Report Number: WUCS-2007-8

2007-02-02

Performance Evaluation for Hybrid Architectures

Praveen Krishnamurthy

In this dissertation we discuss methologies for estimating the performance of applications on hybrid architectures, systems that include various types of computing resources (e.g. traditional general-purpose processors, chip multiprocessors, reconfigurable hardware). A common use of hybrid architectures will be to deploy coarse pipeline stages of application on "suitable" compute units with communication path for transferring data. The first problem we focus on relates to the sizing the data queues between the different processing elements of an hybrid system. Much of the discussion centers on our analytical models that can be used to derive performance metrics of interest such as,... Read complete abstract on page 2.

Performance Evaluation for Hybrid Architectures

Praveen Krishnamurthy

Complete Abstract:

In this dissertation we discuss methologies for estimating the performance of applications on hybrid architectures, systems that include various types of computing resources (e.g. traditional general-purpose processors, chip multiprocessors, reconfigurable hardware). A common use of hybrid architectures will be to deploy coarse pipeline stages of application on "suitable" compute units with communication path for transferring data. The first problem we focus on relates to the sizing the data queues between the different processing elements of an hybrid system. Much of the discussion centers on our analytical models that can be used to derive performance metrics of interest such as, throughput and stalling probability for networks of processing elements with finite data buffering between them. We then discuss to the reliability of performance models. There we start by presenting scenarios where our analytical model is reliable, and introduce tests that can detect their inapplicability. As we transition into the question of reliability of performance models, we access the accuracy and applicability of various evaluation methods. We present results from our experiments to show the need for measuring and accounting for operating system effects in architectural modeling and estimation.

2007-8

Peformance Evaluation For Hybird Architectures, Doctoral Dissertation, December 2006

Authors: Praveen Krishnamurthy

February 2, 2007

Corresponding Author: praveenk@cse.wustl.edu

Abstract: In this dissertation we discusses methodologies for estimating the performance of applications on hybrid architectures, systems that include various types of computing resources (e.g., traditional general-purpose processors, chip multiprocessors, reconfigurable hardware). A common use of hybrid architectures will be to deploy coarse pipeline stages of application on "suitable" compute units with communication path for transferring data.

The first problem we focus on relates to the sizing the data queues between the different processing elements of an hybrid system. Much of the discussing center on our analytical models that can be used to derive performance metrics of interest such as, throughput and stalling probability for networks of processing elements with finite data buffering between them.

We then discuss to the reliability of performance models. There we start by presenting scenarios where our analytical model is reliable, and introduce tests that can detect their inapplicability. As we transition into the question of reliability of performance models, we assess the accuracy and applicability of various evaluation methods. We present results from our experiments to show the need for measuring and accounting for operating system effects in architectural modeling and estimation. We also point to lack in the ability of current estimation

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PERFORMANCE EVALUATION FOR HYBRID ARCHITECTURES

by

Praveen Krishnamurthy, M.S.

Prepared under the direction of Professor Roger Chamberlain

A dissertation presented to the Henry Edwin Sever Graduate School of
Washington University in partial fulfillment of the
requirements for the degree of

DOCTOR OF SCIENCE

Dec 2006

Saint Louis, Missouri

WASHINGTON UNIVERSITY

THE HENRY EDWIN SEVER GRADUATE SCHOOL

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

PERFORMANCE EVALUATION FOR HYBRID ARCHITECTURES

by

Praveen Krishnamurthy

ADVISOR: Professor Roger Chamberlain

Dec 2006

Saint Louis, Missouri

In this dissertation we discuss methodologies for estimating the performance of applications on hybrid architectures, systems that include various types of computing resources (e.g., traditional general-purpose processors, chip multiprocessors, reconfigurable hardware). A common use of hybrid architectures is to deploy stages of pipelined applications on "suitable" compute units.

The first problem we focus on is the sizing of data queues between the different processing elements in a hybrid system. The discussion centers on our analytical models that can be used to derive performance metrics of interest, such as throughput and stalling probability for networks of processing elements with finite data buffering between them.

We then discuss the reliability of the performance models. We start by presenting scenarios where our analytical model is reliable, and then introduce tests that can detect its inapplicability. Once we transition into the question of reliability of performance models, we assess the accuracy and applicability of various evaluation methods. We present results from our experiments to show the need for measuring and accounting for operating system effects in architectural modeling and estimation. We also point to a lack in the ability of current estimation methods (primarily simulation-based methods) to detect and analyze rare events in application execution. We use this and typical embedded benchmarks to demonstrate the ability and ease of emulation-based performance analysis.

We use BLASTN, a biosequence similarity search program, as our running example of a pipelined application in the dissertation. We present *Mercury* BLASTN, a reconfigurable hybrid system developed to accelerate BLASTN. We also use the performance evaluation techniques developed in this dissertation to aid in the performance estimates for *Mercury* BLASTN.

To Amma, Appa, Shams and Chigs

Contents

List of Tables

List of Figures

Acknowledgments

Though this work bears my name, there are a lot of people who have contributed very significantly to make this happen. First and foremost I would like to express my gratitude to Dr. Roger Chamberlain, my dissertation adviser. I have had the honor and pleasure of working with Roger for over 6 years. It goes without saying that this work would not have seen the light of day without his guidance over this period. We have worked on a lot of interesting problems in many aspects of Computer Science during this time, and this interaction has helped me mature as a researcher. The characteristic in Roger that I desire most to imbibe is his kindness. I hope to be able to contribute back to society in the many ways he does. Working with him has helped me realize that one can be extremely content/happy in life by being kind/helpful to others.

I am most grateful to Dr. Mark Franklin, who along with Roger gave me the opportunity to come to this institution and learn from the best. He has also been guiding aspects of this dissertation. I have always found his advice, especially when it comes to communicating ideas, extremely useful. I would also like to thank Dr. Ron Cytron, Dr. Jeremy Buhler for accepting me in research groups and helping me learn a lot in their areas of expertise. They along with Dr. Jason Fritts and Dr. Jim Buckley also served on my doctoral committee and guided me along to completion. I truly appreciate their patience and support during this process. I would like to thank Dr. John Lockwood for teaching me about digital design, and for patenting some of our research ideas.

I would also like to thank the CSE dept. at Washington University for accepting me as a student, and giving me a quality education. Phillip Jones, Richard Hough and Justin Thiel helped immensely with the Liquid Architecture infrastructure. Arpith Jacob and Joseph Lancaster helped implement the solution for *Mercury* BLASTN. I would also like to thank my colleagues who have made this long process fun. Sarang Dharmapurikar, Prashanth Pappu, Jai Ramamirtham, Todd Sproull, Naveen Singla, Eric Tyson and many others have been a good source of both intellectual discussions and good natured graduate student fun. I also thank Myrna Harbison, Jean Grothe, Peggy Fuller, Sharon Matlock and Stella Sung for making life simple and taking care of a plethora of problems over these years.

On a personal note, I take this opportunity to express my deepest gratitude to my parents Shri. N. Krishnamurthy & Smt. K. Karpagam for nurturing me over all these years with unconditional love and care, and making me the person I am. I would like to thank my dear

wife, Sharmila, for her love and patience. She has been a constant source of encourgement over the last two years. None of this would have been possible without their support and prayers. It's to them that I dedicate this work. My heartfelt thanks also go out to my in-laws, Shri. R. Sridharan and Smt. Nirmala Sridharan for their trust and encouragement. Lastly, I would like to thank Dr. P. K Rajagopalan and Mrs. Sivananda Rani for teaching me the value of engineering and math.

I would like to acknowledge the financial support from the following grants: NSF grants ITR-0313203, ITR-0427794 and CCR-0217334, NIH/NGHRI grant 1 R42 HG003225-01, and NSF Career grant DBI-0237902.

Sarve Janaah Sukhino Bhavantu

Krishnaarpanamastu

Praveen Krishnamurthy

Washington University in Saint Louis
Dec 2006

Chapter 1

Introduction

1.1 Hybrid Systems

Advances in new compute architectures, such as reconfigurable hardware via Field Programmable Gate Arrays (FPGAs), Chip Multiprocessors (CMPs), Graphics Processing Units (GPUs), Digital Signal Processors (DSPs) and Network Processors (NPs), have presented us with a variety of computational units. These computational units themselves evolved from different objectives, and have been very successful in their domains. However, integrating them effectively to harness their combined potential becomes an interesting problem in itself. An effective mix of these compute resources can be used to accelerate a wide variety of applications, without having to build customized solutions (logic) from scratch. Including FPGAs to the mix provides reconfigurability to the system. This enables developers to build customized acceleration engines for different applications from the same underlying hardware components.

We define "hybrid architectures" as systems where a variety of compute engines, such as FPGAs, CMPs, DSPs, NPs and GPUs, are put together to improve the performance of applications. We assume here that these engines are connected using an arbitrary interconnect.

We can tune the performance of several applications by using the resources of such a system synergistically. A system such as this provides us with a mix of high-performance hardware and highly flexible software. The designer can move functionality between hardware and software to meet the desired objectives. Besides performance measured in terms of throughput or operations per unit time, a hardware software co-design has to take into consideration other factors such as power, reliability, cost, and number of components used. An efficient

design process can be successful in decreasing the time to market of such systems, which in turn contributes to improving the design process.

1.1.1 Application Deployment on Hybrid Systems

The process of deploying an application on such systems starts with with a good understanding of the application kernels. Once we are able to partition the application into different kernels $\boldsymbol{E_i} \in \boldsymbol{E}$, we assign them to the different resources at our disposal. The resources $\boldsymbol{R_j} \in \boldsymbol{R}$ are connected using some suitable interconnection network.

For any arbitrary application, a mapping $\boldsymbol{M_q}$ between the resources and the application kernels is a function $\boldsymbol{M_q : E \rightarrow R}$ that assigns each kernel to a unique resource. Also, we assume that kernels have buffers in front of their inputs (provided by the resources) to store data when the kernel is busy. Now, the objective is to choose the best mapping $\boldsymbol{M_q}$, which meets the design requirements. Figure 1.1 illustrates this process.

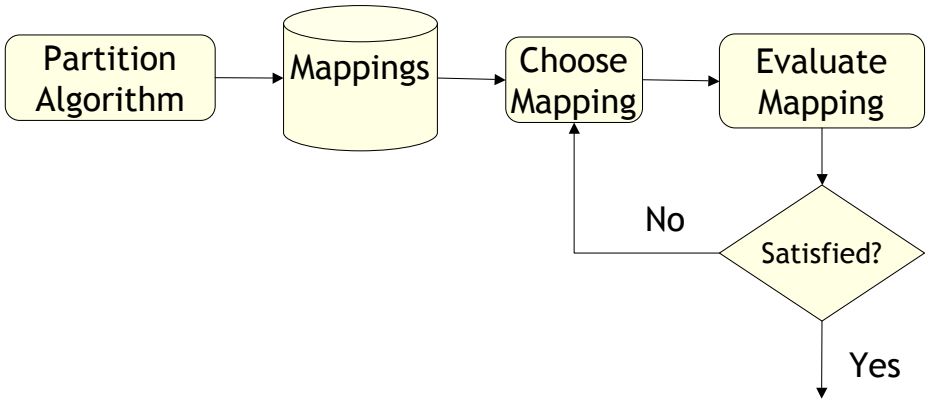


Figure 1.1: Process of selecting an acceptable mapping from a set

The evaluation process assesses a particular mapping and potentially guides the selection of the next mapping to examine. Typical estimation methods include analytical models, simulation models, and emulation models. Each of these have tradeoffs associated with them and are used at different stages of the design process.

1.1.2 Performance Estimation Techniques

Performance estimation models generally fall under the four classes, viz., analytical, simulation, emulation and direct measurement. The level of abstraction differs at each level, and

hence the detail and accuracy with which the system is represented varies. Input parameters to all of these models are derived by understanding the mapping of the application to the resources. The different estimation models are summarized as:

- ***Analytical Models***

  These models, in cases where they are applicable, are the easiest to implement. They use simple representations of the system, i.e. they adopt a rather liberal definition of "approximate" when modeling subsystems. These methods exploit highly abstract models and study the components which are anticipated to have the primary effect on the system behavior. Assumptions like Poisson distributions for arrival processes into the system are typical of these models. The key feature of such models is ease of evaluation, which directly reflects in the execution time of these models. This class of estimation is the fastest among the fore mentioned classes.

  Typically these are used for coarse-grain exploration of the design space, and are used to predict bottleneck stages. It is important to initially verify their reliability by comparing them with some other estimation model.

- ***Simulation Models***

  These methods currently enjoy the lion's share of performance estimation techniques used in systems research. In the design of application specific hybrid systems, they are employed to validate analytical models and also explore the design space. As simulation can model system characteristics without as many simplifying assumptions (as in the analytic models), these generally provide more accurate performance estimates compared to analytical models.

  Simulation models are both coarse-grain and fine-grain. In the coarse-grain simulation model, service processes are modeled as in the analytic model. The arrival process can be modeled from representative traces of application data. Fine-grain simulations, on the other hand, are complex simulation models which attempt to give cycle accurate performance estimates. Simulators have also been designed for hardware software codesign (co-simulators), which are used in the partitioning process.

  Simulation models are often more accurate than analytical models, but are time consuming to implement (code). Also, these methods (esp. fine-grained simulations) can potentially end up having prohibitively long execution times. We have earlier worked on the accuracy of "federated" simulation, a modeling scheme where we model a hypothetical system by combining simulation models of individual subsystems [23].

- ***Emulation Models***

  In these models we use FPGAs to model designs what will eventually be deployed

using other technologies (e.g., ASIC, ASIP). In this dissertation we use the Liquid Architecture platform [92] to assess the performance of an application (or subsystem) for its final deployment platform. This method of estimation is typically more accurate than the simulation models because an actual implementation of the application is what is used to measure performance. We use the platform to help model the performance of certain subsystems which are not well understood and are abstracted away in many of the above estimation methods. The downside of this type of model is the time required to set up an emulation system.

- ***Direct Measurement***
  Direct measurement based evaluation is the most accurate and reliable in the spectrum of performance assessment. It is primarily used to explore the design space with a fine-toothed comb. The accuracy here comes at the cost of prototyping individual design points, which typically consumes significant design time. This kind of evaluation is effective in cases where reconfiguring a system for individual design points is relatively inexpensive, or where the quantity measured cannot be obtained, with the desired degree of accuracy, from earlier methods.

  This technique is also used to validate all the assumptions that are made in the earlier models, and in this way also help determine the models of subsystems for future explorations.

Figure 1.2 illustrates the tradeoff between accuracy and execution time for the different performance estimation methods discussed earlier. Though important, the cost of coming up with the model is not shown in the picture. The level of abstraction at each level of modeling differs, and the accuracy of the model depends of its ability to assume "safe" abstractions. Based on the abstractions, there is a trade-off between accuracy of the model and the time it takes to obtain estimates from the model.

An important aspect in performance estimation (using any of the methods) is to derive models of individual components from a good representative set of data points. As this is typically difficult in initial designs, we explore techniques by which a model not only gives a performance estimate, but also hints as to its applicability. In other words, the model provides clues as to the range of the input parameter space over which it is applicable.

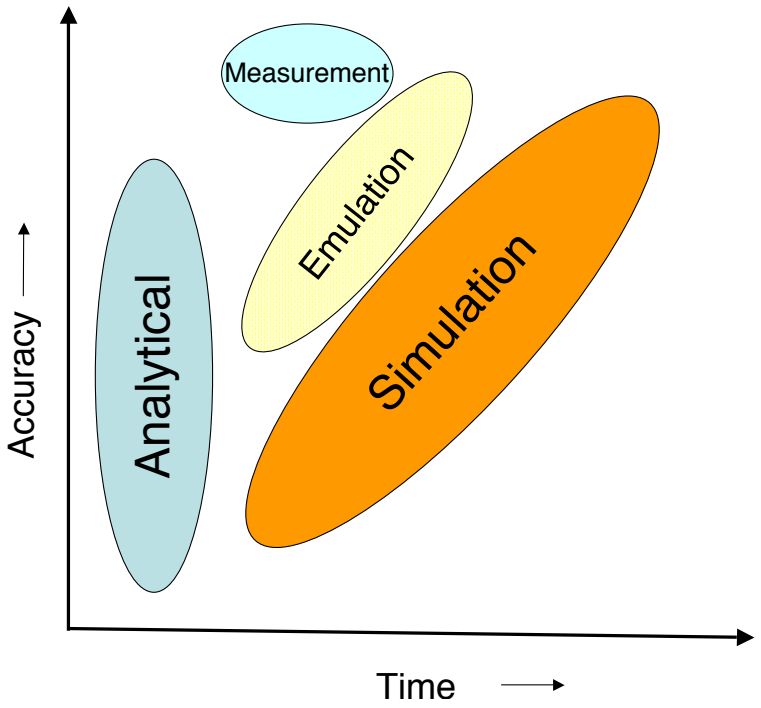


Figure 1.2: Time/Accuracy tradeoff for different estimation methods

1.2 Dissertation overview

As we have described before, an arbitrary application can yield multiple mappings of interest with the resources at hand. Also, these resources will be connected using some interconnection network. To choose the best among the available options, we would like a way to evaluate them. For this part of the problem we start by assuming that we know the characteristics of application kernels on the different resources they are deployed, and the uncertainty in the performance comes about because of interactions between kernels, including the impact of finite buffers.

Finite buffers in the datapath between kernels have been shown to have undesired effect of stalling upstream and downstream kernels. This makes a system with finite buffers non-work conserving, which makes their analysis by simple analytical methods difficult. Now, given that the search space to determine the optimal mapping between application kernels and resources can be large, we wish to investigate analytical (fast) methods to determine performance, and reduce the search space faster than other estimation methods. As part of this dissertation we analytically model the effect of finite buffers between different stages of pipelined applications. We choose to study pipelined applications, as we can easily map

the different stages of the application pipe to the different resources. Also, any application which is eventually deployed on such resources has a inherent pipeline structure to it.

Thus far we have assumed that we can predict the performance of an application kernel on the compute resource its deployed on, and we would use this to predict the performance of the entire system as a whole. Given that we need to estimate this parameter only a few number of times, we can resort to expensive (in time) simulation based methods. However, this leads to another question; how much does one believe the numbers from the simulation? Simulators often execute kernels in isolation and discount the interaction of the application with other processes that could be present in some compute resources. As part of this dissertation we address this particular issue, showing that performance of different applications can be significantly influenced by the execution environment of the system.

When deploying applications on hybrid systems, one needs to understand the application being deployed in detail. When presented with a variety of compute resources, one has to understand how to best use the resources available. Algorithm kernels can take different forms, while maintaining functionality, based on the nature of the resource they are deployed on. This is especially true when migrating an application kernel from the software domain to hardware or other parallelism rich domains. A mere migration of algorithms from one domain to another without fully utilizing the domain capabilities will almost always yield sub-optimal performance, and at times the performance can degrade, even significantly.

To illustrate the above point, we finish the dissertation by describing our algorithmic improvements to the BLASTN algorithm on the *Mercury* system [26], which is one such hybrid architecture. We demonstrate how smart changes to the algorithm and a good hybrid design can yield significant advantages. In this case, we not only migrate the functionality of the most expensive kernel of BLASTN to a hardware resource, but employ an efficient data structure that further improves the performance of the application.

1.2.1 Research questions

As part of this dissertation we attempt to answer the following research questions:

- The deployment of applications on hybrid systems often follows a narrowing down approach, in which the initial design space exploration is performed using analytical models. Thus, it becomes important to have a good reliability estimate of the models.

In the context of this dissertation, we are primarily interested in interstage queues in the hybrid system and hence attempt to answer the question:

> Can we develop analytical models that include both blocking and realistic service and arrival distribution?

also,

> How accurate are the predictions of analytical models that estimate the performance of queueing networks with blocking?

- Next we attempt to evaluate the effectiveness of methods that predict the performance of application kernels on resources. We are in particular interested in the effects of an operating system on an application kernel. In essence we are trying to answer the question:

  > How and when does the performance of an application, as predicted by standalone models, differ from its performance under an operating system?

- We finish by demonstrating the effectiveness of hybrid systems for improving the performance of applications. To be specific, we answer the question:

  > What algorithmic changes can improve the performance of BLASTN when it is deployed on the *Mercury* system?

1.3 Contributions

The specific contributions of this dissertation include the following items.

- *Mercury* BLASTN: We proposed the use of Bloom filters to improve the performance of BLASTN. We made effective use of the hardware resources available on the *Mercury* system, and our solution predicts performance of BLASTN with greater than an order of magnitude performance gain over software solutions.
  - We enhanced the first stage (Word Matching) of the BLASTN pipeline by using Bloom filters [70, 69].
- Queueing Models: We transformed the problem of assessing performance of an application on heterogeneous resources to that of a queueing network. Our contribution here lies in assessing the quality of analytical models used to determine the performance of queueing networks with blocking.

  - We extend the general method of assessing the performance of queueing network with blocking to phase-type service times and bursty arrival processes.
  - We compare the effectiveness of analytical models for a 2-node system with exponentially distributed service times and Poisson arrivals. We characterize the accuracy in terms of system parameters such as service rates and queue capacities.
  - We compare the effectiveness of analytical models for a 2-node system with phase-type service times and Poisson arrivals. We characterize the accuracy in terms of system parameters such as service rates and queue capacities.
  - We also study the effect of bursty departures in a 2-node system with phase-type service times and Poisson arrivals into the network. Again we characterize the accuracy in terms of input parameters like service time distribution, burst sizes, and queue capacities.

- Performance Validation: We investigate how the performance of applications is influenced under the presence of an operating system. We investigate cases when the performance of an application as predicted by standalone models (e.g., simulators) differs significantly from its performance under an operating system running other processes [71].

- Rare Events in Application Execution: We present a study of rare-events which affect the application total runtime. We use the statistics module on the Liquid Architecture platform to detect and explain the occurance of rare events.

- Dissertation related contributions: These are some of the things we have worked on which relate to the issues described in this dissertation:

  - We developed a write-back and a "dusty" cache policy [40] for LEON2, a 5-stage SPARC compatible processor.
  - We evaluated the effectiveness of the dusty cache policy for embedded workloads [71].
  - We developed Bloom filter based word matching stage for *Mercury* BLASTN [70].
  - We verified the effectiveness of a federated modeling scheme, to evaluate an optical path between memory and processors on a multiprocessor system [23].

- Other contributions: These are some of the significant contributions we have made in related areas of computer science, which are not described in the dissertation:

- Longest Prefix Matching: We helped develop and evaluate the use of Bloom filters for Longest Prefix Matching for route lookups in network routers [33, 34].
- Deep packet inspection using Bloom filters: We helped develop and evaluate the use of Bloom filters to detect malicious content in data transferred on network links [31, 32].

1.4 Dissertation Outline

With the objective of the work defined, this section gives the organization of the content of the thesis. Chapter 2 gives the reader an overview of the prior work in relation to the general theme of this dissertation.

Chapter 3 delves into developing analytical models to estimate the performance of applications deployed on hybrid systems. We use queueing models to study the effect of integrating different components of the applications. In this chapter we focus on the effect of downstream blocking on the overall performance.

In Chapter 4 we answer the question *Can we predict when the analytical models are giving us false estimates?* We describe our experiments which attempt to answer this question and the conclusions we derived from them.

In Chapter 5 we present results of our experiments that attempt to characterize the differences between the performance of an application under an operating system and a standalone estimation.

Chapter 6 is about *Mercury* BLASTN, which is our design of the well studied BLASTN application for hybrid *Mercury* system. We describe our enhancement to the BLASTN algorithm to effectively make use to the resources available on the *Mercury* system, and subsequently analyze the performance implications.

Chapter 7 discusses the conclusions we have arrived at as part of this work and the ways this work can be extended.

Chapter 2

Related Work and Background

2.1 Overview

This chapter gives the background and related work in three distinct areas in this dissertation. We start by introducing the reader to some basic queueing theory. We build on this to describe the work in the area of queueing networks with finite queues. Section 2.4 describes the current state of performance evaluation for architectures and microarchitecture. We give an overview of the techniques used and discuss on their pros and cons. We conclude this chapter with examples of applications which can significantly benefit from a hybrid deployment, with a strong emphasis on the BLASTN pipeline.

2.2 Queueing Systems

In this section we provide an overview of queueing models and also describe some basic queueing models we refer to in later chapters. Most of the results in this chapter can be found in standard textbooks on queueing systems [66, 115].

2.2.1 Kendall's notation

The basic queueing model is shown in Figure 2.1. We see a service station marked 'S', and buffer space marked 'Q' which holds and moves jobs to the server 'S'. Other parameters for the queueing system relevant to this dissertation include the arrival process 'A' of the customer, the service process and discipline, and the capacity of the the system. In this dissertation we restrict our investigation to FIFO queueing. Kendall introduced a four-part code $\boldsymbol{a/b/c/K}$ shorthand notation to characterize a range of these queueing models [66].

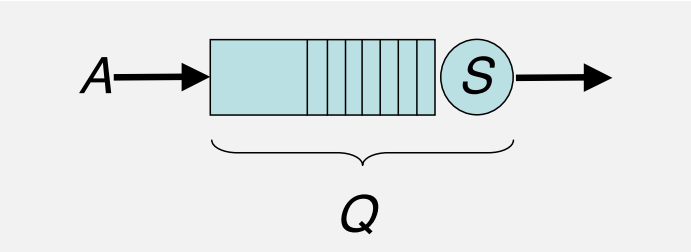


Figure 2.1: A single server queue

The first letter $\boldsymbol{a}$ specifies the interarrival time distribution, the second letter $\boldsymbol{b}$ specifies the service time distribution, the letter $\boldsymbol{c}$ specifies the number of servers in the system and the letter $\boldsymbol{K}$ refers to the capacity of the system, including the jobs being processed. In systems with infinite capacity the last letter 'K' is often dropped. Some common symbols for $\boldsymbol{a}$ and $\boldsymbol{b}$ include $\boldsymbol{M}$ for the memoryless or Markovian processes, $\boldsymbol{G}$ for a general distribution, $\boldsymbol{D}$ for deterministic times, and $\boldsymbol{PH}$ for phase-type.

Performance Measures

In general the mean arrival rate into the system is represented by $\boldsymbol{\lambda_{in}}$, and the mean rate of service is represented by $\boldsymbol{\mu}$. To prevent the queue from growing to infinity we must have $\frac{\boldsymbol{\lambda_{in}}}{\boldsymbol{\mu}} \leq \mathbf{1}$. It is common notation to use $\boldsymbol{\rho} = \frac{\boldsymbol{\lambda_{in}}}{\boldsymbol{\mu}}$ as the server utilization, and the earlier condition for stability is transformed to $\boldsymbol{\rho} \leq \mathbf{1}$. Relevant performance measure in the analysis of the queueing models are

- **Throughput**: The number of jobs that can enter the system in unit time, denoted by $\boldsymbol{\lambda}$.
- **Queue Distribution**: The distribution of number of jobs in the system. We denote $\boldsymbol{p(n)}$ as the probability of $\boldsymbol{n}$ jobs in the system.

Little's law

Another important relationship in queueing systems is given by Little's law [66] which states that

> The average number of customers in the stable system, $\boldsymbol{\rho < 1}$, is equal to their average arrival rate, multiplied by their average time in the system.

A variation of this, when applied to a specific position in the system is

> The probability, $\boldsymbol{\pi}$, that a customer on arrival occupies position $\boldsymbol{n}$ in the system is related to the throughput rate $\boldsymbol{\lambda}$, the service rate $\boldsymbol{\mu}$ and the probability $\boldsymbol{p(n)}$ as $\boldsymbol{\pi = \frac{p(n)\mu}{\lambda}}$.

2.2.2 Queueing models

We now layout, in brief, some queueing models used as part of this dissertation. Models which have a closed form solution for the performance measures of interest are discussed with these solutions. A detailed discussion on these models can be found in [66, 115].

$M/M/1$ Queue

As described in the notation above, the first $\boldsymbol{M}$ here stands for a memoryless arrival process. A Poisson process, which generates jobs with an exponentially distributed interarrival time between them is such a process. The second $\boldsymbol{M}$ represents the service process which is memoryless, i.e., the service time for jobs is exponentially distributed. Again, the average arrival rate into the queueing systems is represented by $\boldsymbol{\lambda_{in}}$ and the mean service rate is denoted by $\boldsymbol{\mu}$. From the notation before, this queue has only 1 service station, and the absence of a fourth symbol in its notation implies an infinite capacity. Also, the server utilization $\boldsymbol{\rho = \frac{\lambda_{in}}{\mu}}$ is assumed to be less than 1.

The performance measures for an $\boldsymbol{M/M/1}$ queue are obtained as

- Throughput ($\boldsymbol{\lambda}$): In a stable system i.e., $\boldsymbol{\rho < 1}$, the throughput of the system is indeed the arrival rate $\boldsymbol{\lambda_{in}}$, i.e., $\boldsymbol{\lambda = \lambda_{in}}$.
- Queue Distribution ($\boldsymbol{p(n)}$): This queue has a closed form solution for the queue distribution given by

$$\boldsymbol{p(n) = (1-\rho)\rho^n, \quad n = 0, 1, 2, \ldots} \tag{2.1}$$

  Note here that the occupancy in the queue is geometrically distributed.

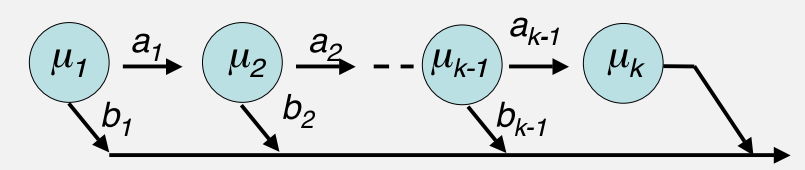


Figure 2.2: A phase-type distribution (Coxian)

A variant on this is the the arrival process where each arrivals brings in a bulk of jobs. The bulk itself has a geometric distribution with a mean of $\boldsymbol{x}$, and the queue model is represented by a $\boldsymbol{M^x/M/1}$ queue. This queue model is analyzed in [113].

$M/M/1/K$ Queue

This is the finite version of the $\boldsymbol{M/M/1}$ queue, where the number of jobs that can be in the system is limited to $\boldsymbol{K}$. This implies that at most $\boldsymbol{K-1}$ jobs can be waiting for service at this node. The performance measures for this queue are obtained as

- Throughput ($\boldsymbol{\lambda}$): As the system has only a finite capacity, jobs that attempt to enter the system when the queue is full will be dropped, which implies that $\boldsymbol{\lambda < \lambda_{in}}$.

$$\boldsymbol{\lambda = (1 - p(K))\lambda_{in}} \tag{2.2}$$

  where, $\boldsymbol{p(K)}$ is the probability that the queue is full.

- Queue Distribution ($\boldsymbol{p(n)}$): This queue has a closed form solution for the queue distribution given by

$$\boldsymbol{p(n) = \frac{(1-\rho)\rho^n}{1-\rho^{K+1}}, \quad n = 0, 1, 2, \ldots, K.} \tag{2.3}$$

$M/PH/1$ Queue

This queueing model is characterized by a single server with a phase-type service distribution [88]. We assume an infinite queue at the server, and that the arrival process to the system is Poisson. A server with a phase-type distribution is equivalent to a series of $\boldsymbol{k}$ exponential servers as illustrated in Figure 2.2. This service distribution is represented by $\boldsymbol{(\alpha, T)}$, where $\boldsymbol{\alpha}$ is a $\boldsymbol{1 \times k}$ row vector containing the probabilities with which the service process starts at each of the $\boldsymbol{k}$ phases, and $\boldsymbol{T}$ is a $\boldsymbol{k \times k}$ matrix containing the transition

rates from the k phases. T^0 is a $1 \times k$ matrix that contains the probabilities of ending the service from any of the k phases. The mean service time $\mu^{-1} = \alpha T^{-1} e$, where e is a column unity vector.

For the phase-type distribution shown in Figure 2.2 we have

$$T = \begin{pmatrix} -\mu_1 & \mu_1 a_1 & & & & & \\ & -\mu_2 & \mu_2 a_2 & & & & \\ & & -\mu_3 & \mu_3 a_3 & & & \\ & & & \vdots & \vdots & & \\ & & & & -\mu_{k-1} & \mu_{k-1} a_{k-1} & \\ & & & & & -\mu_k & \end{pmatrix}, \tag{2.4}$$

$\alpha = (1, 0, \ldots, 0)$, and $T^0 = [(1 - a_1)\mu_1, \ldots, (1 - a_k, \mu_k)]$.

- Throughput (λ): As the system has only a finite capacity, jobs that attempt to enter the system when the queue is full will be dropped, which implies that $\lambda < \lambda_{in}$.

$$\lambda = (1 - p(K))\lambda_{in} \tag{2.5}$$

  where, $p(K)$ is the probability that the queue is full.

- Queue Distribution ($p(n)$): This queue has a closed form solution for the queue distribution given by

$$p(n) = p(0)\alpha R^n, n = 1, 2, \ldots, K-1 \tag{2.6a}$$
$$p(K) = p(0)\alpha R^{K-1}(-\lambda T^{-1}) \tag{2.6b}$$

  where,

$$R = \lambda(\lambda I - \lambda e\alpha - T)^{-1} \tag{2.6c}$$
$$p(0) = \{\alpha[\sum_{n=0}^{K} R^n - \lambda R^{K-1} T^{-1}]e\}^{-1} \tag{2.6d}$$

A special type of phase-type distribution is the $Erlang_k$ distribution, where there are k phases and all of them have the same service rate (exponentially distributed service times) and every job that enters the queue receives service at all phases. This type of queue is represented as an $M/E_k/1$ queue [1].

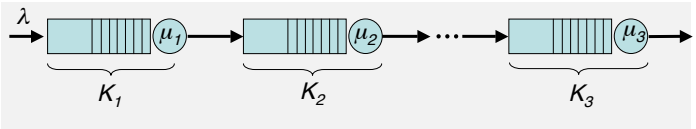


Figure 2.3: A single server queue

2.3 Queueing Networks with Finite Queues

In the previous section we described systems with exactly one server, i.e., jobs entering this node could potentially be queued before service, but once they finish service at this "one" node they exit the system. In a queueing network, any job that enters the system needs to be serviced at an arbitrary set of servers (nodes) before leaving the system. One such configuration is a tandem configuration shown in Figure 2.3.

The use of queueing networks for analyzing communication networks and production lines has been progressing over the last three decades [118]. For the cases where the networks have infinite system capacities there were very important contributions made by Jackson [56, 57], and Baskett, Chandy, Muntz, and Palacios the BCMP theorem [10]. These theorems prove that queueing networks with infinite buffering have product form solutions for the steady state probabilities. In order words, network of queues with infinite resources can be solved by analyzing each node in complete isolation. A queueing network analyzer, often referred to as QNA, based on such mean value methods was proposed and developed by Whitt [122, 121] and was further extended by Tahilramani [64].

However, real computer systems have finite resources and mean value models do not yield solutions for such networks. Finiteness introduces blocking, which in turn leads to interdependencies between adjacent nodes in the network. As a node can now be waiting to be unblocked rather than processing the next job in the queue, it becomes non-work conserving and analytical solutions to such queues become inherently difficult. We define a non-work-conserving node as one which is idle even when it has jobs pending service, in other words it is idle even when it could potentially be serving jobs.

A common way around this problem is to use simulation, or to use numerical state-space models incorporating "blocked" states. The simulation model is accurate when it is statistically valid, but can consume significant wall clock time. A state based model is typically difficult to realize for arbitrary cases, and in the cases where the model is realizable, the number of states in the model does not scale well with increasing nodes in the network [101]. Hillner and Boling [51] came up with approximate expression for the mean queue length for

configuration with exponentially distributed service times using a numerical approach. Simulation to derive performance measures are currently favored because of significant increase in compute power and the memory available to the designer.

2.3.1 Blocking Mechanisms

"Blocking" is the result of finiteness of queues between adjacent nodes in a network. As downstream nodes get filled, jobs finishing service at the immediate upstream nodes are not allowed to propagate further. This implies that even though an upstream server has the capacity to serve jobs (it is not busy processing), it still has to wait until it is unblocked. This makes the queueing network with finite queue a non work-conserving system and hence difficult to analyze analytically. The way a blocked node reacts is termed the "blocking mechanism." There are three different types of blocking mechanisms commonly employed, viz.,

- **Blocking After Service**: In this mode, the node completes service for its job and then waits for the downstream node to have "sufficient" space to accommodate its output.
- **Blocking Before Service**: In this mode, the node ensures that there will be "sufficient" space to accommodate its output in the downstream node, and then proceeds to service its current job.
- **Repetitive Service Blocking**: Upon completion of service, the node checks for the space in the downstream node. If there is not "sufficient" space available in the downstream node, it restarts the service in the current node. This procedure continues until there is space available downstream.

For the purposes of this dissertation we assume that we are evaluating networks which have a Blocking After Service (BAS) service mechanism. In our experiments a node completes service for its current job and waits for the downstream node to have "sufficient" space for accommodating its output, while continuing to accpet inputs. We also assume that there is enough space in the server to hold on to its outputs in the event that the downstream node is blocked. In algorithms described later in the text, this additional storage space is treated as part of the downstream node's queue in the analysis of the queueing network.

2.3.2 Approximate Analysis of Queueing Networks with Finite Queues

As exact solutions for such networks are not easy to derive, there has been work done in the past to approximately analyze such networks. One of the earliest work in the approximate analysis of such queues was by Caseau [21] in which the stability condition for such networks and also maximum sustainable throughput for these networks were discussed. The idea of analyzing nodes in isolation to derive the throughput of such system was first proposed by Caseau [21].

One configuration of a queueing network with finite queues that is of interest is one where the arrival process is Poisson and the service time distribution at the nodes is exponential. It was described in Asare [8] that a two node network with exponentially distributed service times and a Poisson arrival does not have a closed form solution because of its non-work conserving nature. Later two important methods were developed to analyze such networks:

- Memoryless Blocking Method: This method relates to blocking after service (BAS) blocking mechanism, and assumes that the time for which a job is blocked can be modeled by a memoryless process.
- Generalized Expansion Method: This method is was developed for queueing networks with repetitive service blocking. In this case the node that is blocked re-services the job until there is space for it downstream. In this method one inserts an artificial node between each pair of adjacent nodes, and the arrival and service process for that node are used to model the blocking between the adjacent nodes. We do not use this method as part of this dissertation.

This this dissertation, we extend the techniques of [96] and assess its range of applicability.

2.4 Performance Analysis of Hybrid Architectures

Performance analysis and models to estimate performance are an integral part of the hybrid design process. Analytical models are used to trim the space that the designer needs to explore to come up with a "good" architecture for the application of interest. However, one needs to use a more detailed model to fine-tune the architecture and closely analyze the tradeoffs. The direct measurement approach that would work the best for fine-tuning is often not feasible, and designers often resort to the traditional next best approach- *simulation*.

Simulation models often give designers more insight into the performance issues compared to an analytical model. They often have a finer resolution into the workings of a system compared to analytical models, and consequently can help designers make better decisions. Simulation is generally used to evaluate performance of a system before building the physical system.

Simulation is also a "model" of a real system and needs to be verified and validated. These steps to make the simulator credible often end up consuming much time and effort. A good simulation model lets the designer know of its underlying assumptions and the range of experimental conditions under which it is valid and those under which its results should not be trusted. Models in general should be tested for the effect of variability of input parameters, and the validity of the assumptions made in the model development process. This essentially points towards choosing and validating the level of abstraction in the simulation model. There is always a tradeoff between the accuracy of the simulation model and the execution time of the model to produce those performance predictions.

As hybrid architectures can potentially be needed to deployed a variety of applications, one needs a fast and accurate estimation method to evaluate each possible deployment of the hybrid architecture and choose the most appropriate configuration.

2.4.1 Simulator based system evaluation

Most tools for architecture evaluation are software based. They leverage a fast software development cycle for quickly exploring the design space. Also, the flexibility of software gives the ability to tailor the abstraction vs. accuracy tradeoff. Based on which aspect is traded we have either a functional simulator or a performance simulator. A simulator that trades accuracy for time is called a functional simulator, and when execution time is not as important as the accuracy of the architecture being evaluated, we term it a performance simulator. Also, a very significant advantage for the software based estimators has been the cost (dollars) associated with building and testing these models. A comprehensive list of simulators currently used in research is available at [105]. Here we describe two of the most widely used simulators viz. SimpleScalar [9], SimOS [102].

SimpleScalar

SimpleScalar is a software based simulator that comes in a variety of modes trading-off between execution time and detail. The most elaborate, performance mode, of SimpleScalar is called Sim-Outorder where SimpleScalar supports out-of-order issue and execution, and can provide information on micro-architecture details like branch-prediction/miss rates etc. In this mode SimpleScalar executes about 200 KIPS (Kilo Instructions Per Second).

Note here that SimpleScalar can estimate the performance of an architecture by executing standalone programs to stress the features of the architecture one is interested in, and cannot estimate performance for multiple applications executing together or the behavior of the application under a real operating system for this architecture.

SimOS

Simply put, SimOS is a very complex integrated (functional + performance) simulator which has the ability to run application traces under real operating systems. It is a superset of SimpleScalar and offers its users a higher level of detail compared to SimpleScalar. Like any other efficient performance estimation tool, it offers a suite of modes, which have the detail to speed tradeoffs. SimOS can run large commercial benchmarks for evaluating new architectures, and the size of the application does not force the speed-detail tradeoff. SimOS also supports both arbitrary point execution, and checkpoint based execution techniques.

2.4.2 Emulation based evaluation

As we saw in the earlier section, most architecture evaluation is currently done by software simulations. Unfortunately, computer systems are getting more and more complex, and subsequently it is becoming harder and more time consuming to validate simulation models for modern computer systems. It has been shown that simulation can be fraught with potential pitfalls and can systematically produce misleading results [6, 59]. In particular evaluating multiprocessor systems using uniprocessor simulators is shown to be very difficult [106].

Emulators allow designers to implement a circuit using FPGA devices instead of ASICs. This allows the "simulations" of the circuits to run much faster than software simulation. The use of FPGAs for performance monitoring has recently received a fair amount of attention. In terms of functionality, our approach resembles SnoopP [104], in that we both

augment a soft-core architecture (e.g., Microblaze for SnoopP) with logic to capture information based on instruction ranges. Our model, however, utilizes additional logic that allows users to correlate event behavior with the program counter, specific instruction address ranges, and also the process IDs in the operating system. The recently initiated RAMP [7, 95] project uses FPGA technology to perform architectural performance analysis, especially focusing on parallel computing architectures, but they have not yet described the specific mechanisms they intend to use. In a similar vein, although on a production chip rather than an emulation, IBM's Cell processor has extensive on-chip mechanisms for performance monitoring [44], of course limited to the specific architecture of the Cell processor itself.

The main drawback of emulation is the speed of compilation, which include synthesis, partitioning, and place and route. With the ongoing developments in these areas, engineers can now not only do these tasks faster, but can also change only part of their design to speed up the recompilation.

2.5 The Liquid Architecture System

The Liquid Architecture [62] system takes advantage of reconfigurable logic to permit timely design, prototyping, and analysis of new hardware modules. In this section, we describe the features of the Liquid Architecture project that were used to conduct experiments for this work.

2.5.1 Profiling

Programmers often want to know how their software utilizes the underlying micro-architecture. With an accurate view of what happens on-chip during a program run, a programmer may optimize his or her software to take better advantage of the hardware beneath. Feedback on such software-micro-architecture interaction is surely useful, but very difficult to gather. Unfortunately, many methods of gathering accurate software performance data have fundamental flaws in accuracy and timeliness.

Profiling software performance with other instrumented software can yield skewed results. In most cases, the profiling software adds extra overhead and provides a faulty report of processor activity. Other times, software profiling does not provide sufficient resolution

of performance information so the results are too vague to draw conclusions. Simulations can provide better detail, but they can take an extremely long time to evaluate the simplest of programs. Moreover, many software profilers and simulators do not account for (or cannot adequately model) some of the rare or less probable events that occur during normal execution such as memory stalls, dynamic scheduling, operating system interactions, multithreading effects, or external interrupts.

The Liquid Architecture system combines reconfigurable logic with a soft core processor, adding micro-architecture support for monitoring on-chip events and a web-based configuration and analysis interface. This system offers an effective solution to the above profiling problems enabling real-time, cycle-accurate performance analysis and permitting rapid design and testing of hardware and software structures. This infrastructure was used to provide results for this disserataion.

2.5.2 The Liquid Processor Module

The Liquid Architecture processor began as LEON [79], a standard SPARC V8 ISA for embedded systems, developed by the European Space Agency. Illustrated in Figures 2.4 and 2.5, the LEON processor provides typical micro-architecture features such as instruction and data caches, the entire SPARC V8 instruction set [108], and buses for high-speed memory access (the AHB) and low-speed peripheral control (the APB) [5].

The LEON processor is deployed on the Field-programmable Port Extender (FPX) platform [86]. The FPX provides an environment where FPGA designs can be interfaced with external memory and a high-speed network interface. OS support includes both uClinux [117] when the memory management unit (MMU) is absent and Linux kernel 2.6.11 when the MMU is present [27, 92].

2.5.3 Statistics Module Architecture

The statistics module exists as a custom VHDL core resident inside of the LEON2 processor [54]. Communications to and from the processor are handled by a wrapper interface connected to the APB. In addition to this standard interface, the statistics module receives processor state information via a custom *event bus*. This event bus essentially serves as a point-to-point connection between the statistics-capturing engine and various architectural

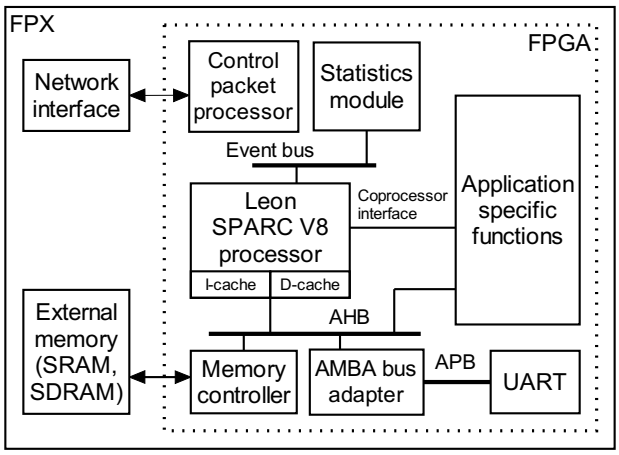


Figure 2.4: Liquid architecture block diagram.

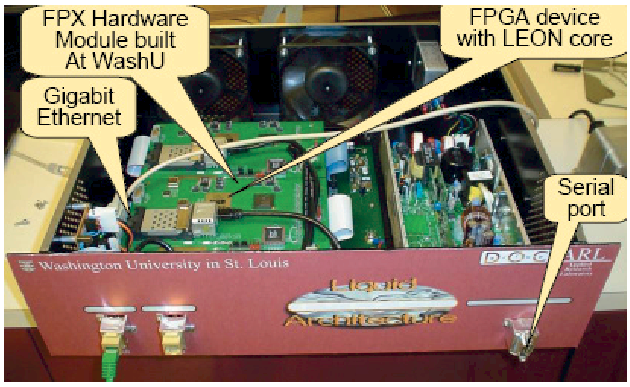


Figure 2.5: Liquid architecture photograph.

"hooks" placed throughout the system. Currently our event bus is configured to carry information regarding the state of the instruction cache, data cache, and PC address; however, as the LEON2 is an open processor, one could easily register new hooks to monitor any given architectural feature with minimal modifications to the VHDL.

The statistics-capturing engine is designed to count designated events, as described above, when they occur within a specified region of the executing program. A region is defined as a particular address range for the program counter, and the program's load map (with a suitable GUI) assists a developer in specifying address ranges of interest.

The statistics module provides a distinct advantage for gathering data in that it is fully programmable by the user at run time; the bitfile does not need to be resynthesized to accommodate new combinations of address ranges and events. To operate the statistics module, the user must program the collection mechanism with the following three different

values: the address range of interest, a timer duration between statistic collection, and the 32-bit counter that should be associated with a particular address range and event. By allowing the free association of counters, events, and address ranges, we are able to cover every possible instrumentation combination while utilizing a relatively small portion of the chip resources [54]. Once the user has instrumented their design by communicating the necessary information over the APB bus, the program will be executed and the module alerted. During the program's operation the statistics module will increment its internal counters that track each desired combination of events and address ranges, and store the resulting values for later retrieval every time the user-timer expires.

2.5.4 Operating System Operation

The instrumentation necessary for our purposes must track performance both within and outside of the application at hand. We next describe enhancements to the statistics module that accommodate performance profiling among processes running under an operating system. Such an environment requires consideration of two additional factors: the effect of the MMU and the scheduling of multiple competing processes. Fortunately, the former reduces to a simple matter of merely changing the address ranges that the module should watch for a given program. However, since each program shares the same virtual memory address space, a distinction between processes must be instituted to isolate individual program statistics.

To accommodate this, the user can associate a particular counter with a specific Process ID (PID). Counters with an associated PID will only increment if the corresponding process is in control of the CPU. If the user does not assert an association, the given counter will increment regardless of the current PID. A modification was made to the Linux scheduler so that, just before the scheduler finishes switching to a new process, it writes the PID of that process over the APB to the module.

2.5.5 PID Logging

Due to the competitive nature of thread scheduling in the Linux kernel, the variance in statistical results from one run to the next is greater than one would find when running a stand alone application. In particular, one could potentially see large differences in execution time if an infrequent process, such as a kernel daemon, should be scheduled in one experimental run but not the other.

To assist in tracking down these rare events, a PID log was added to the module. When the statistics module is started, a 32-bit PID timer is continually incremented. Once the scheduler writes the PID of the new running thread to the statistics module, the value of the PID counter and the new PID are stored into a BlockRAM FIFO. The PID timer is then cleared, and the counting continues until the next context switch by the scheduler. At the end of the run the user may then read back the log and get a clock-cycle accurate picture of the order and duration of the context switches within their system. Extensive details on the design and the operation of the Statistics module is presented in [54, 55].

In this dissertation we assess the impact of the operating system (OS) on the performance metrics one would be interested in measuring for evaluating architectures.

2.6 *Mercury* BLASTN

Computational search through large databases of DNA and protein sequence is a fundamental tool of modern molecular biology. Rapid advances in the speed and cost-effectiveness of DNA sequencing have led to an explosion in the rate at which new sequences, including entire mammalian genomes [119], are being generated. To understand the function and evolutionary history of an organism, biologists now seek to identify discrete biologically meaningful features in its genome sequence. A powerful approach to identify such features is *comparative annotation*, in which a *query sequence*, such as new genome, is compared to a large database of known biosequences. Database sequences exhibiting high similarity to the query, as measured by string edit distance [107], are hypothesized to derive from the same ancestral sequence as the query and in many cases to have the same biological function.

BLAST, the **B**asic **L**ocal **A**lignment **S**earch **T**ool [4], is the most widely used software for rapidly comparing a query sequence to a biosequence database. Although BLAST's algorithms are highly optimized for efficient similarity search, growth in the databases it uses is outpacing speed improvements in general-purpose computing hardware. For example, the National Center for Biological Information (NCBI) Genbank database grew exponentially between 1992 and 2003 with a doubling time of 12–16 months [87]. The problem is particularly acute for BLASTN, the BLAST variant used to compare DNA sequences, because each new genome sequenced from animals or higher plants produces between $\mathbf{10^8}$ and $\mathbf{10^{10}}$ bytes of new DNA sequence.

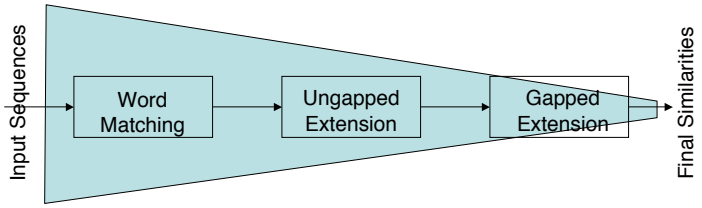


Figure 2.6: Pipeline stages of NCBI BLAST algorithm

BLASTN Pipeline

BLASTN is a 3-state pipeline (Figure 2.6), viz., word matching, ungapped extension, and gapped extension, where the volume of data processed decreases along the pipeline. However, the computational complexity increases down the pipe. An analysis of this pipe shows the bulk of time is spent in the first stage (*word matching*) of the pipe, where the database is scanned for seed matches for further inspection down the pipe. The two following stages (the ungapped and the gapped extension) extend the seed matches, extending it to check for significant similarity between the query and database around the seed.

2.6.1 Solution Strategies

One obvious approach to runaway growth in biosequence databases has been to distribute BLAST searches across multiple computers, each responsible for searching only part of a database. This approach requires both a substantial hardware investment and the ability to coordinate a search across processors. An alternate approach that makes more parsimonious use of hardware is to build a specialized BLAST accelerator. By using an application-specific architecture and exploiting the high I/O bandwidth of modern storage systems, an accelerator can execute the BLAST algorithms much faster than a general-purpose CPU.

The *Mercury* system [26] is a prototype architecture that supports disk-based computation at very high data rates using reconfigurable hardware. Computing applications historically have been coded using the following paradigm: read input data into main memory with explicit I/O calls, compute on that data writing results back to main memory, and send the output from main memory with explicit I/O calls. In contrast, the Mercury system is built around the concept of continuous data flow. Data from disk(s) flow into the computational resource(s); one or more functions (often physically pipelined) are performed on the data; and the results flow to the intended destination. As the computational resources include reconfigurable hardware, application deployment requires hardware/software codesign. The

Mercury system builds upon the work of Reidel [100] (active disks), Dally [30] (stream processors), and a host of work developed in the reconfigurable computing community.

State-of-the-art solutions

Several software tools exist that seek to accelerate BLASTN-like computations through algorithmic improvements. MegaBLAST [129] is used by NCBI as a faster alternative to BLASTN; it explicitly sacrifices substantial sensitivity relative to BLASTN in exchange for improved running time. The SSAHA [89] and BLAT [65] packages achieve higher throughput than BLASTN by requiring that the entire database be indexed offline before being used for searches. By eliminating the need to scan the database, these tools can achieve more than an order of magnitude speedup versus BLASTN; however, they must trade off between sensitivity and space for their indices and so in practice are less sensitive. In contrast, Mercury BLASTN aims for at least BLASTN-equivalent sensitivity.

Other software, such as DASH [67] and PatternHunter II [83], achieves both faster search and higher sensitivities compared to BLASTN using alternative forms of pattern matching and dynamic programming extension. DASH's reported speedup over BLASTN is less than 10-fold for queries of 1500 bases, and it is not clear how it performs at our query sizes, which are an order of magnitude larger. DASH's authors have also reported on a preliminary FPGA design for their algorithm [68]. PatternHunter II achieves a reported two-fold speedup relative to BLASTN, with substantially greater sensitivity, through judicious modification of its pattern-matching stage. We plan to implement similar improvements, based on Buhler et. al's studies of BLASTN-like pattern matching [18], in a future version of our system.

In hardware, numerous implementations of the Smith-Waterman dynamic programming algorithm have been reported in the literature, using both non-reconfigurable ASIC logic [35, 52] and reconfigurable logic [53, 93, 127]. These implementations focus on accelerating gapped alignment, which is heavily loaded in proteomic BLAST comparisons but takes only a small fraction of running time in genomic BLASTN computations. Our work instead focuses on accelerating the bottleneck stages of the BLASTN pipeline, which reduces the data sent to later stages to the point that Smith-Waterman acceleration is not necessary.

High-end commercial systems have been developed to accelerate or replace BLAST [94, 116]. The Paracel GeneMatcher^{TM} [94] relies on non-reconfigurable ASIC logic, which is inflexible in its application and cannot easily be updated to exploit technology improvements. In

contrast, FPGA-based systems can be reprogrammed to tackle diverse applications and can be redeployed on newer, faster FPGAs with minimal additional design work. RDisk [76] is one such FPGA-based approach, which claims a 60 Mbases/sec throughput for stage 1 of BLAST using a single disk.

Two commercial products that do not rely on ASIC technology are BLASTMachine2^{TM} from Paracel [94] and DeCypherBLAST^{TM} from TimeLogic [116]. The highest-end 32-CPU Linux cluster BLASTMachine2^{TM} performs BLASTN with a throughput of 2.93 Mbases/sec for a 2.8 Mbase query. The DeCypherBLAST^{TM}solution uses an FPGA-based approach to improve the performance of BLASTN. This solution has throughput rate of 213 Kbases/sec for a 16-Mbase query.

In this disseration we use BLAST as a recurring example of a pipelined application. We demostrate the use of hybrid architecture to improve the performance of the BLASTN pipe and also use the models developed in this dissertation to estimate its performance.

Chapter 3

Networks of Processing Elements with Finite Intermediate Queues

3.1 Introduction

Queueing networks have extensively been used to model real-life systems. These models can be easily parameterized and evaluated at low cost. Our interest in queueing networks stems from our interest in modeling application kernels mapped to distributed hybrid systems as a queueing network and estimating performance or buffer requirements by solving for relevant parameters of the queueing network.

We are particularly interested in methods to solve tandem queueing networks with bounded capacity queues (Figure 3.1), where data comes into the network only from the ingress node. In addition, once processed, not all data moves to the next computational stage, some is discarded. The probability that an input to stage $\boldsymbol{i}$ generates an input to stage $\boldsymbol{i+1}$ is represented by deliver probability, $\boldsymbol{d_i}$. Each stage of the pipelined application is treated as a server in the tandem network. Also, we assume that the Blocking After Service (BAS) blocking mechanism is employed in this queueing system. General parameters (with their notation) are presented in Table 3.1.

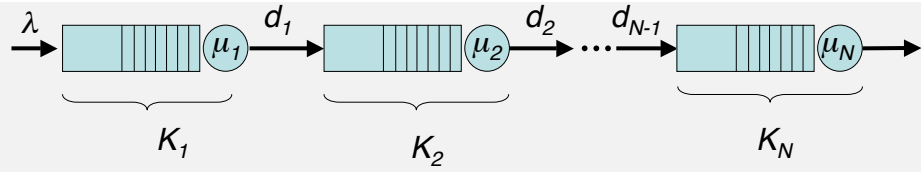


Figure 3.1: A tandem queueing network

Table 3.1: Notation and description of the terms in queueing networks

Parameter	Symbol	Description
Number of Nodes	N	The number of the nodes (server units) in the tandem network
Mean Arrival Rate	λ	Mean number of jobs that arrive into the system or node per unit time
Mean Service Rate	μ_i	Mean number of jobs that are processed by node i per unit time
Capacity	K_i	The maximum number of jobs that can be present at any node i, including the one (if any) in service
Deliver probability	d_i	The probability that an output from node i is an input to the immediate downstream node, node $i+1$.

We start with a technique from existing algorithms in the literature [96] that assumes exponential service times to obtain the throughput of such a system and the distribution of queue lengths for the individual queues. We have modified this algorithms to accounted for the deliver probabilities (d_i's) shown in Figure 3.1. We subsequently extend this method to incorporate phase-type service distributions and then extend them further to include bursty arrival processes. Throughout the modeling, we will validate the models against discrete-event simulations.

The algorithms attempt to account for blocking, or stall, which occurs when a node that is full "blocks" its upstream node from processing any further jobs. We assume blocking after service, where the upstream node completes processing its "current" job and then waits to be "unblocked". Throughout this modeling process, a distinction is maintained between a "true" parameter (one that reflects the properties of the physical system) and an "effective" parameter (one that has been altered to model some effect, typically blocking, in the physical system). The first example of this is the distinction between true and effective service rates at each node. In the case that a node is blocked by the downstream node, the effective service rate of that particular node decreases. The final node in the network is never blocked and hence its effective service rate is the same as its true service rate.

3.1.1 Simulation Procedure

As we are comparing predictions from an analytical model to discrete-event simulation it is important for the reader to understand our simulation methodology. Our simulation numbers come from a "home bred" queueing network simulator, whose correctness was

verified by testing it with known cases including $M/M/1$ and $M/M/1/K$ queues and Jacksonian Networks [56, 57].

Statistical Validity

For every network we simulate, we sample parameters of interest over 5 distinct intervals, each comprising of a million jobs processed. For each interval we compute the sample mean for all parameters of interest. We the use the technique of batch means to compute 95% confidence intervals for all the reported results from the simulator [77].

Simulation Procedure

As throughout the dissertation we are interested in the maximum throughput that can be supported by the queueing network, we need numerous simulations of a given network to arrive at that throughput. We start by assuming that the network is driven by an arrival rate, say $\boldsymbol{\lambda_{sim}}$ which equals the minimum service rate over all of the nodes. We determine the throughput of the of the network for this arrival rate, say $\boldsymbol{T_{sim}}$. If the difference between these values, $\boldsymbol{\delta = \lambda_{sim} - T_{sim}}$ is less than $\boldsymbol{\epsilon}$, our acceptance criteria, we stop. Else, we reduce the arrival rate into the network as $\boldsymbol{\lambda_{next} = \lambda_{sim} - \frac{\delta}{2}}$ and proceed to determine the new throughput of the system. We do this so that we can obtain an arrival rate where none of the arrivals are lost. When we do reach a value of the mean arrival rate which meets our acceptance criteria, we then increase the arrival rate by $\boldsymbol{\epsilon}$ until it fails the criteria. The last value of the mean arrival rate which meets the acceptance criteria is our estimate of the maximum throughput that can be sustained in the queueing network.

3.2 Analytical Approach to Solve Queueing Networks

In this section we introduce the algorithm described in [96], which attempts to determine the throughput and queue length distributions at the different nodes in the queueing network. The algorithm assumes exponentially distributed service times at the nodes in the network. Also, the arrivals into the network are assumed to be exponentially distributed, i.e., a Poisson process with exponentially distributed inter arrival times. In this algorithm we start at the very last node in the network, which is never blocked, and work our way backwards.

When analyzing any node i in the network, we modify its service time distribution using our understanding of nodes downstream of node i.

We start by assuming an overall throughput of the system, say T_0, which is the minimum of the service rates of the individual nodes. Now, a mean throughput of T_0 of the system implies that node i in the network will have a mean throughput rate of $t_i \leq T_0$, accounting for filtering downstream. This throughput, t_i, at node i includes the jobs that are dropped after they are serviced. We then proceed to analyze all the nodes within in this network in isolation moving from back to front.

At each node i we determine the mean arrival rate of a hypothetical lossy arrival process, which has exponentially distributed interarrival times, for which the node has a throughput of t_i. For this arrival process, we determine the probability that the queue at node i is full, say p_{full}. This probability is then transformed to the probability that a job leaving the node $i-1$ will be blocked by node i, say π_{block}. π_{block} is derived using Little's law [66], from the assumed departure rate of the upstream node, the effective service time distribution of node i and the probability that node i's queue is full p_{full}.

The first node that can experience blocking is node $N-1$. The effective service distribution of node $N-1$, immediately upstream to the last node, is modified (to reflect the impact of blocking) as shown in Figure 3.2. It illustrates that all jobs entering node $N-1$ will be serviced with a mean service rate of μ_{N-1} and will experience an additional exponentially distributed delay with a mean of $\frac{1}{\mu_N}$. In the general case (i.e., when analyzing node i) we will have a total of $N-i+1$ phases of service as shown in Figure 3.3.

We proceed in a similar way back to the very first node. Once we evaluate the arrival process at the very first node, we check to see if the rate of jobs discarded at the very first node is within ϵ. If so, we have determined the throughput of the system and the probability of blocking at all nodes. If not, we reiterate with a lower value of system throughput than previously assumed.

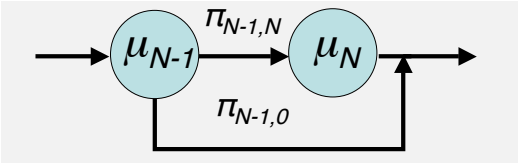


Figure 3.2: Coalescing nodes $N-1$ and N to obtain the effective service time distribution of node $N-1$.

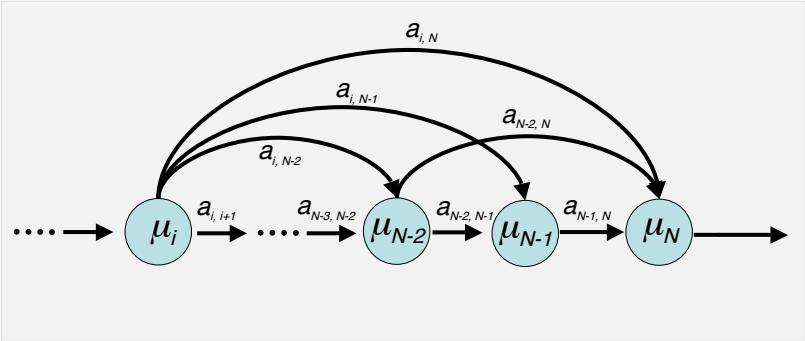


Figure 3.3: Effective service time of an interior node i accounting for potential blocking downstream from it.

The assumption made by this algorithm is that the departures from nodes have an inter-departure time that is exponentially distributed. This assumption justifies the modeling of the arrival process, when analyzing intermediate nodes in isolation, as being memoryless. We will be studying the impact of this assumption in the following chapter.

3.2.1 Exponentially Distributed Service Times

In this section we describe the algorithm in detail when the arrivals into the network of queues have an exponentially distributed interarrival time and the service times themselves are exponentially distributed. To this we add the probability that a job leaving node i will enter node $i+1$ with a probability d_i. Our objective here is to determine the system throughput and probability of blocking in the network. We compute the results from this algorithm and compare them against those derived from our simulation infrastructure.

Procedure

Table 3.2 lists the variables used in the solution for stalling probabilities and throughput for the backward traversing algorithm with exponentially distributed service times. As mentioned before, we start at the last node in the network and work our way backwards. We analyze each of the nodes in isolation, wherein we alter the service distribution of the nodes based on estimated blocking by downstream nodes. Figure 3.3 shows the service time modeled at any interior node i of the network.

We start by assuming that the mean throughput supported by the network is the minimum rate of service at the different nodes in the network. Our initial estimate of the maximum

Table 3.2: Notation and description of the terms as applicable to this particular algorithm

Symbol	Description
T_j	The estimated (assumed) mean throughput of the queueing network for the j^{th} iteration of the algorithm
t_i	The expected throughput at node i of the network
λ_i	Hypothetical lossy mean arrival rate into node i of the network, the jobs are assumed to have an exponentially distributed interarrival time
μ_i	Mean service rate of node i of the network, the service time distribution at each of the nodes is given to be exponential
d_i	Probability that a job completing service at node i needs service at node $i+1$
$p_i(x)$	Probability of having x jobs at node i in the network
π_i	Probability of a job leaving node i being blocked from entering node $i+1$ of the network
$w_i(m)$	Probability that a job leaving node i is blocked at service phase m of node $i+1$
$a_{m,n}$	Probability of transition from phase m to phase n of phase-type distributions modeling effective service times

ingest rate is $T_0 = min_i(\frac{\mu_i}{\prod_{\forall j, j<i} d_j})$. We traverse back from the last node, assuming that the expected throughput at node N is $t_N = \prod_{\forall i} d_i T_0$.

1. Analyze the N^{th} node as a $M/M/1/K_N+1$ queue. The capacity of the node is incremented by 1 to account for the customer that has completed service in node $N-1$ and is waiting for space in the queue for node N.

$$\lambda_N = t_N/[1 - p_N(K_N+1)], \tag{3.1a}$$

where,

$$p_N(K_N+1) = (1-\rho_N)\rho_N^{K_N+1}/(1-\rho_N^{K_N+2}) \tag{3.1b}$$

and $\rho_N = \lambda_N/\mu_N$

$$\pi_{N-1} = \mu_N p_N(K_N+1)/t_N \tag{3.1c}$$

2. Analyze each node i as an $M/PH_{N-i+1}/1/K_i+1$ node, for $i \in [N-1, \ldots, 2]$. The expected throughput at each of these nodes, t_i, is obtained as $t_i = \frac{t_{i+1}}{d_i}$. The effective service at the i^{th} node, as described in [96], is given by the phase-type distribution

$(\boldsymbol{\alpha_i}, \boldsymbol{Q_i})$, where $\boldsymbol{\alpha_i}$ is a row vector with $\boldsymbol{N-i+1}$ elements all equal to zero except the first element that is set to 1, and

$$\boldsymbol{Q_i} = \begin{pmatrix} -\mu_i & \mu_i a_{i,i+1} & \mu_i a_{i,i+2} & \cdots & \mu_i a_{i,N-1} & \mu_i a_{i,N} \\ & -\mu_{i+1} & \mu_{i+1} a_{i+1,i+2} & \cdots & \mu_{i+1} a_{i+1,N-1} & \mu_i a_{i+1,N} \\ & & -\mu_{i+2} & \cdots & \mu_{i+2} a_{i+2,N-1} & \mu_{i+2} a_{i+2,N} \\ & & & & \vdots & \vdots \\ & & & & -\mu N - 1 & \mu_{N-1} a_{N-1,N} \\ & & & & & -\mu_N \end{pmatrix} \tag{3.2}$$

$\boldsymbol{\lambda_i}$, which is the hypothetical arrival rate into node $\boldsymbol{i}$, is obtained by solving the fixed point expression, $\boldsymbol{\lambda_i = t_i/[1 - p_i(K_i+1)]}$. Where,

$$\boldsymbol{p_i(K_i+1) = p_i(0)\alpha_i R_i^{K_i}(-\lambda_i Q^{-1})}$$

$$\boldsymbol{R_i = \lambda_i(\lambda_i I - \lambda_i e_i \alpha_i - Q)^{-1}}$$

$$\boldsymbol{p_0 = \{\alpha_i[\sum_{r=0}^{K_i} R_i^r - \lambda_i R_i^{K_i} Q_i^{-1}]e_i\}^{-1}}$$

and $\boldsymbol{e_i}$ is a column vector of 1s.

After solving the above fixed point, we compute $\boldsymbol{w_i(m), m = i, i+1, \ldots, N}$ using

$$\boldsymbol{w_i(m) = \frac{p_i(0)\alpha_i R_i^{K_i} e_m}{p_i(K_i)}}, \tag{3.3a}$$

and obtain $\boldsymbol{\pi_{i-1}}$ using the relation,

$$\boldsymbol{t_i \pi_{i-1}[-w_i Q_i^{-1} e_i] = p_i(K_i+1)}, \tag{3.3b}$$

where $\boldsymbol{w_i = (w_i(i), w_i(i+1), \ldots, w_i(N))}$.

3. The quantities $\boldsymbol{w_i}$ and $\boldsymbol{\pi_{i-1}}$ are used to characterize the effective service time for node $\boldsymbol{i-1}$. In particular obtain the transition probabilities $\boldsymbol{a_{m,n}}$ for node $\boldsymbol{i-1}$ as

$$\boldsymbol{a_{i-1,n} = \pi_{i-1} w_i(n)} \tag{3.4}$$

and other already known transition probabilities, i.e., $\boldsymbol{a_{m>i-1,n}}$.

4. The ingress node (node 1) is analyzed as a $M/PH_{N-i+1}/1/K_1$ node using the matrix-geometric procedure [88]. The mean arrival rate at the first node in the network is the mean of the exponentially distributed arrival process that is required to have an output rate equal to the assumed throughput.

If the mean of the arrival rate into the system is within ϵ of the assumed throughput, then we conclude that we have a sustainable arrival process, i.e, there will be no losses at ingress for this rate of arrivals. The sustainable arrival process with the minimum average interarrival time will yield the maximum throughput of the system.

Convergence of the algorithm

We assess the convergence of the algorithm as follows. The initial estimate on the throughput, T_0, is an upper bound on the maximum sustainable throughput T_{max} (the value we are attempting to determine). We represent the upper bound on this T_{max} as T_{ub}.

During an iteration with a throughput estimate of T_j, at each node i the expected throughput is $t_i = \prod_{\forall j<i} d_j T_j$. This is compared to the effective service rate $\bar{\mu}_i$, which includes the effect of blocking downstream. If t_j is greater than $\bar{\mu}_i$, the iteration is terminated and t_i is reduced to $\bar{\mu}_i$. This in turn decreases the system throughput to $T_{j+1} = \frac{t_i}{\prod_{\forall j<i} d_j}$, and T_j becomes the new upper bound on the sustainable throughput, T_{ub}. We then iterate with T_{j+1} as the new estimate for the system throughput.

When an iteration with throughput estimate T_j isn't terminated at an interior node i, the necessary input rate $\lambda = \lambda_1$ at node 1 is compared with T_j. If the required λ is ϵ or more greater than T_j we conclude that the pipeline cannot sustain throughput T_j, and we decrease the expected system throughput for the next iteration to $T_{j+1} = T_j - (\lambda - T_j)$. In this case T_j again becomes the new upper bound on the sustainable throughput, is T_{up}, which is potentially tighter than the previous upper bound.

When an iteration with throughput estimate T_j isn't terminated at an interior node and the input rate λ is within ϵ of T_j, we conclude that the pipeline can sustain a throughput T_j. This value of T_j is now a lower bound on the maximum sustainable throughput, i.e, $T_{lb} = T_j$. This case establishes the window T_{lb} to T_{ub} which contains our maximum sustainable throughput T_{max}. Once we have established this window, we can use one of many methods to search this space efficiently. We use a binary search of this space, where $T_{j+1} = T_j + \frac{windowSize}{2}$, when T_j yields a sustainable throughput and establishes

$T_{lb} = T_j$. Else, $T_{j+1} = T_j - \frac{windowSize}{2}$, when T_j fails to yield a sustainable throughput and subsequently establishes $T_{ub} = T_j$. We stop when $windowSize \leq \epsilon$, and this guarantees convergence.

Performance of the algorithm

To evaluate the effectiveness of the algorithm in determining the throughput of the queueing network, we tested it on 200 synthetically generated queueing networks. The parameters N, $\mu_i \;\; i \in [1, N]$, K_i are synthetically generated with their bounds given in Table 3.3. The particular values of the parameters used for these 200 networks are detailed in Appendix A. For each of the networks we compared the throughput predicted by the algorithm to those obtained using our tandem queueing network simulator. Comparison of the raw throughput

Table 3.3: Range of parameters used for the backward traversing algorithm with exponentially distributed service times

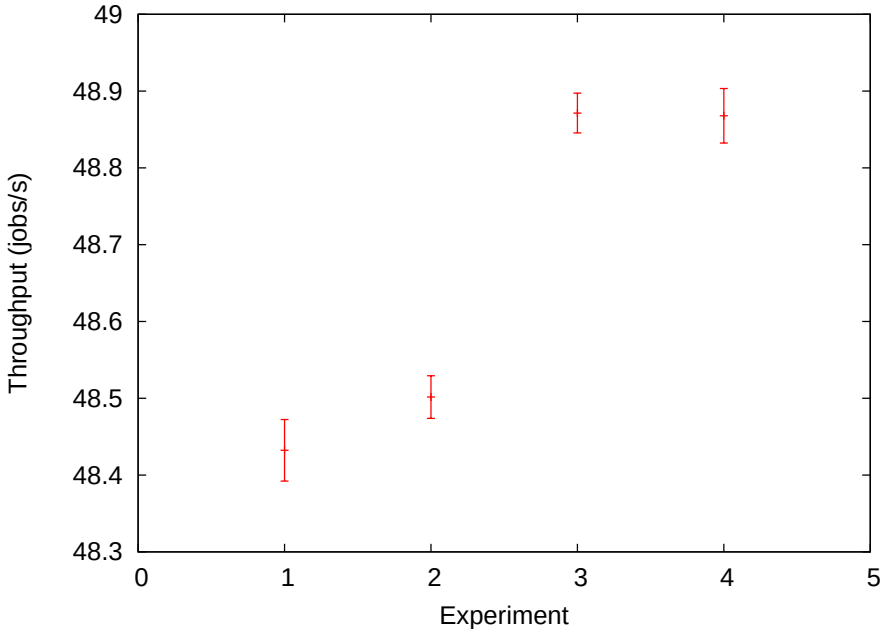
Symbol	**Description**	Range of values
N	The number of nodes in the network	{2, 3, . . . , 10}
K	Capacity of the network	{5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120}
μ_i	Mean service rate of node i	{10, 20, . . . , 1000}

predicted by the two models is illustrated in Figure 3.5(a). The difference is measured as a relative error between the two models, defined as

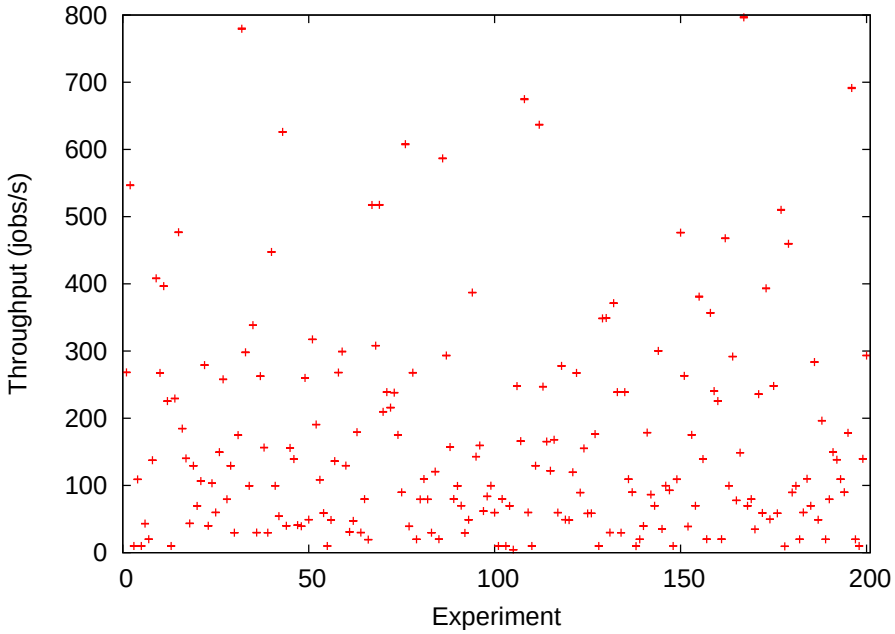
$$RelativeError = \frac{V_{analytical} - V_{simulation}}{V_{simulation}} \tag{3.5}$$

Figure 3.4(a) shows the confidence intervals for 4 of the 200 networks simulated. We show these plot to illustrate that the simulations have very small confidence interval between the results. Figure 3.4(b) shows the throughput as obtained from simulation plotted with 95% confidence intervals.

Figure 3.5 compares the analytical throughput results with the simulation results. Figure 3.5 shows that the analytical model, more often than not, predicts an optimistic value for the throughput. Most of the experiments have a relative error in throughput of less than 5%. The mean of the relative error, when measured in absolute, was 1.273% for the 200 experiments. Of the 200 experiments 163 of them have the analytical model predicting



(a) Throughput predicted by the simulation infrastructure for the 2 queueing networks, with 95% confidence intervals



(b) Throughput predicted by the simulation infrastructure for the 100 queueing networks, with 95% confidence intervals

Figure 3.4: Confidence in simulation results

optimistically with a mean relative error of 1.501%, and the pessitimistic estimates have a mean relative error of -0.272%.

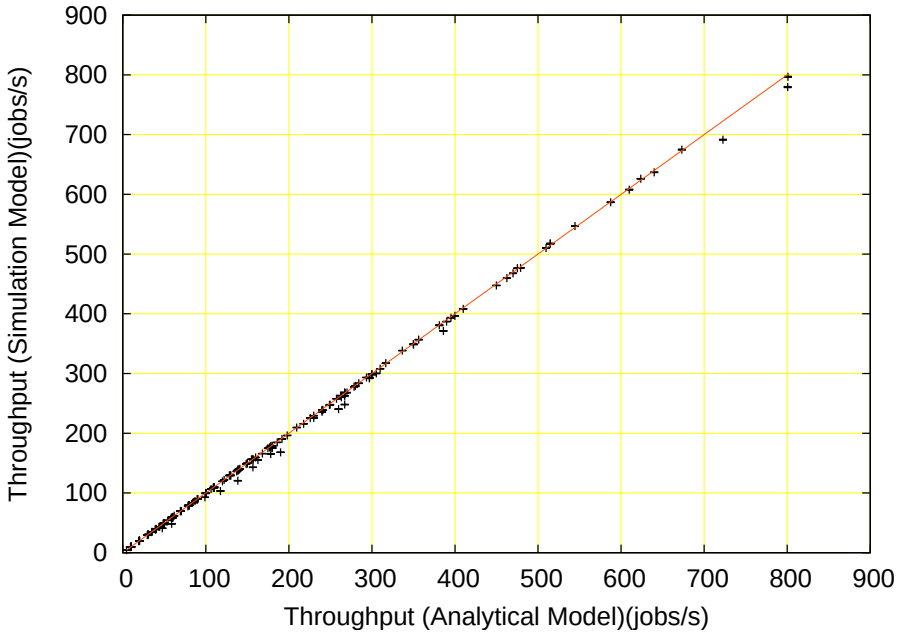
We conclude from the figures above that generally, the model works quite well. However, there are 12 experiments which have a significantly higher relative error compared to the rest of the experiments (greater than 5%). These are the cases that are interesting to us, as these show that we cannot always guarantee that the model is predictable. We will address the specific causes of these errors in the next chapter.

Another parameter of interest in hybrid designs is the frequency with which a node blocks the nodes upstream. This parameter is very important in designs where back pressure results from blocking and results in "expensive" stalls. In Figure 3.6 we compare the probability of blocking predicted by the two models. We see here that the probability of blocking is surprisingly underestimated by the analytical model in a large number of cases, however, the model still yields the right answers for the throughput. We will come back to this issue in the next chapter where we discuss the shortcomings of this model.

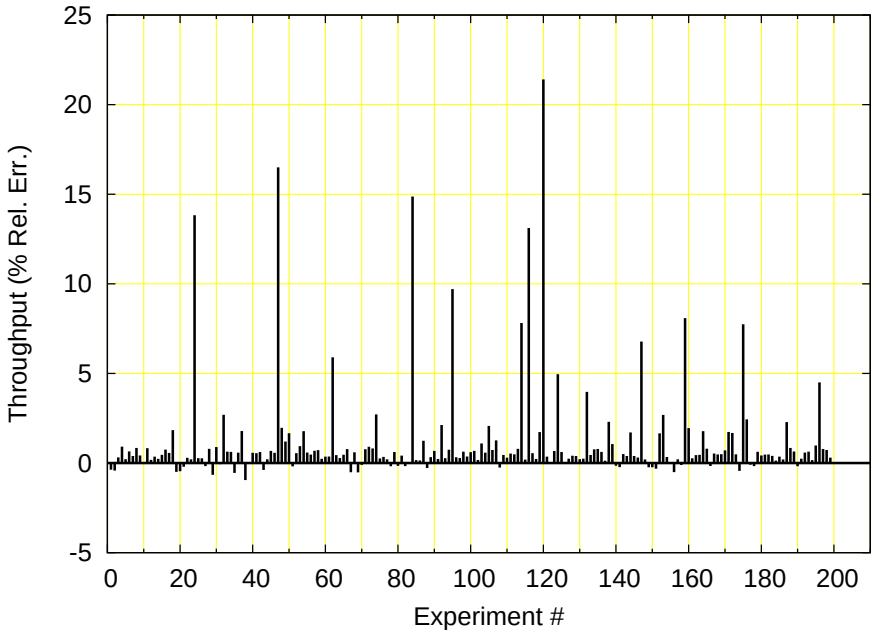
3.2.2 Phase-Type Service Time Distribution

In many scenarios we have to deal with a variety of service distributions, and hence need a model that can accommodate more than exponentially distributed service times at each stage. For this purpose we build on the previous algorithm and assume that the "true" service time distribution at the individual nodes is phase-type. This assumption helps us evaluate the performance of many general service time distributions by modeling them as phase-type. Based on the number of moments known, one can model an "equivalent" phase-type distribution [60]. Phase-type distributions being a series of exponential distributions have the same memoryless property of an exponential distribution and are discussed extensively in [60, 90].

Here we describe service models when we are given only the mean $\boldsymbol{m}$ and variance $\boldsymbol{\sigma^2}$ of the service time distribution [1]. If the distribution satisfies the criteria $\boldsymbol{c^2 = \frac{\sigma^2}{m^2} > \frac{1}{2}}$, we model this distribution as a 2 node phase-type distribution as shown in Figure 3.7, where $\boldsymbol{\mu_1 = \frac{2}{m}}$ and $\boldsymbol{\mu_2 = \frac{1}{mc^2}}$ and $\boldsymbol{a = \frac{1}{2c^2}}$. When the distribution is "tighter" i.e., $\boldsymbol{c^2 < \frac{1}{2}}$, we model this distribution as a mix of 2 Erlang distributions with $\boldsymbol{k}$ and $\boldsymbol{k-1}$ phases, where $\boldsymbol{\frac{1}{k} \leq c^2 \leq \frac{1}{k-1}}$. Each phase of the Erlang distribution has a service rate of $\boldsymbol{\mu}$ (Figure 3.8). A job is always sees $\boldsymbol{k-1}$ phases of service, and with a probability $\boldsymbol{a}$ sees an additional



(a) Difference between analytical and simulation predictions



(b) Relative error of the analytical model w.r.t simulation

Figure 3.5: Throughput predicted by analytical and simulation models

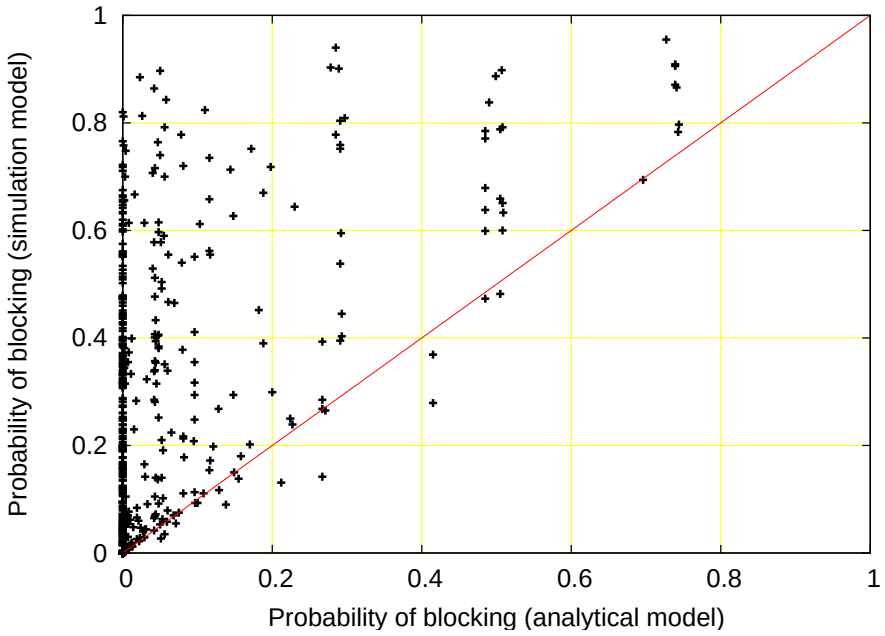


Figure 3.6: Difference between the blocking predicted by analytical and simulation models.

service phase. This probability $\boldsymbol{a}$ is obtained as $\boldsymbol{a = \frac{1}{1+c^2}[kc^2 - \{k(1+c^2) - k^2c^2\}^{\frac{1}{2}}]}$, and $\boldsymbol{\mu = \frac{k-a}{m}}$.

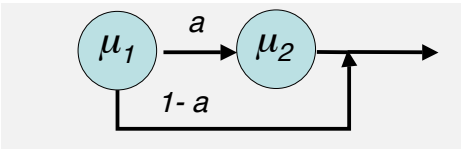


Figure 3.7: A phase type distribution with two phases, used to model servers when the first two moments of their distribution are known and $\boldsymbol{c^2 > \frac{1}{2}}$.

Procedure

Table 3.4 lists the variables used in the solution for queue lengths and throughput for networks with phase-type service time distributions. The nodes in these networks are assumed to have a squared coefficient of variance greater than $\frac{1}{2}$.

As in Section 3.2.1 before, we start at the last node in the network and work our way backwards. We analyze each of the nodes in isolation, wherein we alter the service distribution of the nodes based on estimated blocking by downstream nodes. We start by assuming that the mean throughput supported by the network is the minimum rate of service at

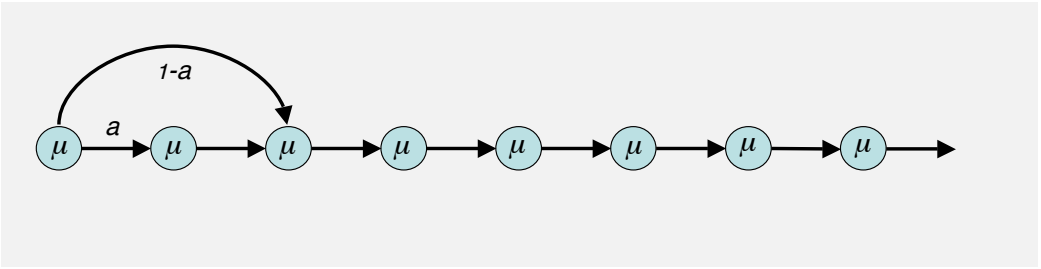


Figure 3.8: A phase type distribution with $\boldsymbol{k}$ phases, used to model servers when the first two moments of their distribution are known and $\boldsymbol{c^2 < \frac{1}{2}}$.

the different nodes in the network. Our initial estimate of the maximum ingest rate is $\boldsymbol{T_0 = min(\frac{\mu_i}{\prod_{\forall j, j<i} d_j} \forall i)}$. We traverse back from the last node, assuming that the output rate corresponding to this ingest rate is $\boldsymbol{t_N = \prod_{\forall i} d_i T_0}$.

1. Analyze the $\boldsymbol{N^{th}}$ node as a $\boldsymbol{M/PH_{r_N}/1/K_N{+}1}$ queue. Where $\boldsymbol{r_N}$ is the number of phases in the service distribution of node $\boldsymbol{N}$.

$$\boldsymbol{\lambda_N = t_N/[1 - p_N(K_N{+}1)],} \tag{3.6a}$$

$$\boldsymbol{\pi_{N-1} = \mu_N p_N(K_N{+}1)/\lambda} \tag{3.6b}$$

   Also calculate the probability of the node being at a particular service phase $\boldsymbol{m}$ when the queue is full, essentially $\boldsymbol{w_N(m)}$. From this we can calculate $\boldsymbol{a_{m,n}}$ for $\boldsymbol{m, n \in [1, r_{N-1}{+}r_N]}$, as illustrated in Figure 3.9.

2. Analyze each node $\boldsymbol{i}$ as an $\boldsymbol{M/PH_{\sum_{\forall q \geq i} r_q}/1/K_i{+}1}$ node, for $\boldsymbol{i \in [N{-}1, \ldots, 2]}$. The expected throughput at each of these nodes, $\boldsymbol{t_i}$, is obtained as $\boldsymbol{t_i = \frac{t_{i+1}}{d_i}}$. The effective service at the $\boldsymbol{i^{th}}$ node is given by the phase-type distribution $\boldsymbol{(\alpha_i, Q_i)}$, where $\boldsymbol{\alpha_i}$ is a row vector with $\boldsymbol{\sum_{\forall q \geq i} r_q}$ elements all equal to zero except the first element that is set to 1, $\boldsymbol{e_i}$ is a $\boldsymbol{\sum_{\forall q \geq i} r_q \times 1}$ unity vector, and $\boldsymbol{Q_i}$ is the rate matrix for effective distribution at node $\boldsymbol{i}$.

   $\boldsymbol{\lambda_i}$, which is essential to solve for node $\boldsymbol{i}$, is obtained by solving the fixed point expression, $\boldsymbol{\lambda_i = t_i/[1 - p_i(K_i + 1)]}$. Where,

$$\boldsymbol{p_i(K_i + 1) = p_0 \alpha_i R_i^{K_i}(-\lambda_i Q_i^{-1})}$$

$$\boldsymbol{R_i = \lambda_i(\lambda_i I - \lambda_i e_i \alpha_i - Q_i)^{-1}}$$

Table 3.4: Notation and description of symbols used in queueing networks with phase-type service time distribution.

Symbol	Description
T_j	The estimated (assumed) mean throughput of the queueing network for the j^{th} traversal of the algorithm
t_i	The expected throughput at node i of the network.
λ_i	Hypothetical mean arrival rate into node i of the network, the jobs are assumed to have an exponentially distributed interarrival time
μ_i	True mean service rate of node i of the network
$\mu_{i,m}$	Mean service rate of phase m of node i of the network, the service time distribution at each of the phases is given to be exponential
$p_i(x)$	Probability of having x jobs at node i in the network
π_i	Probability of a job leaving node i being blocked from entering node $i+1$ of the network
$w_i(m)$	Probability that a blocked job at node i is waiting at service phase m of node $i+1$
$a_{m,n}$	Probability of transition from phase m to phase n of phase-type distributions modeling effective service times

$$p_0 = \{\alpha_i[\sum_{r=0}^{K_i} R_i^r - \lambda_i R_i^{K_i} Q_i^{-1}]e_i\}^{-1}$$

.

After analyzing node i, we compute $w_i(m), m = i, i+1, \ldots, N$ using

$$w_i(m) = \frac{p_i(0)\alpha_i R_i^{K_i} e_m}{p_i(K_i)}, \tag{3.7a}$$

and obtain π_{i-1} using the relation,

$$\lambda \pi_{i-1}[-w_i Q_i^{-1} e] = p_i(K_i + 1), \tag{3.7b}$$

where $w_i = (w_i(i), w_i(i+1), \ldots, w_i(N))$.

3. The quantities w_i and π_{i-1} are used to characterize the effective service time for node $i-1$, essentially calculating $a_{m,n}$ for the downstream nodes.

4. The ingress node (node 1) is analyzed as a $M/PH_{\sum_{\forall i} r_q}/1/K_1$ node using the matrix-geometric procedure. The mean arrival rate at the first node in the network is the

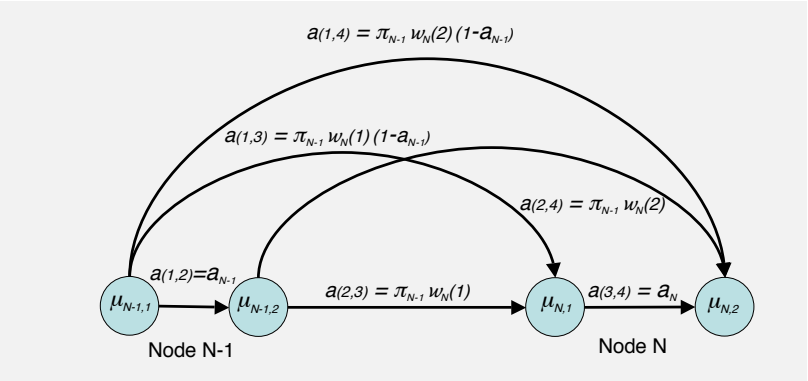


Figure 3.9: Resulting phase-type distribution at node $\boldsymbol{N-1}$.

mean of the exponentially distributed arrival process that is required to have an output rate equal to the assumed throughput, $\boldsymbol{t_1 = T_j}$.

If the mean of the arrival rate into the system is within $\boldsymbol{\epsilon}$ of the assumed throughput, then we claim that we have an sustainable arrival process, i.e, there will be few losses at ingress for this rate of arrivals. The sustainable arrival process with the minimum average interarrival time will yield the maximum throughput of the system. Also, by the same arguments as in Section 3.2.1, this algorithm is guaranteed to converge.

Performance of the algorithm

To evaluate the effectiveness of the algorithm in determining the throughput of the queueing network, we tested it on 200 synthetically generated queueing networks, where for each network the parameters were generated from the set of values in Table 3.5. The particular values of the parameters used for these 200 networks are detailed in Appendix A. For each of the networks we compared the throughput predicted by the algorithm to those obtained using our tandem queueing network simulator.

Comparison of the throughput predicted by the two models is illustrated in Figure 3.10(a). The difference is, as before, measured as a relative error defined as

$$\boldsymbol{RelativeError = \frac{V_{analytical} - V_{simulation}}{V_{simulation}}} \tag{3.8}$$

Figure 3.10(b) shows that the analytical model, more often than not, predicts an optimistic value for the throughput. Most of the experiments have a relative error in throughput of less than 10%. The mean of the relative error, when measured in absolute, was 4.056%

Table 3.5: Range of parameters used for the backward traversing algorithm, with phase-type service time distributions

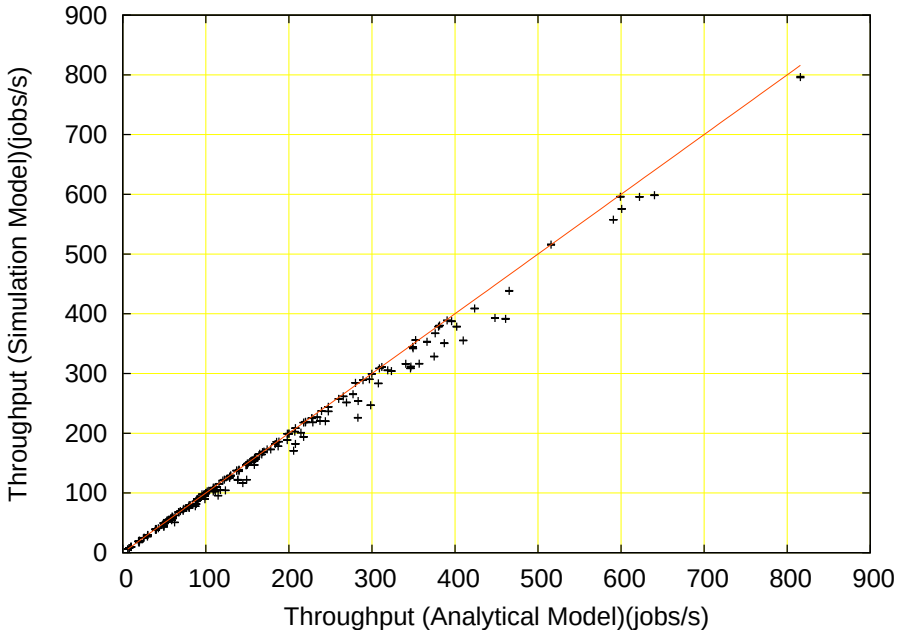
Symbol	**Description**	Range of values
N	The number of nodes in the network	[2, 3, . . . , 10]
K	Capacity of the network	[5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120]
μ_i	Mean service rate of node i in the network	[10, 20, . . . , 1000]
c^2	Squared coefficient of variance of node i's service distribution	[0.5, 0.8, 1.1, 1.3, 1.6, 2.0, 3.0, 4.0, 5.0, 7.5, 10, 15, 20]

for the 200 experiments. Of the 200 experiments 174 of them have the analytical model predicting optimistically with a mean relative error of 4.571%, and pessimistics estimates have a mean relative error of -0.614% We conclude from the figures above, that generally, the model works quite well. However, there are quite a few experiments which have higher relative error compared to the rest of the experiments (greater than 10%). These are the cases that are interesting to us, as these show that we cannot always guarantee that the model is predictable. We will address the specific causes of these errors in the next chapter.

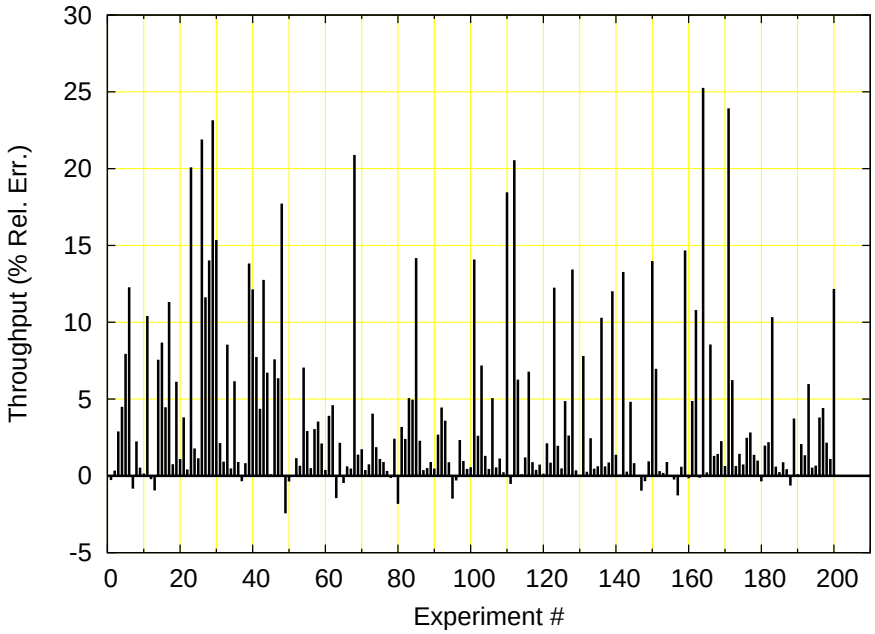
As before, we are also interested in the stalling/blocking probabilities as predicted by the analytical model, and measure the difference between the value predicted by the analytical model and simulation model. This difference is illustrated in Figure 3.11, which are similar to Figure 3.6. Again, this motivates further exploration of the model detailed in the next chapter.

3.3 Network of Queues with Intermediate Bulk Arrivals

We are interested in applications that not only have diminishing volume of data down the pipeline, but also networks where there can be potential expansion in the number of tasks moving downstream. Our experience with applications such as BLAST motivate us to explore the potential of analytical models for such queueing networks. Specifically in BLASTN, the word matching stage frequently yields more than one position in the query that matches a position in the database. This implies that for every job (w-mer) that enters the word matching stage multiple jobs might be produced downstream. Here we modify the algorithm described in Section 3.2.2 to accommodate multiple departures from nodes and explore the effectiveness of this new model. We start by describing the state-space model used to solve a queue with bulk arrivals and a phase-type service time distribution.



(a) Difference between analytical and simulation predictions



(b) Relative error of the analytical model w.r.t simulation

Figure 3.10: Throughput predicted by analytical and simulation models

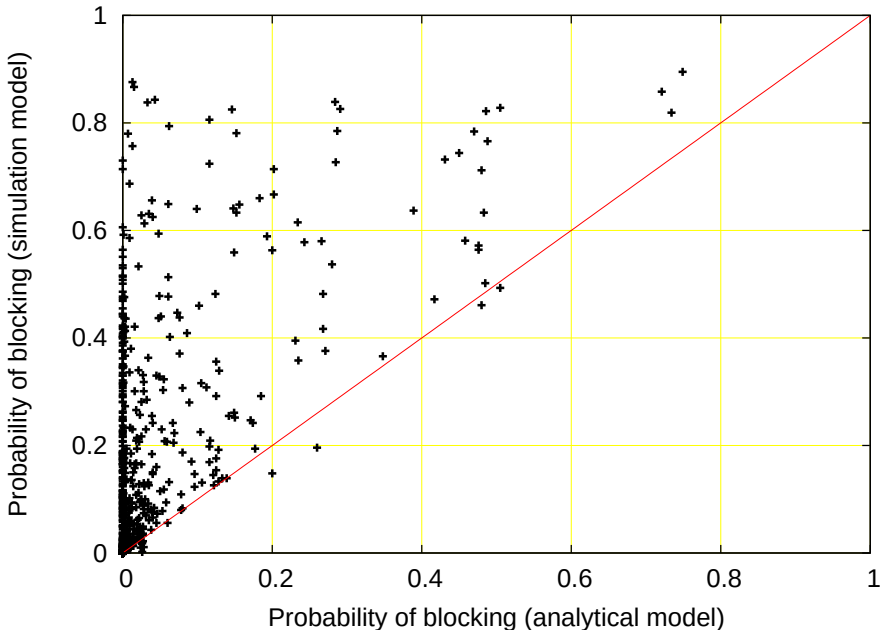


Figure 3.11: Difference between the blocking predicted by analytical and simulation models.

3.3.1 Truncated $M^x/PH/1/K$ Queue

In this section we present a state based model that we developed to solve for the throughput and the queue length distribution of a truncated $\boldsymbol{M^x/PH/1/K}$ queue. In this type of queue, the arrivals have an interarrival times which are exponentially distributed, and the service distribution is a phase-type distribution (Section 3.2.2). Each of the arrivals to this queue can bring in a bulk of customers, where the bulk size is geometrically distributed with a mean of $\boldsymbol{x}$. Further, we assume that the bulk size is bounded, i.e., there can only be a a maximum of $\boldsymbol{maxX}$ jobs in a bulk. All arrivals with bulk size greater than $\boldsymbol{maxX}$ (in a traditional geometric distribution) are considered equivalent to arrivals with a bulk size of $\boldsymbol{maxX}$. Table 3.6 explains the notations used in the solution. In this queue the admission policy allows part of the bulk to enter the queue and the rest to be discarded.

State based model

Figure 3.12 shows the model of the states for an $\boldsymbol{M^x/PH/1/K}$ queue where the service distribution has two phases. Each state is a function of the number of customers in the queue, including the one in service, and the phase of the service process. If we assume that the number of phases in the phase-type service mode is $\boldsymbol{r}$ and the capacity of the

Table 3.6: Notation and description of the terms as applicable to $M^x/PH/1/K$ queues

Symbol	Description
λ	Mean arrival rate into the queue (arrivals/time).
x	Mean number of jobs in each arrival.
$p(1-p)^b$	probability of exactly b jobs in each arrival, where $p = \frac{1}{x+1}$, $b < maxX$
$(1-p)^{maxX}$	probability of exactly $maxX$ jobs in each arrival
r	Number of phases in the phase-type service time distribution
μ_m	Mean service rate of phase m of the phase-type service time distribution, $m \leq r$.
$a_{m,n}$	Probability of job completing service at phase m of the service time distribution joining at phase n.
K	The maximum number of customers that can be present in the system (capacity)

queueing system is K then the total number of states is $rK+1$. As the time spent in each state is exponentially distributed, it satisfies the criteria for a continuous time Markov chain (CTMC) [14]. The probability of being in a particular state, say $s(k,m)$, translates to the probability of having k jobs in the system and the server, if serving any job, being in the m^{th} phase of service. For a server with two-phase phase-type distribution $m \in [0,2]$, $m = 0$ is the case that the server is idle because there are no jobs in the system. Let, $\Pi_{k,m} \quad k \in [0,K], \quad m \in \{0,1,2\}$ be the steady state probability associated with being in state $s(k,m)$. The distribution of number of jobs in the system can be obtained by obtaining $\Pi_k = \Pi_{k,1} + \Pi_{k,2} \ \ \forall k > 1$ and $\Pi_0 = \Pi_{0,0}$.

Steady state probabilities

We solve for the steady state probabilities $\Pi_{k,m}$ of the queue by solving the equation

$$\Pi Q = 0 \tag{3.9}$$

where Q is the rate matrix for the CTMC shown in Figure 3.12. Figure 3.13 shows the rate matrix Q corresponding to a $M^x/PH_2/1/K$ queue (shown to first 9 states).

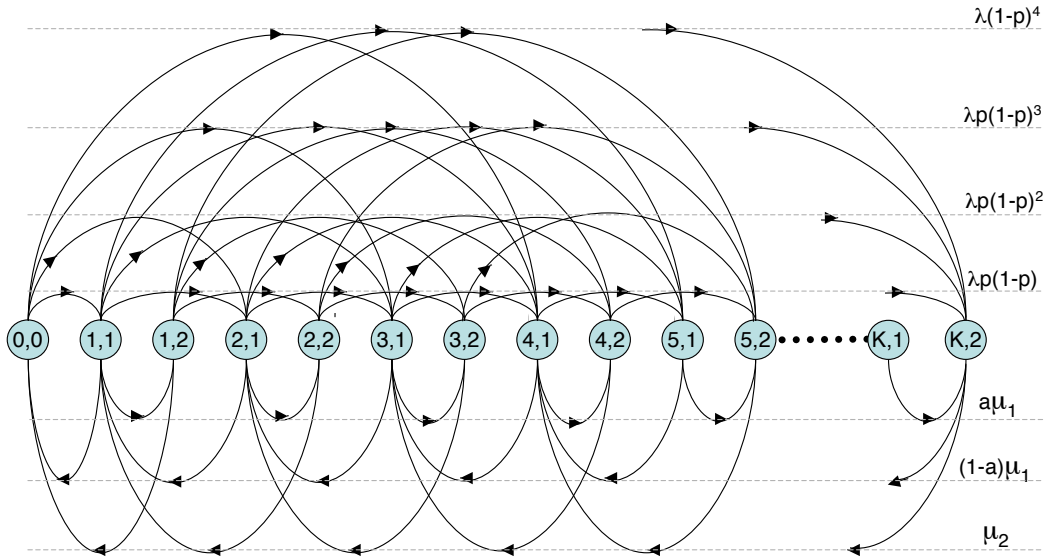


Figure 3.12: States in the state based solution for a $\boldsymbol{M^x/PH/1/H}$ queue, where $\boldsymbol{r = 2}$, $\boldsymbol{maxX = 4}$.

	0,0	**1,1**	**1,2**	**2,1**	**2,2**	**3,1**	**3,2**	**4,1**	**4,2**
0,0	$-\Sigma$	$\lambda p(1-p)$	0	$\lambda p(1-p)^2$		$\lambda p(1-p)^3$		$\lambda p(1-p)^4$	
1,1	$(1-a)\mu_1$	$-\Sigma$	$a\mu_1$	$\lambda p(1-p)$		$\lambda p(1-p)^2$		$\lambda p(1-p)^3$	
1,2	μ_2		$-\Sigma$		$\lambda p(1-p)$		$\lambda p(1-p)^2$		$\lambda p(1-p)^3$
2,1		$(1-a)\mu_1$		$-\Sigma$	$a\mu_1$	$\lambda p(1-p)$		$\lambda p(1-p)^2$	
2,2		μ_2			$-\Sigma$		$\lambda p(1-p)$		$\lambda p(1-p)^2$
3,1				$(1-a)\mu_1$		$-\Sigma$	$a\mu_1$	$\lambda p(1-p)$	
3,2				μ_2			$-\Sigma$		$\lambda p(1-p)$
4,1						$(1-a)\mu_1$		$-\Sigma$	$a\mu_1$
4,2						μ_2			$-\Sigma$

Figure 3.13: Rate matrix for a $\boldsymbol{M^x/PH/1/K}$ queue with a two phase service time distribution

The rate matrix by itself has a rank of $\boldsymbol{numberofstates - 1}$, and hence solution to Equation 3.9 cannot be determined. However, replacing one of the equations with

$$\sum_{\forall k,m} \boldsymbol{\Pi_{k,m} = 1} \tag{3.10}$$

makes $\boldsymbol{Q}$ a full-rank matrix, and we can obtain the solution for $\boldsymbol{\Pi_{i,j}}$ using standard matrix algebra.

3.3.2 Queueing Networks with Bulk Departures

In this section we show how to adapt the algorithm described in Section 3.2.2 to accommodate bulk departures in nodes. As before we are interested in the maximum ingest rate that can be supported by this queueing network, and probability of stalls in the network.

Modeling Blocking

A node which outputs a geometrically distributed number of jobs into its downstream node will be blocked until there is space for all the jobs in a bulk to move to the next node. As we are modeling a BAS blocking mechanism, we assume that all nodes have the capacity to store all the jobs that it can potentially send to its downstream node. Figure 3.14 shows the effective service distribution of node $\boldsymbol{N-1}$, when it has bulk departures and is blocked by node $\boldsymbol{N}$. In this figure $\boldsymbol{\pi_k}$ corresponds to the probability that node $\boldsymbol{N-1}$ has $\boldsymbol{k}$ jobs which cannot enter node $\boldsymbol{N}$, and $\boldsymbol{w_{k,m}}$ is the probability that node $\boldsymbol{N}$ is in the $\boldsymbol{m^{th}}$ phase of its service when it blocks $\boldsymbol{k}$ jobs from node $\boldsymbol{N-1}$. Figure 3.15 shows this effective service time realized as a phase-type distribution, where the probability of moving from the penultimate phase to the last are obtained using Bayes rule on Figure 3.14. In Figure 3.14 we obtain $\boldsymbol{w'_{21}}$ and $\boldsymbol{w'_{22}}$ as

$$\boldsymbol{w'_{2l} = \frac{\pi_2 w_{2l} + \pi_3 w_{3l} + \pi_4 w_{4l}}{\pi_2 + \pi_3 + \pi_4}, \;\; l \in 1, 2}$$

Procedure

As in the earlier variants of the algorithm, the procedure traverses upstream and analyzes each node in isolation. Table 3.7 lists the variables used in the solution for this network.

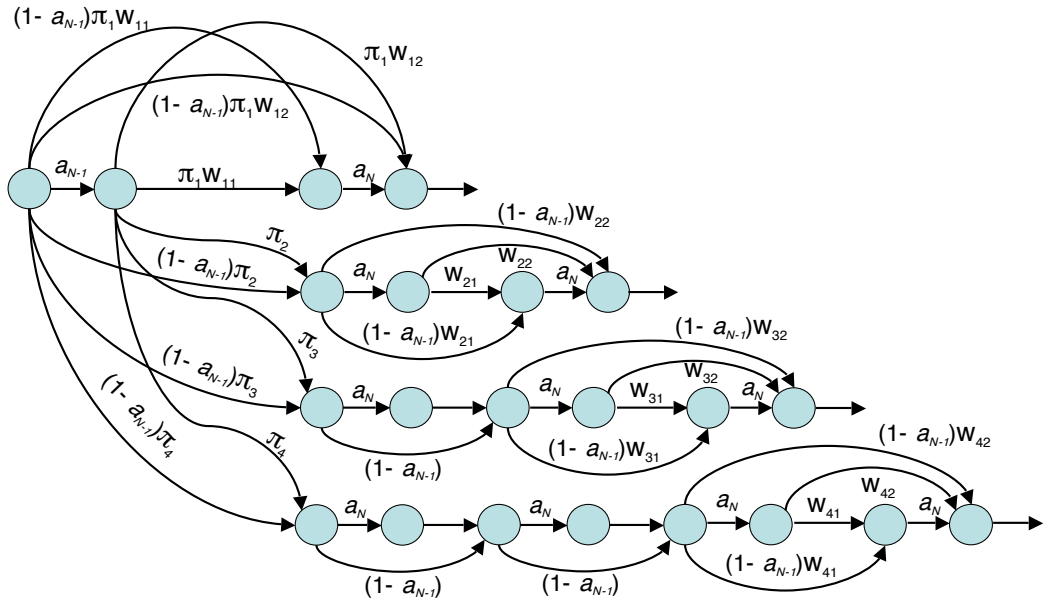


Figure 3.14: Effective service time for the penultimate node because of blocking downstream, when $\boldsymbol{maxX} = 4$.

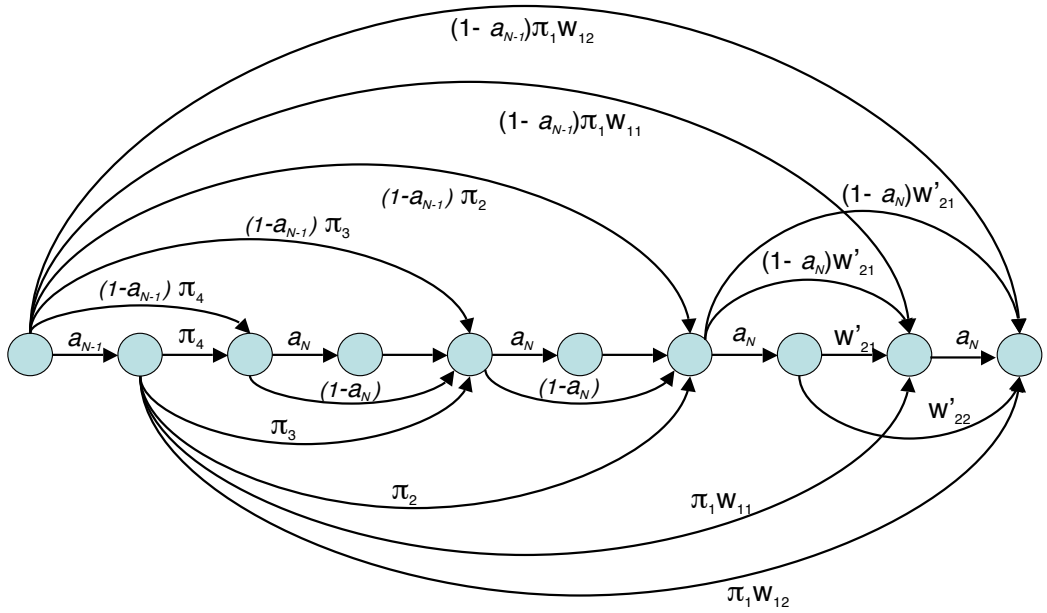


Figure 3.15: Effective service time, realized as the phase-type distribution, for the penultimate node because of blocking downstream, when $\boldsymbol{maxX} = 4$.

The nodes in these networks are assumed to have a squared coefficient of variance greater than $\frac{1}{2}$.

Table 3.7: Notation and description of symbols used in the queueing networks with bulk arrivals and general service time distributions.

Symbol	Description
T_j	The estimated (assumed) mean throughput of the queueing network for the j^{th} forward traversal of the algorithm
t_i	The expected throughput at node i of the network
λ_i	Hypothetical mean arrival rate into any particular node of the network, the jobs are assumed to have an exponentially distributed interarrival time.
μ_i	Mean service rate of node i of the network
$\mu_{i,m}$	Mean service rate of phase m of node i of the network, the service time distribution at each of the phases is given to be exponential
d_i	Mean number of outputs for every departure from node i to node $i+1$.
$p_i(k)$	Probability of having k jobs at node i in the network.
π_i	Probability of a job leaving node i being blocked from entering node $i+1$ of the network.
$w_i(m)$	Probability that a blocked job at node i is waiting at service phase m of node $i+1$
$a_{m,n}$	Probability of transition from phase m to phase n of phase-type distributions modeling effective service times

As before we start at the last node in the network and work our way backwards. We analyze each of the node in isolation, modifying the service distribution to account for downstream blocking and determining a hypothetical (lossy) arrival rate that will support the current throughput. In cases where there are bulk arrivals into a particular node, say node i, we use the following admission policy for our lossy arrivals. A job can only be accepted into the node i iff the queue for node i has at least $maxX$ positions unoccupied. For example, if we are analyzing a queue with a capacity of **10** and the arrivals with a maximum bulk, $maxX$, of **4**, we increase the capacity of that node to **14**, and jobs can enter this node only when there at most **10** jobs in the system.

We start by assuming that the mean throughput supported by the network is the minimum rate of service at the different nodes in the network. Our initial estimate of the maximum

ingest rate is $T_0 = min_i(\frac{\mu_i}{\prod_{\forall j, j<i} d_j})$. We traverse back from the last node, assuming that the output rate at node N corresponding to this ingest rate is $t_N = \prod_{\forall i} d_i T_0$. The expected throughput at each of these nodes, t_i, is obtained as $t_i = d_i t_{i+1}$.

1. Analyze each node i as either an $M/PH_{r_z}/1/K$ queue or an $M^x/PH_{r_z}/1/K_N+maxX$, based on the departure distribution of node $i-1$. r_z is the number of phases in the effective service time distribution of node i.

2. Find the hypothetical (lossy) arrival rate for which the output rate from node i is t_i. In other words, we need to find an arrival rate into the system for which the output rate given by $(1 - p_i(0))\mu_N$ equals t_i. As $p_i(0)$ is dependent on the arrival rate, we solve for this hypothetical arrival rate as a fixed point problem.

3. If the immediate upstream node $(i-1)$ has bulk departures, a job departing node $i-1$ can be blocked until anywhere from 1 to $maxX$ jobs complete service at node N. These probabilities are given by $\pi_{i-1,k}$ and obtained as $\pi_{i-1,k} = \frac{\mu_i p_i(K_i+i)}{\lambda}$ $i \in [1, maxX]$ Also calculate the probability of the node i being at a particular service phase m when it blocks node $i-1$, $w_i(m)$[1]. From this we can calculate $a_{m,n}$ for $m, n \in [1, r_{i-1}+maxXr_z]$.

4. If the immediate upstream node does not have bulk departures, the procedure remains the same as outlined in Section 3.2.2.

Performance of the algorithm

To evaluate the effectiveness of the algorithm in determining the throughput of the queueing network, we tested it on 200 synthetically generated queueing networks, where for each network the parameters were generated from the set of values in Table 3.8. The particular values of the parameters used for these 200 networks are detailed in Appendix A. All the **200** we two node networks and we assumed that the penultimate node always has a bulk departure. For each of the networks we compared the throughput predicted by the algorithm to those obtained using our tandem queueing network simulator.

[1]All probabilities, having certain number of jobs in the system, or being in a particular phase of service, are obtained from the state-space model.

Table 3.8: Range of parameters used for the Backward traversing algorithm, with phase-type service time distributions and bulk departures

Symbol	**Description**	Range of values
$\boldsymbol{N}$	The number of nodes in the network	2
$\boldsymbol{K}$	Capacity of the network	[5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120]
$\boldsymbol{\mu_i}$	Mean service rate of node $\boldsymbol{i}$ in the network	[10, 20, ..., 1000]
$\boldsymbol{c^2}$	Squared coefficient of variance of node $\boldsymbol{i}$'s service distribution	[0.5, 0.8, 1.1, 1.3, 1.6, 2.0, 3.0, 4.0, 5.0, 7.5, 10, 15, 20]
x	Mean burst size	1, 2, 3, 4, 5

Comparison of the throughput predicted by the two models is illustrated in Figure 3.16(a). The difference is, as before, measured as a relative error defined as

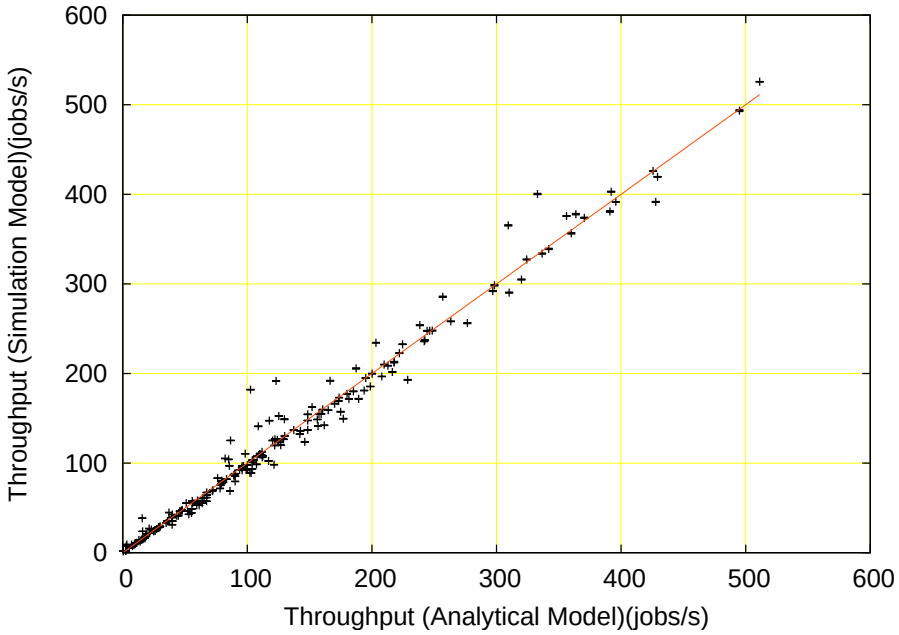
$$\boldsymbol{RelativeError} = \frac{\boldsymbol{V_{analytical}} - \boldsymbol{V_{simulation}}}{\boldsymbol{V_{simulation}}} \tag{3.11}$$

Figure 3.16(b) shows that the analytical model, more often than not, predicts an optimistic value for the throughput. Most of the experiments have a relative error in throughput of less than 10%. The mean of the relative error, when measured in absolute, was 8.202% for the 200 experiments. Of the 200 experiments 137 of them have the analytical model predicting optimistically with a mean relative error of 5.705%, and pessimistics estimates have a mean relative error of -13.632% We conclude from the figures above, that generally, the model works quite well. However, there are quite a few experiments which have higher relative error compared to the rest of the experiments (greater than 10%).

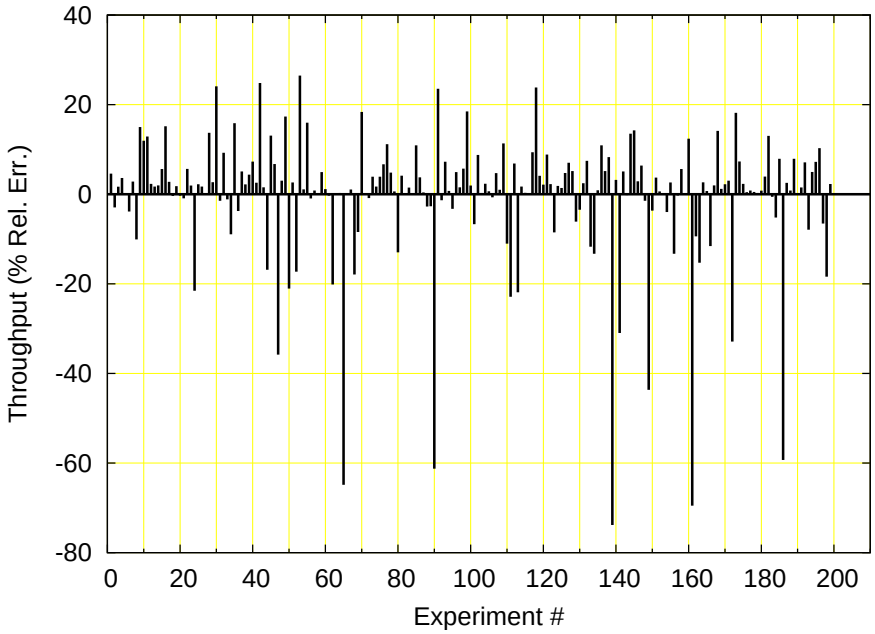
As before, we are also interested in the stalling/blocking probabilities as predicted by the analytical model, and measure the difference between the value predicted by the analytical model and simulation model (illustrated in Figure 3.17). We see here that the stalling probabilities predicted by the analytical model agrees with the simulation model in general, but there still are outliers that warrant our attention. We discuss the effects which lead to these in the next chapter.

3.4 Summary

In this chapter we described an analytical approach to solving for parameters in queueing networks with limited buffering. We evaluated methods to determine the throughput and



(a) Difference between analytical and simulation predictions



(b) Relative error of the analytical model w.r.t simulation

Figure 3.16: Throughput predicted by analytical and simulation models

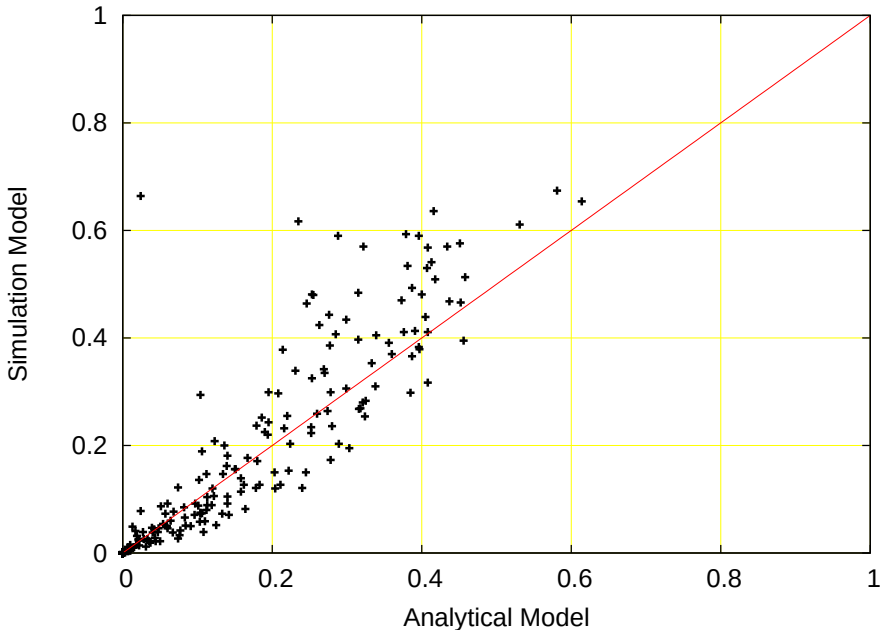


Figure 3.17: Difference between the blocking predicted by analytical and simulation models.

the stalling probabilities in such networks. We evaluated networks that had both a phase-type service distribution and bursty departures. Further, we saw that these models do not guarantee us a bound in their error. As we have said before, a model that does not give a bound on its error, or an idea about its correctness cannot be inherently trusted. In the next chapter we discuss our efforts to assess the validity of these models based on the characteristics of the nodes.

Chapter 4

Analysis of Analytical Models

4.1 Introduction

In the previous chapter we described algorithms which attempt to solve for parameters of interest in networks of queues with blocking. The particular parameters we are interested in are the overall throughput of the system and the probability of "stalls" in the networks. In the results presented in the previous chapter we saw that these algorithms do not yield acceptable answers in all cases, yet in many cases did yield answers that matched simulation. This points us to uncertainty in the model we employed to obtain these results, which in many cases will limit the usability if these models. The algorithms in the previous case were tested with random networks and the estimates from the analytical model were usually within **10%** of those from simulation. However, we also saw cases where the results differed significantly from the simulation model $> \mathbf{25\%}$.

We are motivated to look at these models in detail and present the user with bounds within which the model will yield acceptable results. As we have stressed earlier, an effective model should not only yield good estimates, but should also be able to give the designers hints about its applicability. In other words, in case of models where the error in the estimate is not tightly bound, the model should give the designer an idea about the how applicable the model is to what it being modeled.

4.2 Assessing the Analytic Models

Examination of the randomly generated test cases leads us to pay particular attention to the following aspects of the analytic model.

1. Queueing networks for which there is clearly one bottleneck node do not require this level of analysis. The throughput of the system is dominated completely by the throughput of the slowest node, and all upstream nodes effectively serve as extensions of the queue associated with the bottleneck node.

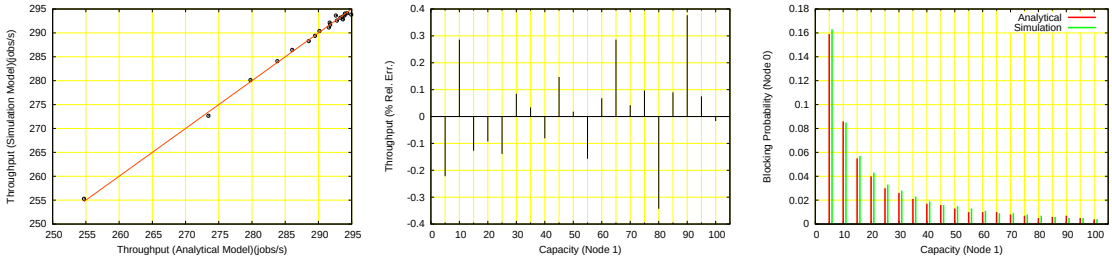
2. A queueing system becomes non-work-conserving when the queue associated with a node alternately is empty (starving the node) and full (blocking the upstream node). This circumstance is more likely as: a) the size of the queue between nodes gets smaller, b) the service rates for two adjacent nodes are similar to one another, and c) the variability in the service distribution of a node increases.

3. The quality of the analytic model throughput results are closely tied to the blocking probability experienced by upstream nodes.

To explore the above observations, a set of test cases were developed to examine the associated parameter space explicitly. Figures 4.1 and 4.2 represent results from 2 node experiments for which each node has an exponentially distributed service time (i.e., the squared coefficient of variation, $\boldsymbol{c_i^2 = 1, 0 \leq i \leq 1}$). In the first experiment (Figure 4.1), the service rates for both nodes are equal ($\boldsymbol{\mu_i = 300}$ jobs/s, $\boldsymbol{0 \leq i \leq 1}$), the capacity of the upstream node is $\boldsymbol{K_0 = 100}$ (chosen to be large enough as to not impact the throughput), and the capacity of the downstream node, $\boldsymbol{K_1}$, ranges from 5 to 100. This first experiment is designed to explore the impact of queue size on the analytic model.

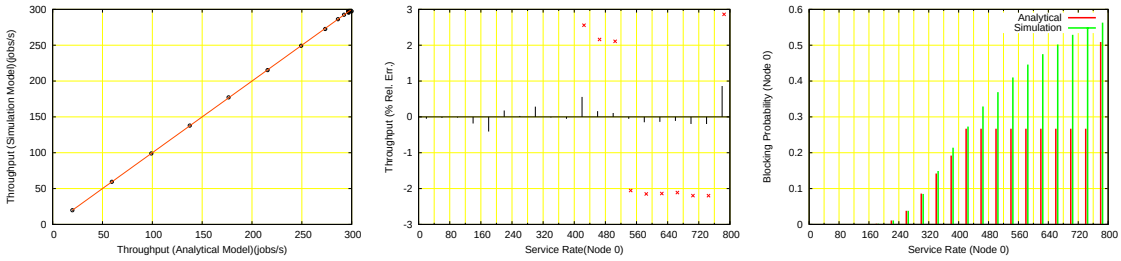
For each of the experiments performed, we plot the throughput as predicted by the simulation model vs. the throughput as predicted by the analytic model. The straight line reference that is added to the plot represents perfect alignment between the two models. The second plot for each experiment shows the relative error in the analytic model as a function of the independent variable being varied for the experiment (e.g., downstream node capacity for Figure 4.1(b)). The third plot for each experiment compares the probability that the upstream node is blocked for each of the analytic and simulation models. Although not shown explicitly in the the first plot, the low-throughput points correspond to the cases of low downstream node capacity (and corresponding high upstream blocking probability).

The graphs of Figure 4.1 correspond to a parameterization explicitly covered by the original models in [96]. Clearly, there is close correspondence between the analytic model predictions and the simulation model predictions, both for overall throughput and blocking probability for the upstream node. For small downstream queue sizes, the throughput drops off and the blocking probability increases, exactly as one would expect.



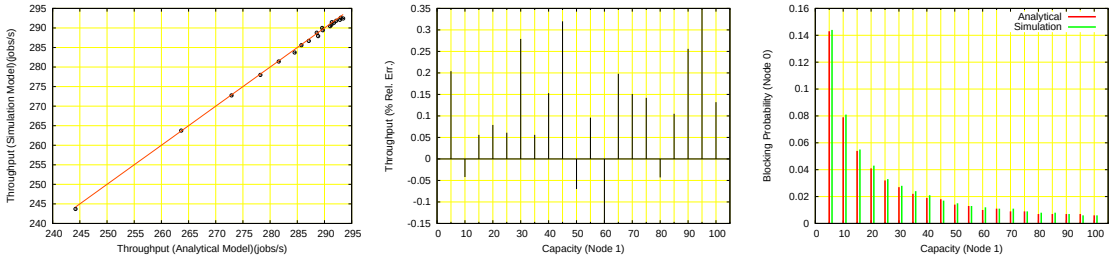
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.1: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$).



(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.2: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 \in [20, 780]}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 = 10}$).



(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.3: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 2}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$).

Further examination of the individual cases (out of the randomly generated tests) with poor correlation between the analytic and simulation throughputs indicates a pair of circumstances under which the analytic model inadequately corresponds to the physical system being modeled. The first circumstance is the case in which a downstream node is sufficiently slower than the immediate upstream node that, in effect, the upstream node (and its associated queue) act as an extension to the queue associated with the downstream node. The second circumstance is the case in which the departure process for the upstream node differs sufficiently from a Poisson model that the arrival process model for the downstream node is no longer effective. We will examine each of these circumstances individually below.

4.2.1 Test 1

Figure 4.2 shows the results of an experiment designed to explore the first circumstance described above. Here, the capacity of the downstream node is fixed at $\boldsymbol{K_1 = 10}$, the mean service rate of the downstream node remains at $\boldsymbol{\mu_1 = 300}$ jobs/s, and the mean service rate of the upstream node, $\boldsymbol{\mu_0}$, is varied between 20 and 780 jobs/s. For all our 2 node experiments we assume a sufficiently large capacity upstream, $\boldsymbol{K_0}$, that does not impact the performance of the network. Here, the system throughput closely tracks the upstream throughput while $\boldsymbol{\mu_0 < \mu_1}$ (the middle and left of the first plot, Figure 4.2(a)), stabilizing near $\boldsymbol{\mu_1}$ as $\boldsymbol{\mu_0}$ exceeds $\boldsymbol{\mu_1}$.

While the throughput results still represent a good match between the analytic and simulation models, there is a clear discrepancy between the two for the upstream blocking probability. We see in these figures that the downstream node does not recognize the fact that the upstream node is faster. This is because the algorithm does not sufficiently account for the upstream node when determining the probability of the downstream node being full. Note that in Figure 4.2(b) both the analytical model and the simulation model are predicting throughputs close the max possible throughput of 300 jobs/s and hence the absolute difference in their numerical values is not large.

We again go back to the question of whether we can add something to the modeling process that can detect this happening. Our first test, test 1, does exactly that. We use the fact that a queueing network consisting of two or more nodes can never have a throughput that exceeds the throughput of a single node system with the following parameters:

- The mean service rate is the same as the service rate of the slowest server in the network (say, S).

- The capacity of the node includes all the capacity of the network upstream from server S.

Our test 1 checks for this condition for every two node pair that is analyzed in the network and triggers a modeling failure (indicated by X on the plots) when the test fails. For example, lets consider experiment where we have exponentially distributed service times for servers $\boldsymbol{S_0}$ and $\boldsymbol{S_1}$, with their individual capacities $\boldsymbol{K_0}$ and $\boldsymbol{K_1}$, and service rates $\boldsymbol{\mu_0}$ and $\boldsymbol{\mu_1}$. Lets say that the throughput predicted by the analytical model is $\boldsymbol{T_a}$.

We consider a single server consisting of server $\boldsymbol{S_1}$, service rate $\boldsymbol{\mu_1}$ and capacity $\boldsymbol{K_{single} = K_0 + K_1}$. As the service distribution of this system is exponential and the arrival into this system is exponential, this system transforms to an M/M/1/K system described in Section 2.2.2. This system has a closed form for its maximum throughput given by $\boldsymbol{T_{single} = (1 - \frac{(1-\rho)}{1-\rho^{K_{single}+1}}) \times \mu_1}$. It is clear that this is the maximum throughput that can sustained by the two node system when the upstream node is assumed to be infinitely fast. Given this fact, if the analytical solution for the queueing network violates the equation $\boldsymbol{T_a < T_{single}}$ for any two node in the system we do not trust the result of the analytical model. For the case when the downstream node is a phase-type distribution instead of the exponential distribution, we can use the $\boldsymbol{M/PH/1/K}$ models described in Section 2.2.2.

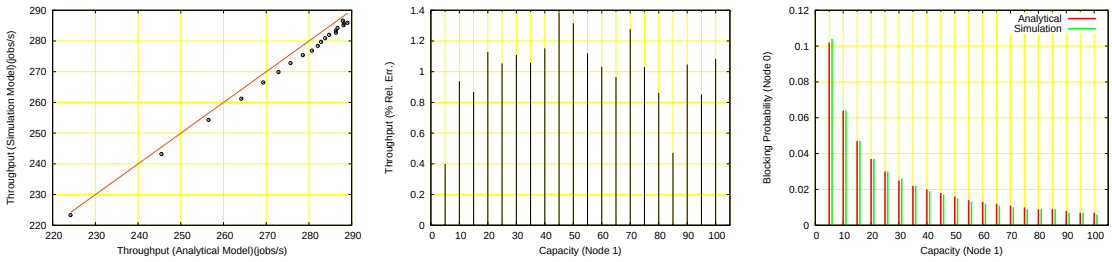
While [96] dealt with exponentially distributed service times, we have extended the model to address more general service distributions. Figures 4.3 through 4.8 show the results of experiments in which there is increased variability in the service time of the downstream node. In the experiments of Figures 4.3, 4.4, and 4.5, the service rates for both nodes are again equal ($\boldsymbol{\mu_i = 300}$ jobs/s, $\boldsymbol{0 \leq i \leq 1}$), the capacity of the upstream node is $\boldsymbol{K_0 = 100}$, and the capacity of the downstream node, $\boldsymbol{K_1}$, ranges from 5 to 100. What differs from the first experiment is the squared coefficient of variation, $\boldsymbol{c_1^2}$, for the downstream node, which is set to 2 in Figure 4.3, 5 in Figure 4.4, and 10 in Figure 4.5. The results show a close match between analytical and simulation models for the entire range of queue sizes explored, both for throughput and for upstream blocking probability.

The sensitivity of the analytic models to dissimilar service rates are again illustrated in Figures 4.6 through 4.8. As in Figure 4.2, the service rate for the upstream node is varied over the range $\boldsymbol{\mu_0 \in [20, 780]}$ jobs/s and the capacity for both nodes is fixed at $\boldsymbol{K_0 = 100}$ and $\boldsymbol{K_1 = 10}$. We see a trend similar to the one seen earlier in Figure 4.2(b). The analytical model does a good job of tracking the simulation results, but starts failing when the upstream node server rate exceeds the downstream node's service rate. In the experiments illustrated in Figures 4.6 through 4.8 we see that the model starts failing after

the upstream service rate starts exceeding 400 jobs/s. However, the thing to note here is that our test does detect this happening and points that out to us.

The next set of experiments investigates the case where the upstream node's service distribution varies from exponential. Figures 4.9 through 4.13 show the results of experiments where the service rates for the two nodes are returned to be equal ($\boldsymbol{\mu_0 = \mu_1 = 300}$ jobs/s), the service distribution of the downstream node is returned to exponential ($\boldsymbol{c_1^2 = 1}$), the capacity of the downstream node is varied ($\boldsymbol{K_1 \in [5, 100]}$), and the squared coefficient of variation of the upstream node is different for each individual experiment. ($\boldsymbol{c_0^2 \in [0.8, 1.3]}$).

We see here that these plots are very similar to Figure 4.1 and point to the fact that performance of the algorithm remains comparable to the case where the service time distribution of the upstream node was exponentially distributed. Also, these cases also suffered from their inherent ability to detect the mismatch between service rates of upstream and downstream nodes. I.e., when the service rate of the upstream node is increased, the model fails and the test indicates failure.



(a) Throughput

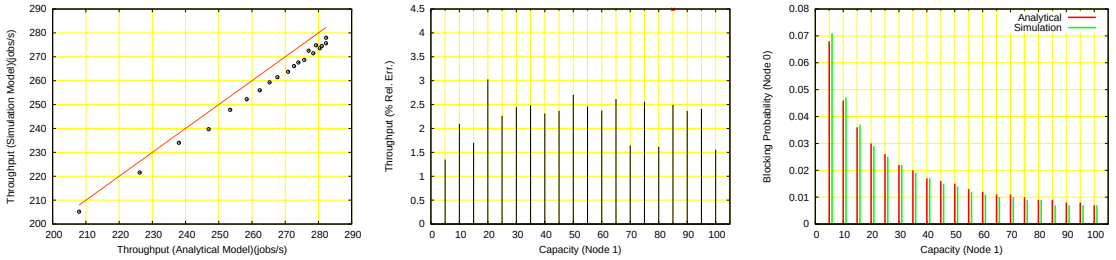
(b) Throughput relative error

(c) Upstream blocking probability

Figure 4.4: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 5}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$).

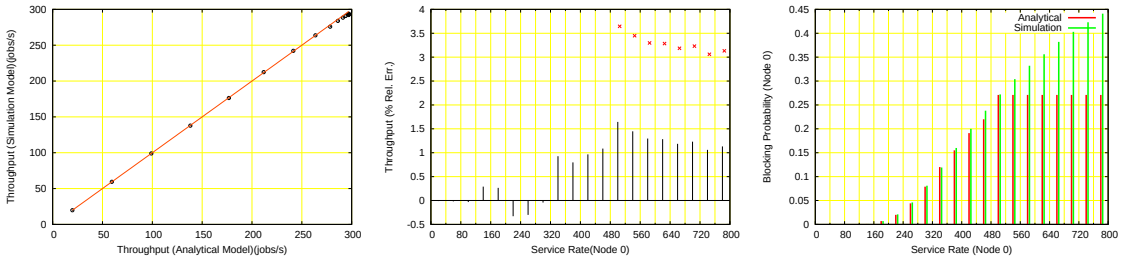
4.2.2 Test 2

Our experiments so far have considered upstream nodes which have a squared coefficient of variance around 1.0. As we have claimed in the earlier chapter that this solution extends to general service times, we wish to extend the experiments to include a larger squared coefficient of variance. In effect, we are studying the behavior of this algorithm for distributions whose tail is heavier than an exponential distribution.



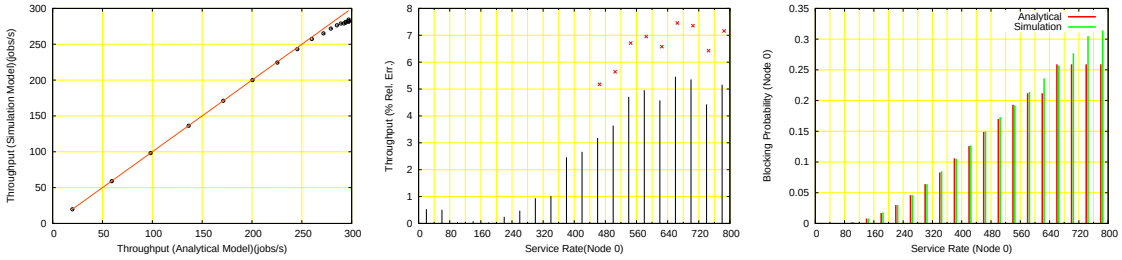
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.5: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 10}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$).



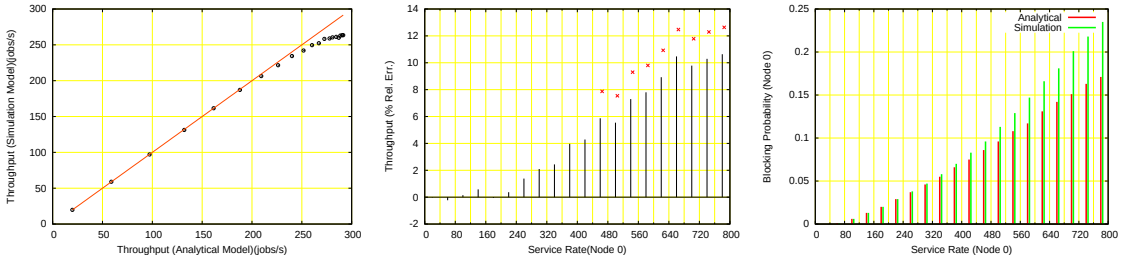
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.6: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 2}$, $\boldsymbol{\mu_0 \in [20, 780]}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 = 10}$).



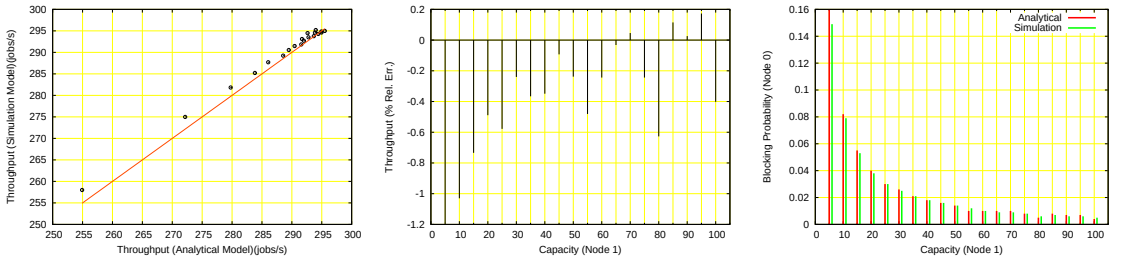
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.7: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 5}$, $\boldsymbol{\mu_0 \in [20, 780]}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 = 10}$).



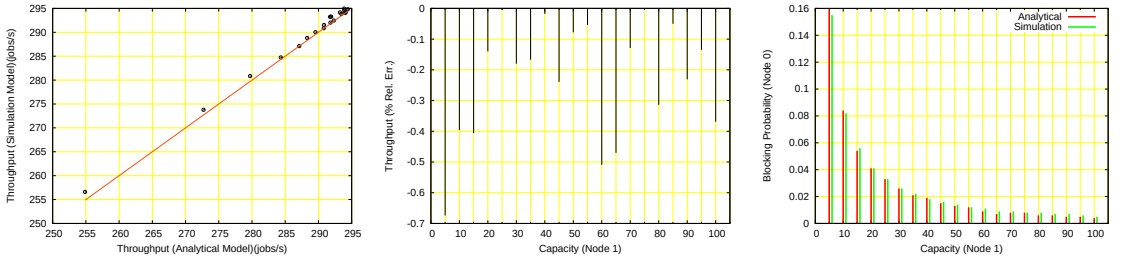
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.8: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 10}$, $\boldsymbol{\mu_0 \in [20, 780]}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 = 10}$).



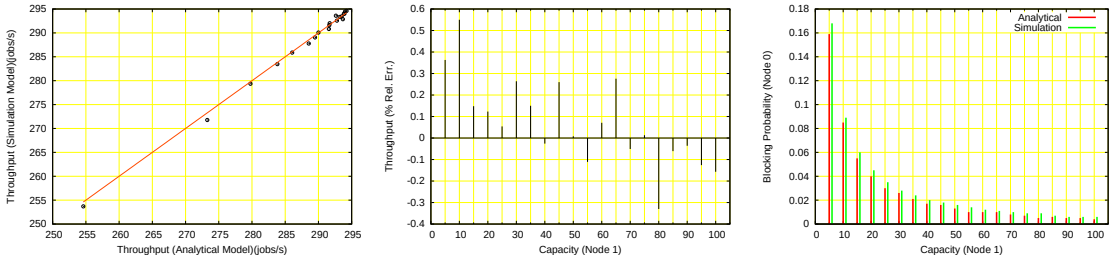
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.9: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 0.8}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$).



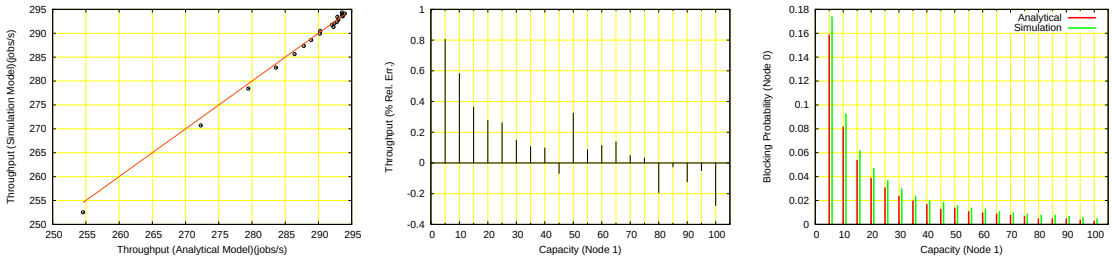
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.10: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 0.9}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$).



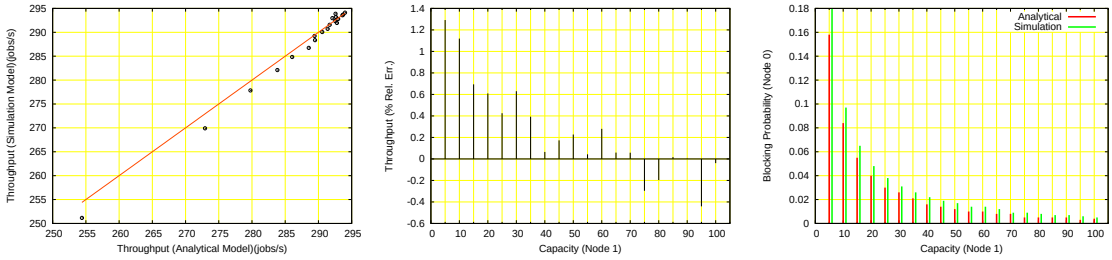
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.11: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1.1}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$).



(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

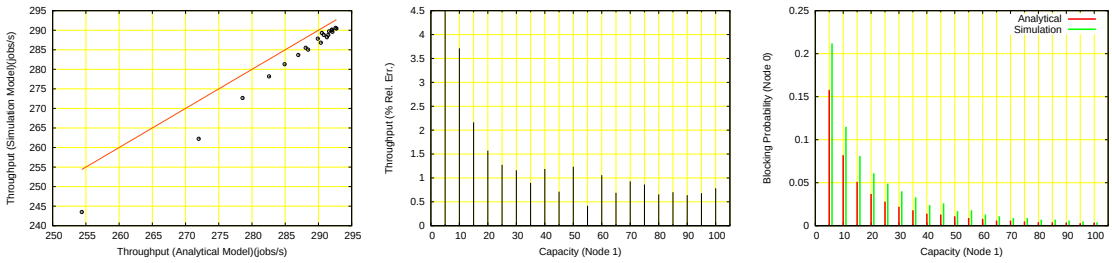
Figure 4.12: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1.2}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$).



(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

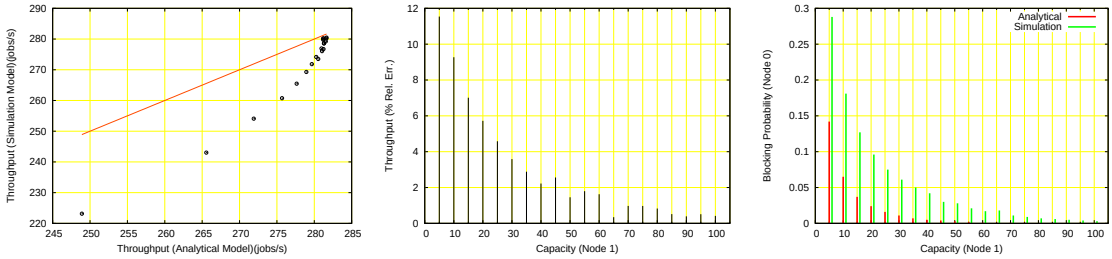
Figure 4.13: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1.3}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$).

In Figures 4.14 through 4.16 we show the effect of increasing squared coefficient of variation on the results predicted by the analytical model. Note here that the downstream node has an exponentially distributed service time and has the same mean service rate as the upstream nodes. We have earlier seen that the case where the upstream and downstream nodes were balanced the algorithm has commendable performance, however in these experiments it is not the case.



(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

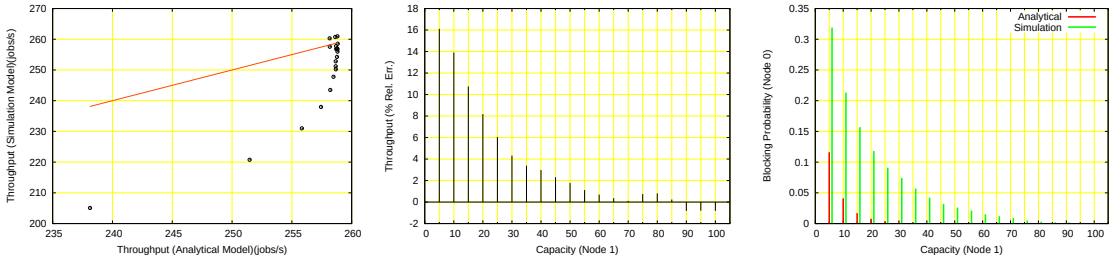
Figure 4.14: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 2}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$).



(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.15: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 5}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$).

We make two observations from these experiments. One that our first test (test 1) fails to detect the deviation from simulation, and second that the analytical model still underestimates the blocking probability. We observe that this is true when the arrival distribution from the upstream node is significantly more bursty relative to an exponential arrival process. Our second test to detect a failing model checks the inputs to the analytical models. We check to see if any of the servers in the network for which analytical model is used to



(a) Throughput

(b) Throughput relative error

(c) Upstream blocking probability

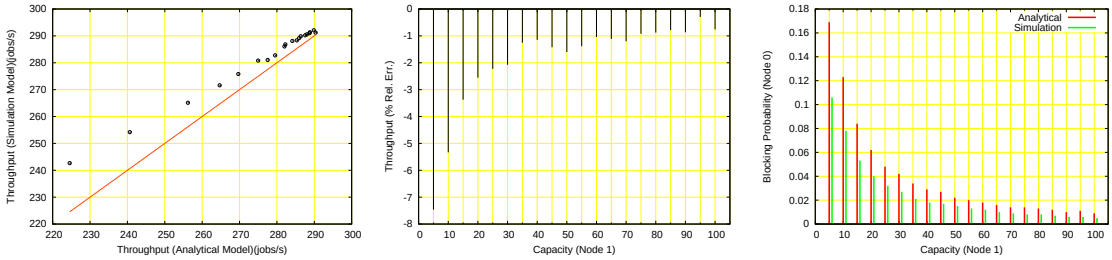
Figure 4.16: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 10}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$).

predict performance have a squared coefficient of variance significantly greater than 1. If it does we err on the side of caution and discount those results. Note here that there could be cases where one or more nodes in the network meeting this criteria could have no impact on the performance of the network and the analytical model could be predicting the right results.

While we only have empirical guidance for guiding the actual test threshold, for the cases studied, $\boldsymbol{c^2 \leq 1.3}$ all have performed well (i.e., close match with simulation) and $\boldsymbol{c^2 \geq 2}$ all have performed poorly.

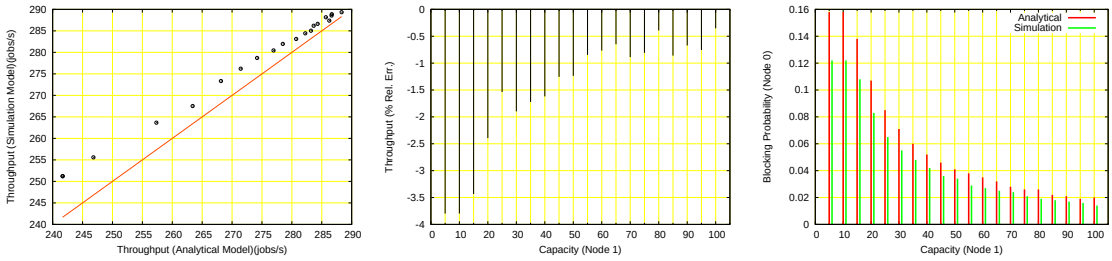
4.2.3 Test for Bursty Departure

We investigate the effect of bursty departures from upstream nodes in a two node scenario. Bursty departures from the upstream nodes implies that that a job processed by the upstream node can produce a burst of jobs for processing by the downstream node. We have similar tests to those described in Section 1.2.1, where we investigated the case of exponentially distributed service times for both upstream and downstream nodes. In Figures 4.17 to 4.20, we show the effect of burstiness for the case when the effective service rate of the two nodes are equal. We see from these figures that the algorithm modified to account for burstiness has similar results as Figure 1.1. An important change here is that the analytical model has pessimistic estimates compared to simulation model. This is a result of the analytical model trailing the simulation model in the blocking probability as illustrated in Figures 4.17(c) to 4.20(c). This we attribute to the fact that in the simulation model, part



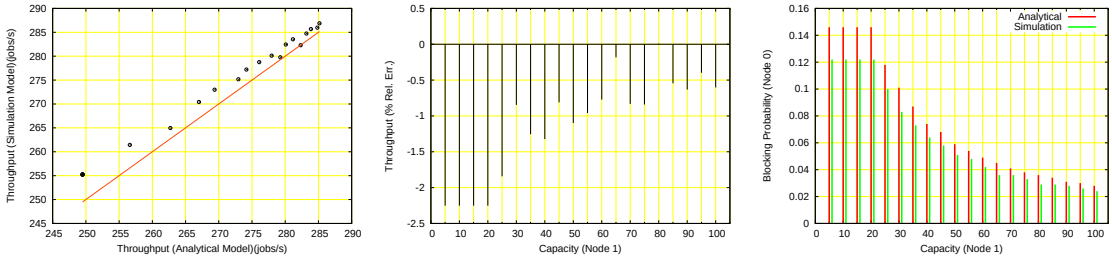
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.17: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$, x = 1, maxX = 7).



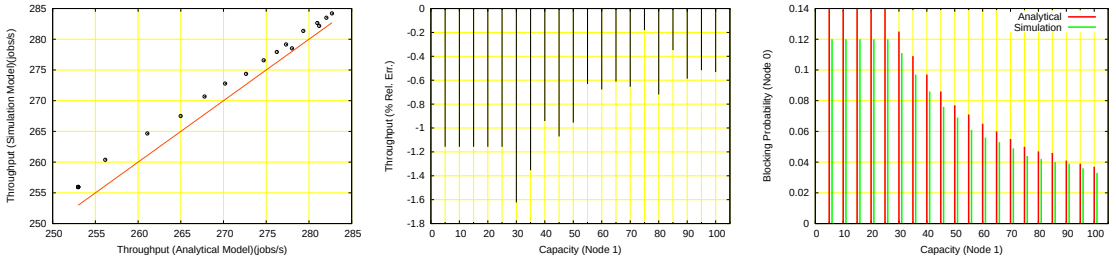
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.18: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 600}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$, x = 2, maxX = 13).



(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.19: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 900}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$, x = 3, maxX = 20).

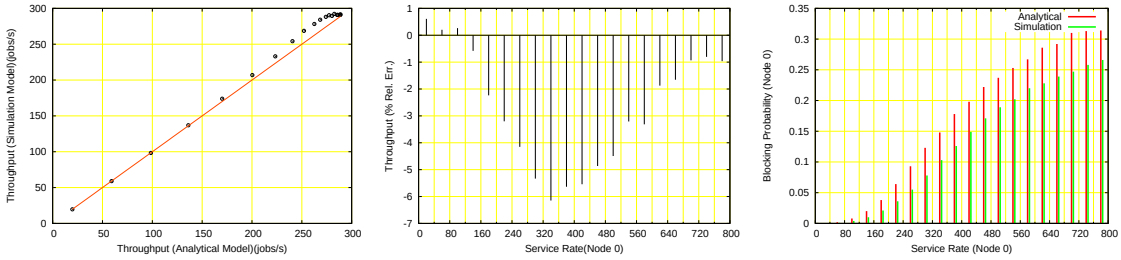


(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.20: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 1200}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [5, 100]}$, x = 4, maxX = 27).

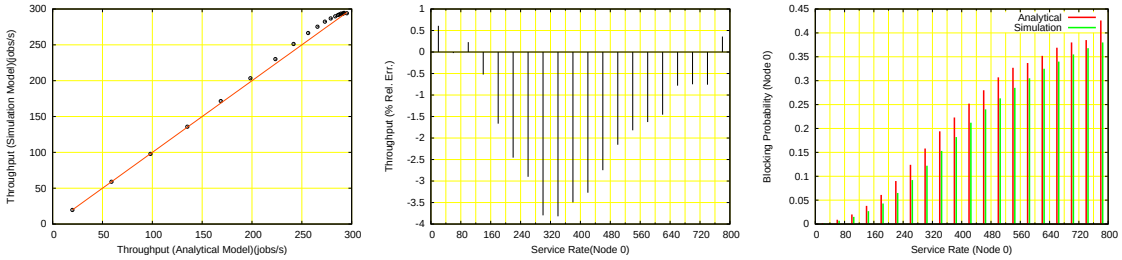
of a burst can proceed to the next node until the downstream node becomes full, however in the analytical model the space has to be available for the entire burst. If all of the jobs in a burst cannot be accommodated downstream we block all those jobs. Note here that the analytical model is not overly pessimistic, and that some pessimism in the analytical model can be a desirable feature to determine the upper limit of performance in certain scenarios.

The next set of plots we present follow the trend earlier in the chapter, where we study the effect of the difference in service rates of upstream and downstream nodes on the results of the analytical models. These are illustrated in Figures 4.21 through 4.24.



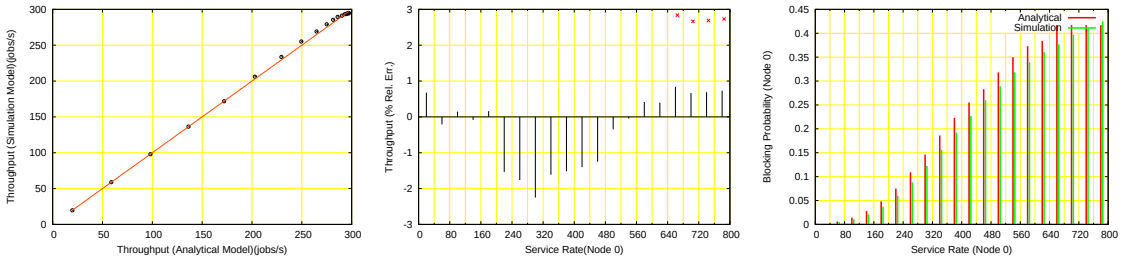
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.21: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 \in [20, 780]}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 = 10}$, x = 1, maxX = 7).



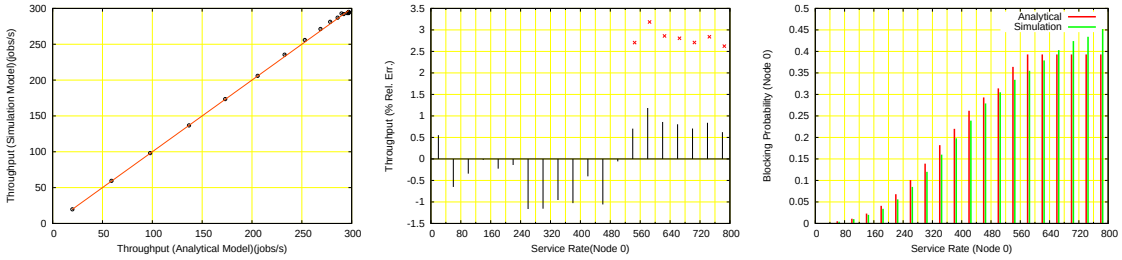
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.22: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 \in [20, 780]}$ jobs/s, $\boldsymbol{\mu_1 = 600}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 = 13}$, x = 2, maxX = 13).



(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.23: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 \in [20, 780]}$ jobs/s, $\boldsymbol{\mu_1 = 900}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 = 20}$, x = 3, maxX = 20).



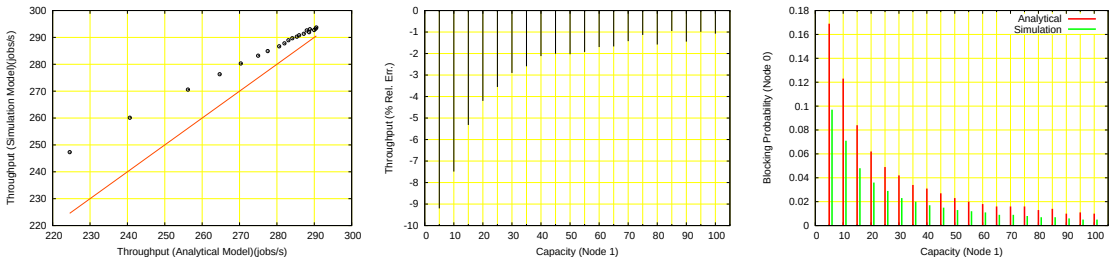
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.24: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 1}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 \in [20, 780]}$ jobs/s, $\boldsymbol{\mu_1 = 1200}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 = 27}$, x = 4, maxX = 27).

These figures illustrate a trend we have seen before, where the model starts failing when there is a significant imbalance between the upstream and downstream rates. Note, here that the effect is more prominent for case where we have a significantly higher imbalance and a longer tail on the bursty departures of the upstream node. Also, note that test 1, which checks if the analytical model predicts higher than the theoretical maximum, still detects errors with the model.

Effect of non-exponential service times of upstream node

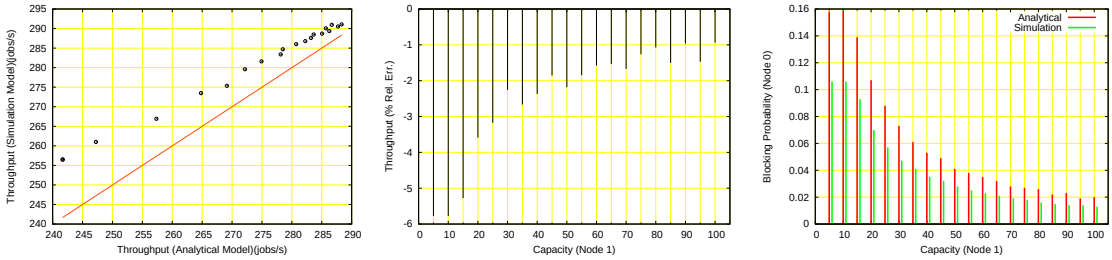
Again following the earlier trend, we check the effect of non-exponential service distribution of the upstream node on the results from the analytical model. Figures 4.25 through 4.28 show how the model reacts to tighter distributions upstream, $\boldsymbol{c_0^2 = 0.5}$ for varying burst sizes. Figures 4.29 through 4.32 show the effect of $\boldsymbol{c_0^2 = 0.8}$ (near exponential). These results are similar to the ones we have seen before in section 4.1.2, where we saw that a tighter than exponential distribution upstream does not negatively affect the performance of the algorithm.



(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

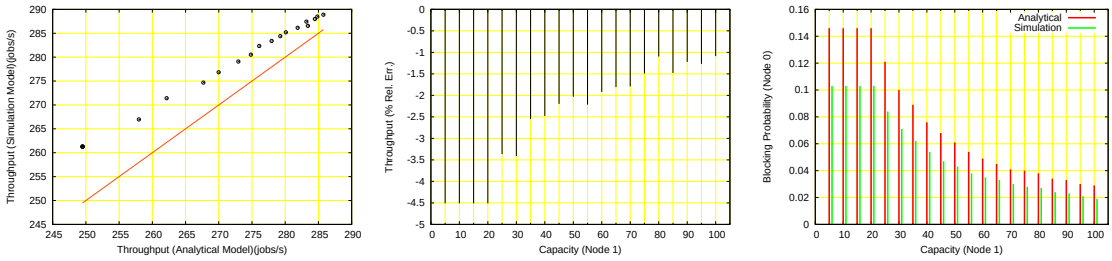
Figure 4.25: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 0.5}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [10, 100]}$, x = 1, maxX = 7).

Figures 4.33 through 4.36 shows that a heavier tail than exponential changes the trend seen earlier where we saw pessimistic results from the analytical model. Here we see that the results are optimistic, and the deviations become wider (from simulations) as the burstiness from the upstream node increases. In these figures we had $\boldsymbol{c_0^2 = 2}$, which was our arbitrary threshold from the earlier experiments. We see here that the trend seen with earlier experiments continues and the model loses trust, as defined by a **10%** deviation from simulation, for coefficient of variance of greater than 2 for the upstream node. Thus, we



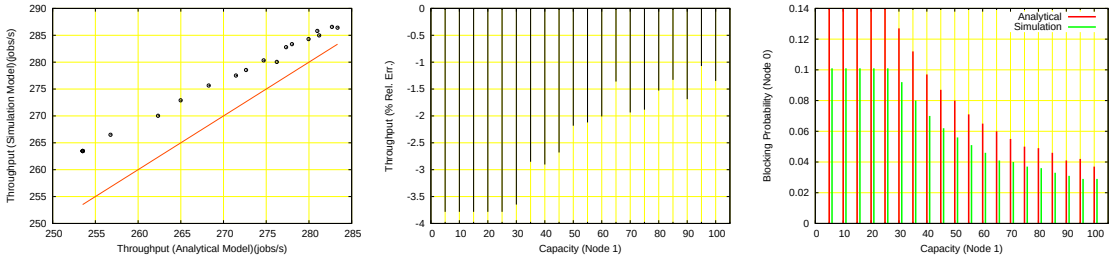
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.26: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 0.5}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [10, 100]}$, x = 2, maxX = 13).



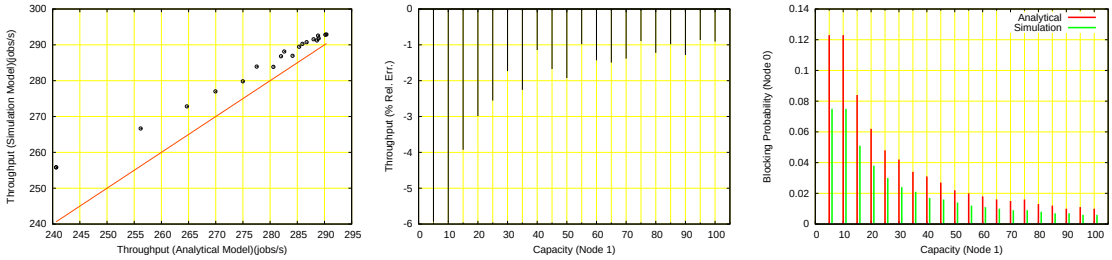
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.27: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 0.5}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [10, 100]}$, x = 3, maxX = 20).



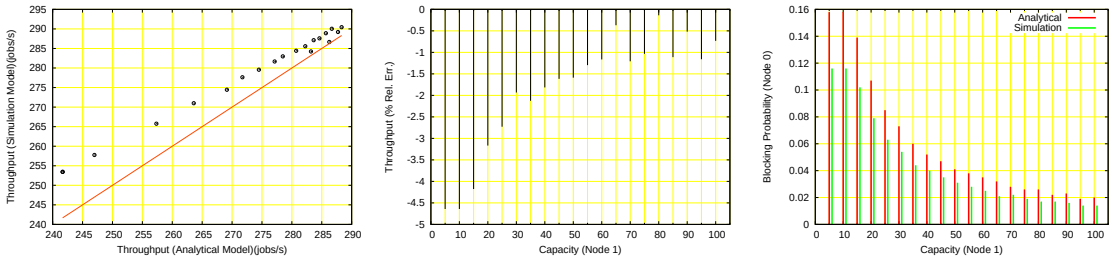
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.28: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 0.5}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [10, 100]}$, x = 4, maxX = 27).



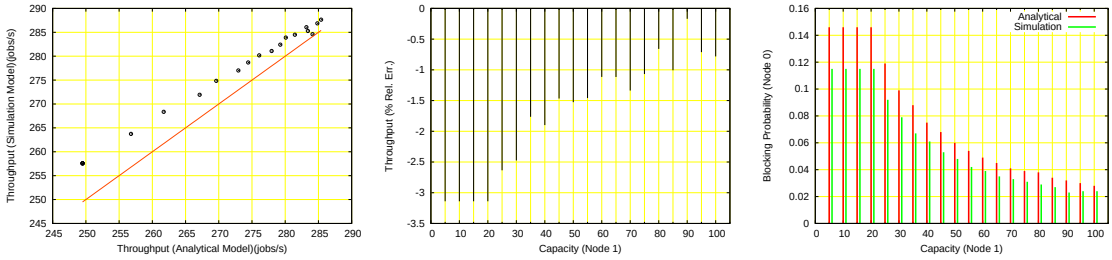
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.29: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 0.8}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [10, 100]}$, x = 1, maxX = 7).



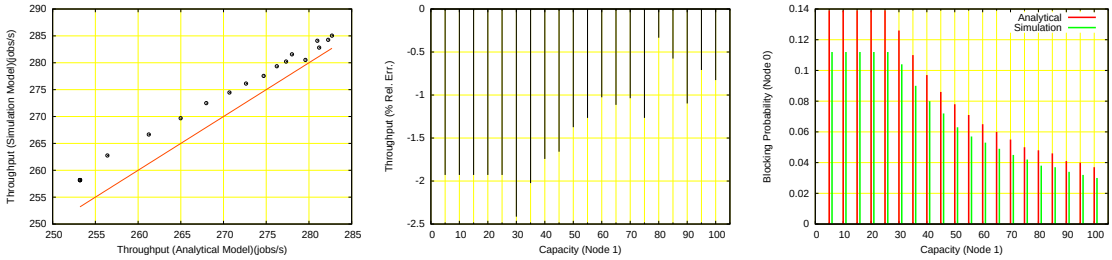
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.30: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 0.8}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [10, 100]}$, x = 2, maxX = 13).



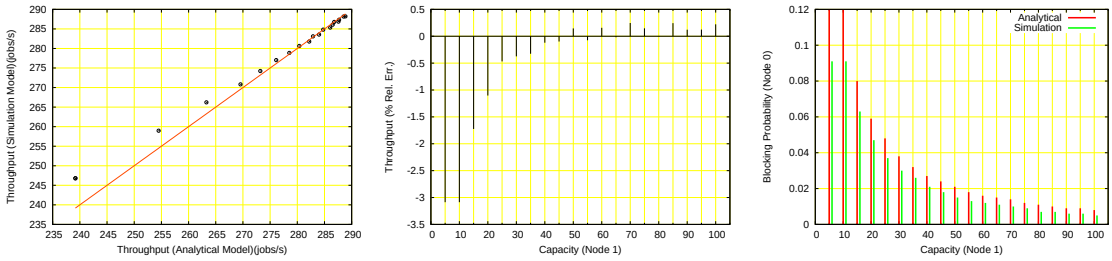
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.31: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 0.8}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [10, 100]}$, x = 3, maxX = 20).



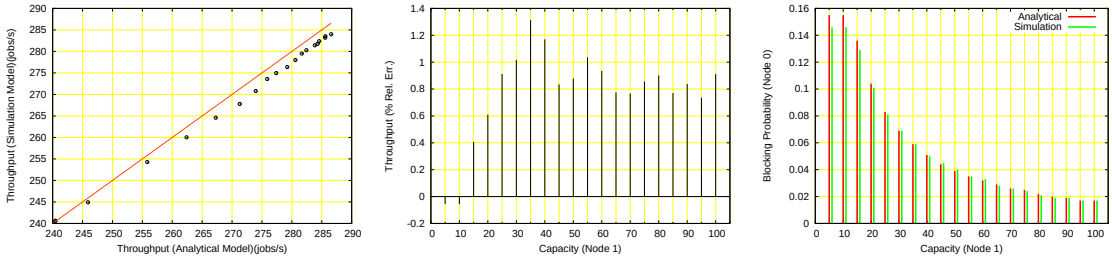
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.32: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 0.8}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [10, 100]}$, x = 4, maxX = 27).



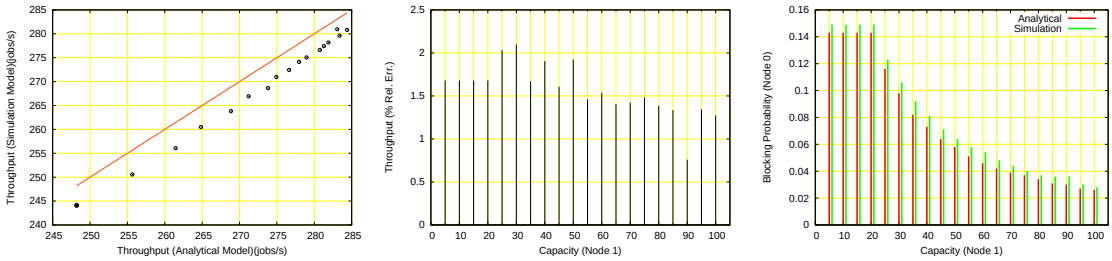
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.33: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 2.0}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [10, 100]}$, x = 1, maxX = 7).



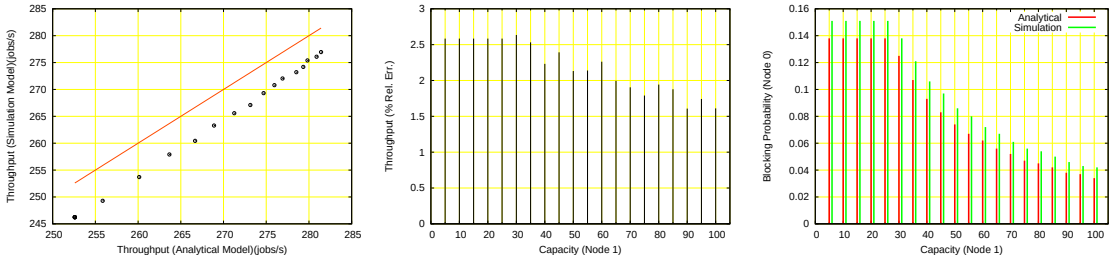
(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.34: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 2.0}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [10, 100]}$, x = 2, maxX = 13).



(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.35: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 2.0}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [10, 100]}$, x = 3, maxX = 20).



(a) Throughput (b) Throughput relative error (c) Upstream blocking probability

Figure 4.36: Throughput and upstream blocking probability predicted by analytical and simulation models ($\boldsymbol{c_0^2 = 2.0}$, $\boldsymbol{c_1^2 = 1}$, $\boldsymbol{\mu_0 = 300}$ jobs/s, $\boldsymbol{\mu_1 = 300}$ jobs/s, $\boldsymbol{K_0 = 100}$, $\boldsymbol{K_1 \in [10, 100]}$, x = 4, maxX = 27).

Table 4.1: Experiments with $>$ **10%** error in Figure 3.5(b).

Experiment #	Analytical (jobs/sec)	Simulation (jobs/sec)	Rel. Err.
120	58.800	48.432	21.407
47	47.607	40.864	16.502
84	138.586	120.643	14.872
24	117.600	103.315	13.827
116	190.060	168.024	13.115

Table 4.2: Test 1 check for experiments with $>$ **10%** error in Figure 3.5(b).

Experiment #	Analytical (jobs/sec)	$\boldsymbol{T_{single}}$ (jobs/sec)	Result
120	58.800	49.074	Fail
47	47.607	40.895	Fail
84	138.586	122.853	Fail
24	117.600	105.302	Fail
116	190.060	177.638	Fail

conclude here that our test 2 will prove a good test for these cases too. Also, we observe here that longer tails on the bursts yield poorer results.

4.3 Validating tests

Now that we have defined the different tests, we investigate if the tests can capture the erroneous results reported for the synthetically generated networks of Chapter 3. We start with the results of Figure 3.5. These experiments included servers whose service times were purely exponentially distributed. We are particular interested in the **5** experiments that have errors greater than 10%. Table 4.1 shows the results predicted by the analytical model for these experiments.

All 5 of these experiments fail test 1 as illustrated in Table 4.2. To perform the test we constructed a single node queue which has the same service rate as the bottleneck node and has a capacity which equals all the capacity upstream from that node. This shows that this test is sufficient in detecting the significant errors in the model which are present with only exponentially distributed service times at the nodes.

We now investigate the results in Figure 3.10 where the service time distributions have a squared coefficient of variance between 0.5 and 20. Of the 200 networks for which we used the test the algorithm 30 of them had a relative error greater than 10%. We tested these 30 networks using test 1 and test 2. 25 of these 30 networks failed test 2 and the rest failed test 1. Note here that we checked for test 2 first, and did not check for test 1 if test 2 failed.

Finally, we revisit the results in Section 3.3.2 to check the results produced by the analytical model. We had tested 200 networks where the upstream node had bursty departures, and for 52 of these the analytical model predicted throughput which were not withing 10% of the simulation results. Again, tests 1 and 2 eliminated all of these errors.

4.4 Chapter Summary

In this chapter we introduce two tests to avoid potential pitfalls with the analytical models described in Chapter 3. The first test checks if the solution performance metrics, throughput, put out by the analytical model exceed known theoretical limits. We were able to avoid the short-comings of the results presented in Section 3.2 of Chapter 3.

The second test is based on our experiments which tried to study the effect of non-exponential service time distributions at upstream nodes. This empirical test concludes that a tail, coefficient of variance of $\geq \mathbf{2}$, on a node upstream from the bottleneck node on the system can potentially lead to erroneous results. Hence, we conclude that our extension to include more general service distribution needs to meet this criteria too. We see that these tests help curtail the relative error between analytical and simulation models to within 10%. This is within the arbitrary threshold we set for the validity of the analytical mode. We note here that there could be cases where one could have false positives from the analytical model. The analytical model could potentially pass these tests and still produce incorrect estimates. Also, test 2 is a conservative in nature and one could potentially be discarding results of the analytical model for networks where the model works well. In essence tests 1 and 2 are necessary but not sufficient checks on the correctness of the analytical model.

Our experiments which aim to study the effect of bursty departures from the upstream node also yielded results similar to the ones which do not have bursty departures. These results show that we can use state-space solutions to solve for single-server models that don't have closed form solutions and use this backward traversing model to yield fast analytical solutions to queueing networks with blocking.

Chapter 5

Performance Evaluation Using Soft-Core Processors

5.1 Motivation

Simulation has continued to be the *de facto* standard method for performance evaluation of newly proposed ideas in computer architecture for more than a decade now. While simulation allows for theoretically arbitrary fidelity (at least to the level of cycle accuracy) as well as the ability to monitor the architecture without perturbing execution itself, it suffers from low effective fidelity and long execution times.

- Fidelity – simulations are typically performed with abstract, incomplete, or missing components. While it is conceptually easy to describe a "cycle accurate" simulation model, there is little hope of the eventual implementation reflecting the details of that simulation exactly.
- Execution time – simulations are typically interpretive so that they can perform sufficient introspection to collect data of interest. As such, they typically run orders of magnitude slower than the system they are modeling.

One viable remediation for the above concerns is to simulate portions of an application's execution by sampling [126]; some segments are simulated in fine detail while the details of executing the other segments are largely ignored. For average behavior, sampling may provide adequate resolution in a reasonable time frame. However, if an execution contains infrequent events whose details are important, sampling may miss the most noteworthy phenomena.

As the research which led to development and the extensive use of SimOS [102] suggests, it is not only important to measure the effectiveness of architectures using standard benchmarks like SPEC-2000, MiBench etc., but also account for the behavior of the application under appropriate operation systems. This additional criteria raises serious issues with trace driven simulators, because the traces which capture the OS effects in detail are long and "need" to be well sampled. Moreover, for some applications, occasional worst-case behavior can be more significant than the application's average-case behavior. For example, in the real-time application we consider below, its worst-case execution time is necessary for proper scheduling within the application. A rare event that can lead to increased execution time can adversely affect scheduling or perhaps cause a real-time program to miss a crucial deadline.

Reconfigurable hardware, in the form of field-programmable gate arrays (FPGAs), can be used to model systems that will ultimately be implemented in custom silicon. In fact, soft-core descriptions of common architecture implementations are becoming widely available [79]. With the appropriate instrumentation of such descriptions, and the addition of logic to log events reliably, execution details at the micro-architectural level can be captured at full (FPGA) speed for an application's entire execution [54].

While much of a program's observed behavior is intrinsic to the application and its host architecture, other processes—some of which may be required for proper system operation—can adversely affect the primary application's performance. System processes responsible for resource management, device allocation, page management, and logging typically execute outside the domain of an application's performance introspection. Thus, debugging and performance monitoring tools can reveal much about about problems within an application, but they are hard pressed to identify interprocess behavior that contributes to poor performance. Moreover interprocess behavior can even mask performance problems within an application.

5.2 Overview

In the chapter, we explore the space in which simulator based performance analysis is classically used, looking to assess whether or not it is sufficiently effective. FPGA-based emulation is presented as an alternative, more effective approach for some performance analysis problems. We start by presenting the evaluation of an micro-architectural technique called "Dusty Cache" on our evaluation platform. We emphasize the difference in

performance metrics between standalone measurements and measurements including the effects of a "tiny" uClinux operating system. We then present results from execution of benchmarks on a full-blown version of Linux.

The last part of the chapter describes the use of our measurement infrastructure to identify and reason rare-events observed during the execution of applications under an operating system. This ability is key to real-time design which needs to meet very stringent deadlines with few or no misses.

5.3 Benchmarks

When proposing some new (or altered) micro-architectural feature, it is standard practice to empirically evaluate the efficacy of that feature across a set of benchmark applications. Similarly, the micro-architectural feature might not be new at all, in an embedded system design we might simply be interested in choosing the parameter setting that best suits the needs of the current application. In our experiments, we illustrate the ability to do this type of investigation by measuring cache hit rates as a function of cache size and associativity for a set of benchmark applications (most of which are traditionally embedded applications). Our particular interest here is the degree to which the performance measures are altered by the presence of both an operating system and other applications present and competing for processor resources.

We create a benchmark suite for experiments/evaluations in this chapter by collecting applications from two different benchmark suites. The MiBench benchmark suite [48] consists of a set of embedded system workloads which differ from standard desktop workloads. The applications contained in the MiBench suite were selected to capture the diversity of workloads in embedded systems. For the purposes of this study, we chose workloads from the networking, telecommunications, and automotive sections of the suite.

CommBench [123] was designed with the goal of evaluating and designing telecommunications network processors. The benchmark consists of **8** programs, **4** of which focus on packet header processing, and the other **4** are geared towards data stream processing.

Following are the set of applications we have used as part of this study:

- From MiBench:

  - *basicmath*: This application is part of the automotive applications inside MiBench. It computes cubic functions, integer square roots and angle conversions.
  - *sha*: This is part of the networking applications inside MiBench. Sha is a secure hash algorithm which computes a 160-bit digest of inputs.
  - *fft*: This is part of the telecommunication applications inside MiBench, and computes the fast Fourier transform on an array of data.

- From CommBench:
  - *drr*, *frag*: These are part of the header processing apps inside CommBench. The drr algorithm is used for bandwidth scheduling for large numbers of flows. Frag refers to the fragmentation algorithm used in networking to split IP packets.
  - *reed_enc*, *reed_dec*: These are part of the packet processing applications in CommBench. They are the encoder and decoder used in the Reed-Solomon forward error correction scheme.

To the above set we added one locally developed benchmark, *blastn*, which implements the first (hashing) stage of the popular BLASTN biosequence alignment application for nucleotide sequences [3][1]. All of the benchmark application either already are, or are candidates to be, individual pipeline stages in a larger complete application.

5.4 Profiling app-only vs. app and uClinux

In the section we present the difference that one could see when incorporating the effects of a light weight OS like uClinux [117]. For this purpose we use the Dusty cache design and evaluate its effectives by measuring application effects under these scenario:

- Standalone: The application is compiled to run on the Liquid architecture platform by itself, and is the only process on the system.
- uClinux: The application is run on the Liquid architecture platform using the uClinux OS. The statstics measured include the activity of both application and the OS.

[1]This does the not reflect any changes made to the BLASTN pipe described in the following chapter.

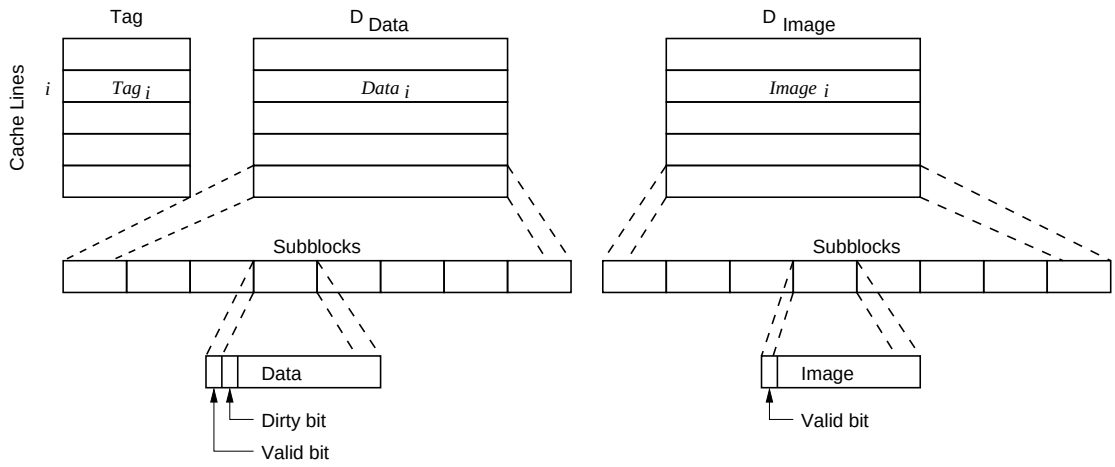


Figure 5.1: Dusty cache structural design.

5.4.1 Dusty Cache

Dusty caches were introduced by Friedman [40, 39] to reduce the number of redundant stores to memory. The architecture eliminates temporally silent stores [80], which attempt to re-write to memory the value that is fetched from memory. In [40], Friedman evaluated the use of this techniques using probabilistic models using Monte-Carlo simulations. We evaluate this architecture using the measurement architecture on the Liquid architecture platform.

In this section we describe the dusty data cache micro-architecture optimization and discuss its design and interaction with the machine architecture. This exposition is a shortened version of what appears in [40]. We classify this policy as an enhancement to a standard write-back policy. This dusty cache specification is implemented in the Liquid Architecture system as a data cache.

5.4.2 Dusty Cache Design

The dusty cache employs the same lines (blocks), subblocks, and valid bits as both traditional write-through and write-back policies. The write-back cache policy uses a dirty bit to decide when to write a value back to main memory. The dusty cache uses a *dusty check* to decide when to write the value back to main memory. The dusty check is not an actual bit (in the sense of a "dirty bit"), but is instead a mechanism for deciding if the cached value duplicates what was fetched from storage initially. Like the write-back policy, the dusty

cache has a dirty bit to decide whether the value has changed since entering the cache. In addition, the dusty cache has a second cache bank that acts as an image of main memory, labeled $\boldsymbol{D_{Image}}$ in Figure 5.1. This bank is readily accessible without incurring the time delay of reading main memory, and does not impact the the access time of $\boldsymbol{D_{Data}}$. In our implementation, we actually duplicate the data cache to realize the image; in systems offering L2 cache, that layer could serve as the image if it can be accessed sufficiently quickly. Lepak [80] describes a number of alternative implementation strategies.

5.4.3 Experiments

For this study we used 7 applications from the set of applications described earlier. For each of the applications the following sequence of actions were taken:

- The application was executed standalone (i.e., no OS) on the Liquid Architecture system. For benchmarks that require disk-based input data, the benchmark was altered to either read compile-time initialized data or synthetically generate the input data. This step is required because there is no file system available for standalone execution. In addition, the benchmark was adapted to initiate the statistics collection subsystem (this required the addition of a single call at the beginning of the code).
- The modified application was also executed under the uClinux OS. Even though we have a file system for this case, our desire to study the impact of applications running with and without an OS motivated us to use the identical (altered) applications from the standalone runs.
- A number of different configurations of the LEON processor were generated. Cache sizes of 1, 2, 4, and 8 Kbytes were included for a traditional direct-mapped, write-back cache, and cache sizes of 1, 2, and 4 Kbytes were included for a dusty cache. For the dusty cache, the sizes above do not include the $\boldsymbol{D_{Image}}$ memory, so the actual on-chip memory usage for a dusty cache configuration is twice that listed above. That is, the listed cache size is the amount visible to the processor, $\boldsymbol{D_{Data}}$.
- Each of the applications was executed on each of the processor configurations, measuring loads, stores, cache hits, cache misses, memory reads, and memory writes.

5.4.4 Performance Results

Table 5.1 shows the total number of loads and stores for each of the benchmark applications.

Table 5.1: Total number of load and store instructions for each benchmark.

Benchmark	Loads	Stores
basicmath	74,151,136	46,577,732
dijkstra	60,985,966	8,925,084
drr	180,171,104	92,676,773
frag	178,058,884	96,099,734
reed_enc	149,063,969	62,261,586
reed_dec	208,593,061	89,553,034
sha	424,476,497	158,084,970

Standalone Execution

The initial performance results are presented for the applications running standalone. Figure 5.2 shows, for each application and each cache size, the total count of memory writes that occur in an individual execution. Given that this is a write-back cache, these writes to the memory subsystem occur primarily as a result of cache evictions. This represents the baseline memory write traffic with the write-back cache, which is what is intended to improve from the dusty cache architecture. We note here that as these experiments were run standalone there was no variability in between different executions of the application.

Figure 5.3 shows the savings in memory writes (as a percentage of the original number of memory writes shown in Figure 5.2) for each application and cache size when using a dusty cache. Note that the on-chip memory requirements have doubled for the dusty cache implementation, so this comparison only makes sense when in a design environment where this extra memory requirement isn't critical. This might be the case, for example, in a multi-level cache system, where the $\boldsymbol{D_{Image}}$ memory is actually implemented as part of the next level in the memory hierarchy. Additional implementation techniques that do not explicitly require double the memory are described in [80]. An alternative way to view this figure is that it is a measure of the frequency of silent stores that potentially can be squashed, irrespective of the cost of the method.

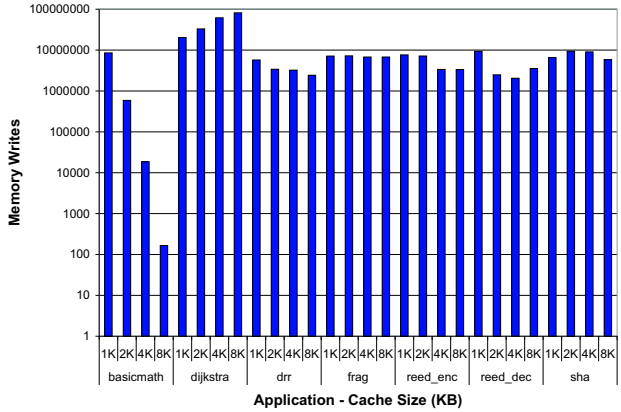


Figure 5.2: Count of memory writes for traditional write-back, direct-mapped cache for each application and cache size. The applications are executing standalone.

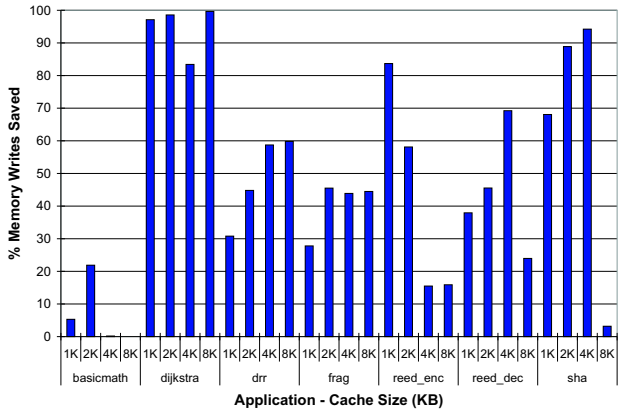


Figure 5.3: Percentage of memory writes saved with a dusty cache. The applications are executing standalone.

What is noteworthy here is the number of cases where the savings are quite substantial, frequently well above 80%. This is an even greater frequency of silent stores than that reported in [82]. The surprisingly large fraction of silent stores in the dijkstra benchmark is due to the repeated traversals of the graph, most of the time writing values that are already present.

As we assume that the cost of the $\boldsymbol{D_{Image}}$ is not prohibitive, we next consider an alternative control mechanism. Figure 5.4 shows the writeback savings at the full block level, rather than the subblock level of Figure 5.3. Under the assumption that memory transactions occur for complete cache lines, the dusty cache will save a memory write only when the entire cache line is either not dirty or matches the contents of $\boldsymbol{D_{Image}}$. As can be seen

in the plot, the savings are qualitatively very similar to the case where decisions are being made at the individual subblock level. We attribute the improvements seen in some cases (e.g., drr, reed_enc, and reed_dec for both 4 KB and 8 KB cache sizes) to write locality.

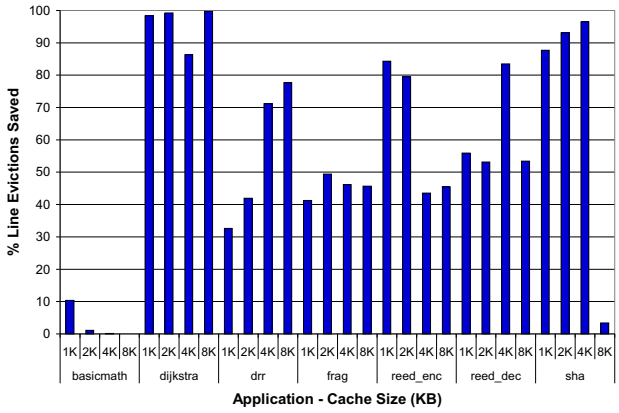


Figure 5.4: Percentage of line evictions saved with a dusty cache. The applications are executing standalone.

Execution with the Operating System

When running the benchmark applications on the OS, we do not separately measure memory operations for the application and the OS, but rather measure aggregate memory writes from all sources. Figure 5.5 plots the number of memory write operations for each application and cache size when running on the uClinux OS. Note that in virtually all cases, the memory performance is noticeably different (including both increases and decreases in memory writes) than the standalone case.

Following in the pattern of the previous subsection, we next show the percentage of memory writes saved with a dusty cache implementation that is of the same size $\boldsymbol{D_{Data}}$ as the write-back cache. This is illustrated in Figure 5.6. While the particular results are distinct from the standalone execution, the savings are still surprisingly large.

Finally, Figure 5.7 shows the savings due to the dusty cache policy if writeback decisions are being made at the full block level rather than the subblock level. An interesting observation to be made here is that there is generally better similarity between word evictions and line evictions (i.e., comparing Figure 5.3 to Figure 5.4 and Figure 5.6 to Figure 5.7) than there is similarity between executing standalone and with the OS (i.e., comparing Figure 5.3 to Figure 5.6 and Figure 5.4 to Figure 5.7).

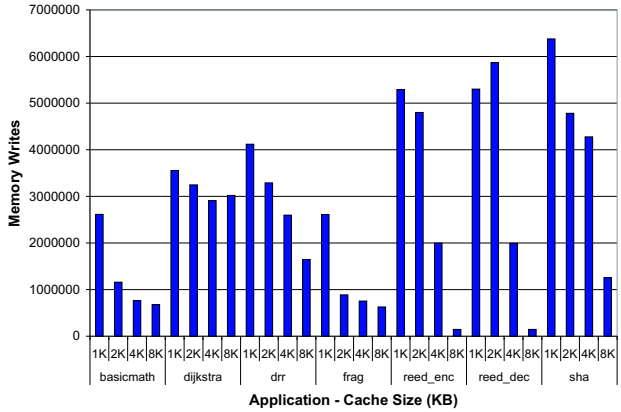


Figure 5.5: Count of memory writes for traditional write-back, direct-mapped cache for each application and cache size. The applications are executing on the OS.

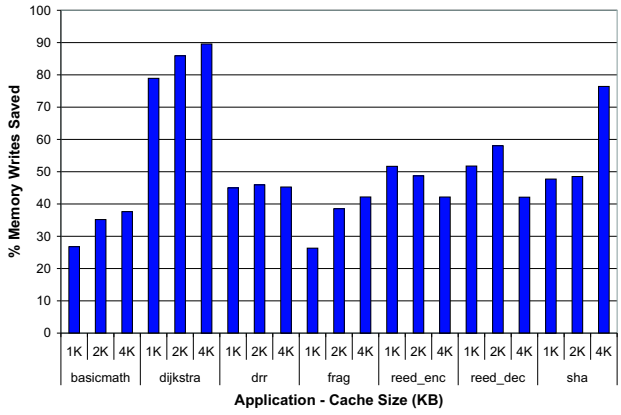


Figure 5.6: Percentage of memory writes saved with a dusty cache. The applications are executing on the OS.

5.5 Summarizing Standalone Vs. uClinux

There are a number of conclusions that can be drawn from the above data. First, temporally silent stores are frequently occurring. This is illustrated in both the experiments that have the applications running standalone, and when the applications are executing under an operating system (uClinux). The embedded benchmarks examined here exhibit even greater frequency of silent stores than previously published results (which focused on scientific and commercial workloads). Also, the effectiveness of the Dusty cache is significantly different between standalone execution and executions under uClinux. Hence, we conclude that a decision on their inclusion should not be be made by examining the application in isolation,

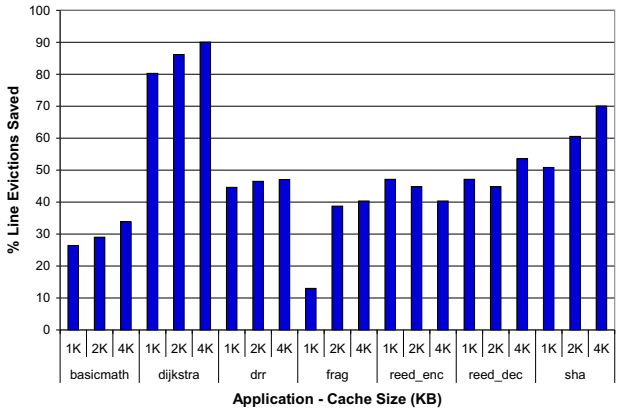


Figure 5.7: Percentage of line evictions saved with a dusty cache. The applications are executing on the OS.

but rather must include the impact of the run time system if that indeed is the way the applications would be executed in the eventual system.

The significant differences seen in the results with and without the OS motivated us to further our investigations in this area. These results are presented in [71].

5.6 Effect of Resource Competition on Application Performance

In the earlier sections we observed that the impact of the operating system cannot be neglected when evaluating architectural changes/ideas using benchmarks. We saw that the conclusions we arrive at depend, to a significant extent, on the inclusion/exclusion of operating system effects. Extending this idea a little we should also be aware, and also account for, other applications besides the benchmark executing on the operating system.

Procedure

For each of the applications, the following sequence of actions was taken:

- The application was executed under the Linux 2.6.11 OS on the Liquid Architecture system. The OS was in single-user mode with no other user applications enabled. The

benchmark was adapted to initiate the statistics collection subsystem (this required the addition of a single call at the beginning of the code).

- A subset of the applications was executed with one or more competing applications concurrently scheduled. The competing applications were drawn from the same benchmark set. Figure 5.2 shows the pairing of primary and competing applications.

Table 5.2: Pairings of primary and competing applications.

primary application	single competing application	set of 3 competing applications
drr	*frag*	—
frag	*reed_dec*	—
reed_dec	*reed_dec*	—
sha	*fft*	—
blastn	*fft*	*fft*, *drr*, *frag*
fft	*reed_enc*	*reed_enc*, *reed_dec*, *fft*

- A number of different configurations of the LEON processor were generated. Data cache sizes of 2, 4, 8, and 16 Kbytes were included for a two-way associative cache, and cache sizes of 4, 8, and 16 Kbytes were included for a four-way associative cache.

- Sets of the applications were executed on each of the processor configurations, measuring loads, stores, cache hits, cache misses, memory reads, and memory writes. It is the variations in these parameters that we wish to examine.

Each execution was repeated five times. The results are reported in Tables 5.3 through 5.5, and the they include both mean and the 95% confidence intervals.

Cache Behavior Results

Figures 5.3 to 5.5 show the mean execution time, data cache read miss rates, and data cache write miss rates for the benchmark applications when executing code in the virtual address range 0 to 0x7FFFFFFF (i.e., the .text segment of the application itself, excluding all system calls). Each of these values is presented twice. The first is for the entire execution (i.e., application, OS, and any competing applications) and the second is for the application

alone (i.e., configuring the statistics module to be sensitive only to the primary application process).

The ability to collect this data illustrates the discriminatory features of the statistics module, both restricting the PID to that of the application and also restricting the address space of the data collection within the application. In addition, the ability to cover this wide of a design space is significantly enabled by the fact that the investigation is executing on an FPGA rather than a simulation model. The execution time required to collect all of the data comprised approximately 4 trillion clock cycles, requiring approximately 40 hours of FPGA execution time. This would have been simply prohibitive in a software simulation environment.

The graphs that follow illustrate some of the interesting features of this data set. Figure 5.8 shows the application-only execution time for *fft* for varying dcache configurations when running on the OS without a competing application. Figure 5.9 shows the complete execution time (*fft* application plus OS) for the same set of experiments. Note that there is only a slight increase in execution time across the board, implying that here the OS is not significantly impacting the execution of the application (i.e., at each scheduling quantum, the scheduler simply returns to the application). Figure 5.10 plots *fft*-only execution time when there is a competing application (*reed_enc*) scheduled concurrently. Note the similarity to Figure 5.8, indicating that the competing application doesn't significantly impact the execution time required for *fft* alone. Contrast this with Figure 5.11, which plots the total execution time for all of *fft*, the competing application (*reed_enc*), and the OS. Here there is clearly an increase in execution time, as expected, due to the significant additional computational requirements associated with both applications vs. just one solo application.

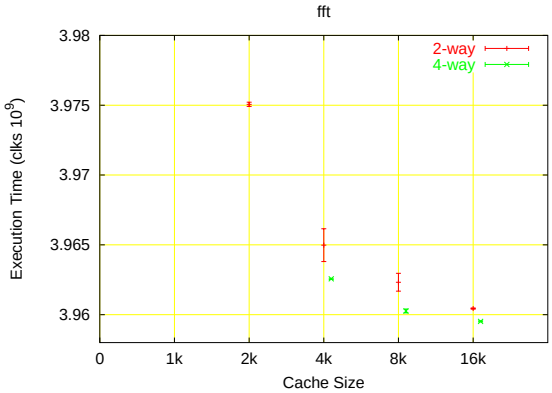


Figure 5.8: Execution time (in billions of clock cycles) for *fft* running on OS with no other competing application. Various dcache configurations are shown.

Table 5.3: Execution time results for 8 benchmark applications.

App	size, assoc.	os + app		os + 2 apps		os + 4 apps	
		os+app 10^9 clks	app 10^9 clks	os+2apps 10^9 clks	app 10^9 clks	os+4apps 10^9 clks	app 10^9 clks
basic-math	2K, 2	8.47 ± 0.0020	8.46 ± 0.0005	—	—	—	—
	4K, 2	8.45 ± 0.0016	8.44 ± 0.0017	—	—	—	—
	8K, 2	8.44 ± 0.0007	8.43 ± 0.0008	—	—	—	—
	16K, 2	8.43 ± 0.0001	8.42 ± 0.0001	—	—	—	—
	4K, 4	8.44 ± 0.0006	8.43 ± 0.0005	—	—	—	—
	8K, 4	8.43 ± 0.0002	8.43 ± 0.0002	—	—	—	—
	16K, 4	8.43 ± 0.0001	8.42 ± 0.0002	—	—	—	—
reed_enc	2K, 2	2.02 ± 0.0001	2.02 ± 0.0002	—	—	—	—
	4K, 2	1.94 ± 0.0005	1.93 ± 0.0011	—	—	—	—
	8K, 2	1.92 ± 0.0011	1.92 ± 0.0002	—	—	—	—
	16K, 2	1.92 ± 0.0000	1.92 ± 0.0001	—	—	—	—
	4K, 4	1.93 ± 0.0004	1.93 ± 0.0009	—	—	—	—
	8K, 4	1.92 ± 0.0002	1.92 ± 0.0001	—	—	—	—
	16K, 4	1.92 ± 0.0001	1.92 ± 0.0001	—	—	—	—
drr	2K, 2	2.71 ± 0.0005	2.70 ± 0.0005	5.24 ± 0.0003	2.70 ± 0.0004	—	—
	4K, 2	2.65 ± 0.0005	2.65 ± 0.0005	5.18 ± 0.0003	2.65 ± 0.0004	—	—
	8K, 2	2.62 ± 0.0007	2.61 ± 0.0004	5.15 ± 0.0005	2.61 ± 0.0002	—	—
	16K, 2	2.61 ± 0.0008	2.60 ± 0.0001	5.13 ± 0.0002	2.60 ± 0.0004	—	—
	4K, 4	2.65 ± 0.0001	2.64 ± 0.0001	5.18 ± 0.0001	2.64 ± 0.0002	—	—
	8K, 4	2.61 ± 0.0001	2.61 ± 0.0000	5.14 ± 0.0002	2.61 ± 0.0003	—	—
	16K, 4	2.61 ± 0.0002	2.60 ± 0.0002	5.13 ± 0.0002	2.60 ± 0.0002	—	—
frag	2K, 2	2.54 ± 0.0000	2.54 ± 0.0006	5.08 ± 0.0202	2.54 ± 0.0008	—	—
	4K, 2	2.54 ± 0.0014	2.53 ± 0.0000	5.08 ± 0.0060	2.53 ± 0.0019	—	—
	8K, 2	2.53 ± 0.0006	2.53 ± 0.0003	5.08 ± 0.0096	2.53 ± 0.0004	—	—
	16K, 2	2.53 ± 0.0000	2.53 ± 0.0003	5.08 ± 0.0167	2.53 ± 0.0002	—	—
	4K, 4	2.53 ± 0.0001	2.53 ± 0.0000	5.09 ± 0.0132	2.53 ± 0.0002	—	—
	8K, 4	2.53 ± 0.0000	2.53 ± 0.0000	5.08 ± 0.0237	2.53 ± 0.0002	—	—
	16K, 4	2.53 ± 0.0000	2.53 ± 0.0001	5.08 ± 0.0244	2.53 ± 0.0003	—	—
reed_dec	2K, 2	4.68 ± 0.0009	4.68 ± 0.0015	9.35 ± 0.0195	4.68 ± 0.0021	—	—
	4K, 2	4.61 ± 0.0049	4.60 ± 0.0008	9.21 ± 0.0032	4.60 ± 0.0051	—	—
	8K, 2	4.60 ± 0.0073	4.59 ± 0.0005	9.19 ± 0.0069	4.59 ± 0.0044	—	—
	16K, 2	4.58 ± 0.0005	4.58 ± 0.0005	9.15 ± 0.0212	4.58 ± 0.0017	—	—
	4K, 4	4.60 ± 0.0013	4.60 ± 0.0004	9.20 ± 0.0023	4.60 ± 0.0010	—	—
	8K, 4	4.59 ± 0.0006	4.58 ± 0.0005	9.15 ± 0.0219	4.58 ± 0.0020	—	—
	16K, 4	4.58 ± 0.0000	4.58 ± 0.0010	9.14 ± 0.0273	4.58 ± 0.0023	—	—
sha	2K, 2	9.17 ± 0.0041	9.16 ± 0.0035	13.13 ± 0.0022	9.14 ± 0.0027	—	—
	4K, 2	9.14 ± 0.0073	9.12 ± 0.0000	13.10 ± 0.0090	9.12 ± 0.0091	—	—
	8K, 2	9.12 ± 0.0054	9.11 ± 0.0006	13.07 ± 0.0058	9.10 ± 0.0008	—	—
	16K, 2	9.11 ± 0.0003	9.11 ± 0.0011	13.04 ± 0.0007	9.08 ± 0.0002	—	—
	4K, 4	9.13 ± 0.0020	9.12 ± 0.0022	13.08 ± 0.0021	9.11 ± 0.0016	—	—
	8K, 4	9.12 ± 0.0017	9.11 ± 0.0002	13.05 ± 0.0011	9.09 ± 0.0009	—	—
	16K, 4	9.11 ± 0.0003	9.10 ± 0.0000	13.04 ± 0.0005	9.07 ± 0.0010	—	—
blastn	2K, 2	4.50 ± 0.0000	4.49 ± 0.0001	8.47 ± 0.0002	4.49 ± 0.0003	13.71 ± 0.0012	4.49 ± 0.0003
	4K, 2	4.49 ± 0.0002	4.48 ± 0.0001	8.45 ± 0.0003	4.48 ± 0.0001	13.63 ± 0.0008	4.48 ± 0.0001
	8K, 2	4.48 ± 0.0005	4.48 ± 0.0001	8.44 ± 0.0004	4.48 ± 0.0002	13.59 ± 0.0002	4.48 ± 0.0002
	16K, 2	4.47 ± 0.0002	4.46 ± 0.0000	8.43 ± 0.0004	4.46 ± 0.0004	13.56 ± 0.0009	4.46 ± 0.0006
	4K, 4	4.48 ± 0.0003	4.48 ± 0.0002	8.44 ± 0.0006	4.48 ± 0.0004	13.62 ± 0.0003	4.47 ± 0.0001
	8K, 4	4.47 ± 0.0004	4.47 ± 0.0004	8.43 ± 0.0003	4.47 ± 0.0002	13.57 ± 0.0002	4.47 ± 0.0003
	16K, 4	4.47 ± 0.0006	4.47 ± 0.0006	8.43 ± 0.0006	4.47 ± 0.0005	13.56 ± 0.0005	4.46 ± 0.0005
fft	2K, 2	3.98 ± 0.0002	3.98 ± 0.0002	6.00 ± 0.0010	3.98 ± 0.0004	13.95 ± 0.0113	3.98 ± 0.0005
	4K, 2	3.97 ± 0.0012	3.96 ± 0.0012	5.90 ± 0.0017	3.97 ± 0.0018	13.84 ± 0.0231	3.97 ± 0.0005
	8K, 2	3.97 ± 0.0000	3.96 ± 0.0006	5.88 ± 0.0004	3.96 ± 0.0005	13.82 ± 0.0096	3.96 ± 0.0009
	16K, 2	3.96 ± 0.0000	3.96 ± 0.0000	5.88 ± 0.0008	3.96 ± 0.0005	13.83 ± 0.0195	3.96 ± 0.0003
	4K, 4	3.97 ± 0.0005	3.96 ± 0.0000	5.89 ± 0.0009	3.96 ± 0.0002	13.85 ± 0.0172	3.96 ± 0.0004
	8K, 4	3.96 ± 0.0001	3.96 ± 0.0001	5.88 ± 0.0004	3.96 ± 0.0007	13.83 ± 0.0100	3.96 ± 0.0003
	16K, 4	3.96 ± 0.0001	3.96 ± 0.0000	5.88 ± 0.0005	3.96 ± 0.0005	13.82 ± 0.0116	3.96 ± 0.0004

Table 5.4: Dcache read miss rate results for 8 benchmark applications.

App	size, assoc.	os + app		os + 2 apps		os + 4 apps	
		os+app miss rate	app alone miss rate	os+2apps miss rate	app miss rate	os+4apps miss rate	app miss rate
basic-math	2K, 2	0.02 ± 0.0008	0.02 ± 0.0002	—	—	—	—
	4K, 2	0.02 ± 0.0007	0.02 ± 0.0007	—	—	—	—
	8K, 2	0.01 ± 0.0003	0.01 ± 0.0003	—	—	—	—
	16K, 2	0.01 ± 0.0001	0.01 ± 0.0001	—	—	—	—
	4K, 4	0.01 ± 0.0002	0.01 ± 0.0002	—	—	—	—
	8K, 4	0.01 ± 0.0001	0.01 ± 0.0001	—	—	—	—
	16K, 4	0.01 ± 0.0001	0.01 ± 0.0001	—	—	—	—
reed_enc	2K, 2	0.04 ± 0.0000	0.04 ± 0.0001	—	—	—	—
	4K, 2	0.01 ± 0.0002	0.01 ± 0.0004	—	—	—	—
	8K, 2	0.01 ± 0.0004	0.01 ± 0.0001	—	—	—	—
	16K, 2	0.01 ± 0.0000	0.01 ± 0.0000	—	—	—	—
	4K, 4	0.01 ± 0.0002	0.01 ± 0.0003	—	—	—	—
	8K, 4	0.01 ± 0.0001	0.01 ± 0.0000	—	—	—	—
	16K, 4	0.01 ± 0.0000	0.01 ± 0.0000	—	—	—	—
drr	2K, 2	0.08 ± 0.0004	0.08 ± 0.0004	0.05 ± 0.0001	0.08 ± 0.0003	—	—
	4K, 2	0.04 ± 0.0003	0.04 ± 0.0003	0.03 ± 0.0001	0.04 ± 0.0003	—	—
	8K, 2	0.02 ± 0.0005	0.02 ± 0.0003	0.02 ± 0.0002	0.02 ± 0.0002	—	—
	16K, 2	0.01 ± 0.0005	0.01 ± 0.0001	0.01 ± 0.0001	0.01 ± 0.0002	—	—
	4K, 4	0.04 ± 0.0000	0.04 ± 0.0001	0.03 ± 0.0000	0.04 ± 0.0002	—	—
	8K, 4	0.02 ± 0.0000	0.02 ± 0.0000	0.01 ± 0.0001	0.02 ± 0.0002	—	—
	16K, 4	0.01 ± 0.0001	0.01 ± 0.0002	0.01 ± 0.0001	0.01 ± 0.0001	—	—
frag	2K, 2	0.02 ± 0.0001	0.02 ± 0.0005	0.02 ± 0.0003	0.02 ± 0.0006	—	—
	4K, 2	0.01 ± 0.0010	0.01 ± 0.0000	0.01 ± 0.0009	0.01 ± 0.0014	—	—
	8K, 2	0.01 ± 0.0005	0.01 ± 0.0002	0.01 ± 0.0002	0.01 ± 0.0003	—	—
	16K, 2	0.01 ± 0.0000	0.01 ± 0.0003	0.01 ± 0.0002	0.01 ± 0.0002	—	—
	4K, 4	0.01 ± 0.0001	0.01 ± 0.0000	0.01 ± 0.0003	0.01 ± 0.0002	—	—
	8K, 4	0.01 ± 0.0000	0.01 ± 0.0000	0.01 ± 0.0003	0.01 ± 0.0002	—	—
	16K, 4	0.01 ± 0.0000	0.01 ± 0.0001	0.01 ± 0.0005	0.01 ± 0.0002	—	—
reed_dec	2K, 2	0.03 ± 0.0002	0.03 ± 0.0003	0.03 ± 0.0003	0.03 ± 0.0004	—	—
	4K, 2	0.01 ± 0.0010	0.01 ± 0.0002	0.01 ± 0.0003	0.01 ± 0.0011	—	—
	8K, 2	0.01 ± 0.0015	0.01 ± 0.0001	0.01 ± 0.0007	0.01 ± 0.0009	—	—
	16K, 2	0.01 ± 0.0001	0.01 ± 0.0001	0.01 ± 0.0003	0.01 ± 0.0004	—	—
	4K, 4	0.01 ± 0.0003	0.01 ± 0.0001	0.01 ± 0.0002	0.01 ± 0.0002	—	—
	8K, 4	0.01 ± 0.0001	0.01 ± 0.0001	0.01 ± 0.0003	0.01 ± 0.0004	—	—
	16K, 4	0.01 ± 0.0000	0.01 ± 0.0002	0.01 ± 0.0001	0.01 ± 0.0005	—	—
sha	2K, 2	0.01 ± 0.0002	0.01 ± 0.0002	0.01 ± 0.0001	0.01 ± 0.0001	—	—
	4K, 2	0.01 ± 0.0004	0.01 ± 0.0000	0.01 ± 0.0004	0.01 ± 0.0005	—	—
	8K, 2	0.00 ± 0.0003	0.00 ± 0.0000	0.00 ± 0.0003	0.00 ± 0.0000	—	—
	16K, 2	0.00 ± 0.0000	0.00 ± 0.0001	0.00 ± 0.0000	0.00 ± 0.0000	—	—
	4K, 4	0.01 ± 0.0001	0.00 ± 0.0001	0.01 ± 0.0001	0.00 ± 0.0001	—	—
	8K, 4	0.00 ± 0.0001	0.00 ± 0.0000	0.00 ± 0.0001	0.00 ± 0.0000	—	—
	16K, 4	0.00 ± 0.0000	0.00 ± 0.0000	0.00 ± 0.0000	0.00 ± 0.0001	—	—
blastn	2K, 2	0.04 ± 0.0000	0.04 ± 0.0000	0.03 ± 0.0000	0.04 ± 0.0001	0.04 ± 0.0002	0.03 ± 0.0001
	4K, 2	0.03 ± 0.0000	0.03 ± 0.0000	0.03 ± 0.0001	0.03 ± 0.0000	0.03 ± 0.0001	0.03 ± 0.0000
	8K, 2	0.03 ± 0.0001	0.03 ± 0.0000	0.03 ± 0.0001	0.03 ± 0.0000	0.02 ± 0.0000	0.03 ± 0.0000
	16K, 2	0.03 ± 0.0000	0.03 ± 0.0000	0.02 ± 0.0001	0.03 ± 0.0001	0.02 ± 0.0001	0.03 ± 0.0002
	4K, 4	0.03 ± 0.0001	0.03 ± 0.0000	0.03 ± 0.0001	0.03 ± 0.0001	0.03 ± 0.0000	0.03 ± 0.0000
	8K, 4	0.03 ± 0.0001	0.03 ± 0.0001	0.02 ± 0.0001	0.03 ± 0.0000	0.02 ± 0.0000	0.03 ± 0.0001
	16K, 4	0.03 ± 0.0002	0.03 ± 0.0002	0.02 ± 0.0001	0.03 ± 0.0001	0.02 ± 0.0001	0.03 ± 0.0001
fft	2K, 2	0.02 ± 0.0002	0.02 ± 0.0002	0.04 ± 0.0003	0.02 ± 0.0004	0.03 ± 0.0001	0.03 ± 0.0004
	4K, 2	0.01 ± 0.0010	0.01 ± 0.0010	0.01 ± 0.0005	0.02 ± 0.0014	0.01 ± 0.0003	0.02 ± 0.0004
	8K, 2	0.01 ± 0.0000	0.01 ± 0.0005	0.01 ± 0.0001	0.01 ± 0.0004	0.01 ± 0.0002	0.01 ± 0.0008
	16K, 2	0.01 ± 0.0000	0.01 ± 0.0000	0.01 ± 0.0002	0.01 ± 0.0004	0.01 ± 0.0001	0.01 ± 0.0003
	4K, 4	0.01 ± 0.0004	0.01 ± 0.0000	0.01 ± 0.0002	0.01 ± 0.0001	0.01 ± 0.0002	0.01 ± 0.0003
	8K, 4	0.01 ± 0.0001	0.01 ± 0.0001	0.01 ± 0.0001	0.01 ± 0.0005	0.01 ± 0.0001	0.01 ± 0.0002
	16K, 4	0.01 ± 0.0001	0.01 ± 0.0000	0.01 ± 0.0001	0.01 ± 0.0003	0.01 ± 0.0001	0.01 ± 0.0003

Table 5.5: Dcache write miss rate results for 8 benchmark applications.

App	size, assoc.	os + app		os + 2 apps		os + 4 apps	
		os+app miss rate	app miss rate	os+2apps miss rate	app miss rate	os+4apps miss rate	app miss rate
basic-math	2K, 2	0.01 ± 0.0001	0.01 ± 0.0005	—	—	—	—
	4K, 2	0.01 ± 0.0003	0.00 ± 0.0003	—	—	—	—
	8K, 2	0.00 ± 0.0003	0.00 ± 0.0004	—	—	—	—
	16K, 2	0.00 ± 0.0000	0.00 ± 0.0001	—	—	—	—
	4K, 4	0.00 ± 0.0001	0.00 ± 0.0001	—	—	—	—
	8K, 4	0.00 ± 0.0000	0.00 ± 0.0000	—	—	—	—
	16K, 4	0.00 ± 0.0001	0.00 ± 0.0001	—	—	—	—
reed_enc	2K, 2	0.02 ± 0.0003	0.01 ± 0.0003	—	—	—	—
	4K, 2	0.01 ± 0.0005	0.01 ± 0.0005	—	—	—	—
	8K, 2	0.01 ± 0.0002	0.01 ± 0.0000	—	—	—	—
	16K, 2	0.01 ± 0.0001	0.01 ± 0.0001	—	—	—	—
	4K, 4	0.01 ± 0.0000	0.01 ± 0.0000	—	—	—	—
	8K, 4	0.01 ± 0.0001	0.01 ± 0.0000	—	—	—	—
	16K, 4	0.01 ± 0.0000	0.01 ± 0.0000	—	—	—	—
drr	2K, 2	0.04 ± 0.0001	0.04 ± 0.0001	0.04 ± 0.0001	0.04 ± 0.0004	—	—
	4K, 2	0.03 ± 0.0004	0.03 ± 0.0004	0.04 ± 0.0001	0.03 ± 0.0003	—	—
	8K, 2	0.03 ± 0.0004	0.03 ± 0.0001	0.03 ± 0.0001	0.03 ± 0.0002	—	—
	16K, 2	0.03 ± 0.0007	0.03 ± 0.0001	0.03 ± 0.0001	0.03 ± 0.0002	—	—
	4K, 4	0.03 ± 0.0001	0.03 ± 0.0000	0.03 ± 0.0001	0.03 ± 0.0001	—	—
	8K, 4	0.03 ± 0.0001	0.03 ± 0.0001	0.03 ± 0.0000	0.03 ± 0.0002	—	—
	16K, 4	0.03 ± 0.0001	0.03 ± 0.0001	0.03 ± 0.0001	0.03 ± 0.0002	—	—
frag	2K, 2	0.04 ± 0.0001	0.04 ± 0.0004	0.02 ± 0.0002	0.04 ± 0.0008	—	—
	4K, 2	0.04 ± 0.0014	0.04 ± 0.0000	0.02 ± 0.0008	0.04 ± 0.0019	—	—
	8K, 2	0.03 ± 0.0006	0.03 ± 0.0004	0.02 ± 0.0000	0.03 ± 0.0004	—	—
	16K, 2	0.03 ± 0.0000	0.03 ± 0.0003	0.02 ± 0.0000	0.03 ± 0.0002	—	—
	4K, 4	0.03 ± 0.0001	0.03 ± 0.0001	0.02 ± 0.0000	0.03 ± 0.0001	—	—
	8K, 4	0.03 ± 0.0000	0.03 ± 0.0000	0.02 ± 0.0000	0.03 ± 0.0002	—	—
	16K, 4	0.03 ± 0.0000	0.03 ± 0.0001	0.02 ± 0.0000	0.03 ± 0.0002	—	—
reed_dec	2K, 2	0.01 ± 0.0002	0.01 ± 0.0003	0.01 ± 0.0001	0.01 ± 0.0001	—	—
	4K, 2	0.01 ± 0.0004	0.01 ± 0.0001	0.01 ± 0.0002	0.01 ± 0.0005	—	—
	8K, 2	0.01 ± 0.0002	0.01 ± 0.0000	0.01 ± 0.0001	0.01 ± 0.0002	—	—
	16K, 2	0.01 ± 0.0001	0.01 ± 0.0001	0.01 ± 0.0000	0.01 ± 0.0000	—	—
	4K, 4	0.01 ± 0.0000	0.01 ± 0.0000	0.01 ± 0.0000	0.01 ± 0.0000	—	—
	8K, 4	0.01 ± 0.0001	0.01 ± 0.0001	0.01 ± 0.0000	0.01 ± 0.0001	—	—
	16K, 4	0.01 ± 0.0000	0.01 ± 0.0000	0.01 ± 0.0000	0.01 ± 0.0001	—	—
sha	2K, 2	0.01 ± 0.0005	0.01 ± 0.0004	0.01 ± 0.0002	0.00 ± 0.0002	—	—
	4K, 2	0.00 ± 0.0008	0.00 ± 0.0000	0.00 ± 0.0007	0.00 ± 0.0009	—	—
	8K, 2	0.00 ± 0.0002	0.00 ± 0.0000	0.00 ± 0.0003	0.00 ± 0.0000	—	—
	16K, 2	0.00 ± 0.0000	0.00 ± 0.0001	0.00 ± 0.0000	0.00 ± 0.0000	—	—
	4K, 4	0.00 ± 0.0002	0.00 ± 0.0003	0.00 ± 0.0002	0.00 ± 0.0002	—	—
	8K, 4	0.00 ± 0.0002	0.00 ± 0.0000	0.00 ± 0.0001	0.00 ± 0.0001	—	—
	16K, 4	0.00 ± 0.0000	0.00 ± 0.0000	0.00 ± 0.0000	0.00 ± 0.0001	—	—
blastn	2K, 2	0.00 ± 0.0000	0.00 ± 0.0000	0.01 ± 0.0001	0.00 ± 0.0001	0.02 ± 0.0003	0.00 ± 0.0001
	4K, 2	0.00 ± 0.0001	0.00 ± 0.0001	0.00 ± 0.0001	0.00 ± 0.0001	0.01 ± 0.0001	0.00 ± 0.0000
	8K, 2	0.00 ± 0.0002	0.00 ± 0.0000	0.00 ± 0.0001	0.00 ± 0.0000	0.01 ± 0.0001	0.00 ± 0.0000
	16K, 2	0.00 ± 0.0000	0.00 ± 0.0000	0.00 ± 0.0000	0.00 ± 0.0001	0.01 ± 0.0000	0.00 ± 0.0000
	4K, 4	0.00 ± 0.0001	0.00 ± 0.0000	0.00 ± 0.0001	0.00 ± 0.0001	0.01 ± 0.0001	0.00 ± 0.0001
	8K, 4	0.00 ± 0.0001	0.00 ± 0.0001	0.00 ± 0.0000	0.00 ± 0.0000	0.01 ± 0.0000	0.00 ± 0.0000
	16K, 4	0.00 ± 0.0000	0.00 ± 0.0000	0.00 ± 0.0000	0.00 ± 0.0000	0.01 ± 0.0000	0.00 ± 0.0000
fft	2K, 2	0.01 ± 0.0005	0.01 ± 0.0005	0.01 ± 0.0007	0.01 ± 0.0001	0.01 ± 0.0001	0.01 ± 0.0004
	4K, 2	0.00 ± 0.0005	0.00 ± 0.0005	0.01 ± 0.0006	0.01 ± 0.0013	0.01 ± 0.0002	0.01 ± 0.0011
	8K, 2	0.00 ± 0.0001	0.00 ± 0.0000	0.00 ± 0.0001	0.00 ± 0.0003	0.00 ± 0.0001	0.00 ± 0.0004
	16K, 2	0.00 ± 0.0000	0.00 ± 0.0000	0.00 ± 0.0001	0.00 ± 0.0001	0.00 ± 0.0000	0.00 ± 0.0002
	4K, 4	0.00 ± 0.0003	0.00 ± 0.0000	0.00 ± 0.0002	0.00 ± 0.0000	0.01 ± 0.0001	0.00 ± 0.0001
	8K, 4	0.00 ± 0.0001	0.00 ± 0.0001	0.00 ± 0.0001	0.00 ± 0.0003	0.00 ± 0.0001	0.00 ± 0.0000
	16K, 4	0.00 ± 0.0000	0.00 ± 0.0000	0.00 ± 0.0000	0.00 ± 0.0001	0.00 ± 0.0000	0.00 ± 0.0001

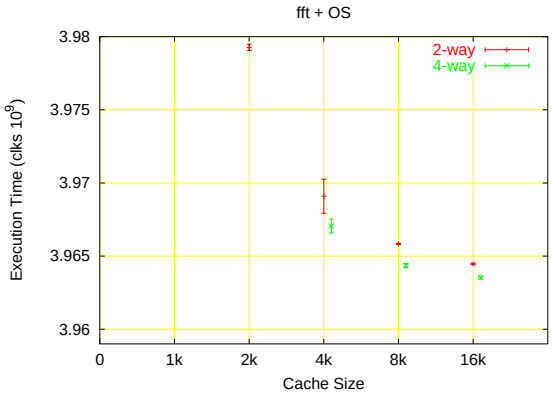


Figure 5.9: Execution time (in billions of clock cycles) for total of *fft* plus the OS with no other competing application. Various dcache configurations are shown.

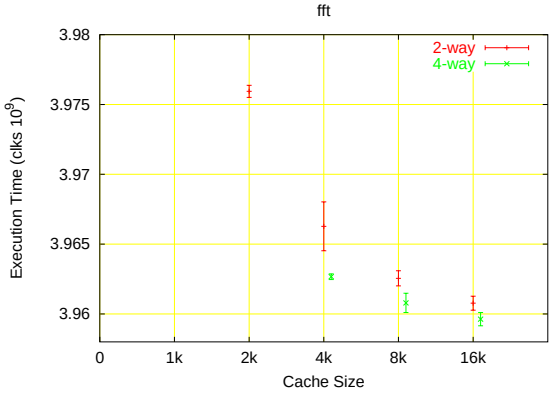


Figure 5.10: Execution time (in billions of clock cycles) for *fft* running on OS with *reed_enc* as a competing application. Various dcache configurations are shown.

With *drr*, the presence of an additional competing application increases the write miss rate for the dcache for a 2 KB and a 16 KB cache size, but does not significantly impact the dcache write miss rate for 4 KB and 8 KB cache sizes. This is shown in Figures 5.12 and 5.13.

With *frag*, the presence of the competing application doesn't have a significant impact on the mean dcache read miss rates, but dramatically increases the variability across individual runs, especially near the knee of the curve for the 2-way associative cache. This is shown in Figures 5.14 and 5.15.

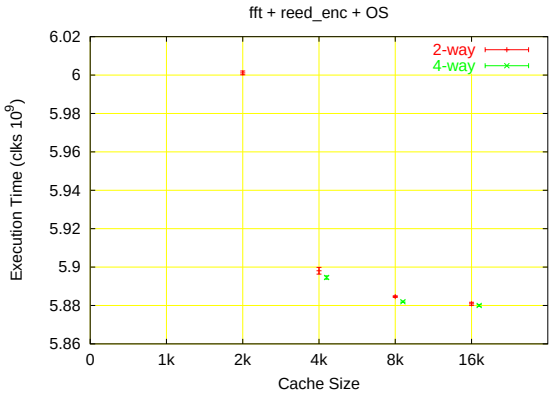


Figure 5.11: Execution time (in billions of clock cycles) for total of *fft* plus *reed_enc* plus the OS. Various dcache configurations are shown.

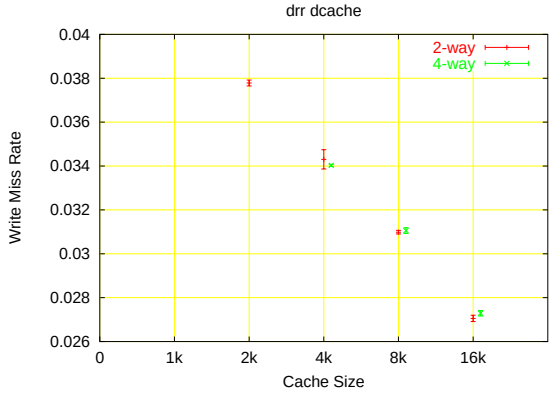


Figure 5.12: Dcache write miss rate for *drr* running on OS with no other competing application.

5.7 Rare Events

While the above section emphasizes the exploration of an architectural design space, we next concentrate on the need to investigate rare events. This is first motivated by a specific case study, which is followed by an illustration of the use of the statistics module to perform this type of investigation. The case study and experiments that follow represent collaborative work reported with others [55].

Motivation

In this section we present a case study that motivates the techniques that will follow. Real-time applications often involve tasks that must be scheduled so as to know they will

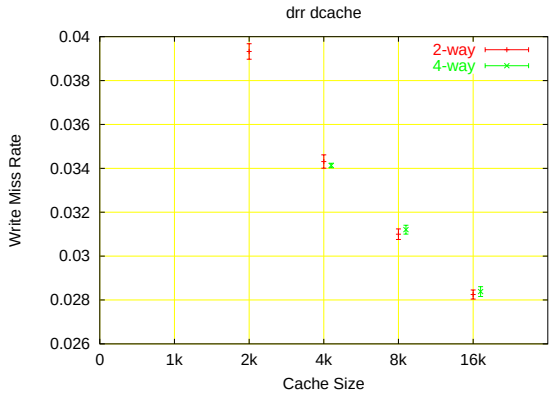


Figure 5.13: Dcache write miss rate for *drr* with one competing application (*frag*).

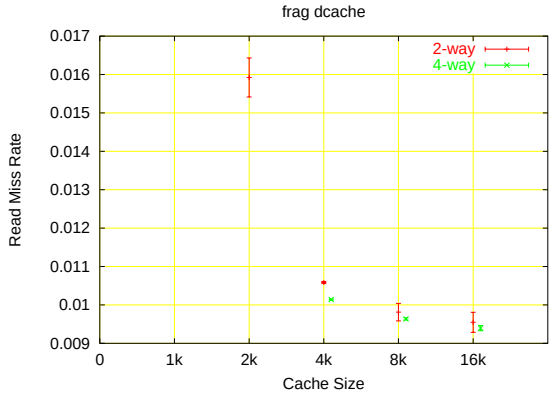


Figure 5.14: Dcache read miss rate for *frag* running on OS with no other competing application.

complete within a given time frame. The analysis [85] required to prove that deadlines are met necessitates knowing the cost (time) of the code that must be scheduled within the tasks, as well as the tasks' deadlines and periodicity. Static scheduling analysis [84] requires a worst-case bound on the tasks' costs, so that scheduling can account for worst-case behavior to ensure that deadlines are met.

Consider a simple hash table, into which data will be inserted and retrieved by a real-time application. The cost of a "`put`" into the table is typically quite small. However, most implementations test the capacity of a hash table during a `put` operation; if the table should be resized, then the table's reorganization is accomplished during the `put`. Thus, the cost of some `put` operations can be much worse than the `put`'s average cost.

Real-time implementations of hash tables [42] amortize the excessive cost over all `put` operations, so that the hash table adapts slightly at every `put` and the cost of each `put` is

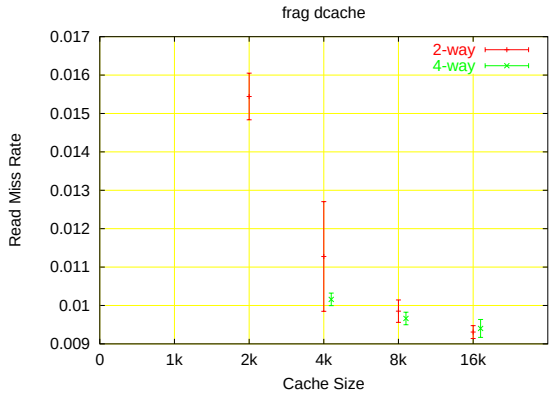


Figure 5.15: Dcache read miss rate for *frag* with one competing application (*reed_dec*).

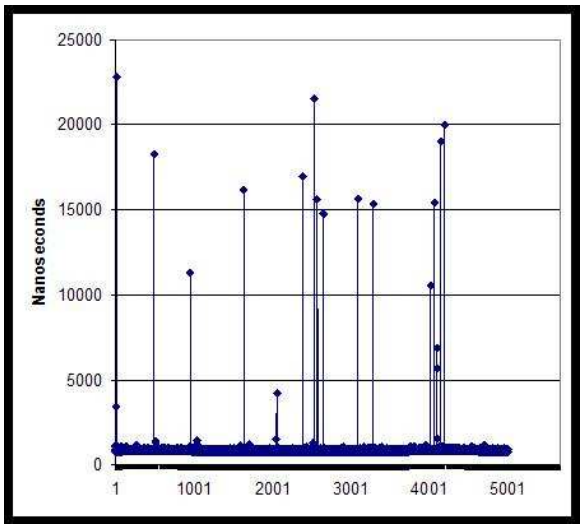


Figure 5.16: Observed execution times for a real-time HashTable `put` operation

theoretically the same. Execution of such an implementation is shown in Figure 5.16 for ∼5,000 `put` operations. The data was collected under Solaris 8 on a Sparc 5 with the application running at the highest priority in real-time mode; no other task supposedly could pre-empt the application. Note that almost every `put` operation is within 980 nanoseconds. Occasionally, a `put` is observed to be significantly more expensive and can take as much as ∼23 microseconds.

Following are results obtained via classical approaches for determining the source of the excessive execution times:

- The code can be instrumented within the `put` to determine which statement or segment of code is responsible for the observed times.

  Standard tools do not instrument at such a level, but manual insertion of timers revealed that the problem could not be attributed to any one section of the code.

- Cache and other such features can be disabled or made useless to determine if the problem arises at a micro-architectural level.

  With the cache effectively disabled, the execution times were uniformly worse (as expected) but there were still occasional `put` operations whose times greatly exceeded the average case.

Based on the unpredictability of the worst-case observed execution times, it is clear that the spikes in Figure 5.16 are due to activity occurring in other processes or threads that cause the CPU to be taken from the real-time task. In theory, such activity should not occur: the application executed in "real-time" mode on an operating system (Solaris) that supposedly supports such applications, and all pages were locked into memory.

Because the events in Figure 5.16 occur rarely and seemingly at random, sampling methods are likely to miss the moment of bad behavior. Moreover, the problem exists between two separate address spaces and between processes that may not have permission to inspect each other. Finally, the code segment of interest is relatively brief; any method for finding the source of the bad behavior must be sufficiently nonintrusive so as not contribute to or mask the actual bad behavior.

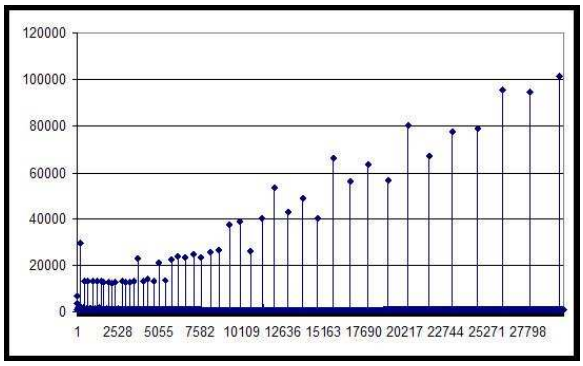


Figure 5.17: Isolated execution times for HashTable `put`

If the sources of the unpredictable behavior are located, then the application's behavior per `put` is shown in Figure 5.17. Note that the data includes data points beyond the first 5000 shown in Figure 5.16. While the times still do not reflect the desired real-time behavior, the

pattern of spikes is now much clearer and was easily resolved to a problem with the storage allocator. When more storage is necessary for expansion of the hash table, the allocator is spending nonlinear time, which contributes to poor real-time performance. By substituting the ACE allocator [103], we obtain the performance shown in Figure 5.18.

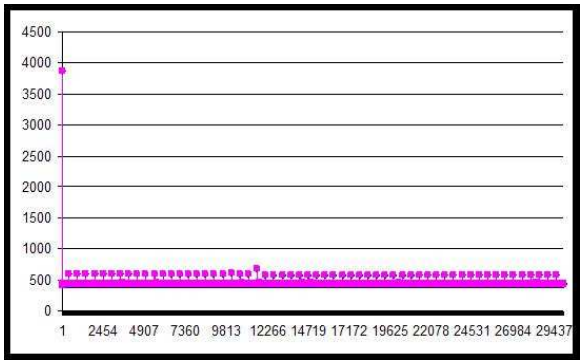


Figure 5.18: Real-time performance obtained with a better allocator

In summary, this case study illustrates the difficulties faced in obtaining an accurate picture of an application's performance when that performance is adversely affected by other processes. Standard debugging and profiling tools are unable to capture system-wide performance data at a resolution and on a scale that allows a developer to appreciate the application's behavior. Indeed, in this case, system interference with the application masked an actual problem in the application that was easily fixed (if not easily found) to obtain real-time performance.

Rare Event Experiment

To illustrate the abilities of the statistics module [54] for investigating events of the type just described, we repeatedly executed the *blastn* application 549 times and measured the total execution times shown in Figure 5.19. Note that the vast majority of the runs complete in 2.717 billion clock cycles, several runs take an additional 3 million clock cycles, but 15 runs take an additional 5.5 million clock cycles to finish. We configured the statistics module to investigate the properties of these incremental 5.5 million clock cycles.

For the rare event experiments we used both the event monitoring and the PID logging features of the statistics module. For one cache configuration (4 KB, direct-mapped data cache) we evaluated the variation in the execution time of the *blastn* application.

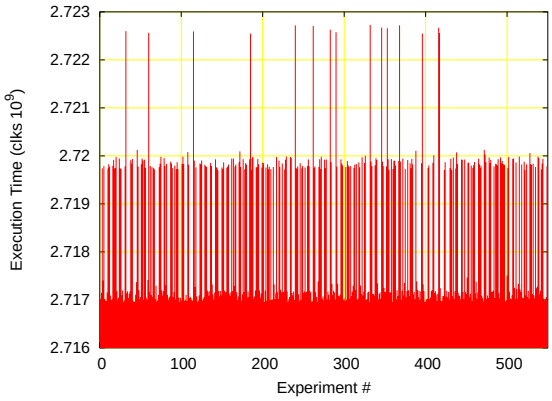


Figure 5.19: Total execution time for the BLASTN application.

Similar to the experiments described in the earlier section, we booted the OS and launched our application to run successively 549 times. Here, *blastn* is the only user application running on the system, and we nominally expect a very low variance in its execution time. For each of the runs we monitored the time spent by the processor in 8 uniform virtual address ranges from 0 to 0xFFFFFFFF. Also, as described in Section 2.5.5, we kept track of the PID changes within the application run and the time spent between changes. At the end of each run, we can examine the division of execution time between all PIDs run in the system during that window of time represented by the log.

Of the address ranges monitored, we observed execution time attributed to only 3 of the 8 ranges. Figure 5.19 shows the total execution time of the application over the 549 runs. To investigate the 15 "rare events," application executions taking an additional 5.5 million clock cycles, we start by examining the activity in the 3 active address ranges. These are plotted in Figures 5.20, 5.21, and 5.22. We continue the investigation by examining the PID log for several individual runs. Figure 5.6 shows this information. Runs 32 and 240 are two of the long runs, and run 50 represents a typical run.

Examination of this data leads us to an important conclusion, the causes of the rare events are not all the same. In run 32, approximately 2 million additional clock cycles can be attributed to the application itself (a fact that is also true of run 66), and the remaining excess clock cycles are in the kernel (PID 0). For run 240, virtually all of the additional clock cycles are in the kernel, not the application. Furthermore, the distribution of excess clock cycles on the two long runs differs in address range as well. About 2 million additional clocks are present in the low address range for run 32, with the remaining 3.5 million clock cycles in the highest address range (which includes the idle loop). For run 240, all of the additional clock cycles are in the high addresses.

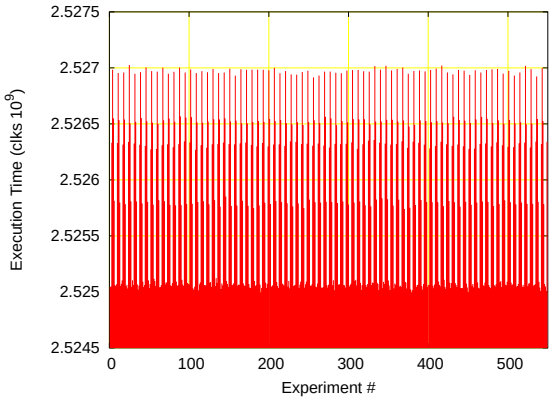


Figure 5.20: Execution time spent in address range 0 to 0x1FFFFFFF for multiple runs of *blastn*.

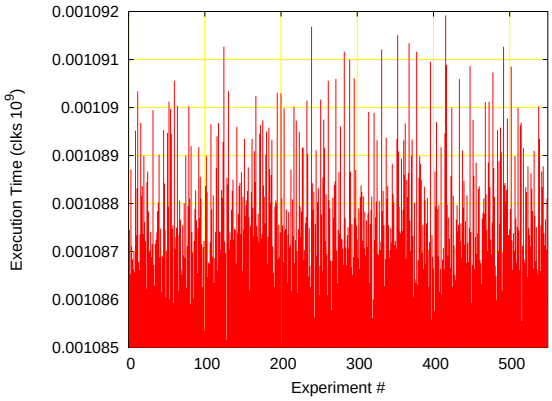


Figure 5.21: Execution time spent in address range 0x40000000 to 0x5FFFFFFF for multiple runs of *blastn*.

Given the above information, it is next reasonable to parameterize the statistics module for a more focused address range investigation, ultimately identifying the specific methods (whether in the kernel or the application) that account for the excess execution time.

5.8 Chapter Summary

We have briefed the user of the measurement infrastructure on the Liquid architecture system and its extremely state-of-the-art characteristics. We have shown here that the inclusion of the OS effect to estimated the performance of an architecture not only adds to the validity of the results presented, but could crucial to the decision about the architecture.

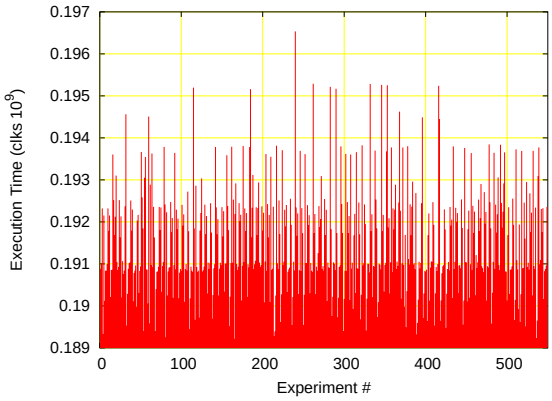


Figure 5.22: Execution time spent in address range 0xE0000000 to 0xFFFFFFFF for multiple runs of *blastn*.

Table 5.6: Execution time split across PIDS.

Experiment number	Process ID					
	0	2	3	8	23	app
32	70,901,766	17538	3,451,566	267,588	4,099,036	2,643,665,649
50	67,517,785	18537	3,337,118	267,970	4,096,603	2,641,632,437
66	67,884,519	16475	3,316,209	268,377	4,160,005	2,643,921,380
240	73,083,068	22986	3,344,711	272,197	4,100,255	2,641,638,131

From our experience with evaluating architecture performance for embedded systems, we see that difference between numbers which include/exclude the OS effect are very much similar. However, this holds true only when the application itself is run on an OS. Standalone executions and executions under operating system can be significantly different.

The addition of the other applications to the mix has an "application-specific" impact on the benchmarking. For any application where the "other" competing applications do not significantly compete for cache resources, we saw that their inclusion were similar to the mean results when these applications were absent. We did, however, see that the inclusion of these competing application affected the variability of the performance measures for the primary application.

An important contribution of this chapter in the realm of performance modeling is the discussion on the rare-events that can affect the the execution time of the applications. Understanding the cause and hence the potential elimination of these become extremely important in the design of real-time system with hard deadlines.

The sheer quantity of data collected, with this level of quality, could not have been accomplished in any reasonable time with simulation. Emulation using FPGAs enabled a wide coverage of the available paramemter space.

Chapter 6

Mercury BLASTN

6.1 Introduction

Computational search through large databases of DNA and protein sequence is a fundamental tool of modern molecular biology. Rapid advances in the speed and cost-effectiveness of DNA sequencing have led to an explosion in the rate at which new sequences, including entire mammalian genomes [119], are being generated. To understand the function and evolutionary history of an organism, biologists now seek to identify discrete biologically meaningful features in its genome sequence. A powerful approach to identify such features is *comparative annotation*, in which a *query sequence*, such as new genome, is compared to a large database of known biosequences. Database sequences exhibiting high similarity to the query, as measured by string edit distance [107], are hypothesized to derive from the same ancestral sequence as the query and in many cases to have the same biological function.

BLAST, the **B**asic **L**ocal **A**lignment **S**earch **T**ool [4], is the most widely used software for rapidly comparing a query sequence to a biosequence database. Although BLAST's algorithms are highly optimized for efficient similarity search, growth in the databases it uses is outpacing speed improvements in general-purpose computing hardware. For example, the National Center for Biological Information (NCBI) Genbank database grew exponentially between 1992 and 2003 with a doubling time of 12–16 months [87]. The problem is particularly acute for BLASTN, the BLAST variant used to compare DNA sequences, because each new genome sequenced from animals or higher plants produces between $\mathbf{10^8}$ and $\mathbf{10^{10}}$ bytes of new DNA sequence.

6.1.1 Solution Strategies

One obvious approach to runaway growth in biosequence databases has been to distribute BLAST searches across multiple computers, each responsible for searching only part of a database. This approach requires both a substantial hardware investment and the ability to coordinate a search across processors. An alternate approach that makes more parsimonious use of hardware is to build a specialized BLAST accelerator. By using an application-specific architecture and exploiting the high I/O bandwidth of modern storage systems, an accelerator can execute the BLAST algorithms much faster than a general-purpose CPU.

The *Mercury* system [26] is a prototype architecture that supports disk-based computation at very high data rates using reconfigurable hardware. Computing applications historically have been coded using the following paradigm: read input data into main memory with explicit I/O calls, compute on that data writing results back to main memory, and send the output from main memory with explicit I/O calls. In contrast, the Mercury system is built around the concept of continuous data flow. Data from disk(s) flow into the computational resource(s); one or more functions (often physically pipelined) are performed on the data; and the results flow to the intended destination. As the computational resources include reconfigurable hardware, application deployment requires hardware/software codesign. The Mercury system builds upon the work of Reidel [100] (active disks), Dally [30] (stream processors), and a host of work developed in the reconfigurable computing community.

The following sections describe the re-engineering of the original BLASTN application for effective deployment on the Mercury system. We examine the existing application to explore its performance properties, propose a novel algorithmic optimization and evaluate the performance potential of the overall application running on the Mercury system. This work is colloborative with others and reported in [69].

6.2 System Architecture

The Mercury system (Figure 6.1) contains reconfigurable logic, associated with the disk controller, that provides computing capability in close proximity to the data flowing off the disk drive(s). Initial processing of the data occurs locally at the disk, prior to delivery to the processor. The reconfigurable logic is implemented via a Field-Programmable Gate Array (FPGA).

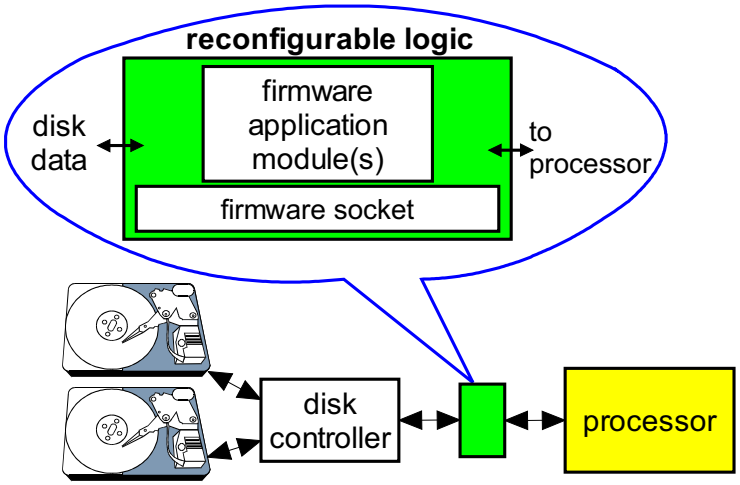


Figure 6.1: Mercury system architecture

Application functionality is divided into two parts executing on the FPGA and the main processor, respectively. Application deployment therefore has the classic components of a hardware/software codesign problem, with the need to map application elements to multiple computational resources (i.e., FPGA and processor). A unique aspect of the Mercury system is that it was designed specifically to work well with high-volume data applications. The computational resource that is best suited to simpler, repetitive operations on a large data set is positioned closer to the data, while the resource best suited to more complex operations on smaller data volumes is (logically) farther away from the data.

The application set that is well matched to the Mercury system architecture is a pipeline that consumes a high data volume at its input, reduces that data volume to a smaller set, and performs higher-level processing on this smaller set. Our previous work has illustrated the use of the system for a number of text search applications [22, 24, 25, 38, 120, 128]. BLASTN has properties that fit well with the Mercury system's capabilities.

While Figure 6.1 illustrates our vision of the system architecture, our prototyping work has so far been limited to a series of implementations that are progressively closer to, but do not yet exactly match, the architecture depicted in the figure. Our earliest prototypes used ATA drives [120, 128] and were severely speed-limited by the disks. Our most recent prototypes are built using a set of 15,000 rpm Ultra320 SCSI drives organized in a RAID-0 configuration. On this configuration, we have demonstrated sustained read performance of over 800 MB/sec for continuous 500 GB reads. The prototype FPGA infrastructure is currently parallel to the disk controller on the I/O bus, which limits throughput into the

FPGA. We have, however, demonstrated sustained data throughput of over 700 MB/sec from the disk array into the FPGA [24, 25].

In what follows, we refer to computations deployed in the FPGA as *firmware* and computations deployed on the processor as *software*. To facilitate the deployment of applications on the FPGA, we have developed a firmware socket interface that provides a consistent environment for the development of firmware application modules. Data from the disk array is delivered to the FPGA via the firmware socket, while outbound data from the reconfigurable logic is delivered into the main memory of the processor for access by software.

The current prototype system uses a Xilinx Virtex-II 6000 series FPGA, which provides 8,448 Configurable Logic Blocks (CLBs), 144 18 Kbit Block RAMs (BRAMs), 144 $\mathbf{18} \times \mathbf{18}$ bit multipliers, and 1104 I/O pins. Each CLB is comprised of 4 slices and each slice provides two 4 input lookup tables (LUTs) for a total of 67,584 LUTs on chip.

6.3 Description of NCBI BLASTN

This section describes the open-source version of BLASTN distributed by the National Center for Biological Information (NCBI) and used by numerous biological research labs. As shown in Figure 2.6, BLASTN is functionally organized as a pipeline with three stages: word matching, ungapped extension, and gapped extension. The inputs to this pipeline are a query sequence and a database, each consisting of a string of DNA *bases*. A base is typically one of $\boldsymbol{\{A, C, G, T\}}$, but other characters (a total of 15) are used to denote uncertainty about or special properties of certain bases. DNA sequences, including these special characters, can be represented using four bits per base; however, to minimize storage and I/O bandwidth, NCBI BLASTN stores its database using only two bits per base.

Each stage of BLASTN's pipeline implements progressively more sophisticated and more expensive computations to identify biologically meaningful similarities between query and database. In stage 1, BLASTN discovers *word matches* between query and database. A word match is a string of some fixed length $\boldsymbol{w}$ (hereafter called a "$\boldsymbol{w}$-mer") that occurs in both query and database. Significantly similar sequences usually share a $\boldsymbol{w}$-mer match for $\boldsymbol{w \approx 11}$, though such matches also occur frequently by chance between unrelated sequences. Each word match is therefore filtered through stage 2, which tries to extend it into an *ungapped alignment* between query and database. An ungapped alignment may contain mismatched bases but consists primarily of matching base pairs. Ungapped alignments with too few matching base pairs are discarded, while the remainder are further filtered

through stage 3, which extends them into *gapped alignments* that permit both mismatches and localized insertion or deletion of bases. In the final operation following the end of stage 3, gapped alignments with sufficiently many matching base pairs are reported to the user. The detailed algorithm for BLASTN stage 2 is described in [73], while that for stage 3 is a variant of the standard Smith-Waterman dynamic programming algorithm [107].

Although each stage of BLASTN is more compute-intensive than the last, each stage also discards a substantial fraction of its inputs. The volume of data that is processed at each stage therefore gradually decreases. Table 6.1 quantifies the data reduction at each stage of the pipeline[1]. The match rate, $\boldsymbol{p_i}$, represents the probability that an output from stage $\boldsymbol{i}$ is generated from an individual input to that stage. For stage 1, $\boldsymbol{p_1}$ measures the number of matches per DNA base read from the database. Stages 1 and 2 are highly effective at reducing the data volume to the next stage. Note that, as the query length increases, the rate at which matches are output from stage 1 into stage 2 also increases, raising the workload for stage 2.

In the performance predictions that follow, we will consider the throughput of individual stages of the pipeline as well as the throughput of the entire pipeline. To make throughputs comparable, they are normalized to be in units of input bases per second from the database. When executing on a single computational resource (i.e., software running on a single processor), the average compute time per input base can be expressed as $\boldsymbol{t_1 + p_1 t_2 + p_1 p_2 t_3}$, where $\boldsymbol{t_i}$ is the compute time for stage $\boldsymbol{i}$ for each input item (base, match, or alignment) to stage $\boldsymbol{i}$. The normalized throughput is then $\boldsymbol{Tput = 1/(t_1 + p_1 t_2 + p_1 p_2 t_3)}$.

Table 6.1: Match rates $\boldsymbol{p}$ across pipeline stages

Query Size (bases)	**Stage 1** ($\boldsymbol{p_1}$)	**Stage 2** ($\boldsymbol{p_2}$)	**Stage 3** ($\boldsymbol{p_3}$)
10 K	0.00858	0.0000550	0.320
25 K	0.0205	0.0000619	0.141
50 K	0.0411	0.0000189	0.194
100 K	0.0841	0.0000174	0.175
1 M	0.851	0.0000172	0.096

6.3.1 Details of BLASTN Stage 1

To facilitate later comparison with our firmware design, we now briefly describe the implementation of NCBI BLASTN's stage 1. This implementation uses a default word match length $\boldsymbol{w = 11}$. Due to the speed advantages of comparing complete bytes at a time,

[1]Reduction measurements for NCBI BLASTN were taken in the same experiments used to generate the timings of Section 6.3.2

discovery of 11-mer matches is implemented in two phases. BLASTN first checks two complete bytes of the database, containing 8 bases, against a lookup table constructed from the query. Only two-byte words occurring on full byte boundaries are checked. If the query contains the same 8-base word, BLASTN tries to extend this 8-base match to 11 bases by seeking additional matching residues on either side.

Two 11-mer matches that occur close to each other in both the query and database are likely to have arisen from the same underlying biological similarity. To avoid having later stages expend the effort to discover this similarity twice, NCBI BLASTN implements a *redundancy elimination* filter at the end of stage 2. The filter checks whether each new 11-mer match overlaps or is close to a previously observed match. If so, the new match is suppressed, since it would likely lead only to rediscovery of any feature found by the previous match.

6.3.2 Performance of NCBI BLASTN

To quantify the performance of NCBI BLASTN on a general-purpose CPU, we measured its execution time with default parameters on a 2.8 GHz Pentium 4 PC, with an L2 cache size of 512 KB and 1 GB of RAM, running Linux. We compared a database containing the mouse genome (1.16 Gbases after removing repetitive sequence) to queries of various lengths selected at random from the human genome. CPU time was measured separately for each of the three pipeline stages.

The length of a typical query sequence in BLASTN is application-dependent. For example, a short DNA sequence obtained in a single lab experiment may be only a few kilobases, while in genome-to-genome comparison, a query (one of the genomes) may be billions of bases long. A BLAST implementation should support the largest computationally feasible query length, both to accommodate long individual queries and to support the optimization of "query packing," in which multiple short queries are concatenated and processed in a single pass over the database. Conversely, queries longer than the maximum feasible length may be broken into pieces, each of which is processed in a separate pass.

In our experiments, we tested queries of 10 Kbases, 25 Kbases, 50 Kbases, 100 Kbases, and 1 Mbase, both to simulate different applications of BLASTN and to assess the impact of query length on the performance of our firmware implementation. One megabase is a reasonable upper bound on query size for NCBI BLASTN with standard parameters, since it generates 11-mer word matches by chance alone at a rate approaching one match for every

base read from the database. Timings were averaged over at least 20 queries for each length, and each query's running time was averaged over three identical runs of BLASTN. It should be noted that, given a query sequence of length n, BLASTN compares the database to both the sequence and its DNA reverse-complement, effectively doubling the query length. The performance numbers reported in this section and throughout the rest of the dissertation reflect such "double-stranded" queries.

Table 6.2 gives the distribution of times spent in each stage of NCBI BLASTN for various query sizes. Times are given with 95% confidence intervals. Time spent in stage 1 dominated that spent in later pipeline stages, while time spent in stage 3 was almost negligible. Although later stages are computationally more intensive, each stage is such an efficient filter that it discards most of its input, leaving later stages with comparatively little work.

Table 6.2: Percentage of pipeline time spent in each stage of NCBI BLASTN

Query Size (bases)	**Stage 1**	**Stage 2**	**Stage 3**
10 K	86.53±1.33%	13.24±1.99%	0.23±0.02%
25 K	83.89±2.56%	15.88±4.40%	0.22±0.04%
50 K	82.63±2.94%	17.28±4.96%	0.09±0.01%
100 K	83.35±1.28%	16.58±2.17%	0.08±0.01%
1 M	85.39±3.34%	14.68±5.24%	0.03±0.01%

From the measured running times of our experiments and the size of the mouse genome database, we computed the throughput (in Mbases from the database per second) achieved by NCBI BLASTN's pipeline for varying query sizes. The results are shown in the first row of Table 6.3. Throughput depends strongly on query length. To explain this observation, we used the predicted filtering efficiencies p_i for each pipeline stage and the distribution of running times by stage to estimate the average time spent to process each base in stage 1, each word match in stage 2, and each ungapped alignment in stage 3. These results are shown in the remaining rows of the table. While the overhead per input remains constant for stage 2 and actually decreases for stage 3, the cost per base in stage 1 grows linearly with query length. This cost growth derives from the linear increase in the expected number of matches per base that occur purely by chance, in the absence of any meaningful similarity.

Table 6.3: Summary of performance results for software runs of NCBI BLASTN

Query Size	**10 Kbases**	**25 Kbases**	**50 Kbases**	**100 Kbases**	**1 Mbase**	**Units**
Throughput	67.0	29.2	14.9	8.76	0.648	Mbases/sec
Stage 1 (time per base, t_1)	0.0129	0.0287	0.0553	0.0951	1.32	μsec/base
Stage 2 (time per match, t_2)	0.231	0.265	0.281	0.225	0.264	μsec/match
Stage 3 (t_3)	71.3	60.4	81.8	58.9	34.4	μsec/alignment

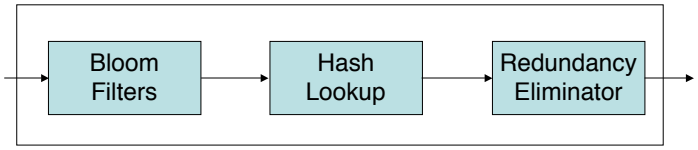


Figure 6.2: Division of BLAST stage 1 (word matching) into 3 substages (1a: Bloom Filters, 1b: Hash Lookup, and 1c: Redundancy Eliminator)

The empirical performance of NCBI BLASTN's pipeline demonstrates that stage 1 is a performance bottleneck and therefore the first target for speedup in firmware.

6.4 Firmware Implementation of Stage 1

Our firmware implementation of stage 1 reflects the overall functionality of stage 1 in NCBI BLASTN but makes no attempt to implement this functionality using the same mechanisms. Our design decomposes stage 1 into 3 substages (Figure 6.2). The initial substage implements a prefilter using Bloom filters; the middle substage determines the query position of w-mers in the database that successfully pass through the Bloom filters (using hashing); and the final substage performs redundancy elimination.

6.4.1 Prefiltering using Bloom Filters

A Bloom filter [13] is a probabilistic algorithm to quickly test membership in a large set using multiple hash functions into a single array of bits. Bloom filters find many uses in networking and other applications [32]. Figure 6.3 illustrates a typical Bloom filter datapath. Programming the filter amounts to setting to '1' each of the bits of the memory locations obtained by the hash functions. Querying the Bloom filter yields a match when all the memory locations in the vector obtained from hashing the query contain '1'.

A Bloom filter yields no false negatives but does yield false positives at a rate f determined by the number of w-mers programmed into it and the length of its memory vector. The rate f can be modeled as $f = (1 - e^{-Nk/m})^k$, where N is the number of entries programmed into the filter (query size), m is the filter memory size in bits, and k is the number of hash functions. Figure 6.4 shows the false positive rate of a Bloom filter, as a function of memory size, for different query lengths. The number of hash functions, k, in Figure 6.4 is obtained as $k = \frac{m}{N} \ln(2)$. The false positive rate is obtained as $f = \frac{1}{2^k}$.

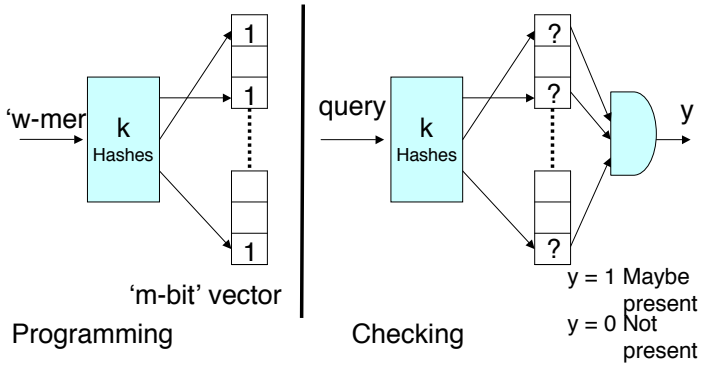


Figure 6.3: Typical Bloom filter functional diagram

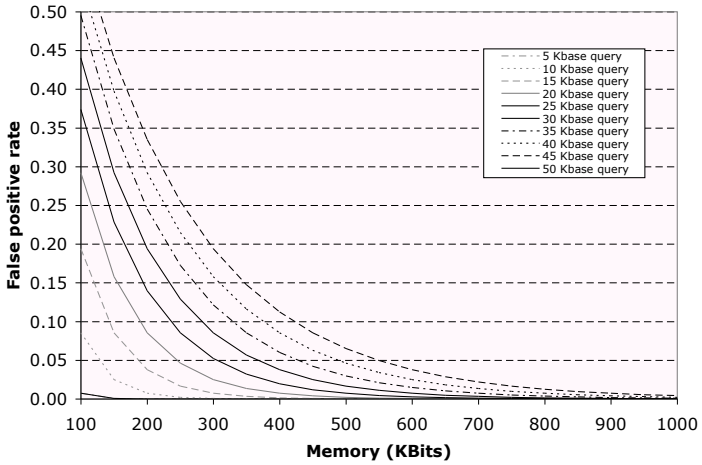


Figure 6.4: Theoretical false positive rate of a Bloom filter vs. memory size for different query lengths

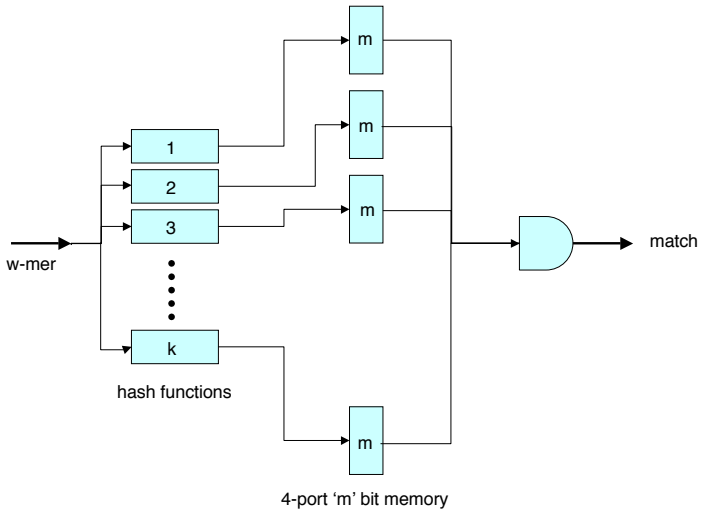


Figure 6.5: Firmware Implementation of Bloom filter using block RAMs

Bloom filters are more efficiently implemented in firmware than in software, as we can store the memory vectors on chip (using sets of block RAMs), calculate the hash functions in parallel, and look up the locations of the memory vector in parallel. However, as the number of ports on these block RAMs is finite, the hash functions are restricted to address only specific block RAMs.

6.4.2 Architecture of Bloom filters

In our implementation, each distinct memory vector made using a set of block RAMs is addressed using a unique hash function (see Figure 6.5, in which each '$\boldsymbol{m}$'-bit memory is constructed out of a set of block RAMs). The probability that a random bit in an '$\boldsymbol{m}$'-bit memory is set to 1 by a hash function is simply $\frac{\mathbf{1}}{\boldsymbol{m}}$. The probability that it is not set is therefore $\mathbf{1} - \frac{\mathbf{1}}{\boldsymbol{m}}$. Now since all $\boldsymbol{N}$ queries are programmed in each of the memories, the probability that a random bit is not set by any of the $\boldsymbol{N}$ queries is $(\mathbf{1} - \frac{\mathbf{1}}{\boldsymbol{m}})^{\boldsymbol{N}}$. Hence, the probability that it is set is $(\mathbf{1} - (\mathbf{1} - \frac{\mathbf{1}}{\boldsymbol{m}})^{\boldsymbol{N}})$. A random query will trigger a false positive if it randomly collides with set bits in all the memories. In our design, the number of memoriess equals the number of hash functions $\boldsymbol{k}$. Hence, the probability that a query triggers a match is obtained as $\boldsymbol{f} = (\mathbf{1} - (\mathbf{1} - \mathbf{1}/\boldsymbol{m})^{\boldsymbol{N}})^{\boldsymbol{k}}$, where $\boldsymbol{m}$ can also be interpreted as the address range of each hash function, $\boldsymbol{N}$ is the length of the query, and $\boldsymbol{k}$ is the number of hash functions used.

Because the Bloom filter implementation is the primary block RAM-intensive stage in our design, we dedicate 96 (about two-thirds) of the 144 block RAMs available on chip to it.

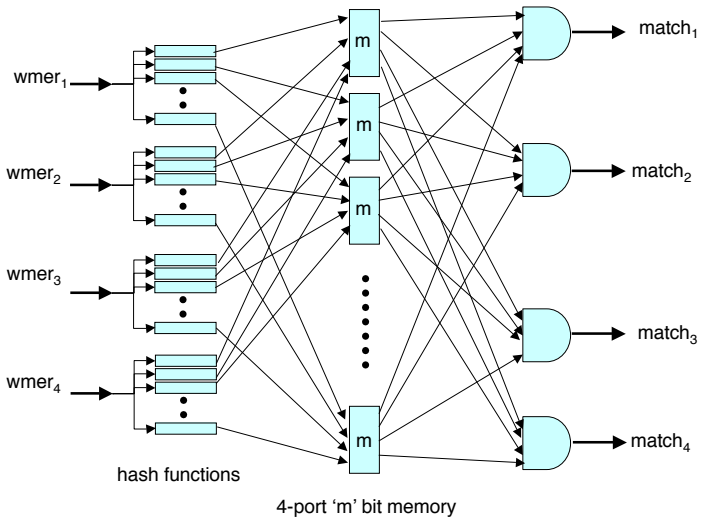


Figure 6.6: Four parallel Bloom filters

The stage is designed to consume 16 bases every clock cycle and to operate at 133 MHz (yielding a potential throughput of 2 Gbases/s). To sustain this rate, we must process 16 $\boldsymbol{w}$-mers every clock cycle and so require 16 identical copies of the Bloom filter.

We reduce the memory requirements for the Bloom filtering stage in half by using both of the ports of our dual-ported block RAMs. Furthermore, using the clock management circuitry on the FPGA, we double-clock the block RAMs and in effect make each block RAM quad-ported [72]. This is illustrated in Figure 6.6. Four copies of the 4-way parallel Bloom filters are therefore sufficient to process all 16 $\boldsymbol{w}$-mers (illustrated in Figure 6.7).

Reducing the number of Bloom filter copies needed helps to decrease the false positive rate of the queries, by dedicating more memory to each Bloom filter, or to process larger

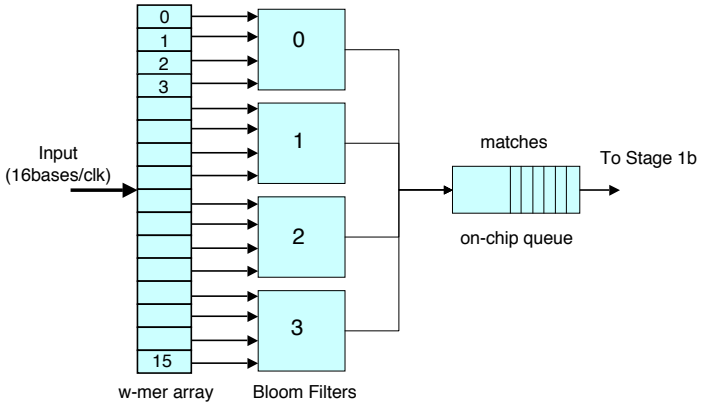


Figure 6.7: Sixteen parallel Bloom filters

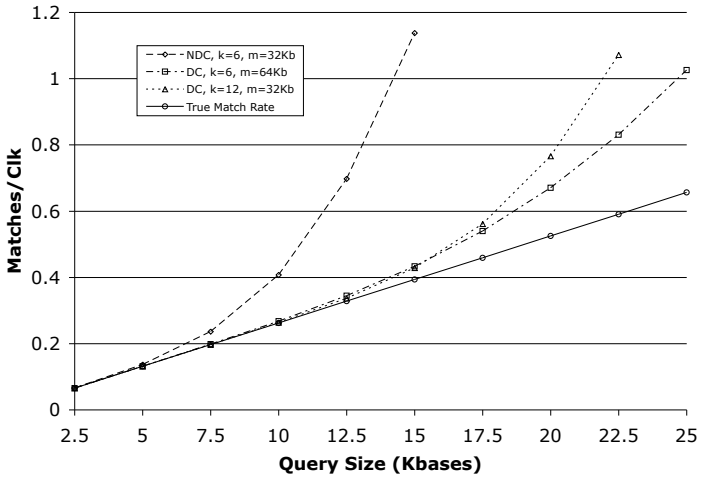


Figure 6.8: Bloom filter output match rate vs. query size, NDC: no double clocking of block RAMs, DC: double clocking of block RAMs

queries with the same false positive rate. These tradeoffs are illustrated in Figure 6.8, which shows the expected true match rate (solid line) and the overall match rate (including false positives) of stage 1a (dashed lines). This figure assumes that the input to stage 1a never stalls (i.e., 16 bases are available each clock cycle). The two double-clocking curves vary in how they utilize the additional effective memory. One doubles $\boldsymbol{k}$, while the other doubles $\boldsymbol{m}$. Doubling the clock rate of the block RAMs not only decreases the false positive rate for smaller query sizes but also helps us support larger queries than would otherwise be possible.

The maximum query size that can be supported on our prototype is partially determined by the rate of matches from stage 1a. Stage 1b (described below in Section 6.4.3) processes matches from this stage, and it is designed to support an input rate of approximately one match every clock cycle. Given an expected average input rate from the disk subsystem of 1.4 Gbases/s (700 MB/s $\times$ 2 bases/byte) and a maximum ingest rate into stage 1a of 2 Gbases/s, 25 Kbase double-stranded queries are reasonably supported.

As similarities can exist between the query and database sequences, there is a good chance that matches from stage 1a will be bursty. We maintain an on-chip queue of size 1000 $\boldsymbol{w}$-mers to accommodate such bursts. In the event that this queue fills up, for example in the case where long genes are conserved between sequences, we can either store the matches from stage 1a in off-chip DRAM or generate backpressure for the upstream stages. The performance implications of this backpressure are minimal, as assessed below.

6.4.3 Hash Lookup

The second substage of stage 1 uses a hash table to identify those $\boldsymbol{w}$-mers from the database that actually occur in the query sequence. Each such $\boldsymbol{w}$-mer must be mapped to its position or positions in the query. Note that, in contrast to NCBI BLASTN, we do not test only those $\boldsymbol{w}$-mers falling on byte boundaries – every $\boldsymbol{w}$-mer in the database is checked. The hash table is implemented in an external SRAM attached to the FPGA, since the latter's internal block RAMs are too limited in size to contain tables built from large query sequences.

The need to access a single external SRAM is a potential source of pipeline bottlenecks. Suppose that the SRAM can sustain only one read per clock cycle, which is a reasonable assumption at high FPGA clock speeds. The data reduction achieved by our Bloom filters is sufficient to ensure an average input rate to this substage of at most one $\boldsymbol{w}$-mer per clock. However, if processing a $\boldsymbol{w}$-mer were usually to entail multiple, serial accesses to the SRAM (e.g. to resolve hash collisions), we could not sustain even this modest data rate. Our design for this stage therefore seeks to dispose of the vast majority of $\boldsymbol{w}$-mers with only a single SRAM lookup.

We have developed an FPGA-friendly approach to hash table design, called *n*ear-perfect hashing [69], that empirically approximates the properties of a perfect hash for this application. Readers are directed to [69] for complete details on this hashing scheme.

6.4.4 Redundancy Filter

To avoid repeated generation of the same sequence alignment, NCBI BLASTN uses a redundancy filter to discard the $\boldsymbol{w}$-mers which fall within the range that already has been inspected by extension in stage 2. We have modified NCBI BLASTN's filter for redundant word matches to work efficiently in firmware. Readers are directed to [69] for complete details on the implementation of this component.

6.5 Performance Analysis

We assess the performance of our design in three phases. First, we assess the performance gain relative to software of stage 1 alone. Second, we assess the overall performance of a design that exploits the firmware implementation of stage 1 and continues to use software

to implement stages 2 and 3. Finally, we discuss the benefit that can be gained from a firmware implementation of stage 2 and provide performance targets for that design.

6.5.1 Word Matching (Stage 1) in Firmware

We have built several firmware prototypes of stage 1 that can consume 16 bases/clk. The latest runs at a clock rate of 133 MHz. The prototype is limited to a query size of 25 Kbases. To process queries of size greater than 25 Kbases, we pass the database through the firmware stage multiple times (say $\boldsymbol{r}$), each pass consuming a fraction of the query $\leq$ 25 Kbases. This results in an effective throughput of $\frac{\mathbf{1}}{\boldsymbol{r}}^{\boldsymbol{th}}$ that of 25 Kbase queries for larger queries. Note that this bound on the query length is due primarily to the number of available block RAMs on our current chip and is not a fundamental limit of the design itself.

To assist in the performance analysis task, and inturn evaluate the analytical models, we developed a detailed simulator that provides a cycle-accurate model of the stage 1 design. This simulation model is used to validate the analytical models used to generate Figure 6.8, and our assumption that we can process a match from stage 1a every clock cycle in stage 1b. We chose 3 of the design points from Figure 6.8 and executed simulations using a minimum of **30** different queries for each configuration. For each of these simulations we assumed that the input rate into stage 1 is 2.128 Gbases/s, which is its maximum ingest rate. Table 6.4 compares the results we obtain from these simulations to the analytical predictions for a number of different parameters. We observe that the simulation performance predictions line up extremely well to the predictions using the analytic models.

While the mean time to process a match in stage 1b is slightly higher than the predicted value of 1, the input rate to this stage is such that the utilization remains $<$ **100%**. Also, the nature of genomic sequences is such that the assumption of uniform distribution of bases in sequences turns out to be pessimistic. The observed mean throughput of 2.128 Gbases/s (with extremely small standard deviations) builds further confidence in our analytic model results.

The raw throughput supported by our stage 1 implementation is about 2 Gbases/sec. Using the Mercury system infrastructure, which currently provides 700 MB/sec of input bandwidth, we can expect a data rate of 1.4 Gbases/sec into stage 1. Note that we use 4 bits per base, thereby eliminating potentially significant post-processing of masked sequences arising from NCBI BLASTN's use of 2 bits/base. Table 6.5 compares the performance of stage 1 in firmware to the software BLASTN.

Table 6.4: Validation of analytic predictions using simulation

Configuration	Parameter	Analytic	Simulation
query=12.5Kbases k=6, m=32Kbits	$\boldsymbol{f}$	0.369	0.34
	$\boldsymbol{f_r}$	0.81	0.82
	Process time stage 1b	$\approx \mathbf{1}$ clk	1.03
	Tput	2.128 Gbases/s	$2.128 \pm \mathbf{2.9 \times 10^{-2}}$ Gbases/s
query=12.5Kbases k=6, m=64Kbits	$\boldsymbol{f}$	0.016	0.014
	$\boldsymbol{f_r}$	0.81	0.82
	Process time stage 1b	$\approx \mathbf{1}$ clk	1.07
	Tput	2.128 Gbases/s	$2.128 \pm \mathbf{7.6 \times 10^{-6}}$ Gbases/s
query=20.0Kbases k=6, m=64Kbits	$\boldsymbol{f}$	0.15	0.12
	$\boldsymbol{f_r}$	0.69	0.71
	Process time stage 1b	$\approx \mathbf{1}$ clk	1.07
	Tput	2.128 Gbases/s	$2.128 \pm \mathbf{7.0 \times 10^{-5}}$ Gbases/s

$\boldsymbol{f}$: false positive rate from stage 1a
$\boldsymbol{f_r}$: $\%\boldsymbol{f}$ removed by stage 1b

Table 6.5: Firmware vs. software stage 1 (throughput and speedup)

Query Size	10 Kbases	25 Kbases	50 Kbases	100 Kbases	1 Mbases	Units
NCBI BLASTN Stage 1 ($\boldsymbol{Tput_1}$)	77.4	34.8	18.1	10.5	0.771	Mbases/sec
Mercury BLASTN Stage 1($\boldsymbol{Tput_1}$)	1400	1400	700	350	35	Mbases/sec
Speedup ($\boldsymbol{S_1}$)	18.1	40.2	38.7	33.4	45.4	

As shown in Figure 6.7, a queue is present in the design to smooth the bursty nature of matches that are generated in stage 1a. The detailed simulation model described above was used to assess the usage of this queue, and also to assess the performance impact of bursty matches. Figures 6.9 through 6.11 plot the maximum queue length for 32 executions each of 3 system configurations. The 3 configurations shown are the same as those used for validation purposes above. Across the board, the maximum queue lengths are reasonable, with an overall maximum under 1600 matches. Note that should the queue fill, there is appropriate backpressure built into the design so that correcness is maintained. At issue is simply the performance impact that a full queue would have on overall throughput.

Using the models we developed in Chapter 3, we now estimate the amount of buffering needed between stages 1a and 1b. We assume here that the bursts from stage 1a are geometrically distributed with a mean of **1** and the maximum burst size of **16**, and the input to this system is Poisson. We also assumed both stage 1a and 1b service times have a squared coefficient of variation of **0.5**. Figure 6.12(a) shows the fraction of maximum ingest rate (16 bases/clk) that can be accepted as a function of the buffer size between stages 1a and 1b. We see that around 100 units of buffering our ingest rate is around the maximum

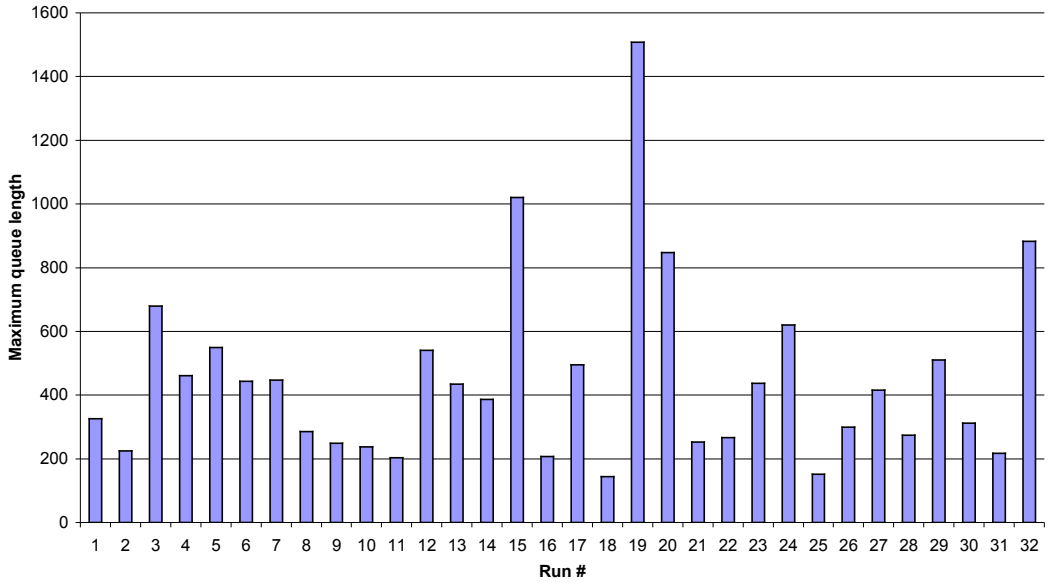


Figure 6.9: Maximum length of queue between stages 1a and 1b, query size = 12.5 Kbases, k=6, m=32Kb

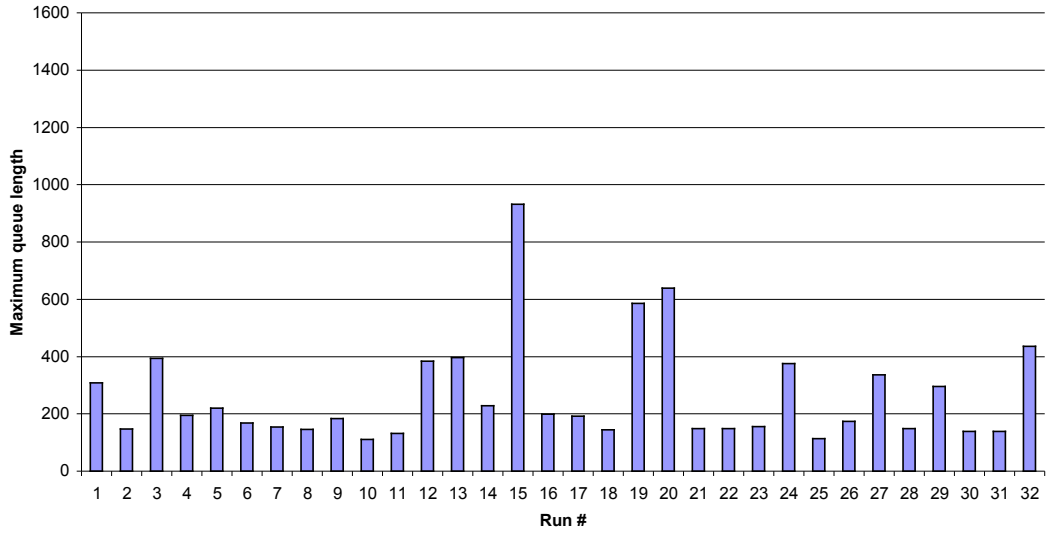


Figure 6.10: Maximum length of queue between stages 1a and 1b, query size = 12.5 Kbases, k=6, m=64Kb

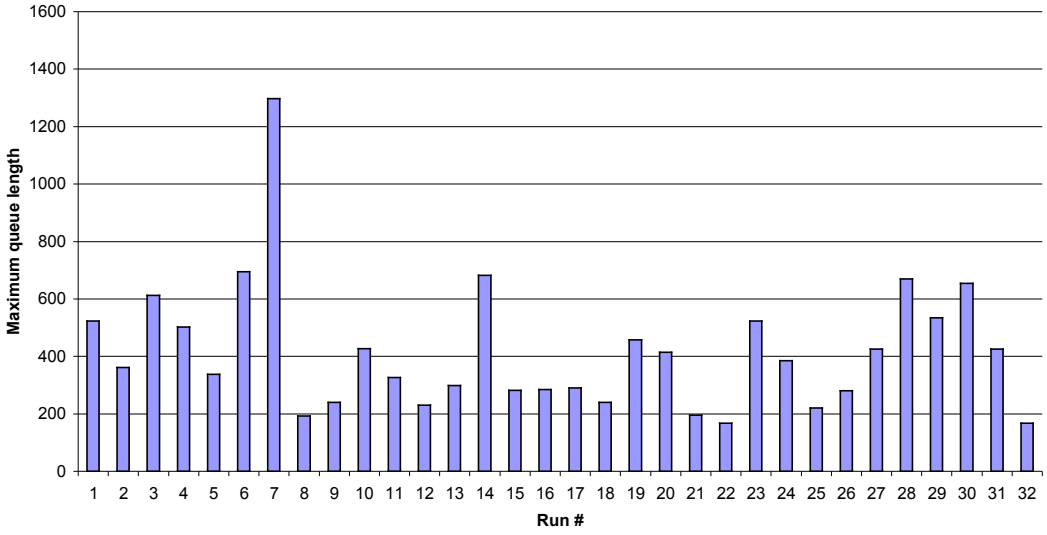
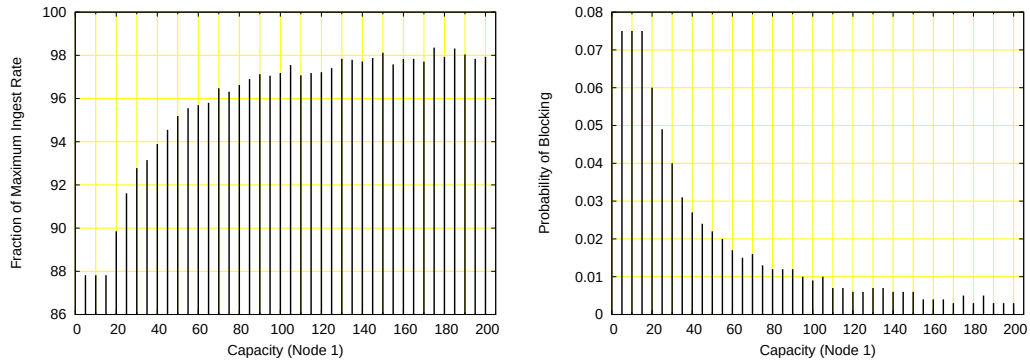


Figure 6.11: Maximum length of queue between stages 1a and 1b, query size = 20 Kbases, k=6, m=64Kb

ingest rate. Also from Figure 6.12(b) we see here that after around 200 units of buffering there is almost no blocking seen in the system.



(a) Fraction of maximum ingest rate

(b) Upstream blocking probability

Figure 6.12: Sustainable throughput and upstream blocking probability predicted by analytical model.

The bursty nature of the stage 1a match process has a negligible impact on the overall throughput performance beyond buffer size of 200. We also observe that the throughput predictions from two-node analytical model (see Table 6.4), the performance is quite accurate and that we are indeed able to process at ingest rate for a queue size of around 200.

6.5.2 Overall Performance of BLASTN on the Mercury System

We now consider *overall* pipeline performance. While the component pipeline stages above have been constructed individually and verified to work together, the complete system has yet to be integrated at full speed (we are currently undertaking this activity). As a result, the performance numbers that follow are model-based predictions.

When executing the application across multiple resources, the overall throughput is determined by the minimum throughput achieved on any one resource. Here, stage 1 executes in firmware, while stages 2 and 3 execute in software. The throughput is therefore

$$\boldsymbol{Tput_{overall} = \min\left(Tput_1, \frac{1}{p_1(t_2 + p_2 t_3)}\right)},$$

where $\boldsymbol{Tput_1}$, stage 1 throughput, is from Table 6.5; $\boldsymbol{t_i}$, the time to perform stage $\boldsymbol{i}$ in software, is from Table 6.3; and $\boldsymbol{p_i}$, the probability of an output from stage $\boldsymbol{i}$, is from Table 6.1.

Table 6.6 compares the overall performance of Mercury BLASTN with that of NCBI BLASTN. Though we have shown significant speedup for stage 1 in firmware (refer to Table 6.5), the overall speedup is limited to a factor of 5 to 8. Overall performance is now limited by the software-based stage 2. Hence, though we successfully deployed stage 1 in firmware with high throughput, the overall application still suffers from limitations imposed by the remaining pipeline stages.

Table 6.6: Overall performance (throughput and speedup)

Query Size	**10 Kbases**	**25 Kbases**	**50 Kbases**	**100 Kbases**	**1 Mbase**	**Units**
NCBI BLASTN	67.0	29.2	14.9	8.76	0.657	Mbases/sec
Mercury BLASTN	497	181	86.0	52.6	4.44	Mbases/sec
Speedup	7.42	6.21	5.76	6.01	6.84	

If $\boldsymbol{t_3}$ is the software compute time per match from stage 3 (from Table 6.3), the maximum pipeline throughput that can be sustained by stage 3 is $\boldsymbol{Tput_3 = 1/p_1 p_2 t_3}$ (as above, normalized to be in units of input bases per second from the database). For 1 Mbase query sizes, this rate is approximately 2 Gbases/sec, which matches the input rate supported by the firmware stage 1. Hence, stage 3 is unlikely to be a bottleneck to overall performance.

We next consider the overall performance impact of accelerating stage 2. This impact can be modeled as $\boldsymbol{Tput_{overall} = \min(Tput_1, Tput_2, Tput_3)}$, where $\boldsymbol{Tput_1}$ and $\boldsymbol{Tput_3}$ are

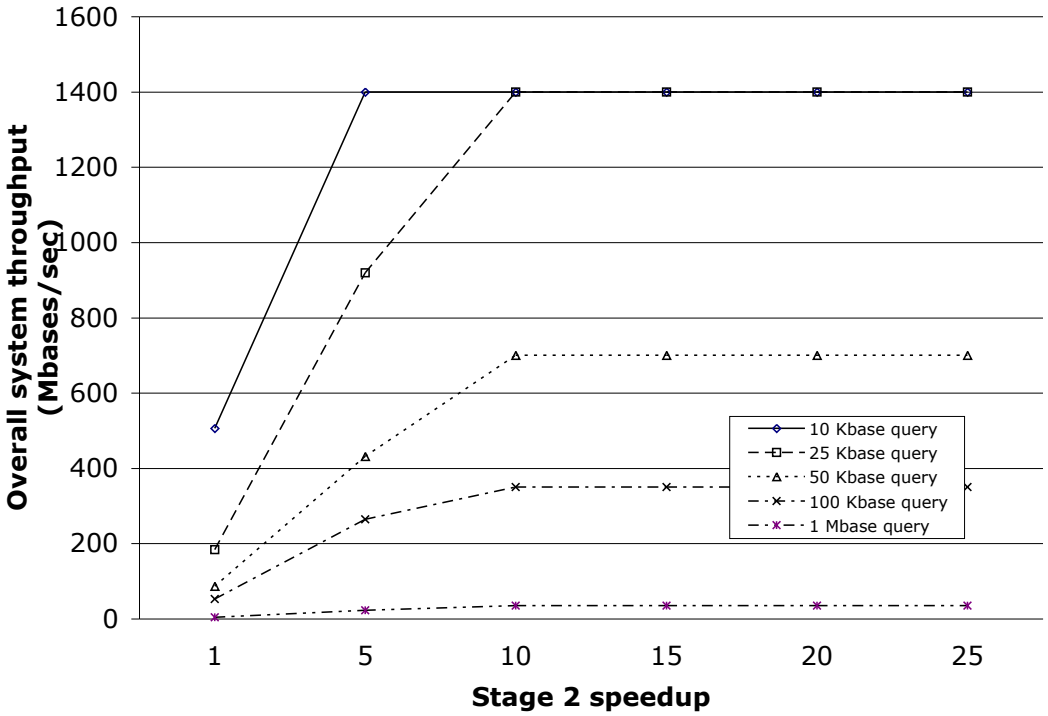


Figure 6.13: Throughput of Mercury BLASTN with improved stage 2

as above and $\boldsymbol{Tput_2}$ is now $\boldsymbol{S_2/p_1t_2}$. $\boldsymbol{S_2}$ is a model input representing the speedup of a hypothetical firmware stage 2 implementation. This model determines the performance required of the stage 2 firmware in order to achieve a given overall pipeline throughput.

Figure 6.13 plots the throughput of the overall application, as a function of the stage 2 speedup $\boldsymbol{S_2}$, for various query sizes. By increasing the performance of the bottleneck stage 2, overall performance improves until the throughput reaches the limit imposed by stage 1, at which point it saturates.

Figure 6.14 plots the speedup, relative to a pure software implementation, of the entire application as a function of stage 2 speedup, again for various query sizes. If, as seems likely, we can achieve even modest speedup in a firmware stage 2, we predict that overall performance of Mercury vs. NCBI BLASTN will improve by two orders of magnitude.

6.6 Chapter Summary

This chapter presented the design of BLASTN, an important biosequence search application, for the Mercury system, an architecture that provides both FPGA and traditional processor

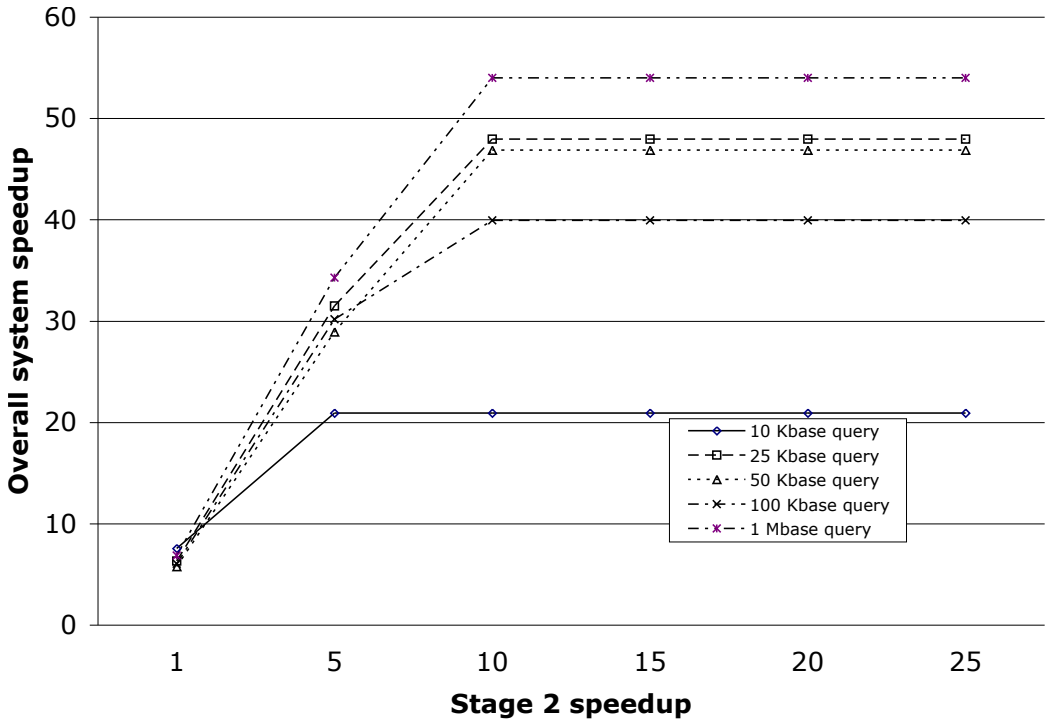


Figure 6.14: Speedup of Mercury BLASTN over NCBI BLASTN with improved stage 2

computing resources and is optimized for disk-based, data-intensive applications. We have since constructed prototype application components for a firmware (FPGA-based) stage 1 of the BLASTN pipeline, including the addition of a Bloom filter-based prefilter, a firmware hash table, and a match redundancy eliminator.

Because of the strong predicted impact of stage 2 speedups on overall application performance, we have proceeded with a firmware implementation of stage 2, which is reported in [73]. Currently, we have an end-to-end deployment of BLASTN on the Mercury prototype. Each of the component pieces is now in place, functionally correct and running at speed on the reconfigurable hardware. This overall is an excellent example of an application utilizing the potential of hybrid archiectural design.

Chapter 7

Summary and Future Work

7.1 Dissertation Summary

We have a covered a wide variety of work related to hybrid architectures as part of this thesis. We have demonstrated the use of hybrid architectures to improve the performance of applications, in our example BLASTN, without resorting to building customized hardware. Our hybrid technology based solution lines up well against the high-end commercial systems that are not as scalable a solution, and also have poor performance/dollar measures compared to this solution.

As part of this dissertation, we have introduced new algorithms to analytically predict the performance of queueing networks with blocking, with phase-type service distributions and with bulk departures from one node to another. We have also analyzed the applicability of such models under different scenarios and have provided a good understanding of cases where such models will and will not provide good results.

We also introduced the concept of self-aware estimation methods which validate the results they provide. This will help eliminate some of the skepticism associated with analytical models, which stems from the reason that the assumptions in the analytical model grossly understate the real system.

We have also demonstrated the use of FPGA based profiling techniques to measure performance metrics for architectural evaluation. We have shown here that the conclusions from such investigation can be impacted by the inclusion of OS characteristics. This tool was also used to detect and reason about rare events of interest during an application run, which has not been possible before in such reasonable amount of time. This is a very strong motivation for furthering the state-of-the-art of such emulation based systems.

7.2 Future Work

The work presented in this dissertation has many facets than can be extended beyond their current state. We summarize some extension we have contemplated in this section.

- Queueing networks: In Chapter 3 we have demonstrated the use of queueing networks to assess the buffer requirements between stages of a pipelined application deployed on hybrid systems. The analytical models we have developed can be extended in spirit to any general queueing network where we can predict the departure process from one node to another (we have assumed that this process is Poisson in nature). One could develop heuristics on the kind of departure process to expect based on the information about the nodes and the blocking experienced in the network.

  Though closed form solution rarely exist for single server nodes with arbitrary arrival processes, we have shown models of general distributions by phase-type distributions and use of state-space models to solve the single node systems encountered.

- Self-aware estimation: We have presented the need and use of self-aware estimation methods, which inform the user of potential failure, in Chapter 4. This should be made a common practice in all estimation techniques, where the model check to make sure that the "targer" for which it estimates the performance meets its trustable-input criteria. Also, sound theoretical bounds on performance should be well understood, and used when using approximate models to estimate performance.

  In our tests, particularly test 2, we used heuristics to eliminate certain test cases based solely on the input. This is generally a pessimistic approach, and a better alternative would be for the model to check if any core assumption in the model is being violated. In our case, this would have been to check if the departure process at intermediate nodes are Poisson.

- In Chapter 5 we presented emulation based techniques for estimating the performance of architectural changes and estimating the performance of benchmarks including the impact of the OS. We wish to further extend this work by running commercial benchmarks such as SPEC, server benchmarks to further drive home the need for such an emulation based estimation method.

  In Chapter 5 we also investigated the occurance of rare-events in the execution of the BLASTN application on the Liquid architecture platform. The infrastructure supports further exploration of the problem and in particular can help determine the reason for the unexpected spikes. It is possible to look at routines the kernel or user

code to reason about these spikes. This powerful capability of the profiling tool once deployed can be used for a wide variety of run-time applications to guarantee better than worst-case deadlines (smaller times).

Appendix A

Parameters for experiments in Chapter 3

This appendix contains the queueing networks used to evaluate the algorithms in Chapter 3. Tables A.1 to A.4 contain the networks that were used the test the algorithm when all the nodes in the network had exponentially distributed service times (Section 3.2.1). Tables A.5 to A.8 have the networks that were used to evaluate algorithms in Section 3.2.2, where the nodes in the network had phase-type service time distributions. Tables A.9 to A.12 are the networks that were used to evaluated the case when we had bursty departures in a network of queues. The results from these tests are presented in Section 3.3.2.

Table A.1: List of experiments for results in Figure 3.5

Expt #	Nodes	Node Parameters $< \mu_i d_i K_i >$
1	6	< 310.00 0.88 50 > < 485.48 0.95 30 > < 227.52 0.90 80 > < 740.02 0.93 100 > < 647.92 0.87 110 > < 390.03 0.94 60 >
2	2	< 660.00 0.87 60 > < 480.80 0.96 100 >
3	10	< 160.00 0.93 20 > < 633.84 0.84 60 > < 101.55 0.75 5 > < 440.19 0.81 5 > < 228.08 0.82 30 > < 54.61 0.85 70 > < 249.99 0.73 90 > < 185.12 0.97 20 > < 2.36 0.85 50 > < 142.19 0.92 120 >
4	4	< 510.00 0.85 20 > < 708.49 0.94 80 > < 88.15 0.74 10 > < 277.41 0.79 30 >
5	10	< 870.00 0.81 70 > < 734.33 0.93 70 > < 353.28 0.88 60 > < 118.56 0.89 120 > < 11.70 0.77 20 > < 207.00 0.95 70 > < 4.26 0.81 90 > < 126.95 0.76 100 > < 145.50 0.82 30 > < 145.14 0.93 40 >
6	4	< 50.00 0.83 20 > < 566.10 0.93 30 > < 519.81 0.74 50 > < 351.57 0.87 10 >
7	6	< 650.00 0.82 80 > < 351.31 0.88 5 > < 252.93 0.78 5 > < 520.31 0.77 20 > < 8.69 0.89 100 > < 225.05 0.79 60 >
8	2	< 990.00 0.74 80 > < 104.29 0.74 5 >
9	4	< 550.00 0.92 30 > < 887.73 0.97 30 > < 602.51 0.94 120 > < 348.16 1.00 60 >
10	3	< 960.00 0.94 20 > < 262.42 0.74 80 > < 186.96 0.83 70 >
11	3	< 900.00 0.80 40 > < 318.21 0.99 70 > < 739.70 0.81 40 >
12	4	< 320.00 0.74 10 > < 725.14 0.92 80 > < 162.61 0.84 40 > < 506.54 0.79 80 >
13	3	< 660.00 0.83 110 > < 91.60 0.84 60 > < 6.98 0.86 120 >
14	9	< 720.00 0.85 70 > < 263.97 0.83 100 > < 297.05 0.94 120 > < 212.40 0.94 10 > < 444.88 0.98 20 > < 245.11 0.85 80 > < 119.46 0.84 20 > < 122.36 0.92 80 > < 220.32 0.90 30 >
15	2	< 550.00 0.75 20 > < 442.12 0.77 80 >
16	6	< 420.00 0.86 5 > < 736.10 0.83 120 > < 284.80 0.93 5 > < 404.62 0.97 5 > < 244.34 0.84 40 > < 366.97 0.82 20 >
17	3	< 200.00 0.84 10 > < 428.41 0.91 90 > < 192.15 0.78 5 >
18	5	< 100.00 0.88 5 > < 297.64 0.98 10 > < 532.27 0.87 5 > < 505.33 0.77 20 > < 160.67 0.77 50 >
19	8	< 360.00 0.78 5 > < 380.86 0.89 30 > < 207.55 0.82 30 > < 142.50 0.98 120 > < 211.85 0.95 120 > < 68.60 0.77 90 > < 292.03 0.74 10 > < 56.98 0.90 60 >
20	8	< 570.00 0.87 20 > < 86.62 0.86 5 > < 423.38 0.74 20 > < 496.09 0.76 30 > < 142.11 0.95 20 > < 27.67 0.81 110 > < 47.74 0.91 10 > < 57.82 0.74 30 >
21	8	< 240.00 1.00 5 > < 459.54 0.95 70 > < 314.83 0.78 90 > < 156.60 0.73 20 > < 492.34 0.91 100 > < 218.19 0.98 70 > < 471.69 0.83 110 > < 319.19 0.79 10 >
22	7	< 820.00 0.84 10 > < 260.83 0.80 70 > < 283.71 0.95 90 > < 333.32 0.80 60 > < 209.37 0.79 10 > < 185.60 0.82 30 > < 92.35 0.93 60 >
23	6	< 130.00 0.81 50 > < 684.32 0.86 120 > < 368.89 0.97 100 > < 593.89 0.89 80 > < 24.16 0.93 110 > < 180.22 0.94 30 >
24	6	< 300.00 0.74 10 > < 88.49 0.99 10 > < 344.60 0.90 10 > < 536.77 0.84 70 > < 385.64 0.83 30 > < 74.48 0.81 120 >
25	3	< 780.00 0.79 110 > < 95.37 0.79 120 > < 37.89 0.83 70 >
26	8	< 490.00 0.91 20 > < 347.09 0.73 60 > < 287.75 0.89 40 > < 420.83 0.74 30 > < 229.39 0.91 10 > < 96.10 0.89 30 > < 216.40 0.78 90 > < 41.58 0.93 90 >
27	4	< 260.00 0.88 120 > < 573.96 0.78 20 > < 283.99 0.79 20 > < 153.76 0.82 120 >
28	5	< 420.00 0.99 40 > < 139.27 0.84 50 > < 67.12 0.83 40 > < 445.61 0.74 70 > < 479.56 0.91 60 >
29	5	< 290.00 0.90 5 > < 524.85 0.87 10 > < 157.09 0.97 10 > < 718.37 0.90 40 > < 150.72 0.73 30 >
30	4	< 150.00 0.93 20 > < 912.95 0.80 60 > < 478.82 0.76 20 > < 16.96 0.97 5 >
31	10	< 910.00 0.82 120 > < 180.71 0.90 30 > < 659.85 0.73 10 > < 391.37 0.94 30 > < 112.18 0.76 5 > < 163.52 0.84 80 > < 302.87 0.76 5 > < 216.51 0.87 10 > < 38.44 0.87 20 > < 52.30 0.88 5 >
32	2	< 990.00 0.78 20 > < 701.56 0.77 10 >
33	4	< 410.00 0.80 30 > < 553.32 0.84 30 > < 201.18 0.76 100 > < 450.66 0.89 10 >
34	4	< 830.00 0.78 60 > < 132.81 0.76 20 > < 59.73 0.97 60 > < 99.00 0.76 20 >
35	5	< 660.00 0.92 100 > < 683.52 0.91 120 > < 802.62 0.75 80 > < 214.25 0.92 5 > < 557.72 0.99 70 >
36	2	< 740.00 0.76 100 > < 22.78 0.96 70 >
37	5	< 330.00 0.84 70 > < 743.40 0.96 10 > < 218.70 0.93 5 > < 247.62 0.92 50 > < 661.92 0.83 10 >
38	2	< 350.00 0.90 5 > < 243.64 0.87 20 >
39	6	< 30.00 0.98 100 > < 353.91 0.89 110 > < 156.86 0.80 30 > < 153.87 0.90 120 > < 563.34 0.98 40 > < 191.99 0.79 90 >
40	2	< 550.00 0.92 90 > < 414.06 0.75 60 >
41	6	< 600.00 0.93 120 > < 92.71 0.81 60 > < 422.71 0.88 110 > < 353.04 0.94 20 > < 398.99 0.79 5 > < 422.91 0.89 20 >
42	6	< 60.00 0.86 30 > < 60.02 0.84 10 > < 453.82 0.99 120 > < 100.23 0.77 120 > < 387.58 0.88 5 > < 407.78 0.89 5 >
43	2	< 760.00 0.94 120 > < 592.32 0.84 40 >
44	10	< 390.00 0.87 50 > < 642.43 0.96 30 > < 41.80 0.83 50 > < 227.96 0.93 100 > < 64.30 0.85 90 > < 366.18 0.86 120 > < 18.91 0.76 20 > < 343.06 0.93 30 > < 198.92 0.85 120 > < 59.36 0.95 60 >
45	4	< 180.00 0.81 20 > < 713.47 0.76 120 > < 529.55 0.82 40 > < 452.51 0.96 110 >
46	4	< 660.00 0.87 120 > < 121.55 0.94 50 > < 179.50 0.82 70 > < 658.84 0.74 100 >
47	8	< 620.00 0.94 5 > < 47.00 0.74 10 > < 633.16 0.81 70 > < 322.97 0.81 20 > < 64.33 0.88 120 > < 299.84 0.74 50 > < 144.50 0.95 5 > < 57.31 0.98 30 >
48	5	< 880.00 0.75 5 > < 716.92 0.85 40 > < 562.83 0.82 30 > < 20.82 0.77 5 > < 211.11 0.78 30 >
49	10	< 990.00 0.82 10 > < 768.21 0.97 80 > < 246.07 0.91 20 > < 295.45 0.84 5 > < 175.50 0.95 5 > < 406.47 0.84 120 > < 273.63 0.81 10 > < 116.47 0.96 20 > < 297.49 0.74 120 > < 110.30 0.78 10 >
50	10	< 610.00 0.76 5 > < 144.73 0.94 5 > < 627.67 0.80 20 > < 456.07 0.76 50 > < 313.15 0.92 5 > < 19.96 0.78 10 > < 40.27 0.79 120 > < 220.53 0.79 110 > < 103.17 0.80 20 > < 65.35 0.85 70 >

Table A.2: List of experiments for results in Figure 3.5 (continued)

Expt #	Nodes	Node Parameters $< \mu_i d_i K_i >$
51	4	< 320.00 0.94 120 > < 505.46 0.79 40 > < 639.94 0.74 90 > < 231.31 0.74 100 >
52	5	< 220.00 0.85 20 > < 542.30 0.90 60 > < 298.80 0.77 20 > < 177.48 0.84 20 > < 133.57 0.77 120 >
53	2	< 130.00 0.79 30 > < 110.82 0.95 5 >
54	5	< 500.00 0.97 5 > < 765.71 0.99 30 > < 394.81 0.88 50 > < 50.66 0.86 5 > < 254.50 0.91 120 >
55	7	< 670.00 0.86 30 > < 128.65 0.79 90 > < 60.91 0.87 30 > < 5.89 0.92 10 > < 43.24 0.93 70 > < 235.76 0.86 40 > < 69.38 0.97 20 >
56	6	< 110.00 0.90 5 > < 449.98 0.91 90 > < 728.80 0.84 5 > < 499.74 0.91 120 > < 343.65 0.75 30 > < 267.36 0.82 90 >
57	4	< 310.00 0.91 5 > < 418.05 0.85 5 > < 473.62 0.98 10 > < 607.60 0.96 120 >
58	3	< 980.00 0.77 70 > < 208.99 0.78 60 > < 229.70 0.98 110 >
59	9	< 900.00 0.90 60 > < 494.29 0.95 20 > < 314.52 0.85 5 > < 275.28 0.80 30 > < 261.23 0.79 20 > < 347.25 0.76 80 > < 104.79 0.96 120 > < 137.64 0.73 90 > < 219.51 0.87 40 >
60	9	< 430.00 0.93 50 > < 509.00 0.87 10 > < 782.68 0.94 50 > < 309.39 0.91 90 > < 89.05 0.99 10 > < 507.84 0.74 30 > < 317.69 0.87 30 > < 95.97 0.98 50 > < 191.40 0.98 120 >
61	4	< 70.00 0.94 5 > < 651.85 0.81 110 > < 207.03 0.86 5 > < 26.29 0.92 10 >
62	5	< 540.00 0.81 10 > < 333.80 0.78 20 > < 31.90 0.89 10 > < 102.05 0.87 40 > < 466.93 0.81 20 >
63	5	< 300.00 0.87 70 > < 346.72 0.79 70 > < 232.60 0.80 5 > < 98.56 0.98 110 > < 477.22 0.80 10 >
64	10	< 280.00 0.93 10 > < 138.98 0.97 5 > < 225.77 0.98 120 > < 369.89 0.85 20 > < 358.51 0.92 100 > < 426.69 0.76 120 > < 136.52 0.78 50 > < 151.11 0.97 10 > < 11.84 0.79 5 > < 230.64 0.91 30 >
65	2	< 420.00 0.79 90 > < 63.11 0.79 110 >
66	4	< 20.00 0.92 60 > < 220.50 0.93 40 > < 770.10 0.87 100 > < 464.01 0.98 40 >
67	4	< 850.00 0.94 120 > < 555.28 0.88 20 > < 430.03 0.80 50 > < 474.86 0.83 120 >
68	10	< 810.00 0.93 10 > < 703.41 0.83 80 > < 575.02 0.83 20 > < 493.63 0.75 10 > < 445.23 0.87 50 > < 127.98 0.76 10 > < 293.92 0.90 80 > < 123.38 0.88 30 > < 218.08 0.76 40 > < 131.96 0.77 90 >
69	4	< 560.00 0.85 90 > < 440.68 0.75 120 > < 551.72 0.79 20 > < 474.14 0.94 5 >
70	2	< 220.00 0.84 40 > < 430.32 0.84 110 >
71	5	< 260.00 1.00 30 > < 319.75 0.80 60 > < 490.24 0.74 20 > < 565.93 0.81 70 > < 164.16 0.85 100 >
72	3	< 350.00 0.74 10 > < 346.26 0.84 20 > < 135.59 0.76 70 >
73	4	< 910.00 0.89 30 > < 759.08 0.88 30 > < 188.68 0.99 60 > < 441.40 0.81 90 >
74	5	< 940.00 0.76 40 > < 136.14 0.91 20 > < 330.96 0.84 90 > < 453.81 0.84 10 > < 467.47 0.93 20 >
75	6	< 610.00 0.74 10 > < 162.18 0.94 120 > < 545.79 0.88 110 > < 342.21 0.91 120 > < 49.96 0.83 70 > < 50.79 0.94 100 >
76	4	< 810.00 0.90 40 > < 610.48 0.76 90 > < 599.80 0.97 20 > < 405.29 0.87 120 >
77	7	< 40.00 0.88 70 > < 421.02 0.81 90 > < 135.48 0.81 10 > < 362.25 0.82 10 > < 396.76 0.79 60 > < 302.46 0.75 120 > < 258.11 0.94 5 >
78	3	< 270.00 0.76 120 > < 431.74 0.91 120 > < 386.32 0.95 60 >
79	5	< 850.00 0.79 60 > < 269.21 0.79 100 > < 12.52 0.79 10 > < 142.69 0.82 120 > < 56.82 0.84 40 >
80	5	< 80.00 0.99 120 > < 296.00 0.87 120 > < 325.21 0.92 10 > < 188.90 0.86 5 > < 623.12 0.91 120 >
81	4	< 760.00 0.83 110 > < 91.67 0.95 90 > < 339.89 0.86 120 > < 190.69 0.79 20 >
82	2	< 80.00 0.89 120 > < 706.94 0.97 5 >
83	4	< 30.00 0.94 100 > < 383.46 0.83 70 > < 566.25 0.98 20 > < 562.91 0.92 60 >
84	4	< 410.00 0.85 10 > < 119.43 0.79 10 > < 557.58 0.99 5 > < 146.24 0.74 120 >
85	9	< 180.00 0.76 20 > < 336.01 0.79 70 > < 520.46 0.94 90 > < 351.26 0.88 5 > < 69.66 0.78 40 > < 287.94 0.80 100 > < 34.42 0.75 30 > < 172.28 0.79 100 > < 3.75 0.73 20 >
86	4	< 600.00 0.81 70 > < 622.70 0.91 30 > < 643.34 0.91 20 > < 597.87 0.96 5 >
87	2	< 630.00 0.89 10 > < 267.18 0.84 60 >
88	4	< 180.00 0.85 20 > < 238.13 0.96 10 > < 792.92 0.88 60 > < 337.93 0.77 20 >
89	6	< 410.00 0.78 20 > < 297.68 0.86 30 > < 602.22 0.88 120 > < 233.13 0.89 20 > < 74.21 0.95 40 > < 40.46 0.83 20 >
90	5	< 420.00 0.85 90 > < 735.26 0.83 5 > < 197.39 0.86 30 > < 60.68 0.97 10 > < 82.31 0.84 60 >
91	6	< 120.00 0.86 70 > < 679.92 0.81 120 > < 361.15 0.86 60 > < 41.94 0.82 110 > < 291.57 0.98 120 > < 369.89 0.98 10 >
92	5	< 200.00 0.77 60 > < 735.60 0.87 10 > < 20.26 0.83 5 > < 72.81 0.97 50 > < 102.93 0.74 20 >
93	4	< 50.00 0.91 70 > < 281.83 0.98 70 > < 223.79 0.85 120 > < 372.72 1.00 20 >
94	7	< 990.00 0.91 20 > < 354.29 0.87 110 > < 690.14 0.80 120 > < 484.25 0.89 50 > < 243.21 0.97 10 > < 324.60 0.89 20 > < 469.83 0.75 70 >
95	4	< 400.00 0.83 20 > < 132.15 0.91 5 > < 263.80 0.93 120 > < 153.87 0.98 80 >
96	4	< 400.00 0.76 40 > < 378.26 0.82 80 > < 382.52 0.85 100 > < 83.84 0.87 40 >
97	3	< 140.00 0.78 5 > < 195.42 0.87 10 > < 642.90 0.85 5 >
98	4	< 190.00 0.73 5 > < 204.92 0.74 50 > < 250.66 0.77 100 > < 406.37 0.94 50 >
99	5	< 410.00 0.81 40 > < 251.81 0.74 100 > < 60.06 0.94 80 > < 113.34 0.86 30 > < 401.52 0.98 40 >
100	5	< 910.00 0.84 120 > < 50.19 0.85 50 > < 506.42 0.75 10 > < 372.27 0.75 100 > < 273.48 0.87 110 >

Table A.3: List of experiments for results in Figure3.5 (continued)

Expt #	Nodes	Node Parameters $< \mu_i d_i K_i >$
101	4	< 170.00 0.98 120 > < 655.90 0.90 5 > < 765.84 0.84 10 > < 7.42 0.78 5 >
102	9	< 940.00 0.76 10 > < 228.70 0.86 20 > < 288.05 0.75 5 > < 142.28 0.92 120 > < 63.11 0.80 120 > < 209.73 0.95 20 > < 248.04 0.92 40 > < 85.82 0.95 10 > < 24.10 0.84 120 >
103	6	< 540.00 0.92 60 > < 887.72 0.74 5 > < 6.87 0.97 30 > < 551.63 0.74 120 > < 442.67 0.77 20 > < 197.88 0.82 120 >
104	10	< 690.00 0.83 30 > < 430.37 0.83 120 > < 48.28 0.84 20 > < 417.46 0.94 40 > < 71.16 0.75 120 > < 184.69 0.88 50 > < 140.61 0.79 90 > < 82.38 0.88 5 > < 164.93 0.85 20 > < 114.26 0.96 120 >
105	4	< 10.00 0.81 5 > < 688.17 0.95 30 > < 7.73 0.83 5 > < 328.46 0.81 90 >
106	6	< 960.00 0.99 5 > < 297.80 0.80 40 > < 705.57 0.96 90 > < 190.21 0.75 5 > < 432.25 0.87 5 > < 452.03 0.75 120 >
107	3	< 860.00 0.78 60 > < 179.07 0.86 10 > < 120.22 0.93 5 >
108	2	< 680.00 0.97 120 > < 835.46 0.86 50 >
109	9	< 630.00 0.99 120 > < 603.10 0.92 50 > < 657.40 0.94 20 > < 51.53 0.87 50 > < 622.35 0.79 20 > < 59.40 0.85 20 > < 267.53 0.76 10 > < 53.46 0.87 70 > < 30.06 0.73 50 >
110	10	< 860.00 0.77 5 > < 680.90 0.75 60 > < 353.59 0.83 110 > < 244.11 0.81 30 > < 281.63 0.91 10 > < 3.49 0.96 90 > < 194.94 0.81 40 > < 139.01 0.84 80 > < 84.67 0.95 40 > < 8.70 0.98 5 >
111	4	< 610.00 0.73 70 > < 555.45 0.80 20 > < 76.48 0.76 120 > < 156.36 0.86 10 >
112	3	< 720.00 0.83 90 > < 687.07 0.93 20 > < 490.66 0.91 100 >
113	4	< 270.00 0.88 30 > < 263.58 0.88 20 > < 623.99 0.95 5 > < 313.16 0.80 5 >
114	10	< 540.00 0.94 20 > < 168.59 0.99 10 > < 708.01 0.95 120 > < 449.36 0.85 110 > < 201.80 0.79 70 > < 573.89 0.97 20 > < 263.45 0.97 5 > < 121.81 0.74 20 > < 324.09 0.99 20 > < 400.98 0.91 10 >
115	6	< 980.00 0.81 110 > < 104.73 0.75 10 > < 102.34 0.98 5 > < 580.37 0.96 60 > < 460.14 0.99 110 > < 437.59 0.97 30 >
116	5	< 590.00 0.90 5 > < 224.42 0.98 5 > < 665.87 0.90 5 > < 220.93 0.74 5 > < 313.81 0.91 30 >
117	5	< 570.00 0.96 20 > < 841.21 0.86 5 > < 409.58 0.75 30 > < 49.08 0.78 30 > < 28.56 0.91 120 >
118	8	< 320.00 0.96 110 > < 538.43 0.80 90 > < 216.59 0.81 20 > < 434.93 0.76 5 > < 408.57 0.88 70 > < 362.49 0.75 90 > < 278.32 0.98 90 > < 256.99 0.87 40 >
119	4	< 230.00 0.79 80 > < 39.32 0.77 10 > < 332.34 0.85 90 > < 165.25 0.92 80 >
120	6	< 310.00 0.78 10 > < 47.10 0.86 5 > < 557.15 0.94 100 > < 498.04 0.99 40 > < 149.53 0.98 110 > < 103.74 0.96 40 >
121	5	< 830.00 0.87 90 > < 548.08 0.97 20 > < 685.53 0.87 120 > < 473.06 0.81 50 > < 71.85 1.00 5 >
122	4	< 270.00 0.81 100 > < 550.99 0.81 30 > < 335.61 0.85 120 > < 168.60 0.76 20 >
123	4	< 790.00 0.81 10 > < 798.01 0.97 40 > < 776.53 0.80 5 > < 57.19 0.93 90 >
124	6	< 390.00 0.78 10 > < 132.12 0.85 40 > < 272.24 0.89 5 > < 112.09 0.97 5 > < 96.91 0.93 20 > < 138.48 0.84 30 >
125	4	< 70.00 0.77 20 > < 69.43 0.91 5 > < 507.47 0.92 110 > < 442.85 0.84 60 >
126	9	< 60.00 0.87 60 > < 468.19 0.89 30 > < 552.51 0.94 70 > < 503.72 0.87 110 > < 199.94 0.74 10 > < 422.21 0.82 120 > < 236.96 0.95 10 > < 207.05 0.93 60 > < 221.86 0.88 110 >
127	8	< 680.00 0.87 70 > < 530.97 0.95 120 > < 649.90 0.81 10 > < 119.45 0.93 70 > < 191.69 0.74 5 > < 454.21 0.84 110 > < 180.48 0.80 30 > < 253.81 0.81 30 >
128	8	< 800.00 0.92 10 > < 351.34 0.99 120 > < 137.93 0.79 5 > < 626.95 0.86 90 > < 6.30 0.93 20 > < 432.08 0.98 20 > < 534.84 0.88 10 > < 476.26 0.87 5 >
129	6	< 550.00 0.78 20 > < 374.91 0.89 20 > < 332.99 0.85 30 > < 388.79 0.84 100 > < 261.08 0.79 30 > < 135.67 0.87 30 >
130	6	< 880.00 0.76 50 > < 571.58 0.79 30 > < 333.12 0.87 60 > < 210.06 0.91 90 > < 409.12 0.82 120 > < 136.78 0.99 120 >
131	5	< 740.00 0.78 80 > < 521.23 0.80 50 > < 546.73 0.98 5 > < 36.36 0.89 120 > < 16.11 0.90 120 >
132	3	< 980.00 0.86 20 > < 335.04 0.76 20 > < 536.27 0.75 90 >
133	7	< 330.00 0.96 100 > < 602.64 0.74 30 > < 169.30 0.96 120 > < 398.96 0.79 10 > < 336.13 0.89 80 > < 347.42 0.78 10 > < 358.33 0.78 30 >
134	6	< 830.00 0.95 30 > < 28.63 0.86 100 > < 451.28 0.94 30 > < 84.73 0.78 30 > < 341.78 0.76 100 > < 213.18 0.97 120 >
135	5	< 260.00 0.84 30 > < 588.04 0.78 120 > < 498.61 0.99 30 > < 498.69 0.78 80 > < 166.67 0.94 60 >
136	3	< 340.00 0.86 120 > < 94.96 0.93 20 > < 711.06 0.93 120 >
137	8	< 320.00 0.73 120 > < 241.48 0.86 20 > < 372.31 0.80 90 > < 55.38 0.75 110 > < 259.20 0.90 10 > < 54.08 0.74 40 > < 95.24 0.96 120 > < 21.57 0.76 30 >
138	7	< 80.00 0.79 40 > < 7.85 0.90 30 > < 170.08 0.74 10 > < 161.51 0.83 100 > < 82.64 0.82 100 > < 154.10 0.91 110 > < 221.24 0.83 20 >
139	9	< 160.00 0.91 50 > < 209.13 0.95 120 > < 17.23 0.74 40 > < 95.56 0.93 120 > < 321.52 0.77 5 > < 9.22 0.77 20 > < 254.64 0.74 80 > < 36.68 0.97 20 > < 208.73 0.94 90 >
140	5	< 490.00 0.89 10 > < 35.47 0.76 110 > < 127.84 0.83 5 > < 518.15 0.98 5 > < 443.54 0.95 120 >
141	5	< 550.00 0.89 5 > < 204.51 0.91 80 > < 177.78 0.85 10 > < 619.68 0.99 60 > < 122.20 0.90 5 >
142	5	< 90.00 0.90 50 > < 890.38 0.95 10 > < 829.06 0.87 20 > < 418.00 0.81 5 > < 557.88 0.83 10 >
143	4	< 170.00 0.79 80 > < 333.44 0.82 10 > < 397.53 0.90 20 > < 41.20 0.98 120 >
144	4	< 450.00 0.83 50 > < 299.91 0.94 20 > < 250.46 0.82 10 > < 410.19 0.76 5 >
145	8	< 80.00 0.79 5 > < 70.94 0.99 20 > < 202.43 0.88 70 > < 547.10 0.90 70 > < 153.31 0.91 50 > < 178.79 0.83 20 > < 328.14 0.99 100 > < 182.60 0.76 30 >
146	10	< 840.00 0.88 90 > < 787.95 0.86 10 > < 165.08 0.79 10 > < 359.85 0.96 20 > < 401.42 0.81 110 > < 45.74 0.80 110 > < 340.44 0.90 120 > < 235.18 0.98 120 > < 212.06 0.96 20 > < 94.10 0.87 20 >
147	5	< 810.00 0.81 20 > < 80.84 0.97 10 > < 516.16 0.91 120 > < 314.02 0.84 30 > < 539.26 0.76 5 >
148	7	< 670.00 0.99 40 > < 983.17 0.95 120 > < 300.78 0.74 120 > < 599.09 0.91 20 > < 254.87 0.75 30 > < 314.97 0.91 40 > < 4.35 0.90 120 >
149	3	< 110.00 0.90 120 > < 621.40 0.82 20 > < 213.62 0.84 120 >
150	2	< 480.00 0.83 120 > < 645.08 0.93 70 >

Table A.4: List of experiments for results in Figure 3.5 (continued)

Expt #	Nodes	Node Parameters $< \mu_i d_i K_i >$
151	3	< 270.00 0.78 60 > < 638.82 0.82 5 > < 397.44 0.92 50 >
152	4	< 80.00 0.75 80 > < 30.16 0.97 5 > < 388.66 0.83 100 > < 79.43 0.96 5 >
153	4	< 520.00 0.85 20 > < 152.35 0.81 40 > < 406.14 0.99 90 > < 306.03 0.91 120 >
154	10	< 700.00 0.79 60 > < 435.21 0.90 50 > < 608.61 0.82 50 > < 336.19 0.77 120 > < 185.15 0.74 5 > < 23.34 0.97 20 > < 219.92 0.96 120 > < 180.48 0.87 20 > < 259.57 0.90 80 > < 119.12 0.95 10 >
155	6	< 860.00 0.80 5 > < 573.48 0.98 60 > < 741.71 0.92 5 > < 366.41 0.76 30 > < 373.30 0.74 20 > < 226.23 0.74 5 >
156	4	< 260.00 0.75 10 > < 616.27 0.81 120 > < 170.75 0.81 60 > < 69.16 0.81 40 >
157	6	< 650.00 0.98 100 > < 274.44 0.97 60 > < 589.40 0.87 100 > < 485.66 0.82 120 > < 654.95 0.86 120 > < 11.59 0.94 90 >
158	3	< 360.00 0.97 110 > < 725.53 0.83 70 > < 409.95 0.88 120 >
159	5	< 790.00 0.84 5 > < 227.24 0.84 20 > < 688.64 0.91 30 > < 535.96 0.95 5 > < 573.91 0.90 120 >
160	5	< 650.00 0.87 30 > < 201.24 0.83 50 > < 313.77 0.85 20 > < 403.01 0.88 30 > < 376.09 0.73 10 >
161	3	< 120.00 0.94 120 > < 873.33 0.89 120 > < 16.78 0.86 70 >
162	3	< 550.00 0.78 120 > < 367.36 0.88 120 > < 379.21 0.78 80 >
163	9	< 590.00 0.80 60 > < 79.86 0.92 120 > < 725.94 0.99 120 > < 217.62 0.86 5 > < 155.98 0.76 120 > < 147.24 0.86 5 > < 180.31 0.85 70 > < 313.22 0.92 10 > < 159.89 0.87 5 >
164	4	< 900.00 0.76 10 > < 250.71 0.80 40 > < 182.82 0.73 30 > < 409.85 0.73 50 >
165	7	< 90.00 0.94 20 > < 178.65 0.82 30 > < 239.73 0.74 120 > < 358.15 0.80 110 > < 244.14 0.81 110 > < 77.10 0.81 20 > < 65.32 0.80 40 >
166	5	< 700.00 0.85 50 > < 434.47 0.93 60 > < 118.91 0.96 5 > < 721.92 0.87 5 > < 238.34 0.90 10 >
167	2	< 920.00 0.95 20 > < 795.44 0.86 90 >
168	3	< 720.00 0.82 70 > < 565.94 0.78 5 > < 44.75 0.94 120 >
169	5	< 240.00 0.92 40 > < 701.11 0.81 120 > < 59.65 0.99 20 > < 562.50 0.73 70 > < 249.42 0.89 70 >
170	5	< 40.00 0.77 20 > < 263.25 0.94 70 > < 320.20 0.88 20 > < 261.71 0.95 10 > < 297.35 0.97 20 >
171	4	< 340.00 0.92 10 > < 905.97 0.91 20 > < 381.44 0.79 30 > < 306.59 0.75 90 >
172	5	< 850.00 0.75 30 > < 45.00 0.98 60 > < 501.17 0.80 30 > < 559.66 1.00 90 > < 506.18 0.81 100 >
173	2	< 560.00 0.96 10 > < 758.81 0.92 120 >
174	5	< 70.00 0.99 10 > < 954.37 0.76 110 > < 458.68 0.97 10 > < 393.87 0.83 80 > < 516.73 0.86 10 >
175	2	< 700.00 0.86 20 > < 233.12 0.84 10 >
176	9	< 840.00 0.79 5 > < 363.33 0.95 20 > < 722.29 0.80 20 > < 36.27 0.83 20 > < 115.93 0.98 20 > < 349.95 0.78 70 > < 358.51 0.77 30 > < 261.51 0.97 40 > < 135.64 0.77 10 >
177	4	< 520.00 0.95 80 > < 614.61 0.93 60 > < 676.61 0.78 70 > < 525.09 0.79 60 >
178	4	< 10.00 0.99 30 > < 473.65 0.94 20 > < 902.66 0.93 30 > < 259.95 0.89 5 >
179	3	< 500.00 0.73 30 > < 496.56 0.73 20 > < 401.39 0.90 100 >
180	3	< 750.00 0.77 80 > < 269.15 0.83 20 > < 57.50 0.75 120 >
181	7	< 530.00 0.77 90 > < 77.16 0.93 110 > < 379.72 0.86 40 > < 413.91 0.93 20 > < 103.65 0.81 20 > < 360.15 0.84 10 > < 106.31 0.94 80 >
182	5	< 640.00 0.96 100 > < 19.26 0.94 100 > < 354.45 0.96 110 > < 773.04 0.79 80 > < 34.22 0.93 10 >
183	10	< 640.00 1.00 20 > < 209.38 0.97 5 > < 269.84 0.77 120 > < 244.41 0.94 50 > < 672.94 0.74 50 > < 468.83 0.85 5 > < 26.31 0.88 10 > < 69.48 0.74 10 > < 119.75 0.83 120 > < 156.19 0.98 20 >
184	10	< 710.00 0.87 20 > < 620.12 1.00 5 > < 853.46 0.97 120 > < 734.78 0.87 90 > < 406.27 0.75 110 > < 448.95 0.99 20 > < 120.30 0.97 30 > < 264.52 0.94 20 > < 348.19 0.95 10 > < 52.18 0.77 120 >
185	6	< 450.00 0.95 110 > < 444.89 0.97 30 > < 64.01 0.74 120 > < 573.43 0.90 20 > < 78.57 0.86 30 > < 119.29 0.87 100 >
186	4	< 290.00 0.87 70 > < 637.97 0.78 80 > < 558.01 0.89 40 > < 180.72 0.78 120 >
187	5	< 220.00 0.78 20 > < 38.82 0.97 50 > < 735.84 0.97 5 > < 408.67 0.78 120 > < 160.28 0.87 80 >
188	4	< 660.00 0.90 5 > < 179.07 0.94 90 > < 236.63 0.85 30 > < 676.64 0.86 30 >
189	10	< 550.00 0.74 5 > < 253.08 0.86 110 > < 561.54 0.86 80 > < 10.95 0.85 120 > < 385.47 0.78 5 > < 28.82 0.97 120 > < 7.00 0.90 120 > < 122.66 0.94 60 > < 169.06 0.86 5 > < 206.86 0.97 30 >
190	4	< 290.00 0.78 5 > < 93.98 0.91 30 > < 56.95 0.82 120 > < 129.06 0.99 80 >
191	7	< 620.00 0.91 30 > < 246.98 0.75 90 > < 599.70 0.94 20 > < 571.54 0.90 5 > < 359.41 0.77 120 > < 66.95 0.76 100 > < 294.75 0.87 5 >
192	5	< 160.00 0.80 20 > < 343.51 0.92 5 > < 278.97 0.80 5 > < 140.24 0.79 5 > < 243.30 0.75 50 >
193	3	< 180.00 0.88 80 > < 96.61 0.91 70 > < 789.65 0.92 30 >
194	9	< 160.00 0.94 30 > < 440.60 0.90 5 > < 505.37 0.86 5 > < 689.47 0.76 10 > < 448.29 0.92 80 > < 361.68 0.90 30 > < 442.64 0.92 60 > < 83.70 0.86 30 > < 32.56 0.80 120 >
195	4	< 260.00 0.91 40 > < 163.70 0.81 70 > < 595.97 0.96 20 > < 324.96 0.92 10 >
196	3	< 930.00 0.85 20 > < 747.52 0.79 10 > < 594.18 0.93 5 >
197	2	< 160.00 1.00 20 > < 19.99 0.97 100 >
198	2	< 250.00 0.88 50 > < 8.77 0.76 90 >
199	6	< 200.00 0.81 30 > < 591.97 0.99 120 > < 288.08 0.98 20 > < 110.14 1.00 110 > < 746.03 0.85 10 > < 421.28 0.98 40 >
200	3	< 300.00 0.97 70 > < 330.30 0.79 40 > < 546.90 0.96 5 >

Table A.5: List of experiments for results in Figure 3.10

Expt #	Nodes	Node Parameters $< \mu_i K_i C_i^2 >$
1	5	< 220 5 1.3 > < 980 50 0.8 > < 210 60 1.1 > < 110 40 5 > < 390 30 20 >
2	4	< 430 20 3 > < 780 120 4 > < 790 90 15 > < 540 80 2 >
3	2	< 210 50 20 > < 360 5 3 >
4	8	< 570 110 1.3 > < 240 90 5 > < 100 5 1.6 > < 770 80 1.1 > < 860 5 2 > < 570 10 15 > < 380 50 15 > < 210 20 2 >
5	5	< 460 20 1.1 > < 610 5 3 > < 60 110 10 > < 490 50 4 > < 510 80 3 >
6	4	< 940 80 1.6 > < 370 10 10 > < 780 110 5 > < 350 70 1.6 >
7	5	< 430 5 15 > < 730 10 7.5 > < 250 50 1.6 > < 510 5 3 > < 180 5 0.8 >
8	5	< 410 120 5 > < 40 30 0.5 > < 560 10 2 > < 20 40 5 > < 860 5 0.5 >
9	5	< 670 40 1.3 > < 120 50 0.8 > < 480 120 7.5 > < 490 10 20 > < 40 20 1.6 >
10	8	< 510 50 2 > < 810 70 1.1 > < 820 40 0.8 > < 540 100 0.5 > < 300 100 0.5 > < 470 60 0.5 > < 550 30 1.6 > < 820 20 20 >
11	4	< 200 40 15 > < 330 110 2 > < 570 20 20 > < 100 10 10 >
12	3	< 870 5 5 > < 760 100 0.8 > < 510 10 1.1 >
13	9	< 890 5 15 > < 980 70 0.5 > < 240 20 20 > < 340 20 20 > < 440 80 4 > < 970 30 0.8 > < 790 40 7.5 > < 560 20 0.5 > < 450 120 15 >
14	6	< 80 120 3 > < 850 40 1.3 > < 50 5 10 > < 620 10 3 > < 550 70 0.5 > < 960 20 10 >
15	6	< 740 5 1.6 > < 870 70 0.5 > < 120 30 3 > < 120 120 20 > < 560 40 20 > < 220 50 0.5 >
16	6	< 760 90 10 > < 620 120 3 > < 320 110 10 > < 590 10 0.8 > < 580 90 10 > < 820 120 1.6 >
17	5	< 870 40 3 > < 640 10 3 > < 770 40 0.5 > < 350 10 7.5 > < 760 10 1.3 >
18	8	< 960 20 1.3 > < 620 30 2 > < 440 70 5 > < 110 120 3 > < 240 120 0.5 > < 420 50 20 > < 740 5 0.5 > < 540 10 3 >
19	3	< 570 40 4 > < 980 110 2 > < 470 30 7.5 >
20	4	< 510 10 1.3 > < 230 60 0.8 > < 520 50 1.1 > < 370 5 1.3 >
21	6	< 100 30 2 > < 130 30 20 > < 700 5 15 > < 810 20 1.6 > < 600 100 20 > < 210 120 3 >
22	6	< 700 30 3 > < 460 70 7.5 > < 790 30 3 > < 620 80 20 > < 800 80 5 > < 200 120 3 >
23	5	< 900 10 0.8 > < 150 5 3 > < 790 30 3 > < 740 20 1.6 > < 850 110 5 >
24	4	< 880 5 0.5 > < 990 5 20 > < 430 110 1.3 > < 780 5 5 >
25	5	< 360 5 10 > < 400 40 2 > < 320 10 3 > < 610 30 1.3 > < 740 100 4 >
26	3	< 240 30 2 > < 900 30 1.1 > < 180 30 20 >
27	4	< 700 5 0.8 > < 950 120 2 > < 330 10 20 > < 530 60 1.6 >
28	2	< 810 20 1.3 > < 520 10 4 >
29	3	< 560 5 3 > < 70 5 0.8 > < 830 70 10 >
30	5	< 910 30 4 > < 740 5 1.3 > < 520 5 5 > < 820 20 15 > < 990 20 5 >
31	5	< 150 20 15 > < 810 110 20 > < 620 5 1.3 > < 550 5 3 > < 10 40 4 >
32	7	< 250 20 3 > < 690 10 10 > < 220 90 0.5 > < 960 5 15 > < 160 90 3 > < 480 30 2 > < 240 5 1.3 >
33	6	< 350 5 3 > < 520 10 15 > < 90 60 3 > < 720 5 7.5 > < 730 120 20 > < 670 5 3 >
34	5	< 230 10 3 > < 620 5 1.1 > < 310 60 1.6 > < 800 70 1.3 > < 740 110 10 >
35	10	< 860 10 4 > < 160 40 0.8 > < 660 30 1.6 > < 680 10 7.5 > < 360 60 4 > < 330 100 2 > < 760 10 7.5 > < 80 30 15 > < 310 80 1.1 > < 850 90 3 >
36	4	< 70 70 5 > < 930 120 2 > < 410 20 7.5 > < 150 30 1.6 >
37	6	< 430 5 3 > < 240 120 1.6 > < 560 50 5 > < 750 60 4 > < 820 10 0.8 > < 740 120 10 >
38	5	< 190 20 2 > < 780 70 3 > < 630 30 15 > < 880 100 7.5 > < 600 60 7.5 >
39	4	< 880 90 5 > < 480 60 5 > < 140 10 15 > < 830 30 1.1 >
40	5	< 300 20 0.8 > < 50 5 1.3 > < 670 10 3 > < 380 70 10 > < 870 80 3 >
41	2	< 620 20 4 > < 350 60 5 >
42	5	< 730 30 1.1 > < 940 5 2 > < 630 40 3 > < 770 50 2 > < 940 30 5 >
43	5	< 570 120 2 > < 420 20 15 > < 680 40 15 > < 510 20 20 > < 450 70 5 >
44	4	< 470 100 3 > < 160 110 1.6 > < 830 5 0.5 > < 80 70 15 >
45	10	< 100 60 3 > < 460 120 20 > < 100 60 1.1 > < 380 120 1.6 > < 880 5 1.6 > < 410 10 1.6 > < 590 70 0.8 > < 250 30 1.3 > < 980 100 15 > < 550 5 1.6 >
46	5	< 550 20 1.1 > < 240 40 3 > < 540 30 7.5 > < 630 10 1.1 > < 720 20 5 >
47	5	< 330 120 1.6 > < 630 20 3 > < 660 5 1.6 > < 360 120 15 > < 450 20 2 >
48	4	< 840 110 7.5 > < 810 20 10 > < 490 20 15 > < 510 120 0.8 >
49	5	< 280 5 2 > < 680 20 4 > < 370 20 0.8 > < 800 80 2 > < 850 20 3 >
50	3	< 300 40 20 > < 420 90 4 > < 630 30 0.8 >

Table A.6: List of experiments for results in Figure 3.10 (continued)

Expt #	Nodes	Node Parameters
51	5	< 240 30 5 > < 170 120 0.5 > < 930 50 1.6 > < 340 20 3 > < 630 80 10 >
52	5	< 10 30 4 > < 550 20 20 > < 450 20 3 > < 870 20 3 > < 400 60 1.3 >
53	5	< 660 100 1.3 > < 330 10 3 > < 580 20 20 > < 840 30 7.5 > < 100 30 1.6 >
54	7	< 640 50 3 > < 650 70 2 > < 330 20 2 > < 340 20 15 > < 680 40 20 > < 780 10 15 > < 590 10 7.5 >
55	4	< 760 30 1.1 > < 90 70 3 > < 440 60 1.1 > < 430 10 5 >
56	2	< 780 120 1.1 > < 150 20 0.5 >
57	6	< 640 30 1.3 > < 180 70 3 > < 930 70 1.1 > < 520 40 2 > < 900 10 4 > < 320 50 1.3 >
58	7	< 840 120 0.5 > < 650 50 15 > < 90 5 4 > < 690 50 20 > < 310 5 3 > < 620 30 10 > < 140 20 20 >
59	3	< 860 90 15 > < 270 20 5 > < 160 50 3 >
60	6	< 840 80 5 > < 370 110 4 > < 160 40 1.6 > < 110 10 5 > < 50 120 1.6 > < 80 50 3 >
61	3	< 560 20 7.5 > < 990 80 10 > < 100 5 4 >
62	4	< 770 5 0.5 > < 690 90 5 > < 250 5 1.6 > < 650 60 0.8 >
63	3	< 200 5 2 > < 140 60 15 > < 720 20 0.8 >
64	4	< 860 30 2 > < 510 40 4 > < 880 30 20 > < 400 90 4 >
65	3	< 250 80 20 > < 680 120 1.1 > < 590 40 20 >
66	4	< 340 30 15 > < 940 120 0.8 > < 190 30 7.5 > < 740 5 0.8 >
67	5	< 360 80 3 > < 350 30 3 > < 480 80 3 > < 160 90 1.6 > < 240 110 1.1 >
68	3	< 520 80 20 > < 660 5 3 > < 310 10 3 >
69	9	< 790 5 1.3 > < 790 80 0.8 > < 940 80 5 > < 140 5 1.1 > < 180 5 3 > < 210 90 0.5 > < 980 110 1.6 > < 30 40 7.5 > < 420 20 7.5 >
70	5	< 380 5 3 > < 80 5 7.5 > < 990 50 2 > < 680 110 0.8 > < 80 80 4 >
71	3	< 70 100 3 > < 800 120 1.6 > < 380 70 2 >
72	8	< 960 110 1.3 > < 300 80 0.8 > < 560 70 2 > < 760 10 3 > < 40 120 3 > < 160 5 3 > < 630 70 0.5 > < 540 30 15 >
73	7	< 670 20 2 > < 990 5 4 > < 700 120 1.1 > < 440 120 10 > < 860 30 15 > < 760 80 4 > < 110 30 15 >
74	5	< 880 5 1.3 > < 150 30 3 > < 340 10 15 > < 680 70 3 > < 50 60 3 >
75	10	< 660 70 20 > < 740 40 1.1 > < 430 5 1.6 > < 680 10 7.5 > < 840 10 1.3 > < 620 90 0.5 > < 630 70 3 > < 190 5 3 > < 890 20 4 > < 800 120 20 >
76	6	< 800 30 0.5 > < 250 80 20 > < 260 50 1.6 > < 550 30 15 > < 690 40 1.6 > < 50 5 3 >
77	5	< 10 50 15 > < 470 5 0.8 > < 490 20 3 > < 900 120 1.6 > < 570 10 20 >
78	9	< 110 10 1.3 > < 680 60 5 > < 430 60 15 > < 340 20 7.5 > < 750 30 1.1 > < 970 90 1.6 > < 90 30 0.8 > < 740 90 10 > < 230 110 20 >
79	3	< 910 80 0.8 > < 380 10 2 > < 480 110 1.1 >
80	4	< 610 5 20 > < 320 90 20 > < 630 20 3 > < 510 80 20 >
81	5	< 580 20 0.5 > < 700 5 20 > < 780 5 1.1 > < 490 10 0.5 > < 470 50 1.1 >
82	10	< 260 120 0.8 > < 460 20 1.1 > < 640 70 0.8 > < 410 120 4 > < 90 20 10 > < 720 30 10 > < 950 100 15 > < 140 50 15 > < 640 30 3 > < 300 20 3 >
83	2	< 890 20 0.5 > < 40 10 0.5 >
84	3	< 590 50 3 > < 200 110 7.5 > < 720 50 15 >
85	6	< 740 120 10 > < 840 10 5 > < 700 20 15 > < 710 110 5 > < 420 5 20 > < 860 20 1.6 >
86	4	< 500 50 1.3 > < 670 5 1.1 > < 520 90 4 > < 300 10 1.1 >
87	6	< 700 50 1.1 > < 190 100 10 > < 260 70 7.5 > < 810 20 7.5 > < 20 40 1.3 > < 920 30 4 >
88	5	< 620 5 3 > < 880 100 3 > < 300 90 5 > < 390 10 10 > < 480 20 0.5 >
89	7	< 210 30 2 > < 940 100 7.5 > < 280 60 1.6 > < 320 110 5 > < 620 20 3 > < 340 30 2 > < 230 10 20 >
90	6	< 560 10 15 > < 830 70 0.8 > < 10 80 0.5 > < 700 20 10 > < 720 120 20 > < 170 5 7.5 >
91	8	< 490 30 15 > < 600 110 1.1 > < 270 5 1.1 > < 130 100 7.5 > < 710 20 2 > < 710 10 10 > < 990 80 1.1 > < 190 120 3 >
92	5	< 620 120 10 > < 690 50 0.5 > < 500 5 0.5 > < 560 120 1.3 > < 280 20 15 >
93	2	< 700 100 2 > < 610 5 15 >
94	3	< 990 60 0.8 > < 420 80 1.3 > < 220 60 3 >
95	3	< 400 70 0.5 > < 290 5 0.5 > < 840 90 1.1 >
96	2	< 220 110 1.3 > < 210 120 1.3 >
97	6	< 440 50 5 > < 120 110 2 > < 90 20 4 > < 680 5 7.5 > < 810 5 4 > < 190 90 10 >
98	9	< 560 90 5 > < 130 110 3 > < 390 100 3 > < 180 40 0.8 > < 780 120 3 > < 260 5 2 > < 140 100 4 > < 570 20 3 > < 430 70 20 >
99	5	< 720 110 4 > < 700 5 2 > < 850 90 3 > < 130 60 1.6 > < 790 20 1.6 >
100	6	< 60 60 1.3 > < 560 20 15 > < 120 30 7.5 > < 270 20 1.1 > < 430 5 1.1 > < 590 5 3 >

Table A.7: List of experiments for results in Figure3.10 (continued)

Expt #	Nodes	Node Parameters
101	4	< 410 40 15 > < 380 10 5 > < 210 20 2 > < 600 80 0.8 >
102	5	< 610 50 1.6 > < 160 90 3 > < 190 80 10 > < 860 40 1.3 > < 550 10 7.5 >
103	6	< 640 10 0.8 > < 380 110 10 > < 570 100 1.1 > < 10 30 15 > < 460 100 20 > < 420 5 20 >
104	6	< 410 60 0.5 > < 800 10 20 > < 470 70 3 > < 840 20 7.5 > < 590 50 1.1 > < 460 5 7.5 >
105	6	< 230 120 1.3 > < 610 5 5 > < 120 90 0.5 > < 220 110 0.8 > < 80 80 3 > < 520 20 7.5 >
106	5	< 250 30 3 > < 270 10 4 > < 190 50 1.1 > < 320 40 5 > < 470 30 3 >
107	2	< 750 120 1.1 > < 770 5 3 >
108	6	< 660 110 3 > < 380 70 4 > < 960 90 3 > < 850 40 0.5 > < 250 100 10 > < 140 120 10 >
109	9	< 360 70 3 > < 290 80 3 > < 530 5 2 > < 590 90 15 > < 240 5 0.8 > < 50 20 0.5 > < 280 120 1.6 > < 270 40 5 > < 800 5 3 >
110	4	< 570 10 4 > < 150 5 2 > < 130 30 1.6 > < 420 5 1.6 >
111	5	< 730 40 3 > < 250 30 3 > < 800 120 0.8 > < 200 20 0.5 > < 800 30 15 >
112	9	< 610 30 10 > < 690 90 0.8 > < 560 5 10 > < 210 10 15 > < 330 120 1.6 > < 610 5 10 > < 290 10 1.3 > < 860 40 1.6 > < 290 70 3 >
113	4	< 140 80 10 > < 620 30 3 > < 110 5 0.5 > < 880 110 0.5 >
114	6	< 520 30 10 > < 680 120 1.6 > < 930 5 1.1 > < 890 120 7.5 > < 800 5 15 > < 400 10 1.1 >
115	9	< 970 30 20 > < 990 30 0.5 > < 550 5 1.3 > < 670 30 1.3 > < 390 5 1.1 > < 230 60 3 > < 690 80 1.1 > < 620 80 3 > < 800 50 4 >
116	4	< 360 50 15 > < 440 5 0.8 > < 910 5 10 > < 280 30 2 >
117	9	< 30 5 1.3 > < 630 40 1.3 > < 690 120 3 > < 420 10 5 > < 60 30 20 > < 760 20 1.3 > < 10 60 5 > < 560 40 0.5 > < 350 30 1.3 >
118	8	< 760 80 0.8 > < 750 50 0.8 > < 850 70 4 > < 280 70 3 > < 670 70 7.5 > < 150 40 3 > < 430 10 0.8 > < 100 80 3 >
119	2	< 180 10 3 > < 690 20 1.1 >
120	6	< 340 120 1.6 > < 780 5 7.5 > < 860 110 4 > < 320 90 0.5 > < 780 10 10 > < 980 70 3 >
121	2	< 860 5 4 > < 410 60 5 >
122	5	< 590 30 20 > < 270 70 0.8 > < 130 20 1.6 > < 200 40 1.3 > < 300 60 5 >
123	5	< 680 60 20 > < 220 5 1.1 > < 980 120 4 > < 840 30 10 > < 420 10 1.6 >
124	7	< 200 5 1.3 > < 440 30 1.1 > < 490 20 20 > < 90 20 0.5 > < 650 40 3 > < 810 5 15 > < 910 40 0.8 >
125	7	< 620 40 4 > < 250 100 4 > < 460 70 1.3 > < 890 20 3 > < 220 50 1.3 > < 170 120 2 > < 330 100 2 >
126	3	< 500 70 2 > < 230 30 4 > < 850 5 1.6 >
127	10	< 990 120 0.5 > < 310 20 0.5 > < 760 30 5 > < 980 80 1.1 > < 870 20 15 > < 720 90 15 > < 540 90 0.5 > < 730 10 10 > < 890 120 4 > < 140 20 20 >
128	5	< 70 50 2 > < 290 30 1.6 > < 20 70 15 > < 640 5 1.6 > < 490 60 1.1 >
129	4	< 710 40 3 > < 590 30 5 > < 190 120 1.1 > < 60 20 0.5 >
130	4	< 660 80 3 > < 520 80 1.1 > < 900 90 10 > < 760 80 0.8 >
131	3	< 580 60 10 > < 670 5 15 > < 160 10 3 >
132	3	< 120 40 4 > < 910 60 15 > < 110 110 1.1 >
133	9	< 730 20 4 > < 80 120 4 > < 680 20 1.6 > < 710 60 10 > < 170 100 7.5 > < 900 5 4 > < 310 30 3 > < 780 60 0.8 > < 660 5 10 >
134	4	< 420 50 3 > < 560 30 1.1 > < 760 30 15 > < 820 70 5 >
135	4	< 980 100 1.1 > < 620 110 2 > < 80 20 1.6 > < 40 5 1.3 >
136	7	< 180 10 5 > < 460 5 2 > < 30 5 0.5 > < 580 70 3 > < 960 120 5 > < 390 120 4 > < 970 120 15 >
137	3	< 800 90 1.6 > < 90 80 1.3 > < 970 5 3 >
138	2	< 290 20 1.6 > < 600 40 4 >
139	4	< 980 30 10 > < 140 5 4 > < 120 60 5 > < 860 20 2 >
140	9	< 970 90 20 > < 810 40 4 > < 210 5 4 > < 150 80 5 > < 580 5 0.8 > < 750 20 0.8 > < 550 110 0.5 > < 770 90 1.1 > < 460 5 10 >
141	3	< 80 5 4 > < 580 30 3 > < 390 20 3 >
142	9	< 380 20 7.5 > < 630 30 7.5 > < 950 20 4 > < 850 60 10 > < 420 20 10 > < 140 20 15 > < 410 20 1.6 > < 800 20 1.3 > < 360 70 0.8 >
143	4	< 450 20 1.3 > < 580 120 1.6 > < 580 80 1.3 > < 420 40 0.5 >
144	2	< 640 5 4 > < 160 5 10 >
145	5	< 20 30 0.5 > < 850 120 20 > < 310 120 3 > < 300 80 10 > < 250 120 1.6 >
146	4	< 220 30 10 > < 300 30 2 > < 240 90 7.5 > < 720 20 2 >
147	5	< 920 100 10 > < 620 50 10 > < 900 5 5 > < 360 100 0.5 > < 440 10 0.8 >
148	2	< 30 20 4 > < 350 30 20 >
149	4	< 180 5 1.3 > < 190 5 0.8 > < 920 20 3 > < 830 40 2 >
150	5	< 920 5 1.1 > < 760 20 5 > < 610 110 10 > < 50 70 20 > < 690 60 0.8 >

Table A.8: List of experiments for results in Figure 3.10 (continued)

Expt #	Nodes	Node Parameters
151	3	< 750 110 3 > < 870 30 3 > < 880 20 20 >
152	9	< 700 60 2 > < 990 5 1.1 > < 540 10 1.1 > < 570 5 2 > < 400 80 7.5 > < 950 120 1.6 > < 840 20 20 > < 440 10 5 > < 220 120 2 >
153	9	< 810 5 1.6 > < 90 120 3 > < 400 20 20 > < 910 110 0.8 > < 460 20 0.5 > < 940 110 0.8 > < 610 100 1.1 > < 400 10 4 > < 60 10 1.1 >
154	3	< 610 10 0.8 > < 570 120 3 > < 260 90 3 >
155	4	< 620 5 0.8 > < 410 20 0.8 > < 220 120 1.3 > < 500 10 0.8 >
156	3	< 260 20 5 > < 280 70 1.6 > < 970 60 0.8 >
157	3	< 80 20 4 > < 750 5 0.5 > < 400 30 7.5 >
158	4	< 610 90 0.5 > < 970 50 2 > < 990 10 5 > < 90 5 0.8 >
159	9	< 730 5 1.3 > < 140 30 3 > < 20 60 10 > < 480 70 3 > < 220 10 10 > < 450 90 1.6 > < 430 40 10 > < 330 90 4 > < 580 10 3 >
160	2	< 140 90 1.1 > < 580 70 10 >
161	3	< 660 100 1.3 > < 340 20 2 > < 60 50 7.5 >
162	2	< 950 40 1.1 > < 250 120 15 >
163	4	< 340 60 1.1 > < 240 80 7.5 > < 110 50 0.8 > < 280 5 7.5 >
164	5	< 440 60 5 > < 560 20 7.5 > < 560 20 10 > < 360 20 20 > < 680 5 20 >
165	10	< 340 50 0.8 > < 600 90 7.5 > < 800 5 1.3 > < 300 20 3 > < 660 120 1.6 > < 990 10 5 > < 890 30 0.8 > < 590 90 5 > < 530 70 4 > < 200 40 1.1 >
166	6	< 800 5 0.8 > < 890 20 3 > < 800 110 3 > < 310 120 7.5 > < 460 10 1.3 > < 490 100 1.3 >
167	7	< 120 60 20 > < 40 60 2 > < 180 20 3 > < 170 20 15 > < 860 40 0.5 > < 170 120 1.3 > < 880 70 20 >
168	3	< 330 20 10 > < 840 80 1.1 > < 100 40 3 >
169	4	< 640 60 3 > < 10 110 4 > < 150 5 2 > < 110 80 10 >
170	9	< 590 120 2 > < 80 100 7.5 > < 320 30 3 > < 630 20 3 > < 150 80 10 > < 160 120 2 > < 970 90 7.5 > < 20 100 7.5 > < 750 20 3 >
171	9	< 990 30 7.5 > < 210 40 20 > < 210 5 1.3 > < 900 50 15 > < 160 90 2 > < 510 5 0.5 > < 540 80 1.3 > < 620 120 4 > < 490 80 3 >
172	8	< 600 50 2 > < 620 30 1.3 > < 570 5 3 > < 740 70 1.1 > < 470 20 7.5 > < 920 5 15 > < 890 30 0.8 > < 720 60 1.3 >
173	8	< 860 70 0.8 > < 260 10 15 > < 250 100 10 > < 110 70 15 > < 310 60 0.5 > < 510 80 5 > < 40 100 5 > < 490 110 0.8 >
174	5	< 270 120 1.1 > < 750 60 0.5 > < 420 10 20 > < 520 10 0.5 > < 880 20 10 >
175	5	< 90 20 5 > < 510 5 3 > < 760 100 10 > < 780 110 1.6 > < 800 80 3 >
176	3	< 860 120 3 > < 960 30 4 > < 850 70 1.3 >
177	5	< 960 60 0.8 > < 930 100 20 > < 160 5 2 > < 490 20 3 > < 540 110 0.5 >
178	10	< 390 10 0.5 > < 150 50 0.8 > < 720 80 3 > < 590 80 20 > < 260 30 20 > < 260 30 0.8 > < 940 100 3 > < 360 100 0.8 > < 690 60 0.5 > < 580 20 10 >
179	3	< 200 20 20 > < 180 30 1.1 > < 320 40 1.6 >
180	8	< 160 5 2 > < 410 70 7.5 > < 860 90 3 > < 450 80 10 > < 130 120 1.3 > < 680 100 4 > < 920 90 7.5 > < 450 50 7.5 >
181	8	< 280 80 7.5 > < 690 90 7.5 > < 610 5 7.5 > < 540 50 20 > < 370 5 7.5 > < 590 110 0.5 > < 770 20 3 > < 150 30 7.5 >
182	4	< 550 10 15 > < 800 120 4 > < 270 80 2 > < 390 110 0.8 >
183	3	< 900 20 2 > < 780 10 5 > < 960 5 20 >
184	5	< 200 60 1.1 > < 260 110 1.1 > < 770 40 3 > < 750 90 3 > < 140 40 3 >
185	10	< 690 30 1.6 > < 630 5 4 > < 450 90 3 > < 940 120 1.3 > < 160 50 0.8 > < 220 80 0.5 > < 280 10 1.6 > < 400 10 15 > < 610 70 1.1 > < 470 90 15 >
186	9	< 930 30 0.8 > < 160 10 3 > < 240 90 1.3 > < 690 50 15 > < 330 10 3 > < 870 5 4 > < 100 5 1.6 > < 450 120 0.5 > < 690 40 0.5 >
187	5	< 640 20 5 > < 700 120 3 > < 590 60 1.1 > < 730 5 1.3 > < 650 30 3 >
188	6	< 350 30 20 > < 170 120 3 > < 150 20 3 > < 230 10 3 > < 70 40 0.5 > < 510 100 15 >
189	6	< 910 120 5 > < 710 120 0.5 > < 450 5 15 > < 370 120 5 > < 810 120 2 > < 750 100 5 >
190	9	< 330 10 1.1 > < 790 80 0.5 > < 280 70 0.8 > < 310 100 7.5 > < 220 80 3 > < 470 20 2 > < 420 80 4 > < 110 5 1.3 > < 760 30 7.5 >
191	8	< 970 120 2 > < 980 5 7.5 > < 40 30 3 > < 650 10 0.5 > < 500 60 1.3 > < 540 100 1.6 > < 250 10 1.1 > < 790 20 7.5 >
192	3	< 420 10 1.6 > < 250 60 0.5 > < 400 30 1.3 >
193	6	< 800 30 3 > < 800 10 4 > < 990 30 1.1 > < 660 70 0.5 > < 780 50 3 > < 960 20 10 >
194	9	< 260 80 5 > < 530 120 10 > < 30 120 3 > < 220 10 3 > < 370 90 1.6 > < 440 70 20 > < 710 90 20 > < 70 70 5 > < 640 40 10 >
195	9	< 460 10 15 > < 390 100 1.1 > < 560 30 0.8 > < 720 20 2 > < 610 50 15 > < 300 120 20 > < 570 20 1.6 > < 340 5 1.1 > < 480 120 15 >
196	3	< 510 5 7.5 > < 30 70 10 > < 490 20 1.1 >
197	3	< 970 10 1.3 > < 650 30 1.1 > < 770 120 2 >
198	8	< 950 40 20 > < 730 10 3 > < 730 120 1.1 > < 550 5 7.5 > < 730 5 2 > < 760 80 0.5 > < 470 60 1.3 > < 570 5 20 >
199	2	< 10 10 0.5 > < 970 120 7.5 >
200	4	< 850 30 0.8 > < 570 100 20 > < 100 50 15 > < 220 5 20 >

Table A.9: List of experiments for results in Figure 3.16

Expt #	Nodes	Node Parameters $< \mu_i K_i C_i^2 >$	Burst Parameters Mean	Max
1	2	< 310 50 5 > < 135 50 1 >	4	27
2	2	< 880 120 10 > < 10 60 15 >	4	27
3	2	< 800 70 3 > < 160 120 2 >	2	13
4	2	< 80 70 3 > < 460 27 0.8 >	4	27
5	2	< 70 70 1 > < 760 90 5 >	2	13
6	2	< 960 7 2 > < 510 20 5 >	2	13
7	2	< 200 100 5 > < 960 30 0.8 >	3	20
8	2	< 680 110 5 > < 910 60 15 >	3	20
9	2	< 20 20 1.1 > < 10 30 5 >	4	27
10	2	< 490 20 10 > < 240 40 2 >	4	27
11	2	< 410 7 1.6 > < 190 50 5 >	4	27
12	2	< 730 60 3 > < 90 20 5 >	3	20
13	2	< 920 20 1.6 > < 580 27 10 >	4	27
14	2	< 990 80 0.5 > < 430 13 0.8 >	2	13
15	2	< 960 30 2 > < 410 70 15 >	4	27
16	2	< 960 20 0.8 > < 270 27 5 >	4	27
17	2	< 900 40 1.1 > < 520 20 5 >	3	20
18	2	< 230 10 20 > < 520 80 1.6 >	5	34
19	2	< 890 80 1.3 > < 630 110 1.1 >	4	27
20	2	< 10 120 0.8 > < 830 70 20 >	4	27
21	2	< 420 120 1.6 > < 710 20 1.1 >	2	13
22	2	< 330 30 5 > < 420 80 1.3 >	2	13
23	2	< 280 7 2 > < 600 80 2 >	3	20
24	2	< 940 30 0.8 > < 110 20 15 >	3	20
25	2	< 380 40 3 > < 220 70 1.3 >	1	7
26	2	< 510 90 1.1 > < 90 20 0.5 >	3	20
27	2	< 340 10 0.8 > < 680 60 0.8 >	2	13
28	2	< 800 20 3 > < 490 20 0.8 >	3	20
29	2	< 455 20 1 > < 790 120 1.3 >	2	13
30	2	< 720 10 0.8 > < 290 34 3 >	5	34
31	2	< 100 7 2 > < 450 50 1 >	3	20
32	2	< 445 20 1 > < 470 10 0.8 >	1	7
33	2	< 240 100 1.1 > < 330 30 5 >	4	27
34	2	< 820 10 1.6 > < 890 70 15 >	3	20
35	2	< 790 10 20 > < 650 34 5 >	5	34
36	2	< 420 90 1.6 > < 410 40 3 >	1	7
37	2	< 305 30 1 > < 130 70 1.3 >	3	20
38	2	< 800 110 20 > < 850 80 2 >	5	34
39	2	< 320 30 0.5 > < 160 13 3 >	2	13
40	2	< 470 10 0.8 > < 690 100 5 >	3	20
41	2	< 160 10 20 > < 120 90 15 >	5	34
42	2	< 800 7 5 > < 380 30 1.6 >	4	27
43	2	< 440 30 2 > < 430 20 5 >	3	20
44	2	< 610 90 1.3 > < 940 13 5 >	2	13
45	2	< 650 20 1.3 > < 280 34 1.3 >	5	34
46	2	< 420 40 2 > < 80 20 3 >	3	20
47	2	< 640 60 5 > < 290 30 20 >	1	7
48	2	< 240 50 1.1 > < 450 40 1.1 >	5	34
49	2	< 330 10 1.3 > < 340 60 1.1 >	5	34
50	2	< 30 7 5 > < 480 120 20 >	5	34

Table A.10: List of experiments for results in Figure 3.16 (continued)

Expt #	Nodes	Node Parameters	Burst Parameters	
			Mean	Max
51	2	< 890 10 1.3 > < 110 80 1 >	1	7
52	2	< 120 30 5 > < 310 34 20 >	5	34
53	2	< 280 7 1.1 > < 210 34 0.5 >	5	34
54	2	< 470 10 15 > < 405 30 1 >	4	27
55	2	< 880 10 1.6 > < 310 60 1.3 >	5	34
56	2	< 100 60 1.1 > < 810 20 1.3 >	3	20
57	2	< 740 120 5 > < 340 120 5 >	1	7
58	2	< 50 120 5 > < 660 27 10 >	4	27
59	2	< 650 40 0.5 > < 50 34 1.6 >	5	34
60	2	< 330 50 1.3 > < 80 120 1.1 >	1	7
61	2	< 270 20 1.6 > < 850 100 3 >	1	7
62	2	< 90 30 1 > < 890 20 15 >	3	20
63	2	< 30 20 5 > < 450 50 10 >	4	27
64	2	< 600 120 5 > < 560 34 5 >	5	34
65	2	< 10 40 1.6 > < 630 34 5 >	5	34
66	2	< 95 7 1 > < 900 120 0.8 >	4	27
67	2	< 700 7 1.1 > < 910 100 0.8 >	1	7
68	2	< 630 40 5 > < 410 50 20 >	2	13
69	2	< 50 50 5 > < 100 27 10 >	4	27
70	2	< 330 70 15 > < 910 34 0.5 >	5	34
71	2	< 210 60 3 > < 900 20 1.1 >	1	7
72	2	< 860 40 15 > < 900 120 15 >	2	13
73	2	< 140 50 5 > < 980 34 5 >	5	34
74	2	< 620 7 15 > < 910 27 1.3 >	4	27
75	2	< 450 50 1 > < 100 50 1.1 >	4	27
76	2	< 100 30 1 > < 255 34 1 >	5	34
77	2	< 890 30 20 > < 530 20 0.8 >	3	20
78	2	< 990 10 20 > < 310 120 5 >	2	13
79	2	< 50 30 0.5 > < 230 120 1.6 >	4	27
80	2	< 300 20 3 > < 400 100 20 >	2	13
81	2	< 610 7 20 > < 880 13 0.5 >	2	13
82	2	< 70 100 3 > < 930 10 5 >	1	7
83	2	< 900 110 0.8 > < 420 40 1.3 >	4	27
84	2	< 320 120 1.3 > < 860 40 2 >	4	27
85	2	< 520 30 10 > < 640 27 1.6 >	4	27
86	2	< 145 20 1 > < 30 120 1.6 >	2	13
87	2	< 25 7 1 > < 500 30 1.1 >	1	7
88	2	< 280 90 1 > < 485 7 1 >	1	7
89	2	< 210 90 1 > < 440 90 5 >	1	7
90	2	< 10 10 0.8 > < 470 20 5 >	3	20
91	2	< 315 10 1 > < 500 27 0.8 >	4	27
92	2	< 960 110 0.5 > < 760 30 5 >	3	20
93	2	< 110 20 3 > < 240 20 1.6 >	3	20
94	2	< 800 120 3 > < 640 70 1.1 >	5	34
95	2	< 380 110 1.3 > < 300 60 15 >	5	34
96	2	< 370 7 2 > < 225 20 1 >	2	13
97	2	< 910 90 5 > < 540 90 1.6 >	3	20
98	2	< 580 110 15 > < 275 20 1 >	3	20
99	2	< 410 90 20 > < 750 20 1.3 >	3	20
100	2	< 275 60 1 > < 270 120 1.3 >	3	20

Table A.11: List of experiments for results in Figure 3.16 (continued)

Expt #	Nodes	Node Parameters	Burst Parameters	
			Mean	Max
101	2	< 890 7 0.5 > < 160 13 5 >	2	13
102	2	< 240 30 0.5 > < 240 20 2 >	3	20
103	2	< 180 40 0.5 > < 345 120 1 >	3	20
104	2	< 400 70 1.6 > < 180 34 0.5 >	5	34
105	2	< 950 110 1.3 > < 150 20 1.1 >	1	7
106	2	< 390 20 20 > < 480 100 1 >	3	20
107	2	< 260 50 3 > < 30 20 1.1 >	1	7
108	2	< 395 80 1 > < 180 110 1 >	1	7
109	2	< 500 7 3 > < 530 100 1.3 >	5	34
110	2	< 980 10 3 > < 590 20 10 >	3	20
111	2	< 720 120 2 > < 740 34 20 >	5	34
112	2	< 750 20 5 > < 940 90 1.3 >	3	20
113	2	< 700 20 1.1 > < 520 30 20 >	3	20
114	2	< 700 90 15 > < 900 90 1.3 >	3	20
115	2	< 870 20 2 > < 500 120 0.8 >	1	7
116	2	< 510 110 1.3 > < 610 30 2 >	2	13
117	2	< 610 20 1.6 > < 340 120 5 >	5	34
118	2	< 350 10 1.1 > < 220 30 1.6 >	4	27
119	2	< 910 30 1.6 > < 120 110 1 >	4	27
120	2	< 410 50 10 > < 180 120 2 >	2	13
121	2	< 220 30 10 > < 700 34 5 >	5	34
122	2	< 930 60 1.1 > < 970 110 1.3 >	4	27
123	2	< 90 70 2 > < 520 13 10 >	2	13
124	2	< 680 90 10 > < 610 110 1.6 >	5	34
125	2	< 40 70 5 > < 190 30 5 >	2	13
126	2	< 620 60 1.1 > < 90 120 10 >	5	34
127	2	< 970 10 15 > < 610 120 1.3 >	3	20
128	2	< 640 30 2 > < 240 100 10 >	2	13
129	2	< 290 120 0.5 > < 910 20 2 >	3	20
130	2	< 300 120 0.8 > < 240 20 1.3 >	1	7
131	2	< 310 120 20 > < 110 90 10 >	4	27
132	2	< 280 30 20 > < 80 34 1.3 >	5	34
133	2	< 310 90 0.8 > < 410 27 10 >	4	27
134	2	< 600 80 1.6 > < 410 13 5 >	2	13
135	2	< 20 7 1.1 > < 910 120 10 >	2	13
136	2	< 670 10 1.6 > < 45 90 1 >	5	34
137	2	< 140 40 3 > < 110 80 10 >	3	20
138	2	< 340 80 10 > < 600 27 0.5 >	4	27
139	2	< 165 90 1 > < 10 34 15 >	5	34
140	2	< 620 60 1.3 > < 180 20 1.6 >	2	13
141	2	< 610 20 1.1 > < 650 20 20 >	3	20
142	2	< 420 30 1.3 > < 160 120 1 >	5	34
143	2	< 70 40 1 > < 395 20 1 >	3	20
144	2	< 435 7 1 > < 50 20 1 >	1	7
145	2	< 610 7 3 > < 790 34 5 >	5	34
146	2	< 870 40 3 > < 180 120 0.8 >	3	20
147	2	< 600 30 5 > < 40 34 1.1 >	5	34
148	2	< 130 50 0.5 > < 660 13 1.6 >	2	13
149	2	< 310 70 0.5 > < 490 20 20 >	2	13
150	2	< 990 20 5 > < 870 34 15 >	5	34

Table A.12: List of experiments for results in Figure 3.16 (continued)

Expt #	Nodes	Node Parameters	Burst Parameters	
			Mean	Max
151	2	< 190 90 3 > < 330 20 1 >	3	20
152	2	< 280 10 20 > < 490 7 1 >	1	7
153	2	< 220 80 1.6 > < 710 40 1.6 >	3	20
154	2	< 620 100 2 > < 130 10 2 >	1	7
155	2	< 250 10 1.1 > < 50 120 0.8 >	2	13
156	2	< 280 50 1.1 > < 970 30 10 >	3	20
157	2	< 205 40 1 > < 100 27 10 >	4	27
158	2	< 265 10 1 > < 910 40 10 >	5	34
159	2	< 210 7 2 > < 890 100 5 >	4	27
160	2	< 900 20 5 > < 210 27 1.1 >	4	27
161	2	< 10 7 15 > < 390 34 15 >	5	34
162	2	< 440 7 1.6 > < 140 34 15 >	5	34
163	2	< 530 50 1.3 > < 930 13 5 >	2	13
164	2	< 600 60 0.8 > < 400 34 2 >	5	34
165	2	< 830 120 0.5 > < 520 120 1.3 >	5	34
166	2	< 690 30 20 > < 70 40 15 >	2	13
167	2	< 910 80 2 > < 790 50 0.8 >	3	20
168	2	< 145 7 1 > < 540 70 1.6 >	5	34
169	2	< 10 7 1.3 > < 5 110 1 >	1	7
170	2	< 630 40 5 > < 300 120 0.5 >	3	20
171	2	< 120 110 10 > < 910 27 1.3 >	4	27
172	2	< 620 10 15 > < 120 13 15 >	2	13
173	2	< 940 7 1.1 > < 870 50 5 >	5	34
174	2	< 560 60 20 > < 580 27 3 >	4	27
175	2	< 830 20 0.5 > < 265 7 1 >	1	7
176	2	< 550 7 5 > < 860 120 0.8 >	1	7
177	2	< 860 110 3 > < 170 30 3 >	3	20
178	2	< 10 90 5 > < 510 50 2 >	4	27
179	2	< 60 20 2 > < 610 27 0.5 >	4	27
180	2	< 940 90 1.6 > < 80 13 0.8 >	2	13
181	2	< 280 80 10 > < 390 100 10 >	4	27
182	2	< 140 20 1 > < 70 34 1.6 >	5	34
183	2	< 125 120 1 > < 990 10 2 >	1	7
184	2	< 510 110 5 > < 970 20 5 >	2	13
185	2	< 350 30 0.8 > < 180 27 1.3 >	4	27
186	2	< 850 70 20 > < 60 10 20 >	1	7
187	2	< 980 60 1.1 > < 780 100 15 >	3	20
188	2	< 295 7 1 > < 760 20 0.5 >	1	7
189	2	< 470 20 0.5 > < 570 40 2 >	2	13
190	2	< 430 110 0.8 > < 990 30 2 >	1	7
191	2	< 890 100 0.5 > < 250 90 1.6 >	3	20
192	2	< 690 7 1.6 > < 860 110 5 >	4	27
193	2	< 540 110 5 > < 230 27 10 >	4	27
194	2	< 790 7 1.1 > < 670 80 2 >	2	13
195	2	< 140 20 1.1 > < 500 100 0.5 >	5	34
196	2	< 240 110 10 > < 790 20 2 >	3	20
197	2	< 630 20 10 > < 640 100 20 >	3	20
198	2	< 240 7 0.8 > < 680 20 10 >	3	20
199	2	< 380 40 20 > < 790 20 1.6 >	1	7
200	2	< 110 120 2 > < 720 90 20 >	1	7

References

[1] Ivo Adan and Jacques Resing. Queueing theory. Technical report, Department of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands, 2001.

[2] S. F. Altschul and W. Gish. Local alignment statistics. *Methods: a Companion to Methods in Enzymology*, 266:460–80, 1996.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, et al. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–10, 1990.

[4] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–402, 1997.

[5] AMBA Specification (Rev 2.0). ARM, Ltd., `http://www.arm.com/products/solutions/AMBA_spec.html`.

[6] Todd R. Andel and Alec Yasinac. On the credibility of manet simulations. *Computer*, 39(7):48–54, 2006.

[7] Arvind, Krste Asanovic, Derek Chiou, James C. Hoe, Christoforos Kozyrakis, Shih-Lien Lu, Mark Oskin, David Patterson, Jan Rabaey, and John Wawrzynek. RAMP: Research Accelerator for Multiple Processors – A Community Vision for a Shared Experimental Parallel HW/SW Platform. Technical Report UCB//CSD-05-1412, September 2005.

[8] B.K. Asare. *Queueing networks with blocking*. PhD thesis, Trinity College, Dublin, Ireland, 1978.

[9] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[10] Forest Baskett, K. Mani Chandy, Richard R. Muntz, and Fernando G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *J. ACM*, 22(2):248–260, 1975.

[11] Gordon B. Bell, Kevin M. Lepak, and Mikko H. Lipasti. Characterization of silent stores. In *Proc. of 2000 Int'l Conf. on Parallel Architectures and Compilation Techniques*, pages 133–142, October 2000.

[12] Nathan L. Binkert, Erik G. Hallnor, and Steven K. Reinhardt. Network-oriented full-system simulation using M5. In *Proc. of 6th Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2003.

[13] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, May 1970.

[14] Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor Shridharbhai Trivedi. *Queueing Networks and Markov Chains : Modeling and Performance Evaluation With Computer Science Applications*. Wiley-Interscience, 1998.

[15] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. http://citeseer.nj.nec.com/543187.html.

[16] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17:419–428, 2001.

[17] J. Buhler. Mercury BLAST dictionaries: analysis and performance measurement. Technical Report XX, Washington University in St. Louis, 2006.

[18] J. Buhler, U. Keich, and Y. Sun. Designing seeds for similarity search in genomic DNA. *Journal of Computing and Systems Science*, 70:342–363, 2005.

[19] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural characterization of Java TPC-W. In *Proc. of 7th Int'l Symp. on High Performance Computer Architecture*, January 2001.

[20] L. Carter and M. Wegman. Universal classes of hashing functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[21] P. Caseau and G. Pujjole. Throughput capacity of a sequence of queueing with blocking due to finite waiting room. *IEEE Trans. Software Engineering*, 5:631–642, 1979.

[22] R. Chamberlain and R. Cytron. Novel techniques for processing unstructured data sets. In *Proc. of IEEE Aerospace Conf.*, March 2005.

[23] R. Chamberlain, J. Fritts, P. Krishnamurthy, and H. Zhang. Experimental federated modeling of an optical data path. In *Proc. of The* $\mathbf{4^{th}}$ *IASTED Intl. Conf. on Modelling, Simulation, and Optimization (MSO)*, Kauai, HI, USA, 2004.

[24] R. Chamberlain and B. Shands. Streaming data from disk store to application. In *Proc. of 3rd Int'l Workshop on Storage Network Architecure and Parallel I/Os*, pages 17–23, September 2005.

[25] R. Chamberlain, B. Shands, and J. White. Achieving real data throughput for an FPGA co-processor on commodity server platforms. In *Proc. of 1st Workshop on Building Block Engine Architectures for Computers and Networks*, October 2004.

[26] R. D. Chamberlain, R. K. Cytron, M. A. Franklin, and R. S. Indeck. The *Mercury* system: Exploiting truly fast hardware for data search. In *Proc. of Int'l Workshop on Storage Network Architecture and Parallel I/Os*, pages 65–72, September 2003.

[27] Roger D. Chamberlain, Ron K. Cytron, Jason E. Fritts, and John W. Lockwood. Vision for liquid architecture. In *Proc. of Workshop on Next Generation Software*, Rhodes, Greece, April 2006.

[28] Compugen, Ltd. http://www.cgen.com.

[29] Z. J. Czech, G. Havas, and B. S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182:1–143, 1997.

[30] W. J. Dally et al. Merrimac: Supercomputing with streams. In *Proc. of Supercomputing Conf.*, November 2003.

[31] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel Bloom filters. In *Hot Interconnects, (Stanford, CA)*, pages 44–51, August 2003.

[32] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. Deep packet inspection using parallel Bloom filters. *IEEE Micro*, 24(1):52–61, 2004.

[33] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using Bloom filters. In *Proceedings of SIGCOMM*, pages 201–212. ACM Press, 2003.

[34] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest prefix matching using Bloom filters. *IEEE/ACM Transactions on Networking*, 14(2):397– 409, April 2006.

[35] R. K. Singh et al. BioSCAN: a dynamically reconfigurable systolicarray for biosequence analysis. In *Proceedings of CERCS 96*, 1996.

[36] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[37] E. A. Fox, L. S. Heath, Q.-F. Chen, and A. M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35:105–121, 1992.

[38] M. Franklin, R. Chamberlain, M. Henrichs, B. Shands, and J. White. An architecture for fast processing of large unstructured data sets. In *Proc. of 22nd Int'l Conf. on Computer Design*, pages 280–287, October 2004.

[39] S. Friedman. Dusty caches to save memory traffic. Master's thesis, Washington University in St. Louis, 2005.

[40] S. Friedman, P. Krishnamurthy, R. Chamberlain, and R. Cytron. Dusty caches for reference counting garbage collection. In *MEmory performance: DEaling with Applications, systems and architecture*, September 2005.

[41] Scott Friedman, Praveen Krishnamurthy, Roger Chamberlain, Ron K. Cytron, and Jason E. Fritts. Dusty caches for reference counting garbage collection. In *Proc. of Workshop on Memory Performance: Dealing with Applications, Systems and Architecture*, September 2005.

[42] Scott Friedman, Nicholas Leidenfrost, Benjamin C. Brodie, and Ron K. Cytron. Hashtables for embedded and real-time systems. In *Proceedings of the IEEE Workshop on Real-time Embedded Systems*, 2001.

[43] Jiri Gaissler. The LEON Processor. `www.gaisler.com`, 2005.

[44] Mike Genden, Ram Raghavan, Mack Riley, John Spannaus, and Thomas Chen. Real-time performance monitoring and debug features of the first generation Cell processor. In *Proc. of 1st Workshop on Tools and Compilers for Hardware Acceleration*, September 2006.

[45] R. A. Gibbs et al. Genome sequence of the Brown Norway rat yields insights into mammalian evolution. *Nature*, 428:493–521, 2004.

[46] S. A. Guccione and E. Kellar. Gene matching using jbits. In *Twelfth International Field-Programmable Logic and Application Conference*, 2002.

[47] P. Guerduox-Jamet and D. Lavenier. Systolic filter for fast DNA similarity search. In *Proceedings of IEEE International Conference on Application-specific Systems, Architecture, and Processors*, 1995.

[48] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of IEEE 4th Workshop on Workload Characterization*, 2001.

[49] T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *Journal of Algorithms*, 41:69–85, 2001.

[50] John L. Hennessey and David A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 2003.

[51] F.S. Hillner and R. Boling. Finite queues in series with exponential or Erlang service times - a numerical approach. *Operation Research*, 15:286–303, 1967.

[52] J. D. Hirschberg, R. Hughley, and K. Karplus. Kestrel: a programmable array for sequence analysis. In *Proceedings of IEEE International Conference on Application-specific Systems, Architecture, and Processors*, pages 23–34, 1996.

[53] D. T. Hoang. Searching genetic databases on Splash 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 185–191, 1993.

[54] Richard Hough, Phillip Jones, Scott Friedman, Roger Chamberlain, Jason Fritts, John Lockwood, and Ron Cytron. Cycle-accurate microarchitecture performance evaluation. In *Proc. of Workshop on Introspective Architecture*, February 2006.

[55] Richard Hough, Praveen Krishnamurthy, Ron K. Cytron, and Roger Chamberlain. Performance evaluation using soft-core processors on reconfigurable hardware. Submitted to SIGMETRICS, Nov. 2006.

[56] J. Jackson. Network of waiting lines. *Management Science*, 5(4):518–521, 1957.

[57] J. Jackson. Jobshop-like queueing systems. *Management Science*, 10(1):131–142, 1963.

[58] R. Jain. The art of computer systems performance analysis. *SIGMETRICS Performance Evaluation Review*, 18(3):21–22, 1990.

[59] R. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* John Wiley & Sons, 1991.

[60] M.A. Johnson and M.R. Taaffe. Matching moments to phase distributions: Mixtures of Erlang distributions of common order. *Communication Statistics-Stochastic Models*, 5:711–743, 1989.

[61] P. Jones, S. Padmanabhan, D. V. Schuehler, S. J. Friedman, P. Krishnamurthy, H. Zhang, R. Chamberlain, R. K. Cytron, J. W. Lockwood, and J. Fritts. Extracting and improving microarchitecture performance on reconfigurable architectures. In *CTCES Workshop(at CASES)*, 2004.

[62] Phillip Jones, Shobana Padmanabhan, Daniel Rymarz, John Maschmeyer, David V. Schuehler, John W. Lockwood, and Ron K. Cytron. Liquid architecture. In *Proc. of Workshop on Next Generation Software*, April 2004.

[63] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Attson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, 36(8):54–62, August 2003.

[64] H. T. Kaur, D. Manjunath, and S. K. Bose. The queuing network analysis tool (qnat). *Mascots*, 2000.

[65] W. J. Kent. BLAT: the BLAST-like alignment tool. *Genome Research*, 12:656–64, 2002.

[66] L. Kleinrock. *Queueing Systems, Volume 1: Theory.* John Wiley & Sons, 1975.

[67] G. Knowles and P. Gardner-Stephen. DASH: localizing dynamic programming for order of magnitude faster, accurate sequence alignment. In *Proceedings of the 3rd International IEEE Computer Society Computational Systems Bioinformatics Conference*, pages 732–35, 2004.

[68] G. Knowles and P. Gardner-Stephen. A new hardware architecture for genomic and proteomic sequence alignment. In *Proc. of IEEE Computational Systems Bioinformatics Conf.*, 2004.

[69] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, A. Jacob, and J. Lancaster. Biosequence similarity search on the *Mercury* system. *Journal of VLSI Signal Processing (to appear)*, 2007.

[70] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, and J. Lancaster. Biosequence similarity search on the mercury system. In *15th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2004)*, September 2004.

[71] P. Krishnamurthy, R. Chamberlain, R. Cytron, and J. Fritts. Evaluating dusty caches on general workloads. In *Fifth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2006.

[72] J. Lancaster. Design and evaluation of a BLAST ungapped extension accelerator. Master's thesis, Washington University in St. Louis, 2006.

[73] J. Lancaster, J. Buhler, and R. D. Chamberlain. Acceleration of ungapped extension in Mercury BLAST. In *Proc. of 7th Workshop on Media and Streaming Processors*, November 2005.

[74] E. S. Lander et al. Initial sequencing and and analysis of the human genome. *Nature*, 409:860–921, 2001.

[75] D. Lavenier. Speeding up genome computations with a systolic array. *SIAM News*, 31(8):1–7, 1998.

[76] D. Lavenier, S. Guytant, S. Derrien, and S. Rubin. A reconfigurable parallel disk system for filtering genomic banks. In *ERSA'03, Engineering of Reconfigurable Systems and Algorithms*, 2003.

[77] Averill M. Law and W. David Kelton. *Simulation Modeling and Analysis*. McGraw-Hill Higher Education, 1997.

[78] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *International Symposium on Microarchitecture*, pages 330–335, 1997.

[79] Free Hardware and Software Resources for System on Chip. `http://www.leox.org`.

[80] Kevin M. Lepak. *Exploring, Defining, and Exploiting Recent Store Value Locality*. PhD thesis, University of Wisconsin–Madison, 2003.

[81] Kevin M. Lepak and Mikko H. Lipasti. Silent stores for free. In *Proc. of the 33rd ACM/IEEE Int'l Symp. on Microarchitecture*, pages 22–31, 2000.

[82] Kevin M. Lepak and Mikko H. Lipasti. Temporally silent stores. In *Proc. of the 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 30–41, 2002.

[83] M. Li, B. Ma, D. Kisman, and J. Tromp. Patternhunter II: highly sensitive and fast homology search. *Journal of Bioinformatics and Compuational Biology*, 2:417–39, 2004.

[84] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-time Environment. *JACM*, 20(1):46–61, January 1973.

[85] Jane W. S. Liu. *Real-time Systems*. Prentice Hall, New Jersey, 2000.

[86] John W Lockwood. Evolvable Internet hardware platforms. In *The Third NASA/DoD Workshop on Evolvable Hardware*, pages 271–279, July 2001.

[87] National Center for Biological Information. Growth of GenBank, 2002. http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.

[88] M.F Neuts. *Matrix-Geometric Solutions in Stochastics Models: An Algorithmic Approach*. John Hopkins University Press, 1981.

[89] Z. Ning, A. J. Cox, and J. C. Mullikin. SSAHA: a fast search method for large DNA databases. *Genome Research*, 11:1725–9, 2001.

[90] T. Osogami and M. Harchol-Balter. Necessary and sufficient conditions for representing general distributions by Coxians. Technical Report CMU-CS-02-178, Carnegie Mellon University, 2002.

[91] S. Padmanabhan, R. K. Cytron, R. Chamberlain, and J. W. Lockwood. Automatic application-specific microarchitecture reconfiguration. In *Proc. of 13th Reconfigurable Architectures Workshop*, Rhodes, Greece, 2006.

[92] Shobana Padmanabhan, Phillip Jones, David V. Schuehler, Scott J. Friedman, Praveen Krishnamurthy, Huakai Zhang, Roger Chamberlain, Ron K. Cytron, Jason Fritts, and John W. Lockwood. Extracting and improving microarchitecture performance on reconfigurable architectures. *Int'l Journal of Parallel Programming*, 33(2–3):115–136, June 2005.

[93] N. Pappas. Searching biological sequence databases using distributed adaptive computing. Master's thesis, Virginia Polytechnic Institute and State University, 2003.

[94] Paracel, Inc. http://www.paracel.com.

[95] David Patterson, Arvind, Krste Asanovic, Derek Chiou, James C. Hoe, Christoforos Kozyrakis, Shih-Lien Lu, Mark Oskin, Jan Rabaey, and John Wawrzynek. RAMP: Research Accelerator for Multiple Processors. In *Hot Chips*, August 2006.

[96] H.G Perrros and T Altiok. Approximate analysis of open networks of queues with blocking: tandem configurations. *IEEE Trans. Soft. Eng.*, 12:450–461, 1986.

[97] P. A. Pevzner and M. S. Waterman. Multiple filtration and approximate pattern matching. *Algorithmica*, 13(1/2):135–154, 1995.

[98] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. A performance study of hashing functions for hardware applications. In *Proc. of Int'l Conf. on Computing and Information*, pages 1621–1636, 1994.

[99] M. V. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient hardware hashing functions for high performance computers. *IEEE Transactions on Computers*, 46:1378–1381, 1997.

[100] E. Reidel, C. Faloutsos, G. Gibson, and D. Nagle. Active disks for large-scale data processing. *IEEE Computer*, 34(6):68–74, June 2001.

[101] T. Altiok R.O Onvural, H.G Perros. On the complexity of the matrix-geometric solution of exponential open queueing networks with blocking. In *Proc. of Int'l Workshop on Modelling Techniques and Performance Evaluation*, pages 3–12, 1987.

[102] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[103] Douglas C. Schmidt. ACE: an Object-Oriented Framework for Developing Distributed Applications. In *Proceedings of the* $\mathbf{6^{th}}$ *USENIX C++ Technical Conference*, Cambridge, Massachusetts, April 1994. USENIX Association.

[104] Lesley Shannon and Paul Chow. Using reconfigurability to achieve real-time profiling for hardware/software codesign. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pages 190–199, New York, NY, USA, 2004. ACM Press.

[105] WWW Computer Architecture Page. `http://www.cs.wisc.edu/arch/www/tools.html`.

[106] K. Skadron, M. Martonosi, D. August, M. Hill, D. Lilja, and V. Pai. Challenges in Computer Architecture Evaluation. *IEEE Computer*, 36(8), August 2003.

[107] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–97, March 1981.

[108] SPARC International. *The SPARC Architecture Manual Version 8*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[109] Standard Performance Evaluation Corporation. `http://www.spec.org`.

[110] SPEC newsletter, 1989.

[111] SPEC newsletter, 1990.

[112] R. Sprugnoli. Perfect hashing functions: a single probe retrieving method for static sets. *Communications of the ACM*, 20(11):841–850, 1977.

[113] W. Stadje. Some exact expressions for the bulk-arrivals queue $\boldsymbol{M^x/M/1}$. *Queueing Systems*, 4(1):85–461, 1989.

[114] R. E. Tarjan and A. C. C. Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, 1979.

[115] H.C. Tijms. *Stochastic Models: An Algorithmic Approach*. John Wiley & Sons, 1994.

[116] TimeLogic Corporation. http://www.timelogic.com.

[117] $\boldsymbol{\mu}$Clinux – Embedded Linux/Microcontroller Project. `http://www.uClinux.org`.

[118] J. Warland. *An Introduction to Queueing Networks*. Prentice Halls, 1988.

[119] R. H. Waterston et al. Initial sequencing and comparative analysis of the mouse genome. *Nature*, 420:520–562, 2002.

[120] B. West, R. D. Chamberlain, R. S. Indeck, and Q. Zhang. An FPGA-based search engine for unstructured database. In *Proc. of 2nd Workshop on Application Specific Processors*, pages 25–32, December 2003.

[121] W. Whitt. Performance of the queueing network analyzer. *Bell System Technical Journal*, 62(9):2817–2843, 1983.

[122] W. Whitt. The queueing network analyzer. *Bell System Technical Journal*, 62(9):2779–2815, 1983.

[123] Tilman Wolf and Mark A. Franklin. CommBench – A telecommunications benchmark for network processors. In *Proc. of IEEE Int'l Symp. on Performance Analysis of Systems and Software*, pages 154–162, Austin, TX, April 2000.

[124] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. of 22nd Int'l Symp. on Computer Architecture*, pages 24–36, June 1995.

[125] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *ACM Computer Architecture News*, 23(1):20–24, March 1995.

[126] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Statistical sampling of microarchitecture simulation. *ACM Trans. Model. Comput. Simul.*, 16(3):197–224, 2006.

[127] Y. Yamaguchi, T. Maruyama, and A. Konagaya. High speed homology search with FPGAs. In *Pacific Symposium on Biocomputing*, pages 271–282, 2002.

[128] Q. Zhang, R. D. Chamberlain, R. S. Indeck, B. West, and J. White. Massively parallel data mining using reconfigurable hardware: Approximate string matching. In *Proc. of Workshop on Massively Parallel Processing*, April 2004.

[129] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning DNA sequences. *Journal of Computational Biology*, 7:203–14, 2000.

Vita

Praveen Krishnamurthy

Date of Birth July 10, 1979

Place of Birth Raebareli, India

Degrees B.E. First Class With Distinction, Electronics and Communication Engineering, May 2000
M.S. Computer Engineering, December 2002
D.Sc. Computer Engineering, December 2006

Professional Societies Association for Computing Machines
Institute of Electrical and Electronics Engineers

Publications

P. Krishnamurthy et. al."Biosequence Similarity Search on the Mercury System," To appear in Journal of VLSI Signal Processing

S. Dharmapurikar, P. Krishnamurthy and D. Taylor "Longest Prefix Matching Using Bloom Filters," To appear in IEEE Transactions on Networking

R. Chamberlain et. al. "VLSI Photonic Ring Multicomputer Interconnect: Architecture and Signal Processing Performance," Journal of VLSI Signal Processing, Vol. 40, 2005

S. Padmanaban et. al. "Extracting and Improving Microarchitecture Performance on Reconfigurable Architectures," International Journal of Parallel Programming , Vol. 33, 2005

S. Dharmapurikar, P. Krishnamurthy et. al. "Deep Packet Inspection using Parallel Bloom Filters," IEEE Micro, Vol. 24, 2004

Dec 2006