Throughput Monte Carlo,” *Proc. of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, 1999.

[26] Baumgartner, K. and Wah, B.W., “Computer Scheduling Algorithms: Past, Present and Future,” *Information Sciences*, vol. 57 & 58, pp. 319-345, ELSEVIER Science, New York, NY, Sept. - Dec. 1991.

[27] 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.

[28] Bokjari, Shahid H., “Partitioning Problems in Parallel, Pipelined, and Distributed Computing,” *IEEE Transactions on Computers*, vol. 37, no. 1, pp. 48-57, January 1988.

[29] 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.

[30] 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.

[31] Bubendorfer, K. and Hine, J.H., “A Compositional Classification for Load-Balancing Algorithms,” Technical Report CS-TR-99-9, Victroia University of Wellington, July 1998.

[32] Cantu-Paz, E., “Designing Efficient Master-Slave Parallel Genetic Algorithms,” in *Genetic Programming: Proc. of the 3rd Annual Conference*, San Francisco, Morgan Kaufmann, 1998.

[33] Cap, Clemens H., and Volker Strumpen, “Efficient Parallel Computing in Distributed Workstation Environments,” *Parallel Computing*, vol. 19, pp. 1221-1234, 1993.

[34] Casanova, H., Dongarra, J., and Jiang, W., “The Performance of PVM on MPP Systems,” CS-95-301, pp. 1-19, Knoxville, TN, University of Tennessee, 1995.

[35] Casanova, H., Kim, M., Plank, J.S., and Dongarra, J.J., “Adaptive Scheduling for Task Farming with Grid Middleware,” *Intl. Journal of Suptercomputer Applications and High Performance Computing*, vol. 13, no. 3, pp. 231-240, 1999.

[36] 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.

[37] Chamberlain, R. D., and Mark A. Franklin, “Hierarchical Discrete-Event Simulation on Hypercube Architectures,” *IEEE Micro*, vol. 10, no. 4, pp. 10-20, August 1990.

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

[39] Choi, Tik-Hing, “Floating-Point Matrix-Vector Multiplication Using Reconfigurable System,” Master of Science Electrical Engineering, The University of Tennessee, 2003.

[40] Chou, Timothy C.K. and Abraham, Jacob A., “Load Balancing in Distributed System,” *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 401-412, July 1982.

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

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

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

[44] David, H.A., *Order Statistics*, Wiley, 1970.

[45] Davis, M. and Putnam, H., A Computing Procedure for Quantification Theory. *Journal of the ACM*, vol. 7, pp. 201-215, 1960.

[46] Davis, M., Logemann, G., and Loveland, D., A Machine Program for Theorem Proving.*Communications of the ACM*, vol. 5, pp. 394-397, 1962.

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

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

[49] de Souza e Silva, E. and Gerla, M., 'Queueing Network Models for Load Balancing in Distributed Systems,” *Journal of Parallel and Distributed Computing*, vol. 12, no. 1, pp. 24-38, May 1991.

[50] Dongarra, J., H. Meuer, H. Simon, and E. Strohmaier, “High Performance Computing Today,” http://icl.cs.utk.edu/publications/pub-papers/2000/hpc-today.pdf, 2000.

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

[52] Dongarra, J., and H. Simon, “High Performance Computing in the U.S. in 1995,” Technical Report UT-CS-95-318, University of Tennessee Computer Science Department, 1995.

[53] 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.

[54] Dubois, Michel, and Faye A. Briggs, “Performance of Synchronized Iterative Processes in Multiprocessor Systems,” *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 419-431, July 1982.

[55] Efe, Kemal, and Schaar, Margaret A., “Performance of Co-Scheduling on a Network of Workstations,” In *Proceedings of 13th International Conference on Distributed Computing Systems*, pp. 525-531, 1993.

[56] Efe, Kemal, and Krishnamoorthy, Venkatesh, “Optimal Scheduling of Compute-Intensive Tasks on a Network of Workstations,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 6, pp. 668-673, June 1995.

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

[58] Feitelson, Dror G. and Rudolph, Larry, “Metrics and Benchmarking for Parallel Scheduling,” In *Job Scheduling Strategies for Parallel Processing: IPPS/SPDP98 Workshop Proceedings*, LNCS 1459, pp. 1-24, Springer-Verlag, 1998.

[59] Flynn, M. J., "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948-960, Sept. 1972.

[60] Franklin, Mark A. and Govindan, Vasudha, “The N-Body Problem: Distributed System Load Balancing and Performance Evaluation,” In *Proceedings of the Sixth International Conference on Parallel and Distributed Computing Systems*, October 1993.

[61] Garey, Michael R. and Johnson, David S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

[62] 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.

[63] 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.

[64] 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.

[65] Govindan, Vasudha and Franklin, Mark A., “Application Load Imbalance on Parallel Processors,” In *Proceedings of 10th International Parallel Processing Symposium (IPPS96)*, 1996.

[66] 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.

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

[68] 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.

[69] Harchol-Balter, Mor and Downey, Allen B., “Exploiting Process Lifetime Distributions for Dynamic Load Balancing,” *ACM Transactions on Computer Systems*, vol. 15, no. 3, pp. 253-285, 1997.

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

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

[72] 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.

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

[74] 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.

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

[76] Jacqmot, C. and Milgrom, E., “A Systematic Approach to Load Distribution Strategies for Distributed Systems,” in *IFIP Transactions: International Conference on Decentralized and Distributed Systems*, ELSEVIER Science, September 1993.

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

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

[79] 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.

[80] Jones, M. T., Langston, M. A., and Raghavan, P., "Tools for mapping applications to CCMs," *In SPIE Photonics East ‘98*, 1998.

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

[82] 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.

[83] Kremien, Orly and Kramer, Jeff, “Methodical Analysis of Adaptive Load Sharing Algorithms,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 6, pp. 747-760, November 1992.

[84] Krueger, P., and Chawla, R., “The stealth distributed scheduler,” in *Proc. 11th Int. Conf. Distrib. Comput. Syst.*, 1991, pp. 336-343.

[85] 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.

[86] Leland, Will E. and Ott, Teunis J., “Load-balancing Heuristics and Process Behavior,” In *Proceedings of Performance’86 and ACM SIGMETRICS*, pp. 54-69, May 1986.

[87] 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.

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

[89] 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.

[90] Ma, P.R., Lee, E.Y.S., and Tsuchiya, M., “A Task Allocation Model for Distributed Computing Systems,” *IEEE Transactions on Computers*, vol. C-31, no. 1, pp. 41-47, January 1982.

[91] 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.

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

[93] 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.

[94] 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.

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

[96] 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.

[97] 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.

[98] 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.

[99] 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.

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

[101] 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.

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

[103] Peterson, G. D. and Chamberlain, R. D., “Exploiting Lookahead in Synchronous Parallel Simulation,” In *Winter Simulation Conference*, pp. 706-712, 1993

[104] Peterson, G. D. and Chamberlain, R. D., “Performance of a Globally-Clocked Parallel Simulator,” In *International Conference on Parallel Processing*, pp. 289-298, 1993.

[105] Peterson, G. D. and Chamberlain, R. D., “Beyond Execution Time: Expanding the Use of Performance Models,” *IEEE Parallel and Distributed Technology*, vol. 2, no. 2, pp. 37-49, 1994.

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

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

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

[109] Platzner, Marco and De Micheli, Giovanni, “Acceleration of Satisfiability Algorithms by Reconfigurable Hardware,” *Proceedings of the 8th International Workshop on Field Programmable Logic and Applications (FPL98)*, pp. 69-78, Tallinn, Estonia, Springer-Verlag, 1998.

[110] Reed, Daniel A. and Fujimoto, Richard M., *Multicomputer Networks: Message-Based Parallel Processing*, MIT Press, 1987.

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

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

[113] 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.

[114] Shen, Chien-Chung and Tsai, Wen-Hsiang, “A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion,” *IEEE Transactions on Computers*, vol. C-34, no. 3, pp. 197-203, March 1985.

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

[116] 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.

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

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

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

[120] Tananbaum, Andrew S., *Modern Operating Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[121] Tantawi, Asser N. and Towsley, Don, “Optimal Static Load Balancing in Distributed Computer Systems,” *Journal of the ACM*, vol. 32, no. 2, pp. 445-465, April 1985.

[122] 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.

[123] 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,

[124] Underwood, K.D., W.B. Ligon, and R.R. Sass, “Analysis of a prototype intelligent network interface,” *Concurrency and Computation:Practice and Experience*, Vol. 15, 2003.

[125] Underwood, K.D., *An Evaluation of the Integration of Reconfigurable Hardware with the Network Interface in Cluster Computer Systems*, Ph.D. thesis, Clemson University, August 2002.

[126] 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.

[127] Wang, Y.T., and Morris, R.J.T., “Load Sharing in Distributed Systems,” *IEEE Transactions on Computers*, vol. C-34, March 1985, pp. 204-217.

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

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

[130] 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.

[131] 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.

[132] 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.

[133] Zhong, P., Martonosi, M., Ashar, P., and Malik, S., "Solving Boolean Satisfiability with Dynamic Hardware Configurations," *Proceedings of the 8th International Workshop on Field Programmable Logic and Applications (FPL98)*, pp. 326-335, Tallinn, Estonia, Springer-Verlag, 1998.

[134] Zhong, P., Martonosi, M., Ashar, P., and Malik, S., "Using Reconfigurable Computing Techniques to Accelerate Problems in the CAD Domain: A Case Study with Boolean Satisfiability," *Design Automation Conference (DAC) 1998*, pp. 194-199, 1998.

APPENDIX

Table a.1 and Table a.2 list the data used in Figure 5.2. The data collected is for the communication measurements between workstations. Table a.1 lists the round trip time for messages of various data sizes between workstations vlsi4 and vlsi6 and also between vlsi1 and vlsi4. The bandwidth and latency values used in the model calculations are listed as well as the model results. Table a.2 lists five bandwidth samples for each message size.

Table a.3 lists the runtime data from the SAT Solver application running in ideal conditions as discussed in Sec. 5.4.1. The runtime data is listed for six problem sizes (32-bit, 36-bit, 38-bit, 41-bit, 43-bit, and 44-bit). Table a.4 lists similar runtime data from the SAT Solver application running with application load imbalance as discussed in Sec. 5.4.2. Table a.5 lists the runtime data from the SAT Solver application running with five different background load imbalance conditions as discussed in Sec. 5.4.3. Finally, Table a.6 lists the runtime data from the SAT Solver application running with application and background load imbalance as discussed in Sec. 5.4.4.

TABLE a.1 Communication Measurements: Message Round Trip Time

vlsi4-6 Time (us)	vlsi1-4 Time (us)	data size	BW - eta	latency - tau	Model Time
767	766	0	1.7	766	766
740	741	8			770.705882
786	784	80			813.058824
1217	1241	800			1236.58824
4596	5184	8000			5471.88235
43853	54391	80000			47824.8235
425837	537110	800000			471354.235

TABLE a.2 Communication Measurements: Network Bandwidth (vlsi4)

Message Size	Sample1	Sample2	Sample3	Sample4	Sample5
100	0.2	0.252525	0.253165	0.257732	0.222717
1000	1.751313485	1.782531	1.792115	1.795332	1.795332
10000	6.93962526	7.102273	7.189073	7.209805	7.183908
100000	10.73191672	10.83072	10.82017	10.74345	10.83306
1000000	11.6051016	11.63224	11.62885	11.62345	11.62696
10000000	11.70816235	11.7114	11.70738	11.70997	11.6233
1E+08	11.50276095	11.52063	11.52788	11.4835	11.54895

TABLE a.3 SAT Solver No-load Data (sec)

32f	36f	38f	41f	43f	44f
5.1310	65.323	322.463	2571.83	10285.49	20570.592
5.0920	65.354	322.231	2572.188	10284.7	20570.392
4.8510	65.115	322.461	2572.181	10285.47	20570.516
5.0910	65.012	322.461	2571.836	10285.72	
4.7440	65.351	322.463	2572.176	10285.45	
4.6050	65.333	322.461	2572.177	10285.65	
5.0710	65.34	322.461	2572.233	10285.77	
5.0630	65.351	322.373	2571.833	10285.8	
5.0960	65.349	322.466	2571.847	10285.55	
5.0850	65.015	322.461	2572.179	10285.65	
4.8700	65.328	322.134	2571.946		
5.1520	65.361	322.106	2572.176		
4.7410	65.361	322.125	2572.168		
4.7270	65.352	322.461	2571.831		
5.0990	65.329	322.463	2572.156		
4.6150	65.342	322.467	2572.179		
5.0890	65.352	322.462	2571.856		
5.0910	65.352	322.124	2572.179		
5.2060	65.352	322.12	2571.962		
5.0920	65.352	322.455	2571.394		
5.0760	65.352	322.461			
5.0930	65.35	322.457			
4.6150	65.368	322.121			
5.0920	65.351	322.457			
5.1010	65.351	322.134			
4.7480	65.023	322.435			
4.7540	65.104	322.461			
4.8590	65.129	322.24			
5.0920	65.36	321.679			
5.0820	65.352	322.246			
4.7510	65.348				
4.8790	65.351				
4.7570	65.346				
5.0650	65.119				
5.0920	65.351				
5.0920	65.129				
5.1000	65.362				
4.8310	64.991				
4.5850	65.398				
4.5920	64.99				
4.6110	65.09				
4.8680	65.164				
4.7200	64.831				
4.8360	64.81				
4.8320	64.899				
4.8660	65.045				
4.6440	64.949				
4.8030	65.052				
4.8520	64.808				
4.5650	64.849				

TABLE a.4 SAT Solver Application Load Imbalance Data (sec)

32f	36f	38f	41f	43f	44f
34.342	546.561	2732.002	21854.62	87417.6	339402.2
34.498	546.591	2732.003	21855.28	87417.55	
34.367	546.59	2732.005	21854.61	87417.67	
34.398	546.58	2732.04	21856.81	87417.7	
34.352	546.591	2731.999	21855.22	87417.67	
34.381	546.575	2732.028			
34.373	546.597	2732.003			
34.355	546.577	2732.01			
34.366	546.584	2732.012			
34.365	546.603	2732.012			
34.379	546.598	2732.03			
34.361	546.57	2732.022			
34.364	546.571	2732.02			
34.372	546.559	2732.036			
34.361	546.586	2731.996			
34.357	546.584	2732.115			
34.334	546.556	2732.062			
34.355	546.563				
34.384	546.584				
34.383	546.572				
34.354	546.612				
34.352	546.611				
34.339	546.589				
34.36	546.571				
34.356	546.573				
34.349	546.602				
34.356	546.562				
34.354	546.62				
34.354	546.587				
34.355	546.569				
34.381	546.581				
34.373	546.567				
34.368	546.567				
34.399	546.599				
34.384	546.581				
34.383	546.574				
34.352	546.593				
34.342	546.57				
34.401	546.574				
34.363	546.566				
34.359	546.56				
34.377	546.579				
34.36	546.55				
34.34	546.591				
34.396	546.564				
34.341	546.593				
	546.558				
	546.576				
	546.576				
	546.581				

TABLE a.5 SAT Solver Background Load Imbalance Data (sec)

gamma		gamma		gamma		gamma		gamma	
1.476		1.975		2.34		3.436		5.254	
32f	36f	32f	36f	32f	36f	32f	36f	32f	36f
5.099	64.433	6.46	66.153	4.508	65.004	5.724	66.241	5.892	65.984
4.608	64.563	5.641	65.431	4.708	64.873	5.332	65.748	6.321	66.452
4.388	65.171	5.177	65.53	4.546	64.964	5.729	65.526	6.014	66.239
4.419	64.913	4.999	65.564	4.705	64.773	6.367	66.08	6.368	65.956
4.617	64.729	5.011	65.588	4.537	64.838	6.734	66.821	5.799	66.279
4.387	64.812	5.215	65.632	4.452	64.757	6.219	66.541	5.754	66.557
4.328	64.923	4.971	64.443	4.758	64.738	5.956	66.782	6.074	65.712
4.454	64.538	5.013	64.453	4.563	64.727	6.469	66.358	5.509	66.058
4.282	65.07	4.555	65.908	4.501	64.93	5.628	65.63	6.075	66.358
4.371	64.822	4.937	66.076	4.73	65.025	4.833	67.277	6.018	65.884
4.371	64.931	4.182	65.714	4.698	65.055	4.859	66.572	5.588	66.441
4.383	64.81	4.18	65.812	4.625	64.943	4.183	66.538	5.881	65.922
4.603	64.975	5.278	65.791	4.57	64.761	6.104	65.251	5.863	66.275
4.315	64.675	5.889	65.245	4.576	65.022	5.757	65.424	5.925	65.941
4.238	64.445	5.487	64.443	4.682	64.937	5.518	66.3	5.76	66.588
4.472	64.447	5.685	64.449	4.661	64.842	6.41	65.182	5.617	65.944
4.611	64.433	5.733	66.067	4.503	65.05	5.569	65.101	5.874	65.983
4.705	64.432	5.507	65.928	4.738	64.817	5.504	65.466	5.855	66.115
4.528	64.438	5.586	65.748	4.545	64.773	5.793	64.928	5.619	65.991
4.437	64.432	5.529	65.425	4.766	64.852	6.151	66.076	5.666	66.44
4.167	64.434	4.753	65.844	4.591	64.956	5.978	65.178	5.828	65.825
4.444	64.432	4.612	65.803	4.646	65.029	4.889	67.356	5.669	66.118
4.167	64.432	4.178	64.444	4.522	64.985	4.626	66.007	6.578	66
4.368	64.434	4.181	64.445	4.77	64.529	5.018	67.122	5.638	66.816
4.168	64.435	5.524	66.276	4.609	64.683	5.009	66.01	6.557	66.287
4.311	64.432	5.75	66.143	4.604	64.99	4.186	65.591	5.165	65.868
4.561	64.432	4.179	65.552	4.545	64.754	5.522	65.939	5.939	66.017
4.242	64.432	5.849	65.348	4.789	64.786	5.406		6.717	66.025
4.169	64.431	5.522	65.66		65.044	5.264			66.524
4.244	64.44		65.607			5.915			
4.408	64.431								
4.222	64.431								
4.599	64.433								
4.167	64.432								
4.235	64.432								
4.169	64.432								
4.426	64.433								
4.167	64.432								
4.371	64.432								
4.168	64.431								
4.341	64.432								
4.168	64.433								
4.335	64.43								
4.169	64.431								
4.357	64.432								
4.359	64.433								
4.318	64.433								
4.168	64.433								
4.296	64.431								
4.168									

TABLE a.6 SAT Solver Application and Background Load Imbalance Data (sec)

gamma 1.476		gamma 1.975		gamma 2.34		gamma 3.436		gamma 5.254	
32f	36f	32f	36f	32f	36f	32f	36f	32f	36f
34.434	546.615	34.897	547.6	34.709	546.638	35.721	547.866	35.647	0.547866
34.541	546.867	35.286	547.415	34.679	547.096	36.074	548.028	35.533	0.548028
34.744	546.714	35.884	548.486	34.69	546.779	35.123	548.017	35.751	0.548017
34.378	546.512	35.375	547.423	34.818	546.89	35.155	547.872	35.48	0.547872
34.302	546.513	35.74	546.526	34.502	546.705	35.107	547.571	35.873	0.547571
34.303	546.511	35.555	548.156	34.605	546.749	34.923	547.986	35.091	0.547986
34.542	546.616	35.376	548.173	34.478	547.079	36.122	547.087	35.716	0.547087
34.632	546.577	35.856	547.469	34.501	546.969	35.845	547.899	35.809	0.547899
34.301	546.851	35.851	547.966	34.63	547.135	35.466	547.97	35.65	0.54797
34.303	546.846	35.734	546.523	34.826	546.786	34.877	547.623	35.466	0.547623
34.314	546.825	35.391	547.707	34.443	546.767	35.179	549.518	35.788	0.549518
34.7	546.754	35.422	547.7	34.601	546.829	35.326	548.902	35.216	0.548902
34.644	546.511	35.757	547.78	34.522	546.987	35.236	547.422	35.528	0.547422
34.32	546.665	35.431	547.574	34.748	546.693	35.744	547.4	35.733	0.5474
34.301	547.087	35.493	546.521	34.934	546.667	35.857	548.457	35.709	0.548457
34.299	546.945	35.656	546.519	34.69	546.974	34.941	548.039	35.596	0.548039
34.712	546.528	35.318	547.882	34.865	546.938	35.414	548.03	35.336	0.54803
34.863	546.508	35.553	547.263	34.844	546.861	34.798	548.246	35.859	0.548246
34.382	546.512	35.937	547.469	34.697	546.801	35.087	547.302	35.79	0.547302
34.299	546.618	35.637	547.925		546.811	35.041	547.644	35.402	0.547644
34.298	546.546	35.35	547.336		546.916	34.771	547.671	35.777	0.547671
34.554	546.554	35.056	546.535			35.628	547.838	35.727	0.547838
34.642	547.055	35.437	548.118			34.993	548.056	35.927	0.548056
34.298	546.715	36.056	548.006			34.613	547.895	35.82	0.547895
34.299	546.821	35.496	547.733			34.931	548.318	35.16	0.548318
34.299	546.511	35.631	547.15			34.976	547.027	35.665	0.547027
34.972	546.511		546.519			35.427	548.327	35.853	
34.673	546.918		547.778			35.656	548.169	36.056	
34.301	547.006		547.904			34.924	548.364	35.618	
34.3	546.631		547.705						
34.301	546.511								
34.679	546.511								
34.674	546.511								
34.37	546.52								
34.3	546.55								
34.299	546.526								
34.768	546.92								
34.719	546.772								
34.374	546.981								
34.301	546.512								
34.299	546.513								
34.506	546.915								
34.67	546.868								
34.383	546.755								
34.3	546.663								
34.299	546.519								
34.796	546.512								
34.841	546.511								
34.386	546.598								
34.3	546.55								

Table a.7 lists the runtime data from the Matrix Vector Multiplication Algorithm running in ideal conditions and with application load imbalance as discussed in Sec. 5.4.1 and Sec. 5.4.2. The runtime data is listed for three matrix sizes (10x10, 50x50, and 100x100) and two HPRC configurations (a master with two worker nodes and a master with four worker nodes). Table a.8, Table a.9, Table a.10, and Table a.11 list similar runtime data from the Matrix Vector Multiplication Algorithm running with application and background load imbalance as discussed in Sec. 5.4.4. Each table depicts a different background loading condition.

Table a.12 lists the runtime data from the AES Algorithm running in ideal conditions as discussed in Sec. 5.4.1. The runtime data is listed for four HPRC network sizes (2, 3, 4, and 5 nodes). Table a.13 lists similar runtime data from the AES Algorithm running with four different background load imbalance conditions on two nodes as discussed in Sec. 5.4.3. Table a.14 lists the runtime data from the AES Algorithm running with four different background load imbalance conditions on three nodes as discussed in Sec. 5.4.3. Table a.15 lists the runtime data from the AES Algorithm running with four different background load imbalance conditions on four nodes as discussed in Sec. 5.4.3. Finally, Table a.16 lists the runtime data from the AES Algorithm running with four different background load imbalance conditions on five nodes as discussed in Sec. 5.4.3.

Table a.17 lists the runtime and background load data (gamma) calculated from the model for the AES Algorithm running on homogeneous resources as discussed in Sec. 6.1.1.

Table a.18 lists the runtime (first column) and cost function data (columns 3-11) calculated from the model for the AES Algorithm running on homogeneous resources as discussed in Sec. 6.1.2. The second column is the number of workstations in the set. The cost function data columns differ by the value of x/c listed at the bottom of the table. Finally the Optimum P row denotes the optimum set of workstations for the given x/c value.

TABLE a.7 Matrix Vector Multiplication Algorithm No-load and Application Load Data (msec)

Matrix 10x10		Matrix 50x50		Matrix 100x100	
2 work nodes	4 work nodes	2 work nodes	4 work nodes	2 work nodes	4 work nodes
486	1271	1277	1351	3327	3020
487	896	1270	1495	3135	3392
835	1057	1287	1321	3140	3031
643	857	1238	1262	3130	3167
629	829	1272	1627	3123	2936
943	780	1289	1356	3170	3013
565	890	1276	1351	3375	3044
1066	795	1215	1500	3138	2984
526	800	1271	1379	3160	3252
550	866	1263	1380	3265	3026
565	901	1278	1376	3133	3500
619	865	1249	1459	3139	2997
540	1324	1243	1373	3148	3221
546	806	1208	1266	3146	3012
460	980	1279	1380	3145	2964
480	815	1258	1374	3143	3035
656	884	1263	1368	3493	3048
518	854	1249	1289	3144	3344
542	781	1271	1361	3135	3134
632	936	1280	1379	3152	3376
507	865	1283	1304	3140	2972
971	873	1258	1377	3149	2999
525	801	1253	1372	4020	3044
682	800	1233	1352	3144	3015
594	844	1127	1451	3407	2954
567	856	1256	1275	3150	2953
508	903	1289	1359	3170	3014
601	1036	1276	1425	3153	3151
589	904	1230	1330	3136	3005
519	832	1259	1388	3148	3052
778	861	1322	1395	3148	3003
893	795	1194	1378	3524	3012
662	905	1274	1474	3288	2980
585	1159	1281	1368	3143	3153
1136	1058	1284	1368	3141	3085
583	823	1269	1365	3420	2982
630	896	1277	1383	3153	3492
444	797	1278	1254	3147	3006
598	894	1265	7268	3158	3096
573	796	1278	1327	3141	3021
558	854	1276	1357	3271	2977
611	842	1276	1301	3148	2995
589	831	1272	1361	3161	3192
548	820	1271	1325	3162	2999
672	806	1352	1365	3226	3003
925	950	1271	1299	3138	3006
614	749	1270	1375	3131	2981
1042	786	1219	1359	3281	2995
715	806	1284	1296	3136	3006
993	802	1173	1381	3573	2984
940	834	1260	1365	3098	2981
594	832	1258	1316	3179	2991
503	764	1265	1369	3124	2998
585	799	1250	1368	3145	2997
492	775	1247	1332	3127	2990
867	836	1266	1285	3127	2992
517	830	1221	1289	3132	2992
644	825	1286	1342	3445	2958
836	842	1278	1290	3314	3004
761	816	1296	1316	3173	2971

TABLE a.8 Matrix Vector Multiplication Algorithm Background and Application Load Data (msec) Part I

Matrix 10x10		Matrix 50x50		Matrix 100x100	
gamma 1.265 2 work nodes	gamma 1.476 4 work nodes	gamma 1.265 2 work nodes	gamma 1.476 4 work nodes	gamma 1.265 2 work nodes	gamma 1.476 4 work nodes
530	3150	1279	1342	3128	2999
607	826	1267	1334	3284	5080
1532	767	1233	1274	3123	3540
595	764	1216	1300	6388	4390
2390	850	1275	1337	3125	2999
579	2364	1814	4042	3122	2950
546	829	1279	1299	3120	3020
584	2519	1228	3205	3127	2999
468	825	1787	1341	3118	4067
596	811	1249	1344	3363	3090
1621	802	1259	1319	3133	5336
613	793	1344	1362	6773	2994
2162	828	1282	1307	3106	3002
521	2385	1263	1346	3128	2998
601	841	1545	2768	3102	2999
443	1081	1273	1305	3128	3254
629	814	2575	1325	3127	2983
428	786	1223	1312	3443	6116
1792	819	1262	1369	3105	2995
530	825	1276	1331	7607	2983
2053	3134	1226	3959	3124	2983
546	787	1288	1360	3114	2985
517	2664	1484	2781	3119	3210
633	799	1752	1275	3116	2997
614	743	1274	1324	3118	6686
577	791	1170	1356	3544	2993
2206	827	1275	1360	3121	2977
612	823	1282	2390	7196	3000
2048	805	1246	1350	3128	2994
631	1841	1656	3245	3128	3661
533	791	1258	1363	3130	3288
561	803	1212	1268	3129	6382
618	782	2168	1330	3118	3000
508	793	1193	1353	3904	3006
2052	773	1264	1348	3131	2977
610	2767	1277	2877	7357	2999
2311	814	1262	1942	3125	3044
584	2187	1654	2545	3120	3514
586	739	1222	1288	3117	5735
603	748	1244	1324	3122	2986
582	838	2925	1341	3123	3008
558	823	1253	1343	4273	2978
1181	2115	1282	3727	3126	3006
542	838	1252	1360	7430	2994
1853	2279	1264	3114	3122	4792
626	807	1229	1322	3123	4406
560	776	1324	1356	3120	3076
582	826	1237	1257	3124	2998
570	789	2481	1281	3128	3003
589	813	1238	1281	3630	2994
2273	3605	1264	1410	3125	3001
606	745	1268	1572	8064	3553
1821	2049	1247	1701	3120	2997
573	829	1281	1351	3120	5166
620	788	1391	1364	3122	2996
550	830	1234	1277	3126	3000
594	768	2700	1318	3123	2987
621	3315	1242	3393	3723	3006
1636	745	1280	1275	3124	3266
605	2387	1262	3427	7124	3171

TABLE a.9 Matrix Vector Multiplication Algorithm Background and Application Load Data (msec) Part II

Matrix 10x10		Matrix 50x50		Matrix 100x100	
gamma 1.836	gamma 2.336	gamma 1.836	gamma 2.336	gamma 1.836	gamma 2.336
2 work nodes	4 work nodes	2 work nodes	4 work nodes	2 work nodes	4 work nodes
543	3779	1293	1489	3137	2998
618	2687	2731	1365	7940	3014
607	1971	4496	1317	7205	3000
592	833	1280	3682	3123	5466
607	748	1281	4343	3127	6121
558	2449	1232	3744	3205	4323
1894	3852	1990	1364	3999	2962
2717	2633	4473	1347	8136	3002
2318	824	3562	1373	3389	5096
584	837	1268	3884	3129	6144
615	828	1212	3540	3117	5467
607	3778	1271	2298	3429	2979
1725	3047	2302	1363	3813	3004
2909	858	4253	1308	8134	3058
1417	839	3790	4991	3125	6868
605	762	1250	3633	3124	5809
610	2962	1257	2419	3129	3154
1410	3435	1263	1350	3965	2999
1840	2150	1770	1334	6767	2997
1955	823	4027	4413	8216	6213
607	820	1892	4113	3130	7757
587	809	1282	2771	3126	3003
611	3371	1425	1384	3125	3008
1530	2676	1552	1298	4151	2996
2207	808	3423	4553	8775	6181
1995	812	4409	4076	6617	5675
461	898	1280	3620	3124	4030
596	3583	1267	1368	3117	2998
532	2857	1281	1287	3193	3000
1861	2107	2109	2818	3876	5592
2499	829	3770	4138	7826	6087
1899	822	4706	3852	4938	5132
597	2656	1274	1269	3129	3000
618	3306	1275	1236	3117	2933
2104	2857	1268	1362	3236	4109
1419	839	2262	4183	3974	6815
2312	810	4152	3349	8003	6586
629	773	4320	1474	3434	2913
485	3465	1281	1406	3123	2869
583	2536	1214	1319	3131	3257
2618	718	1300	4338	3717	6170
2524	827	2412	4181	8117	6601
2149	762	3692	1921	7777	3259
629	3593	1275	1263	3124	3007
604	2700	1239	1232	3122	3901
1094	2637	1268	4146	3121	6987
1689	790	1914	3413	4117	6245
2756	824	3428	2048	8220	2885
614	1110	4107	1460	5322	2900
597	3945	1206	1246	3128	3284
634	3136	1275	4125	3123	5719
2302	766	1295	2937	3655	6291
2562	800	2438	3230	3656	3956
2738	830	3828	1246	8115	2870
601	3921	3900	1246	3128	2903
558	2419	1204	1303	3126	5410
616	817	1253	3919	3107	7639
1605	814	1259	3461	3423	6462
2260	812	1901	1238	5639	2914
2316	3737	4434	1258	7576	2859

TABLE a.10 Matrix Vector Multiplication Algorithm Background and Application Load Data (msec) Part III

Matrix 10x10		Matrix 50x50		Matrix 100x100	
gamma 1.588 2 work nodes	gamma 1.975 4 work nodes	gamma 1.588 2 work nodes	gamma 1.975 4 work nodes	gamma 1.588 2 work nodes	gamma 1.975 4 work nodes
618	3859	1289	1361	9559	5317
2320	827	1250	1209	3596	3476
531	842	3496	1348	3129	3014
604	804	7037	1361	3123	3894
597	826	1276	5411	3119	7890
598	3506	1190	1486	5076	2978
1133	718	1247	1360	9680	2976
4000	842	1260	1376	3117	3000
542	783	4273	1300	3124	5932
593	808	3805	5158	3124	4331
523	4499	1259	2626	3123	3014
2281	4492	1277	1298	5624	2948
2990	801	1227	1327	4737	3044
626	779	2600	1293	3124	6434
562	826	3662	5732	3131	4063
596	3381	1272	5279	3128	3066
596	4819	1256	1347	4308	2806
2236	815	1253	1378	8086	2791
4477	780	1340	1372	3131	9434
578	810	2501	1288	3109	2844
529	809	6517	5966	3135	2815
606	3317	1244	1378	3120	3145
631	835	1258	1325	4524	7325
2936	798	1274	1374	9661	7473
551	781	1272	1350	3122	3009
592	814	2222	5057	3139	3018
512	5377	2857	4285	3127	3018
562	3650	1278	1355	3415	6073
3760	828	1281	1357	10792	4099
3352	828	1269	1358	3127	3004
545	830	2964	4911	3126	5760
575	5070	5616	6617	3126	7916
478	3685	1205	1369	3123	2994
564	820	1187	1355	5838	3024
1527	817	1268	1320	9321	3031
571	825	1280	3009	3125	7543
618	785	3692	5428	3130	3024
511	1436	5159	1327	3121	3032
601	3047	1250	1333	3161	3007
3280	831	1268	1319	8873	7398
3376	810	1283	1357	3129	7039
601	771	1273	5487	3123	2998
617	5452	3466	1928	3128	2991
555	3369	1278	1294	3128	3228
587	740	1268	1315	5601	6209
1831	815	1248	1310	10134	3828
2177	822	1278	5250	3120	3025
621	799	1622	5671	3131	3006
493	2361	4978	1305	3127	3364
616	2624	1255	1347	5008	8307
2434	808	1246	1368	9326	3187
469	818	1188	5858	3132	3108
585	800	1267	5865	3128	8403
569	4930	3996	1278	3125	7491
618	3822	5509	1344	3251	3049
3201	816	1265	1331	10006	3056
3616	744	1276	4035	3131	5607
518	811	1207	4730	3112	6345
518	4826	1268	1334	3127	3252
569	4424	3256	1313	3128	3052

TABLE a.11 Matrix Vector Multiplication Algorithm Background and Application Load Data (msec) Part IV

Matrix 10x10		Matrix 50x50		Matrix 100x100	
gamma 3.923 2 work nodes	gamma 5.254 4 work nodes	gamma 3.923 2 work nodes	gamma 5.254 4 work nodes	gamma 3.923 2 work nodes	gamma 5.254 4 work nodes
568	4292	1263	4739	3125	6784
692	4818	1265	6696	3953	9078
598	6729	4138	5318	10051	9192
761	3969	6803	5730	8083	7415
651	6564	3751	6626	4900	8680
1857	4495	6230	3550	8339	7696
3976	4869	6962	6618	8436	8470
2961	4644	6924	5501	14530	6333
4264	4683	4463	5680	11291	8592
3219	6822	5245	4908	8492	8133
3051	3442	7370	5614	12600	7699
3429	6292	4319	7026	12868	9101
3136	4500	6690	2922	11993	7367
2841	4363	4138	7682	10134	8883
2653	4186	6637	6098	13371	7328
4044	4964	3297	5626	4382	9441
3291	5387	7905	5365	11769	7639
3732	4302	6419	4839	5225	7538
2941	5737	6444	7555	10703	7686
4221	3945	6741	4637	8037	6956
3021	4839	4537	7540	11581	8720
3641	4326	6284	5363	9828	7156
2835	5287	2969	6528	5247	9909
4346	4720	5496	4812	9934	7304
2568	4386	5489	6342	5844	8041
2788	6291	4625	6152	10395	7267
3502	3762	5085	4965	9180	6494
3363	5463	2999	7161	4421	9218
2505	4135	6809	5375	9972	7202
3959	5047	5942	6785	7626	10470
3005	5260	3856	6424	11151	7099
4294	5463	5761	5846	9760	9809
3043	6248	4218	6139	8234	8051
4112	3742	6709	6734	9821	7967
3058	6853	5007	6423	4862	8364
4899	3994	8079	5552	10542	7043
3664	6903	6441	7275	6093	8449
4951	4128	4001	5878	11554	7501
3038	5429	6618	7052	10305	8984
4248	4567	3043	4462	6848	7205
2853	5009	7380	6406	9199	9711
3815	6010	5482	5838	3881	8026
2376	3735	8566	5337	9482	6656
5035	6646	4934	7245	9741	9557
3825	4752	4265	5595	9109	8166
3819	6022	6272	7152	10169	9386
2986	4934	3335	5780	5126	6707
3569	5945	6773	7260	10949	9792
3999	4444	5048	5764	4403	7116
3239	5575	6557	6817	8608	10270
4457	3968	5865	5609	9044	8580
3738	4100	4327	6624	8660	7810
3482	7528	7004	5953	10117	8060
2741	4808	4018	6316	4936	7008
3519	6271	5138	7367	8765	8942
2780	4225	6297	3832	8879	7029
4091	4857	3256	7423	8851	9882
3714	4980	7069	5577	8830	7674
2969	5542	3589	7098	3686	10758
2926	5021	7662	5703	8820	8078

TABLE a.12 AES Algorithm No-load Data (msec)

2 Node	3 Node	4 Node	5 Node
5141	2624	1843	1480
5172	2623	1844	1480
5149	2628	1841	1478
5146	2631	1842	1476
5133	2621	1844	1479
5138	2630	1845	1482
5160	2625	1842	1476
5150	2629	1845	1480
5158	2627	1843	1474
5147	2628	1843	1495
5139	2629	1844	1479
5149	2630	1844	1480
5155	2629	1847	1479
5144	2630	1840	1476
5149	2619	1843	1479
5141	2629	1845	1481
5154	2738	1844	1475
5126	2621	1846	1481
5143	2626	1845	1481
5147	2628	1860	1551
5152	2631	1845	1595
5141	2625	1847	1477
5144	2629	1975	1481
5131	2621	1845	1476
5146	2630	1845	1477
5139	2624	1843	1531
5133	2625	1843	1479
5133	2622	1847	1480
5141	2628	1843	1476
5138	2623	1846	1486
5128	2630	1842	1481
5133	2639	1844	1480
5144	2628	1842	1480
5142	2617	1844	1477
5199	2629	1853	1475
5131	2626	1842	1476
5134	2627	1846	1481
5141	2627	1842	1592
5139	2628	1843	1476
5137	2628	2020	1477
5131	2629	1843	1476
5136	2629	1839	1477
5147	2630	1844	1573
5130	2630	1845	1476
5145	2625	1845	1473
5143	2631	1846	1475
5146	2623	1843	1478
5153	2631	1845	1480
5156	2623	1846	1477
5147	2631	1842	1483
5153	2635	1845	1482
5148	2630	1848	1481
5154	2630	1845	1482
5152	2630	1844	1484
5136	2627	1845	1476
5140	2631	1843	1479
5151	2620	1842	1478
5130	2629	1844	1480
5146	2626	1847	1474
5138	2623	1842	1478

TABLE a.13 AES Algorithm Background Load Imbalance Data (msec) (2 Nodes)

Gamma	1.16	1.06	1.34	1.12
	5405	5111	5139	5109
	5110	5159	5140	5101
	5102	5108	5144	5112
	5106	5117	5146	5112
	5102	5106	5138	5107
	5321	5099	5132	5109
	5113	5335	5136	5110
	5102	5152	5143	5100
	5107	5103	5144	5109
	5112	5096	5145	5121
	5116	5109	5146	5104
	5101	5113	5138	5102
	5111	5121	5138	5099
	5196	5120	5142	5101
	5175	5106	5143	5572
	5105	5127	5144	5111
	5120	5119	5145	5112
	5114	5113	5146	5104
	5109	5110	5146	5114
	5334	5105	5149	5107
	5109	5324	5151	5100
	5110	5102	5156	5109
	5111	5111	5185	5106
	5105	5108	5617	5152
	5111	5105	5137	5102
	5111	5115	5138	5106
	5111	5099	5138	5103
	5118	5102	5139	5105
	5171	5111	5139	5536
	5105	5128	5139	5106
	5116	5112	5139	5107
	5104	5105	5139	5116
	5110	5111	5139	5102
	5343	5110	5140	5101
	5115	5359	5141	5108
	5103	5103	5141	5111
	5111	5101	5141	5103
	5115	5107	5141	5138
	5108	5105	5141	5098
	5101	5335	5141	5110
	5114	5099	5141	5097
	5104	5109	5142	5104
	5103	5096	5142	5542
	5112	5111	5143	5108
	5117	5107	5143	5109
	5110	5098	5144	5096
	5107	5103	5144	5098
	5340	5095	5144	5109
	5103	5101	5145	5111
	5101	5103	5145	5104
	5106	5109	5145	5100
	5102	5112	5148	5150
	5325	5108	5150	5105
	5126	5327	5154	5100
	5112	5111	5154	5107
	5118	5116	5155	5162
	5111	5101	5161	5573
	5103	5111	5188	5107
	5106	5102	5581	5103
	5125	5116	5599	5113

TABLE a.14 AES Algorithm Background Load Imbalance Data (msec) (3 Nodes)

Gamma	1.23	1.09	1.47	1.18
	2629	2623	2631	2628
	2621	2628	2628	2624
	2626	2624	2625	2626
	2622	2625	2622	2625
	2626	2619	2632	2682
	2630	2621	2632	2627
	2621	2624	2638	2624
	2625	2632	2774	2624
	2746	2627	2624	2623
	2621	2745	2624	2688
	2628	2643	2624	2630
	2624	2622	2625	2620
	2624	2624	2626	2625
	2624	2624	2626	2623
	2738	2627	2627	2627
	2634	2734	2629	2625
	2630	2626	2623	2625
	2623	2631	2624	2624
	2624	2625	2624	2636
	2627	2619	2624	2618
	2633	2623	2624	2628
	2619	2623	2624	2621
	2731	2629	2624	2631
	2622	2740	2624	2851
	2631	2624	2625	2620
	2632	2631	2625	2625
	2622	2626	2625	2627
	2629	2625	2625	2624
	2738	2625	2625	2623
	2625	2704	2625	2627
	2624	2633	2625	2625
	2651	2714	2626	2855
	2624	2626	2626	2692
	2622	2632	2626	2621
	2632	2642	2626	2629
	2626	2634	2626	2622
	2623	2623	2627	2625
	2622	2627	2627	2843
	2623	2632	2627	2624
	2631	2624	2627	2623
	2625	2630	2627	2625
	2624	2624	2628	2623
	2680	2622	2628	2625
	2650	2643	2628	2626
	2627	2619	2628	2632
	2631	2630	2628	2864
	2631	2618	2629	2657
	2628	2628	2629	2629
	2624	2730	2629	2628
	2623	2625	2630	2627
	2624	2622	2630	2631
	2623	2624	2630	2844
	2632	2633	2630	2630
	2621	2625	2630	2627
	2622	2622	2631	2631
	2743	2625	2632	2631
	2623	2783	2658	2620
	2628	2645	2855	2627
	2624	2624	2867	2624
	2622	2624	2902	2858

TABLE a.15 AES Algorithm Background Load Imbalance Data (msec) (4 Nodes)

Gamma	1.3	1.12	1.59	1.23
	1845	1844	1848	1846
	1839	1842	1846	1844
	1845	1845	1847	1844
	1918	1845	1849	1848
	1861	2033	1839	1845
	1843	1846	1840	1844
	1839	1844	1842	1843
	1846	1847	1843	1844
	1844	1846	1845	1844
	1917	1849	1846	1848
	1844	1922	1847	1847
	1842	2007	1893	1844
	1844	1844	1843	2003
	1844	1841	1843	1893
	1842	1842	1844	1843
	1845	1923	1844	1846
	1841	1843	1844	1845
	1845	1843	1844	1845
	1844	1846	1844	1990
	1845	1903	1844	1849
	1848	1846	1844	1845
	1842	1841	1844	1845
	1844	1855	1845	1848
	1844	1841	1845	1843
	1845	1845	1845	1844
	1842	1837	1845	1845
	1844	1842	1845	1931
	1843	1842	1845	1846
	1917	1845	1845	1843
	1844	1850	1846	1839
	1846	1846	1846	1844
	1846	1842	1846	1844
	1844	1841	1846	1843
	1843	1843	1846	1842
	1846	1844	1846	1843
	1845	1847	1846	1846
	1974	1843	1847	1846
	1901	1985	1847	2083
	1843	1887	1847	1845
	1848	1844	1847	1844
	1849	1846	1847	1846
	1840	1842	1847	1843
	1921	1846	1847	1845
	1844	1919	1848	1845
	1842	1846	1848	1842
	2548	1844	1848	1845
	1843	1846	1849	1846
	1845	1844	1849	1840
	1842	1937	1849	1844
	1842	1842	1849	1843
	1916	1847	1849	1846
	1921	1844	1861	1997
	1844	1844	1868	1843
	1844	1865	1920	1846
	1847	1845	1976	1837
	1848	1844	1977	1846
	1915	1844	1985	1843
	1847	1842	1994	1845
	1845	1846	1995	1848
	1845	1845	2004	1847

TABLE a.16 AES Algorithm Background Load Imbalance Data (msec) (5 Nodes)

Gamma	1.36	1.14	1.7	1.28
	1478	1478	1479	1478
	1475	1475	1479	1478
	1479	1481	1479	1477
	1478	1477	1479	1477
	1478	1480	1480	1502
	1477	1482	1481	1478
	1476	1481	1541	1479
	1478	1481	1591	1481
	1479	1477	1476	1478
	1538	1484	1476	1480
	1476	1479	1476	1478
	1477	1476	1476	1483
	1541	1500	1476	1479
	1478	1535	1477	1481
	1478	1480	1477	1479
	1478	1481	1477	1480
	1477	1480	1477	1476
	1478	1480	1477	1483
	1549	1539	1477	1663
	1481	1508	1477	1481
	1478	1478	1477	1479
	1481	1478	1478	1526
	1485	1480	1478	1480
	1516	1478	1478	1474
	1480	1533	1478	1477
	1482	1478	1478	1478
	1481	1476	1478	1477
	1480	1481	1478	1722
	1480	1577	1478	1475
	1510	1480	1478	1479
	1482	1480	1479	1480
	1535	1480	1479	1474
	1508	1489	1479	1740
	1480	1477	1479	1479
	1478	1479	1479	1480
	1476	1480	1479	1595
	1478	1479	1479	1490
	1479	1479	1479	1478
	1477	1478	1480	1478
	1480	1483	1480	1480
	1483	1479	1480	1592
	1484	1481	1480	1489
	1565	1481	1480	1480
	1478	1555	1480	1479
	1479	1481	1480	1481
	1486	1481	1480	1479
	1480	1575	1480	1591
	1477	1480	1481	1482
	1478	1480	1481	1479
	1480	1478	1481	1501
	1479	1478	1483	1476
	1480	1532	1483	1479
	1479	1520	1484	1477
	1482	1480	1495	1480
	1481	1497	1504	1763
	1480	1478	1538	1478
	1542	1477	1555	1480
	1481	1538	1587	1479
	1479	1478	1591	1475
	1535	1476	1592	1481
			1632	
			1651	
			1714	
			1784	

TABLE a.17 Runtime AES Application Homogeneous Resources Data

Workstations	Runtime (msec)	gamma
P=1	4933.236	
P=2	5148.045	1.12E+00
P=3	2688.246	1.18E+00
P=4	1873.48	1.23E+00
P=5	1469.646	1.28E+00
P=6	1229.944	1.32E+00
P=7	1072.123	1.37E+00
P=8	960.944	1.41E+00
P=9	878.801	1.45E+00
P=10	815.919	1.49E+00
P=11	766.439	1.52E+00
P=12	726.641	1.56E+00
P=13	694.049	1.59E+00
P=14	666.952	1.62E+00
P=15	644.133	1.65E+00
P=16	624.703	1.67E+00
P=17	607.996	1.70E+00
P=18	593.506	1.72E+00
P=19	580.843	1.75E+00
P=20	569.699	1.77E+00
P=21	559.829	1.79E+00
P=22	551.038	1.81E+00
P=23	543.166	1.83E+00
P=24	536.083	1.85E+00
P=25	529.681	1.86E+00
P=26	523.87	1.88E+00
P=27	518.575	1.90E+00
P=28	513.732	1.91E+00
P=29	509.287	1.92E+00
P=30	505.196	1.94E+00
P=31	501.417	1.95E+00
P=32	497.919	1.96E+00
P=33	494.671	1.98E+00
P=34	491.648	1.99E+00
P=35	488.828	2.00E+00
P=36	486.191	2.01E+00
P=37	483.721	2.02E+00
P=38	481.403	2.03E+00
P=39	479.224	2.04E+00
P=40	477.171	2.04E+00
P=41	475.235	2.05E+00

TABLE a.18 AES Application Optimum Set Cost Function Homogeneous Resources Data

4933.236	1		5426.56	7399.854	9866.472	29599.42	54265.6	251595	498256.8	2471551	4938169
5148.045	2		10810.89	12870.11	15444.14	36036.32	61776.54	267698.3	525100.6	2584319	5158341
2688.246	3		8333.563	9408.861	10752.98	21505.97	34947.2	142477	276889.3	1352188	2696311
1873.48	4		7681.268	8430.66	9367.4	16861.32	26228.72	101167.9	194841.9	944233.9	1880974
1469.646	5		7495.195	8083.053	8817.876	14696.46	22044.69	80830.53	154312.8	742171.2	1476994
1229.944	6		7502.658	7994.636	8609.608	13529.38	19679.1	68876.86	130374.1	622351.7	1237324
1072.123	7		7612.073	8040.923	8576.984	12865.48	18226.09	61111.01	114717.2	543566.4	1079628
960.944	8		7783.646	8168.024	8648.496	12492.27	17296.99	55734.75	103782	488159.6	968631.6
878.801	9		7997.089	8348.61	8788.01	12303.21	16697.22	51849.26	95789.31	447309.7	886710.2
815.919	10		8240.782	8567.15	8975.109	12238.79	16318.38	48955.14	89751.09	416118.7	824078.2
766.439	11		8507.473	8814.049	9197.268	12263.02	16095.22	46752.78	85074.73	391650.3	774869.8
726.641	12		8792.356	9083.013	9446.333	12352.9	15986.1	45051.74	81383.79	372040.2	735360.7
694.049	13		9092.042	9369.662	9716.686	12492.88	15963.13	43725.09	78427.54	356047.1	703071.6
666.952	14		9404.023	9670.804	10004.28	12672.09	16006.85	42684.93	76032.53	342813.3	676289.3
644.133	15		9726.408	9984.062	10306.13	12882.66	16103.33	41868.65	74075.3	331728.5	653795
624.703	16		10057.72	10307.6	10619.95	13118.76	16242.28	41230.4	72465.55	322346.7	634698.2
607.996	17		10396.73	10639.93	10943.93	13375.91	16415.89	40735.73	71135.53	314333.9	618331.9
593.506	18		10742.46	10979.86	11276.61	13650.64	16618.17	40358.41	70033.71	307436.1	604189.1
580.843	19		11094.1	11326.44	11616.86	13940.23	16844.45	40078.17	69120.32	301457.5	591879
569.699	20		11450.95	11678.83	11963.68	14242.48	17090.97	39878.93	68363.88	296243.5	581093
559.829	21		11812.39	12036.32	12316.24	14555.55	17354.7	39747.86	67739.31	291670.9	571585.4
551.038	22		12177.94	12398.36	12673.87	14878.03	17633.22	39674.74	67226.64	287641.8	563160.8
543.166	23		12547.13	12764.4	13035.98	15208.65	17924.48	39651.12	66809.42	284075.8	555658.8
536.083	24		12919.6	13134.03	13402.08	15546.41	18226.82	39670.14	66474.29	280907.5	548949
529.681	25		13294.99	13506.87	13771.71	15890.43	18538.84	39726.08	66210.13	278082.5	542923
523.87	26		13673.01	13882.56	14144.49	16239.97	18859.32	39814.12	66007.62	275555.6	537490.6
518.575	27		14053.38	14260.81	14520.1	16594.4	19187.28	39930.28	65859.03	273289	532576.5
513.732	28		14435.87	14641.36	14898.23	16953.16	19521.82	40071.1	65757.7	271250.5	528116.5
509.287	29		14820.25	15023.97	15278.61	17315.76	19862.19	40233.67	65698.02	269412.8	524056.3
505.196	30		15206.4	15408.48	15661.08	17681.86	20207.84	40415.68	65675.48	267753.9	520351.9
501.417	31		15594.07	15794.64	16045.34	18051.01	20558.1	40614.78	65685.63	266252.4	516960.9
497.919	32		15983.2	16182.37	16431.33	18423	20912.6	40829.36	65725.31	264892.9	513852.4
494.671	33		16373.61	16571.48	16818.81	18797.5	21270.85	41057.69	65791.24	263659.6	510995.1
491.648	34		16765.2	16961.86	17207.68	19174.27	21632.51	41298.43	65880.83	262540	508364
488.828	35		17157.86	17353.39	17597.81	19553.12	21997.26	41550.38	65991.78	261523	505937
486.191	36		17551.5	17745.97	17989.07	19933.83	22364.79	41812.43	66121.98	260598.4	503693.9
483.721	37		17946.05	18139.54	18381.4	20316.28	22734.89	42083.73	66269.78	259758.2	501618.7
481.403	38		18341.45	18534.02	18774.72	20700.33	23107.34	42363.46	66433.61	258994.8	499696.3
479.224	39		18737.66	18929.35	19168.96	21085.86	23481.98	42650.94	66612.14	258301.7	497913.7
477.171	40		19134.56	19325.43	19564.01	21472.7	23858.55	42945.39	66803.94	257672.3	496257.8
475.235	41		19532.16	19722.25	19959.87	21860.81	24236.99	43246.39	67008.14	257102.1	494719.6

Rp	P	x/c	0.1	0.5	1	5	10	50	100	500	1000
		Optimum P	1	1	7	10	13	23	30	41	41

Table a.19 lists the runtime (first column) and cost function data (third column) calculated from the model for the AES Algorithm running on homogeneous resources as discussed in Sec. 6.1.2. The second column is the number of workstations in the set. The cost function data is calculated based on the background load on the workstations.

Table a.20 lists the runtime (second column) and background load factor gamma (third column) calculated from the model for the AES Algorithm running on heterogeneous resources as discussed in Sec. 6.1.1. The results from two HPRC network sizes are given (8 nodes and 16 nodes).

Table a.21 lists the runtime (first column) and cost function data (columns 3-10) calculated from the model for the AES Algorithm running on heterogeneous resources as discussed in Sec. 6.1.2. The second column is the number of workstations in the set. The cost function data columns differ by the value of x/c listed at the bottom of the table. Finally the Optimum P row denotes the optimum set of workstations for the given x/c value.

TABLE a.19 AES Application Cost Function Based on Load Homogeneous Resources Data

Rp	P	Cost
5148.045	2	22857.32
2688.246	3	16559.6
1873.48	4	14763.02
1469.646	5	14108.6
1229.944	6	13922.97
1072.123	7	13980.48
960.944	8	14183.53
878.801	9	14482.64
815.919	10	14849.73
766.439	11	15267.46
726.641	12	15724.51
694.049	13	16212.98

TABLE a.20 Runtime AES Application Heterogeneous Resources Data

Workstations	Runtime (msec)	Gamma
P=1	4933.236	
P=2	5235.925	1.27E+00
P=3	2826.208	1.53E+00
P=4	2036.825	1.75E+00
P=5	1780.229	2.40E+00
P=6	1625.309	2.88E+00
P=7	1847.784	4.61E+00
P=8	1972.558	5.83E+00

Workstations	Runtime (msec)	Gamma
P=1	4933.236	
P=2	5235.925	1.27E+00
P=3	2826.208	1.53E+00
P=4	2036.825	1.75E+00
P=5	1649.485	1.93E+00
P=6	1421.359	2.08E+00
P=7	1306.71	2.35E+00
P=8	1226.075	2.57E+00
P=9	1166.757	2.75E+00
P=10	1121.73	2.91E+00
P=11	1125.305	3.23E+00
P=12	1127.788	3.50E+00
P=13	1129.874	3.74E+00
P=14	1131.881	3.95E+00
P=15	1347.254	5.21E+00
P=16	1507.915	6.20E+00

TABLE a.21 AES Application Optimum Set Cost Function Heterogeneous Resources Data

4933.236		1	5426.56	7399.854	9866.472	14799.71	29599.42	54265.6	251595	498256.8
5235.925		2	10995.44	13089.81	15707.78	20943.7	36651.48	62831.1	272268.1	534064.4
2826.208		3	8761.245	9891.728	11304.83	14131.04	22609.66	36740.7	149789	291099.4
2036.825		4	8350.983	9165.713	10184.13	12220.95	18331.43	28515.55	109988.6	211829.8
1780.229		5	9079.168	9791.26	10681.37	12461.6	17802.29	26703.44	97912.6	186924
1625.309		6	9914.385	10564.51	11377.16	13002.47	17878.4	26004.94	91017.3	172282.8
1847.784		7	13119.27	13858.38	14782.27	16630.06	22173.41	31412.33	105323.7	197712.9
1972.558		8	15977.72	16766.74	17753.02	19725.58	25643.25	35506.04	114408.4	213036.3
Rp	P	x/c	0.1	0.5	1	2	5	10	50	100
		P	1	1	1	4	5	6	6	6

Table a.22 lists the runtime and cost function data calculated from the model for the SAT Solver Application running on homogeneous resources as discussed in Sec. 6.1.3. The costs are calculated for three different cases of hardware size or number of hardware solver engines (4, 8, and 16), three different cases of hardware speed (based on the memory buss speed - divide by 3, 4, and 5), and three different cases of application load imbalance (beta equal to 6, 8.5, and 12).

TABLE a.22 SAT Solver Application Optimization Space Homogeneous Resources Data

	8 Copies		16 Copies		4 Copies	
Workstations	Runtime	Cost	Runtime	Cost	Runtime	Cost
2	4.13E+04	82674.71	2.07E+04	41337.63	82670.31	165348.89
3	2.76E+04	82673.67	1.38E+04	41337.27	55113.66	165346.49
4	2.07E+04	82673.35	1.03E+04	41337.31	41335.33	165345.45
5	1.65E+04	82673.35	8.27E+03	41337.50	33068.35	165345.06
6	1.38E+04	82673.52	6.89E+03	41337.80	27557.03	165344.94
7	1.18E+04	82673.77	5.91E+03	41338.15	23620.38	165345.02
8	1.03E+04	82674.07	5.17E+03	41338.56	20667.89	165345.19
9	9.19E+03	82674.47	4.59E+03	41338.99	18371.51	165345.43

	Divide by 5		Divide by 4		Divide by 3	
Workstations	Runtime	Cost	Runtime	Cost	Runtime	Cost
2	4.13E+04	82674.71	33068.29	66139.887	24801.28	49605.04
3	2.76E+04	82673.67	22045.64	66139.125	16534.3	49604.553
4	2.07E+04	82673.35	16534.32	66138.933	12400.82	49604.52
5	1.65E+04	82673.35	13227.54	66139.023	9920.736	49604.672
6	1.38E+04	82673.52	11023.02	66139.222	8267.352	49604.939
7	1.18E+04	82673.77	9448.368	66139.521	7086.367	49605.278
8	1.03E+04	82674.07	8267.381	66139.875	6200.63	49605.66
9	9.19E+03	82674.47	7348.838	66140.277	5511.726	49606.085

	Beta = 8.5		Beta = 6		Beta = 12	
Workstations	Runtime	Cost	Runtime	Cost	Runtime	Cost
2	351348	702699.513	2.48E+05	496023.28	496020.5	992045.96
3	234232.1	702698.642	1.65E+05	496022.85	330680.5	992044.81
4	175674.2	702698.557	1.24E+05	496022.84	248010.5	992044.48
5	140539.4	702698.405	9.92E+04	496022.94	198408.4	992043.98
6	117116.2	702698.371	8.27E+04	496023.17	165340.4	992044.05
7	100385.4	702698.804	7.09E+04	496023.51	141720.4	992044.22
8	87837.3	702699.278	6.20E+04	496023.90	124005.4	992044.44
9	78077.66	702699.721	5.51E+04	496024.30	110227.1	992045

VITA

As a devoted wife of Harrison Smith and the mother of twin daughters Allison Melissa and Courtney Diane, Melissa has carved a path through life with the utmost precision and detail. Melissa was born October 17th, 1970 to Donald and Diane Crawley in Pensacola, Florida. She attended J.M. Tate High School where she participated in the marching band and graduated with honors in 1988. After graduation, she enrolled in pre-engineering courses at The University of West Florida which allowed her to focus on her Electrical Engineering courses while gaining valuable experience as a co-operative education student with the Naval Aviation Depot at NAS Pensacola. In 1991 Melissa transferred to The Florida State University where, in 1993, she graduated Magna Cum Laude, earning a Bachelors of Science degree in Electrical Engineering. Melissa remained at The Florida State University as a graduate research assistant for the *High-Performance Computing and Research Laboratory* where she was a project team leader in ICASE tools and graduated with a Masters of Science in Electrical Engineering in 1994. While attending Florida State University, Melissa was selected as a member of the nationally distinguished engineering honor society, Tau Beta Pi. She proudly served as President of the Florida Eta chapter of Tau Beta Pi during the 1993-94 term. She is also a member of the Electrical Engineering Honor Society Eta Kappa Nu, Golden Key National Honor Society, and Phi Kappa Phi.

In 1994 Melissa accepted a research position with Oak Ridge National Laboratory (ORNL) in Oak Ridge Tennessee where she works as a member of the Monolithic Systems Development Group. Her research at ORNL has included the use of neural networks in control systems, sensor development, and specialized data acquisition systems for high energy physics experiments such as PHENIX and SNS. She is an active member in IEEE and ACM.

Melissa resides in East Tennessee where she is an active mother of twins, Allison and Courtney. She is an instrument rated private pilot and her hobbies, when she is not busy with her girls, include playing guitar and piano, golfing with her husband, watching football, and staying active in fitness and body building.

By her husband,

Harrison G. Smith II