Online Modeling and Tuning of Parallel Stream Processing Systems

A dissertation by: Jonathan C. Beard

Dept. of Computer Science and Engineering
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science
Department of Computer Science and Engineering

Dissertation Examination Committee:
Roger Dean Chamberlain, Chair
Jeremy Buhler
Ron Cytron
Roch Guerin
Jenine Harris
Juan Pantano
Robert B. Pless

Online Modeling and Tuning of Parallel Stream Processing Systems

by
Jonathan Curtis Beard

A dissertation presented to the
Graduate School of Arts and Sciences
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

August 2015
Saint Louis, Missouri

Table of Contents

List of Figures

List of Tables

Acknowledgements

Any great accomplishment is rarely the act of a single person, this one is no exception. I was fortunate to receive a little help from those who have traveled the path before me and enjoy the company of some great companions along the way. I would like to thank you all by name, but my allotted space is quite finite. Thank you, despite your collective anonymity, your contributions are no less appreciated.

I would like to thank Roger Chamberlain for his patience, mentorship, and friendship over the last few years. I would also like to thank Robert Pless for his advice throughout this process, and for yanking my application from the department to which I originally applied. I would undoubtedly not be here if it were not for him. I would also like to thank Ron Cytron and David Rubenstein for their mentorship and encouragement over the years.

It would be negligent of me to not thank my colleagues who have befriended me along the way: Meenal Kulkarni, Michael Hall, Adina Stoica, Peng Li, Hongtao Sun, Steve Cole, Dan Lazewatsky, David Ferry, David Lu, and countless others. I can't acknowledge friends without thanking my very good friends Joe Wingbermuehle and Sara Melnick, who spent countless hours listening to me babble within the confines of my office. We shared many successes and failures, neither will soon be forgotten.

Lastly I want thank the office staff for their help, humor, and friendly reminders: Madeline Hawkins, Kelli Eckman, Jayme Moehle, Lauren Huffman, Myrna Harbison, and Sharon Matlock. Thank you all.

Jonathan Curtis Beard

Washington University in Saint Louis
August 2015

Dedicated to my family, Kate and Aiden.

ABSTRACT OF THE DISSERTATION

Online Modeling and Tuning of Parallel Stream Processing Systems
by
Jonathan Curtis Beard
Doctor of Philosophy in Computer Science
Washington University in St. Louis, 2015
Professor Roger Dean Chamberlain, Chair

Writing performant computer programs is hard. Code for high performance applications is profiled, tweaked, and re-factored for months specifically for the hardware for which it is to run. Consumer application code doesn't get the benefit of endless massaging that benefits high performance code, even though heterogeneous processor environments are beginning to resemble those in more performance oriented arenas. This thesis offers a path to performant, parallel code (through stream processing) which is tuned online and automatically adapts to the environment it is given. This approach has the potential to reduce the tuning costs associated with high performance code and brings the benefit of performance tuning to consumer applications where otherwise it would be cost prohibitive. This thesis introduces a stream processing library and multiple techniques to enable its online modeling and tuning.

Stream processing (also termed data-flow programming) is a compute paradigm that views an application as a set of logical kernels connected via communications links or streams. Stream processing is increasingly used by computational-x and x-informatics fields (e.g., biology, astrophysics) where the focus is on safe and fast parallelization of specific big-data applications. A major advantage of stream processing is that it enables parallelization without necessitating manual end-user management of non-deterministic behavior often characteristic of more traditional parallel processing methods. Many big-data and high performance applications involve high throughput processing, necessitating usage of many parallel compute kernels on several compute cores. Optimizing the orchestration of kernels has been the focus of much theoretical and empirical modeling work.

Purely theoretical parallel programming models can fail when the assumptions implicit within the model are mis-matched with reality (i.e., the model is incorrectly applied). Often it is unclear if the assumptions are actually being met, even when verified under controlled conditions. Full empirical optimization solves this problem by extensively searching the range of likely configurations under native operating conditions. This, however, is expensive in both time and energy. For large, massively parallel systems, even deciding which modeling paradigm to use is often prohibitively expensive and unfortunately transient (with workload and hardware). In an ideal world, a parallel run-time will re-optimize an application continuously to match its environment, with little additional overhead. This work presents methods aimed at doing just that through low overhead instrumentation, modeling, and optimization. Online optimization provides a good trade-off between static optimization and online heuristics. To enable online optimization, modeling decisions must be fast and relatively accurate.

Online modeling and optimization of a stream processing system first requires the existence of a stream processing framework that is amenable to the intended type of dynamic manipulation. To fill this void, we developed the RaftLib C++ template library, which enables usage of the stream processing paradigm for C++ applications (it is the run-time which is the basis of almost all the work within this dissertation). An application topology is specified by the user, however almost everything else is optimizable by the run-time. RaftLib takes advantage of the knowledge gained during the design of several prior streaming languages (notably Auto-Pipe). The resultant framework enables online migration of tasks, auto-parallelization, online buffer-reallocation, and other useful dynamic behaviors that were not available in many previous stream processing systems. Several benchmark applications have been designed to assess the performance gains through our approaches and compare performance to other leading stream processing frameworks.

Information is essential to any modeling task, to that end a low-overhead instrumentation framework has been developed which is both dynamic and adaptive. Discovering a fast and relatively optimal configuration for a stream processing application often necessitates solving for buffer sizes within a finite capacity queueing network. We show that a generalized gain/loss network flow model can bootstrap the process under certain conditions. Any modeling effort, requires that a model be selected; often a highly manual task, involving many expensive operations. This dissertation demonstrates that machine learning methods (such as a support vector machine) can successfully select models at run-time for a streaming application. The full set of approaches are incorporated into the open source RaftLib framework.

Chapter 1

Introduction

Gordon Moore conjectured that the number of transistors on a computer chip would double every 18 months[157]. This became known as "Moore's Law," which up until recently has been assumed to imply that computing power will roughly double every two years [169]. Increasing frequency and subsequently pipeline depth of processors led the charge to increase computing performance until the start of the 21^{st} century. Diminishing returns due to power limitations has led to the end of this trend. The next (and current) driver of performance is attributed to continuing miniaturization of feature sizes enabling more compute cores to be placed on a chip. Increasing the number of cores on a chip means higher performance for parallel applications, but it also means more power is needed per unit area. With high density compute cores comes the need to dissipate heat efficiently. This problem is known as the "Power Wall" [69]. Yet another limitation, termed the "Memory Wall," came about because memory access times (i.e., latency and throughput) have stagnated while compute power has doubled roughly every two years [200]. Engineers and scientists saw these trends, often long before they were named. Solutions vary widely, from bigger general purpose processors and specialized cores to highly distributed "cloud" computing.

General purpose multi-core processors are excellent at multi-purpose computation, however increasingly more efficient special purpose processors are used for targeted computational tasks, examples include: General Purpose Graphics Processors (GPGPUs) [147], the Xeon Phi [98], customized embedded processors [174] and Field Programmable Gate Arrays (FPGAs) [50]. Most commercially available general purpose processors are actually systems on a chip (or more commonly SoC). Within a SoC multiple sub-processors are found on a chip network, often co-located with multiple memory layers. Systems that include multiple types of processors are generally termed "heterogeneous." This term can be used for systems composed of multiple processor types and SoC solutions (within this work we will use the later rather than the former version). Almost all of these systems are capable of some sort of heterogeneous parallel processing (i.e., executing cooperative code on multiple types of resources concurrently). Increasingly architectures are distributed, even at the SoC level (see compute near memory technologies [80]). At a macro level distributed computation is now mainstream, spawning consumer friendly buzz words like “cloud-computing” and “internet-of-things.” How do we effectively program heterogeneous and distributed systems? In multiple instruction set architectures, byte orders, network types, protocols and a dizzying array of other such terms must come together to form a coherent computing platform on which an end user can compute seamlessly.

1.1 Industry Trends

The era of “big-data” is here to stay. Indeed, what is considered “big-data” today will undoubtedly pale in comparison to what will face us in the coming decades. By-in-large the data we have drives innovation, either directly or indirectly. How innovators process that data will directly reflect the rate of innovation in society. The processing of “big-data” is fundamentally changing the way we live. No longer do we have to drive around for hours for the right adapter or the perfect red shirt, search engines have scoured the web for matches to what we want and quickly show us the prices for each. The programmers of big-data applications are often not computer scientists. They are stock analysts, physicists, etc. Their code works but not nearly as well as well modeled code could. A second group of people spend months (even years) re-factoring applications every time a new supercomputer or HPC cluster is built. Both classes of user could benefit from dynamically adaptive code. If automatically tuned code could perform within a fraction of manually tuned code, then many would benefit. HPC users could save millions of dollars in labor costs. Analytics providers could have faster, more actionable data. Even novice programmers could have more efficient programs.

The vast majority of processing cores in the world today are not large desktop-class cores, but smaller, more energy efficient mobile ones. These devices are in our pockets, our watches, and our glasses. The Internet-of-Things revolution has only just begun, with it the need to quickly process “small-data” streams created by these devices (note: small is relative to true big-data loads which are currently above the terabyte range in many cases[133]). Users want image recognition, voice recognition and many other computationally intensive tasks that are far out of reach with the power levels available to mobile units. The solution is simple: compress and ship the necessary data to processors near the edge of the network. This creates an interesting problem, very similar to the aforementioned heterogeneous compute one. A run-time must find a compute configuration that minimizes latency and maximizes throughput for individual users. The data must be streamed from the user and the processed meta-data streamed back very quickly.

1.2 Paradigm Flux

Effectively programming high performance applications for heterogeneous architectures is currently the domain of expert programmers. One of the best programming paradigms to tackle heterogeneous distributed systems is a very old one: stream processing. Why are the current approaches to programming not effective at managing these systems? Specifying computation has gone from languages specified by human modulated switch positions to vibrant written languages of many types. As the designers of all these languages are (we expect) all human, it is only natural that they are largely sequential in their semantics. Humans spend much of their early brain development learning to focus on a single specific strand of thought, logically maneuvering from one task to the next. Many current and popular languages are excellent for expressing this type of paradigm: C, Java, C++, to name but a few.

Extracting parallelism from sequential code is typically done at the compiler and hardware layers[85, 146, 189]. Modern compilers can perform loop unrolling, blocking and insert single instruction multiple data (SIMD) instructions which can perform vector operations in fewer clock cycles than equivalent single arithmetic operation instructions. Hardware can view dependencies within the instruction window and issue multiple instructions whose dependencies are already met, these are re-assembled to restore the illusion that they were executed in serial. The hardware-only approaches are generally limited to a few processors on shared memory architectures, while compiler based approaches have been shown to scale with stencil computations it remains to be seen if they can scale to multi-node distributed computation [57, 85]. While shared memory architectures have been shown to scale, the performance is generally poor when expanding past a few hundred nodes (even for advanced non-uniform memory access machines [144]). It remains to be seen if newer inter-connect technologies can handle larger distributed shared uniform memory networks.

Explicit user threading is one approach to extract more parallelism from an application, however it is best left to experts who know how to manage non-deterministic behavior inherent with concurrent execution [122]. Approaches such as OpenMP [45] reduce the need to handle threads directly, however they have another issue that limits their performance on large heterogeneous hardware: split/join parallelism. Streaming systems explicitly enable pipelined and task parallelism; tasks can execute as soon as the required resources are available. More distributed approaches such as MPI have done quite well in high-performance and general computation. The code bases underlying most MPI systems are quite optimized through decades of development. MPI programs are often hard to read, and difficult to debug. MPI for high performance computing must also be tuned for each new generation of processor and interconnect combination, this often takes months to years for the types of large scale simulation code used within HPC. MPI also doesn't adapt well to heterogeneous systems where a distant piece of code can't always query an accelerator for a particular piece of data that it needs (or ensure its proximity within the network).

Anyone who has had the experience of management, or operations planning, or any other related field that requires utilizing multiple independent resources knows that scheduling parallel independent tasks is often difficult. It is often far easier to give workers completely independent tasks to work on and coordinate synchronization events when they are complete. This is often wasteful if one worker finishes before the other, however with a steady state of continuous work this happens infrequently. The other possibility is to take some tasks originally assigned to another individual and give them to the free worker (this has been given the term "work-stealing" in computer science [5], although earlier usage of "load-balancing" algorithms that are extremely similar are found in the operations research literature [59]). Assembling these workers into a collective unit gives rise to the classic "job-shop" problem where queueing behavior is observed (more on this later). Stream processing is a paradigm that embodies this thought process.

Stream processing (also termed data-flow programming) is a compute paradigm that views an application as a set of compute kernels connected via communications links or "streams." Each compute kernel is like an independent worker, performing a relatively sequential task (perhaps however with instruction level parallelism) on a finite (but unknown quantity) of streaming data (it is often convenient to think of this stream as infinite with a multi-phase distribution). Streaming languages and libraries include RaftLib [25, 159] (my own), StreamIt [187], S-Net [84], and others. Stream processing is increasingly used by multi-disciplinary fields with names such as computational-x and x-informatics (e.g., biology, astrophysics) where the focus is on safe and fast parallelization of a specific application [130, 190]. With non-expert programmers in mind, intuitive linguistic semantics are quite important to parallel systems. Linguistically, stream processing systems can take many forms. Traditionally programmers assemble building blocks of "kernel" compute units. With RaftLib (described in Chapter 3) I took a mixed approach, attempting to balance C++ style semantics with the more traditional building block approach.

1.3 Turning Streams into Torrents

Many "big-data" applications involve real-time or latency sensitive processing, necessitating usage of many parallel kernels on several compute cores. Optimizing the orchestration of kernels has been the focus of much theoretical and empirical work [19, 20, 23, 124, 148]. Purely theoretical parallel programming models can fail when the assumptions implicit within the model are mis-matched with reality (i.e., the model is incorrectly applied). Full empirical optimization solves this problem by extensively searching the range of configurations under native conditions, this however is expensive in both time and energy. This dissertation focuses on the middle ground; how to quickly apply theory and knows when to fall back to tried and true empirical evaluation.

Many largely theoretical stream processing optimizations focus on synchronous data flow with fixed input and output buffers. By synchronous flow, we assume that there is always only one item removed from the input queue and one item sent to the output queue (and each item is typically the same size). Clearly synchrony and fixed buffers heavily limit the design space over which one can optimize. When these conditions are met, then optimizations can take place statically. What happens when these conditions are not met? Applications such as image processing over the entire web (say picking out red shirts or toasters with associated reviews) are bursty in data-flow and present non-identical workloads to the processors searching them. Clearly a dynamic approach is desirable to optimize this traffic.

The optimization and tuning of stream processing applications on anything other than a single core platform must start with a k-way partitioning of the streaming graph onto each core. Once the k-way partition is made, each compute kernel must be scheduled to run on each core. Multiple strategies exist for partitioning, even some with provable performance bounds (for synchronous data flow applications). Mapping full applications to diverse sets of hardware requires a multi-objective k-way partitioning. One of the key objectives within streaming graph partitioning is to reduce the movement of data and maximize re-use of cache-lines. The partitioning problem itself is provably NP-Hard [76] (although certain special cases can simplify it greatly).

Aside from managing the placement of compute kernels at partitioning, changing the nature of the buffers themselves or even changing the degree of parallelism within the graph can improve or worsen communications costs within the application. Managing communications links within a parallel streaming system is analogous to optimizing a finite capacity queueing network. Optimizing a streaming application is often most efficiently done using a combination of stochastic queueing and other performance models. Performance models of complex systems are often computationally intensive [63], hard to validate, and error prone. As systems progress from the relatively simple (single processor) to the extremely complex (k non-uniform processors) they become extremely hard to model and therefore optimize. Essentially a "Modeling Wall" is reached. With increasing complexity, comes increased modeling cost. What we propose to break this wall is an adaptive process that utilizes online, dynamic optimization. Selecting models by hand for an online tuning process is clearly impractical, humans are too slow too make effective decisions. Deciding what to model and how must be done by algorithm. This dissertation demonstrates that, for the stream processing paradigm, machine learning techniques can be effective at online performance model selection for a target application and platform.

Parallel processing systems and stream processing systems in particular must adapt to changing environments. Cloud computing is largely supplanting dedicated high performance compute clusters. The workloads that parallel systems must manage are also not static, they change from one execution to the next and within a single execution. We demonstrate that a modified Levy distribution can approximate the distribution of the "best-case" execution time variation for an application running on a multi-core processor. This demonstrated that multiple types of hardware (`x86`, `ARM`, `PowerPC`) and schedulers exhibit a quantifiable and regular noise pattern that can be used to accept or reject performance models. We also demonstrate that two differing machine learning (ML) techniques, Support Vector Machine (SVM) and an Artificial Neural Network (ANN), can distinguish similar patterns which can be used to reliably classify stochastic queueing models into two categories “use” and “don’t use.” The ML process takes the place of someone trained to utilize stochastic models and performs the same selection work in microseconds instead of minutes. This enables fast online model selection, making dynamic optimization and re-optimization possible.

In order for online modeling to be practical, while still viewing each kernel as a “black-box,” instrumentation is required to provide data to the modeling system. For streaming systems some metrics of interest are: queue occupancy, process distribution, non-blocking service rate, to name but a few. A very real risk is that the monitoring used to inform dynamic change utilizes more compute power and bandwidth than the computation itself. In order to minimize the overall effect of instrumentation, we develop a series of techniques to measure (and where necessary approximate) the values of interest with little overhead. Instrumentation combined with analytic models and machine learning techniques described in this dissertation can enable adaptive monitoring of the streaming system and placement of instrumentation only where necessary (e.g., the variables of interest have a high probability of changing, or there is a high probability that the model used to build the system is incorrect). As an example, if a tandem queue is operating at a low utilization then most basic queueing models can be used to give a very stable result. Systems operating outside of this low-utilization range require more information to make a good estimate of buffer size. To quote Rumsfeld “There are known unknowns. That is to say there are things that we now know we don’t know. But there are also unknown unknowns.” [163] This dissertation shows that we can instrument for the known unknowns and adaptively target some of the unknown unknowns without inadvertently influencing the behavior of the application.

Online modeling of streaming systems involves many pieces (machine learning), adaptive efficient instrumentation, and a good starting partition. This dissertation takes each of these pieces and puts them together to produce applications that are performant and straightforward. Easier to use modeling tools such as the one described within this work, enable novice users to write relatively performant distributed parallel streaming applications for today’s modern hardware.

1.4 Contribution and Structure

This dissertation makes the following contributions to the online modeling and execution of stream processing systems:

1. We developed the RaftLib stream processing library for C++ [25]. RaftLib is built as a template library, enabling end users to utilize the robust C++ standard library along with RaftLibs streaming / data-flow parallel framework. It is designed to be both a research vehicle and a productivity tool for parallel programmers. RaftLib supports dynamic queue optimization, automatic parallelization, real-time low overhead performance monitoring, heuristically optimized partitioning, and online modeling of performance (through techniques described in this work).

2. Finding an application topology mapping that maximizes throughput on a given set of hardware is a hard problem. We show that with a few modifications, a generalized gain/loss maximum flow model is appropriate to model the throughput of a queueing network [20, 19] under certain conditions.

3. We show that an appropriately modified Levy distribution (to ensure finite support) can approximate a lower bound on the distribution of the variation expected from a process executing on modern multi-core hardware. The modified Levy distribution is reparameterized in terms of the target modeling application so that it can be used without expensive model fitting. This distribution we call mLevy. We go on to show how mLevy can be used to accept or reject a queueing model based [21].

4. Many models commonly used for streaming applications require some idea of the non-blocking service rate (i.e., how fast a compute kernel is able to execute in isolation. Many of the models discussed within this dissertation require it. We give a technique that approximates the non-blocking service rate while the application is executing with extremely low overhead. We also give an implementation of this technique within the RaftLib streaming framework [22].

5. Many methods are available to assess when a model can be trusted (i.e., where is the error within a given model). Generally these methods are used by a human, guided by intuition and based on the overall problem structure. Many cases are easy to find, such as when assumptions used to derive a model are broken (e.g., exponential inter-arrival

and service time distributions for a $M/M/1$ queueing model). Having a human in the loop isn't fast enough to adapt while an application is executing. We show that two differing machine learning methods can take a fingerprint, consisting of hardware and application features, to decide if a model is to be trusted or not. We show that this method is effective when utilizing both Support Vector Machines and Artificial Neural Networks [23, 24].

6. What can be done at run-time to optimize the execution of an application? Often is the only time a streaming application can be optimized. What can be done when modeling assumptions turn out to be false during execution? We show that using an initial mapping and partition as a starting point, local search with the aid of our online modeling techniques and instrumentation can effectively optimize the application with little additional overhead (often within a very high percentage of optimal, see Chapter 8).

In the following chapters I'll delve deeper into the background for this work, describe the RaftLib C++ streaming library, several modeling techniques, a distribution to describe minimal execution time variation, dynamic instrumentation, and machine learning for online model selection. The final chapter will put all these elements together and describe how they coalesce to make a state of the art streaming framework. Lastly we'll conclude by discussing future directions, and where we believe the future of parallel processing will lead.

Chapter 2

Background and Related Works

The background and related works of others for the contributions contained within this thesis is quite deep and must obviously be truncated at some point. In this chapter we will quickly go through the necessary background, history and related works for each contribution in their order of presentation. We will outline the differences from prior techniques to those presented here, if necessary (or expedient) bits of background information have been included within the appropriate chapters, this chapter is intended to be a bit more comprehensive.

2.1 Stream Processing

The concept behind stream processing is a very simple one. By connecting compute kernels via buffers so that each compute kernel can compute independently, a model to construct parallel applications emerges. It naturally arises from the job shop [96] (or assembly line model) common to modern factories throughout the world. The first data flow language is believed to be credited to Jack Dennis [60]. A highly cited early summary by David Culler [56] provides a brief overview of both data flow hardware and the stream processing paradigm. Since Culler's synopsis there has been an explosion of data flow hardware, languages, and libraries both academic and commercial. Before data-flow languages or stream processing, data-flow was the domain of hardware.

Much work has been done in the field of data-flow hardware. It is largely tangential to this work, however many of the modeling techniques apply to software stream processing systems as well. Early hardware in this field is described by Papadopoulos [151]. The most notable hardware to date has been produced through the Imagine streaming engine project from

Stanford University [106]. True data-flow processors have not gained commodity acceptance for various reasons, however the basic concept of non-Von Neumann data-flow architectures remains promising for performance, energy, and general efficiency reasons. The most recent example of a data-flow architecture combines the low latency Von Neumann architecture with a high throughput data-flow processor, dynamically sending instructions to each as necessary [146]. Our work, both Auto-Pipe and our successor RaftLib (see Chapter 3) execute the data-flow model on top of the Von Neumann abstraction. Linguistically and conceptually stream processing languages, models, and techniques are generally transferable from Von Neumann architectures to data-flow ones. The stream processing paradigm (data-flow) is also ideal for exploiting massively parallel computation due to many salient features discussed within Chapter 3. Next we will discuss the multitude of recent notable stream processing languages (both academic and commercial).

Brook [35] is a language designed for streaming processing on GPUs. StreamIt [187] is a streaming language and compiler based on the synchronous data-flow model. Google Cloud Dataflow [155] is another proprietary stream processing system. Storm [181] and Samza [164] are open-source streaming platforms that are focused on message-based data processing. RaftLib and the other truly "streaming" languages differ from these in that they use point to point communication instead of centralized brokers to distribute data. A central broker makes providing fault tolerance simpler, however it also stifles many opportunities for optimization available to traditional streaming systems. Another potential disadvantage is that queueing behavior within the backing store could require infinite storage when consumers cannot keep up with the volume of data [111]. Blocking and decentralized control of classic streaming systems enable throttling at a local level eliminating the potential for an infinitely sized data broker.

Many of the full languages and frameworks have until recently been suited only to "niche" applications, often with steep learning curves. RaftLib's advantage over them is that a C++ template library is easy to integrate with legacy code, enabling more general usage. ScalaPipe [198] and StreamJIT [28] are two similar language extensions for streaming processing with Scala and a Java front-ends, respectively. Other C++ parallelization template libraries include Threading Building Blocks [162] and Concurrency Collections [110], which are both Intel products. RaftLib differs from the last two in that it aims to provide a fully integrated, distributed, and dynamically optimized stream parallel platform.

2.2 Modeling

Streaming applications can be thought of as a series of queues and servers. Each compute kernel is a server which draws data from a queue. The background needed for most of the queueing models utilized within this thesis are given by Kleinrock [109], although they are also covered in more recent texts such as [88, 179]. Notation for many of these models comes from the classic work of Kendall [103]. Necessary explanations are included as needed within the text. Work by Dor et al. [64] shows that simple queueing networks can accurately model the performance of a heterogeneous streaming application. Many earlier works, including Schweitzer [172], demonstrated that maximum throughput can be determined analytically for a finite-capacity open queueing network. These works have shown that queueing networks can be used for modeling throughput, however they often assume that the queueing capacity is known. Lancaster et al. [117] showed that virtual queues have many of the same properties as a single abstract queue and associated server. The model presented in Chapter 4 solves for maximum throughput assuming unbounded buffering capacity and then follows with an analysis to capacitate them.

One of the primary themes throughout this work is buffer allocation. Determining the correct size of buffers within an open queueing network has been the subject of much research over the past half century (see Cruz et al. [55], for a summary of applications or Smith and MacGregor [177] for $M/G/1/K$ approaches). A general summary on the buffer allocation problem (or BAP, as it is often abbreviated) including its hardness in general is described by Anantharam [9]. Simulation methods to solve the general cases are common, some highly cited examples are [8, 152]. There are also analytic approaches for probabilistic upper bounds, the easy to derive $M/M/1$ is given by Kleinrock [109] for single stations and methods to find the allocation for some networks are given by Neuts [143]. There are also "rule-of-thumb" methods such as those given by Anantharam and Ganesh [10]. One thing that each of these methods has in common is that they require characterization of the distribution of each process (often including multiple central moments). These methods are often expensive in their computation, especially the simulation approaches, which are not suitable for time sensitive online modeling tasks. Our buffer allocation techniques differ from these previous approaches in that we focus on simple models (often separable) that can be executed quickly and give a good enough approximation for relatively high performance.

Queueing networks have a close relationship with flow networks. Recent work by Boudec [121] considers not individual jobs on a network but flows of jobs within a network. Work by Pourbabai [156] utilizes a maximum flow model to solve a queuing network with side constraints. Unlike the target of these works, most real applications have data-flow routing requirements that are critical to the correctness of the application. For example, the RGB2YCbCr module in Figure 4.1 takes in a stream of RGB data and outputs three separate streams of Y, Cb and Cr data. A typical formulation of the maximum flow problem, assume that any path from source to sink can be taken. That is they assign maximum flow to a graph without regard to application imposed data distribution requirements. Using a standard maximum flow model with no further constraints might result in all the data being sent along the Y channel but none to the Cb and Cr channels. The flow model described in Chapter 4 places volume constraints on each out edge that are derived directly from data routing requirements of the underlying application. This is distinct from the previous work of Pourbabai [156] in that we have added routing constraints so that this method is usable for compute applications.

Many applications exhibit some form of data filtering, that is they change the form of the data from that which is originally received. Two ways in which applications can change data include: increasing or decreasing the volume of data (e.g., a basic block that calculates matrix eigenvalues might take in a grid of data and output a short vector of values) or changing the width of the individual data elements (e.g., expanding a single unsigned byte to a 64-bit floating point value). Filtering presents an interesting problem for standard maximum flow algorithms [71, 73]. Work by Jewell [99] outlined algorithms for calculating a maximum flow of a network with gain or loss. Using the theoretical work of Jewell the flow model utilized in Chapter 4 is a generalized gain/loss flow network with a fixed branching probability at each out-edge. Chapter 4 uses the gain loss foundations of Jewell [99] directly (with optimizations due to Goldberg et al. [81] and later work by Goldfarb et al. [82]), as they are relative optimal. The distinction of this work is that it is the first to use this type of model applied to an open queueing network and streaming systems. A slightly tangential use of flow models within performance engineering is for multi-processor scheduling [180].

More advanced methods of modeling behavior of applications on shared resources have been used and shown to be relatively effective [39, 180]. Contrary to these complex models, the methods outlined in Chapter 4 demonstrate that simple models can be as effective as the former complex ones for certain classes of applications.

2.3 Instrumentation

At its core, Chapter 6 is about low-overhead instrumentation of software systems. The same techniques could also be used for co-designed hardware/software systems. Many others have produced low overhead instrumentation systems. Amongst the earliest type of performance oriented instrumentation tools were call graph tools such as `gprof` [83]. Other instrumentation tools such as TAU [175] provide low overhead instrumentation and visualization for MPI style systems. What these tools don't provide is a mechanism for reporting performance during execution (which our system does). The TimeTrial performance monitor [116], which Chapter 6 describes an extension of, laid the groundwork for real time monitoring for streaming systems.

Modern stream processing systems such as RaftLib (see Chapter 3) can dynamically re-optimize in response to changing conditions (workload and/or computing environment). To re-optimize buffer allocations there are generally two choices, either branch and bound search or analytic queueing model. Branch and bound search has the disadvantage of requiring multiple allocations and re-allocations until a semi-optimal buffer size is found. Analytic queuing models are highly desirable for this purpose since they can divine a buffer size directly, eschewing many unnecessary buffer re-allocations. Compute kernel mean service rates are, at a minimum, typically required for these types of models. Utilizing these models dynamically therefore requires dynamic instrumentation. Tools such as DTrace [38], Pin [132], and even analysis tools such as Valgrind [141] can provide certain levels of dynamic information on executing threads. Other, more modern performance monitoring tools of note are Paradyn [137] and Scalasca [78]. These toolkits provide a multitude of information for parallel systems, however not quite the same type of information that our instrumentation provides. They've pioneered things like trace compression for instrumentation, however we're interested in eliminating traces all together by using the data in real time then throwing it away. Another way our approaches differ from the aforementioned ones is that we are specifically targeting methods for estimating online service rate in a low overhead manner, as opposed to being a general purpose instrumentation toolkit.

Work by Lancaster et al. [118] laid out logic that could ostensibly make online service rate determination possible. They suggest measuring the throughput into a kernel when there is sufficient data available within it's input queue(s) and no back-pressure from its output queue(s). This logic works well for FPGA-based systems where hardware is controlled by the developer. For multi-core systems, however, this logic breaks down, for several reasons which are enumerated within this section. The need for low overhead online service rate determination motivates this work.

The work of Lancaster et al. [42, 118] assumes that the measurements of a non-blocked service rate are all equal (i.e., the full service rate is observed at every sample point). In reality, modern parallel system conspire against this logic with delays due to NUMA, coherence, etc. making observations gleaned from this logic unusable (due to many factors, discussed in detail within Chapter 6). This work is distinct from these previous works, while we marginally extend the logic, in Chapter 6 a technique is described to approximate the non-blocking service rate while an application is executing (online), enabling optimizations not possible using previous techniques.

2.4 Online Modeling & Performance Tuning

There has been considerable work investigating the efficient execution of streaming applications, both on traditional multi-cores and on heterogeneous compute platforms, from the early work on dedicated data flow engines [61], to the synchronous data flow model [123]. Lancaster et al. [116, 117] have built low-impact performance monitors for streaming computations, and Padmanabhan et al. [148, 149, 150] have shown how to efficiently search the design space given a model that connects tuning parameters to application performance. Beard and Chamberlain [19] showed how to approximate flow through a streaming application efficiently. RaftLib intends to leverage and expand upon the above work as it seeks to efficiently execute streaming applications.

In order to dynamically "tune" RaftLib, online instrumentation is required. Much previous work has been done in this area as well, although not as much for streaming systems. Tools such as DTrace [38], Pin [132], and even analysis tools such as Valgrind [141] can provide certain levels of dynamic information on executing threads. The approach taken by RaftLib differs from the aforementioned ones in that we are estimating the service rate while the application is executing and the instrumentation can be turned on and off as needs dictate.

In order to tune an application, models are typically used. When do we trust the model? When to use and not to use a performance model has been studied extensively [88, 97, 179].

Model selection and validation in general has also been studied quite heavily over the last century [30]. Multiple related fields from Operations Research [119] to agriculture [77] all rely on models and their subsequent validation. Most of the methods in the aforementioned resources rely on either physically observing or simulating a model in order to validate it. For most applications, this is quite expensive (in terms of time and labor). Validation of simulation models has also been covered quite well by the works of Sargent [167, 168]. The methods described in Chapter 7 differ in that they are not meant to be perfect validation, rather a highly probable and fast result.

2.5 Dynamic Adaptation

Dynamically adapting to changes in the environment for software systems is not a new concept, in and of itself. There have been several realizations of this idea. One of the most notable are "hot-spot" compilers which re-compile segments of code based upon its relative frequency of execution and other cost factors. One such scheme is the Jalapeño JVM [14]. There is also the classic Sun Microsystems developed hotspot compiler [142]. Before hotspot compilation was a very similar concept, iterative compilation [108], which searches the design space of instruction sequences to find the best sequence for a target architecture. These systems make code changes, our code changes are limited in nature. They are not intended (in general) to extract more parallelism from code, we do. They are also designed to work with intermediate byte-code (save for the earlier iterative compilation works) which gives a JVM based adaption the benefit of having essentially an original source to re-compile at will to a given architecture. What these systems don't necessarily take into account is the performance of threads within a system and attempt to optimize the whole, this is where our work distinguishes itself.

Other forms of adaption take the form of compile time static adjustments to code. The most notable example is the ATLAS BLAS package [196]. Several others have developed statistical methods to reduce the search space for this type of offline search for optimal library code [32, 194]. How this work differs from these is that we focus no on optimizing the generation of code itself, but tuning the topology so that the parallel execution itself is performant.

Other forms of adaptation take place dynamically for fault tolerance reasons. There are both active and passive forms of fault tolerance [18, 31, 93, 173] , as well as hybrid approaches [203]. While fault tolerance is a necessary trait of a high performance distributed system, it is out of scope. This work makes no claims that ensure fault tolerance as these methods do, we aim to provide performance tolerance.

Chapter 3

RaftLib Streaming Library

Decries touting the end of frequency scaling and the inevitability of a massively multi-core future are frequently found in current literature [69]. Equally prescient are the numerous papers with potential solutions to programming multi-core architectures [13, 125, 154, 201]. One of the more promising programming modalities to date is a very old one: stream processing [60, 135] (the term "stream processing" is also used by some to refer to online data processing [40, 70]). Until recently it has garnered little attention, we hope to help change that by enabling performant and automatically tuned stream processing within the highly popular C++ language.

Stream processing is a compute paradigm that views an application as a set of compute kernels (also sometimes termed "filters" [187]) connected by communication links that deliver data streams. Each compute kernel is typically programmed as a sequentially executing unit. Each stream is a first-in, first-out (FIFO) queue whose exact allocation and construction is dependent upon the link type (and largely transparent to the user). Sequential kernels are assembled into applications that can execute in parallel. Figure 3.1 is an example of a simple streaming `sum` application, which takes in two streams of numbers, adds each pair, and then writes the result to an outbound data stream.

A salient feature of streaming processing is the compartmentalization of state within each compute kernel [3], which simplifies parallelization logic for the run-time [61] as well as the programming API (compared to standard parallelization methods [4]). Stream processing has two immediate advantages: 1) it enables a programmer to think sequentially about individual pieces of a program while composing a larger program that can be executed in parallel, 2) a streaming run-time can reason about each kernel individually while optimizing globally [148]. Moreover, stream processing has the fortunate side effect of encouraging

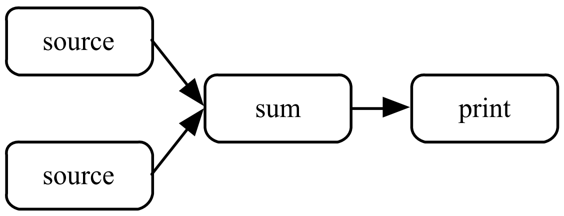


Figure 3.1: Simple streaming application example with four compute kernels of three distinct types. From left to right: the two source kernels each provide a number stream, the "sum" kernel adds pairs of numbers and the last kernel prints the result. Each kernel acts independently, sharing data via communications streams depicted as arrows.

developers to compartmentalize and separate programs into logical partitions. Logical partitioning is also beneficial for the optimization and tuning process.

In addition to simpler logic, stream processing also enables easier heterogeneous and distributed computation. A compute kernel could have individual implementations that target an FPGA and a multi-core running within the same application, known as "heterogeneous computation" or "hybrid computing" [43]. As long as the inputs and outputs are matching, the application will run correctly regardless of which resource a kernel is scheduled to execute on. Brook [35], Auto-Pipe [72], GPU-Chariot [74], and ScalaPipe [198] are examples of such systems. Stream processing also naturally lends itself to distributed (network) processing, where network links simply become part of the stream.

Despite the promising features of stream processing, there are hurdles that affect programmers' decision to use the paradigm. One hurdle to adoption is communication cost. There are many potential solutions to reducing communications cost such as shared stack space/private stack combinations which facilitate fast lock-free inter-thread communication and the more traditional (albeit difficult) optimization of FIFOs through queueing network models (this work focuses primarily on the later). Substantive solutions to reducing communications cost are hampered by the sheer difficulty of fully tuning a general parallel application. Optimizing the communications cost requires a graph-partition with multiple objectives which is NP-hard for the general case [76]. Much work has been done in the VLSI community for the similar problem of optimal chip layout. A second hurdle is simply the bulk of legacy code and the requirement on the part of most streaming frameworks that applications be re-authored or substantially modified to conform [136]. The most popular languages for the past two decades have been C, C++ and Java [188] (each designed for a sequential mode of execution). One way to reach a tipping point towards acceptance of stream processing is to conform to one of these languages, and that is the path that we chose.

RaftLib [159] is a C++ template library aimed at enabling safe and fast stream processing. By leveraging the power of C++ templates, RaftLib can be incorporated with a few function calls and the linking of one additional library. It is completely self container, only a C++ compiler is needed. RaftLib aims to transparently parallelize an application, while minimizing re-factoring for legacy code. RaftLib is an online auto-tuned streaming system. As such it tries to maximize throughput of the application by: adaptively scheduling compute kernels, providing low overhead instrumentation to the run-time so it can make informed decisions, and selecting when to model and how to model each part of the streaming system. RaftLib dynamically optimizes the streaming system in a multitude of ways, drawing on research from past works described within Chapter 2 and the work within this dissertation. Machine learning techniques (Chapter 7) are used to model buffers within the streaming graph and select progressively more appropriate buffer sizes while the application is executing. The framework incorporates low overhead instrumentation which can be turned on and off dynamically to monitor such metrics as queue occupancy, non-blocking service rate, and utilization. All of these pieces put together make a highly usable and adaptive stream parallel system which is integrable with legacy C++ code.

3.1 Design considerations

Several properties of streaming applications that must be exploited or overcome by streaming systems have been noted by others. The stream access pattern is often that of a sliding window [186], which should be accommodated efficiently. RaftLib accommodates this through a `peek_range` function. Streaming systems, both long running and otherwise, often must deal with behavior that differs from the steady state [186, 126] (see also Chapter 5). Non-steady state behavior is often also observed with data-dependent behavior, resulting in very dynamic I/O rates (behavior also observed in [186]). This dynamic behavior, either at start-up or elsewhere during execution, makes the analysis and optimization of streaming systems a slightly more difficult prospect, however it is not insurmountable. We will demonstrate empirically how RaftLib handles dynamic rates through a text search application at the end of this chapter. For example, text search has the property that while the input volume is often fixed, the downstream data volume varies dramatically with algorithms, which heuristically skip, as does the output (pattern matches). Kernel developers, as should be the case, focus on producing the most efficient algorithm possible for a given kernel. Despite this, kernels can become bottlenecks within the streaming system. RaftLib dynamically monitors the system to eliminate the bottlenecks where possible.

At one time it was thought that end users were probably best at resource assignment [58], whereas automated algorithms were often better at partitioning an application into compute kernels (synonymous to the hardware-software co-design problem discussed in [12]). Anecdotal evidence suggests that the opposite is often true. Programmers are typically very good at choosing algorithms to implement within kernels, however they have either too little or too much information to consider when deciding where to place a computation and how to allocate memory for communications links. There is simply too much information which changes too quickly for a human to comprehend in real-time. The placement of each kernel changes not only the throughput but also the latency of the overall application. In addition, it is often possible to replicate kernels (executing them in parallel) without altering the application semantics [127]. RaftLib exploits this ability to blend pipeline and data parallelism as well.

3.2 RaftLib description

Writing parallel code traditionally has been the domain of experts. The complexity of traditional parallel code decreases productivity which can increase development costs [90]. The streaming compute paradigm generally, and RaftLib specifically, enables the programmer to compose sequential code and execute not only in parallel but distributed parallel (networked nodes) using the same code.

RaftLib has a number of useful innovations as both a research platform and a programmer productivity tool. As a research platform, it is first and foremost easily extensible; modularized so that individual aspects can be explored without a full system re-write. It enables multiple modes of exploration: 1) how to effectively integrate pipeline parallelism with standard threaded and/or sequential code, 2) how to reduce monitoring overhead, 3) how best to algorithmically map compute kernels to resources, 4) how to model streaming applications quickly so that results are relevant during execution. It is also fully open source and publicly accessible [159]. As a productivity tool it is easily integrable with legacy C++ code. It allows a programmer to parallelize code in both task and pipelined fashions.

We introduce RaftLib via the following example application. The `sum` kernel from Figure 3.1 is an example of a kernel written in a sequential manner (code shown in Figure 3.2). It is authored by extending a base class: `raft::kernel`. Each kernel communicates with the outside world through communications “ports.” The base kernel object defines `input` and `output` port user accessible objects. These are inherited by sub-classes of `raft::kernel`. Port container objects can contain any type of port. Each port itself has the behavior of a FIFO queue. The constructor function of the `sum` kernel adds the ports. In this example, two input ports are added of type `A` & `B` as well as an output port of type `C`. Each port gets a unique name which is used by the run-time and the user. The real work of the kernel is performed in the `run()` function which is called by the scheduler. The code within this section can be thought of as a “main” function of the kernel. Input and output ports can access data via a multitude of intuitive methods from within the `run()` function. Accessing a port is safe, free from data race and other issues that often plague traditional parallel code [15]. Figure 3.3 shows the full application topology from Figure 3.1 assembled in code. Assembling the topology is akin to connecting a series of actors, each sequential in an of themselves but executing independently. Each call to the link function connects the specified ports from the source and destination kernels. The function call returns a structure with references to the linked source and destination kernels for re-use by the programmer if needed. The run-time itself brings the parallel power to these sequential actors.

Once the kernel “actors” are assembled into a full application, the run-time starts to work parallelizing the application with the `exe()` function call. This feat is performed by mapping kernels to appropriate resources, sizing buffers, selecting the appropriate algorithm when more than one exists, scheduling kernels for execution, and tuning any remaining performance impacting run-time parameters.

Scheduling, mapping, and queueing behavior are each important to efficient, high-performance execution. RaftLib is intended to facilitate empirical investigation within each of these areas. RaftLib implements a simple but effective scheduler that is straightforward to substitute with new algorithms. Similarly, the modular mapping algorithms used in RaftLib can easily be altered for comparative study. Each communication link between compute kernels exhibit queueing behavior. RaftLib serves as a platform for optimizing the queueing network,

```
template< typename A, typename B, typename C > class sum : public raft::kernel
{
public:
   sum() : raft::kernel()
   {
      input. template addPort< A >( "input_a" );
      input. template addPort< B >( "input_b" );
      output.template addPort< C >( "sum" );
   }

   virtual raft::kstatus run()
   {
      A a;
      B b;
      input[ "input_a" ].pop( a );
      input[ "input_b" ].pop( b );
      auto c( output[ "sum" ].template allocate_s< C >() );
      (*c) = a + b;
      return( raft::proceed );
   }
};
```

Figure 3.2: A simple example of a `sum` kernel which takes two numbers in via `input_a` and `input_b`, adds them, and outputs them via the `sum` stream. The `allocate_s` call returns an object which releases the allocated memory to the downstream kernel with the call of its destructor as it exits the stack frame.

not only statically but dynamically. RaftLib supports continuous optimization of a host of run-time settable parameters.

There are many factors that have led to the design of RaftLib. Chief amongst them is the desire to have a fully open source framework to explore how best to integrate stream processing with legacy code. Secondly it serves as an experimental platform for investigating optimized deployment and optimization of stream processing systems. In the following sections we discuss why we need the features included in the library, the science and engineering behind them and some examples of how those features are executed by the user. This will be followed by benchmarking a text searching application against other leading parallel text search applications.

```
const std::size_t count( 100000 );
using ex_t = std::int64_t;
using gen  = raft::random_variate< ex_t, raft::sequential >;
using sum  = sum< ex_t, ex_t, ex_t >;
auto linked_kernels( map.link( kernel::make< gen >( 1, count ),
                               kernel::make< sum >(), "input_a" ) );
map.link(   kernel::make< gen >( 1, count ),
            &linked_kernels.getDst(), "input_b" );
map.link( &linked_kernels.getDst(), kernel::make< print< std::int64_t ,'\n'> >() );
map.exe();
```

Figure 3.3: Example of a streaming application map for a "sum" application (topology given in Figure 3.1). Two random number generators are instantiated, each of which sends a stream of numbers to the `sum` kernel which sends the sum to a `print` kernel. The call to `link()` returns a struct (`linked_kernels`) with references to the kernels used within the `link()` function call (`linked_kernels.getSrc()` and `linked_kernels.getDst()` respectively) so that they may be referenced in subsequent link calls.

3.2.1 RaftLib as a research platform

As a research platform, RaftLib is designed to enable the investigation of a number of questions that impact the performance of streaming applications. We will address a number of these questions in the paragraphs below, with a focus not on the answer to the research question, but instead on how RaftLib facilitates the investigation.

We start with the ability to blend pipeline parallelism with data parallelism. Some applications require data to be processed in order, others are okay with data that is processed out of order, yet others can process the data out of order and re-order at some later time. RaftLib accommodates all of the above paradigms. Streams that can be processed out of order are ideal candidates for the run-time to automatically parallelize. Li et al. [127] describe algorithms for replicating kernels in a pipelined environment, both for homogeneous compute resources and for heterogeneous compute resources. Parallelization decisions can also be guided empirically through low-overhead instrumentation described below.

Automatic parallelization of candidate kernels is accomplished by analyzing the graph for segments that can be replicated preserving the application's semantics (indicated by the user specifying out of order processing at link time with the appropriate template parameter). As part of the graph analysis process, single entry single exit [100] segments are identified (with respect to user indicated out of order links) and indexed as potential parallelization points. Split and reduce adapters are inserted where needed. Custom split/reduce objects can be created by the user by extending the default split/reduce objects. Split data distribution can be done in many ways, and the run-time attempts to select the best amongst round-robin and least-utilized strategies (queue utilization used to direct data flow to less utilized servers). As with all of the specific mechanisms that we will discuss, each of these approaches is designed to be easily swapped out for alternatives, enabling empirical comparative study between approaches.

Given an application topology to execute, the kernels need to be assigned to specific compute resources and scheduled for execution. Scheduling of compute kernels within a streaming application has been the subject of much research. Conceptually it has two parts, initial resource assignment or “mapping” of kernels to compute resources and then scheduling the kernels. The abstract layout of the RaftLib scheduler is depicted in Figure 3.4. The problem of partitioning and mapping a streaming application to compute resources is nearly identical however to the decades old problem of partitioning and mapping a circuit. The graph partitioning problem itself is NP-hard in general, however several heuristics have been developed that are quite good. Derivatives of the Kernighan-Lin algorithm [104] are used for the initial partition. The heuristic originally developed within their seminal work to minimize communications cost between graph partitions and maximize within partition communication is ideal for improving data-locality when scheduling. RaftLib uses a multi-partitioning strategy for the initial placement from the work of Sanchez [166] to maximize within core and minimize between core communications. An online global scheduler (one of many which can be used as a tuning knob) has the option of using the OS scheduler (one thread per kernel), round-robin, work-stealing and cache-weighted work stealing. Cache-weighted work stealing simply adds a weight to encourage the scheduler to steal across cores only if the utilization advantage gained by stealing is greater than a given threshold. Local, within-thread schedulers can also be used although the best scheme for an optimal local scheduler has still to be researched. Currently a simple, but effective, round robin policy is used.

As illustrated in Figure 3.5, the allocated size of each queue of a streaming application can have a significant impact on performance (the data from the figure is drawn from a matrix multiply application, performance based on overall execution time). One would assume perhaps that simply selecting a very large buffer might be the best choice, however as shown

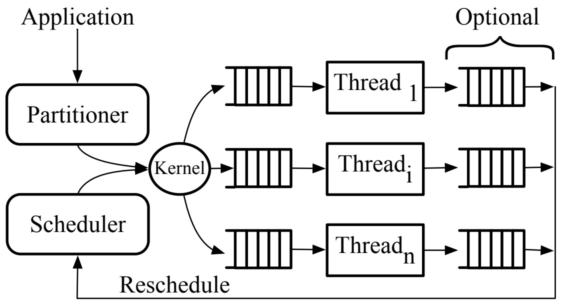


Figure 3.4: The RaftLib scheduler is designed to allow testing and evaluation of multiple schedulers. A basic graph partitioner clusters the graph to minimize communications between cores and maximize caching. Once partitioned, compute kernels are sent to executing threads. The threads themselves have a scheduler which decides which kernels to execute. Optionally the thread can send a kernel back to a "global" scheduler to load balance the application.

the upper confidence interval begins to increase after about eight megabytes. Queueing models are often the fastest way to estimate an approximate queue size, however service rates and their distributions must be determined, which is hard to do during execution. In general, two options are available for determining how large of a buffer to allocate: branch and bound search or analytic modeling. Branch and bound searching has the advantage of being extremely simple, and eventually finds some reasonable condition. If the queue is destined to be of infinite size, a simple engineering solution presents itself in the form of a buffer cap. Model based solutions are also often straightforward to calculate (assuming an appropriate model can be selected), if the conditions are right for considering each queue individually (e.g., the queueing network is of product form).

While treating compute kernels as a "black" box, queue sizing approaches must accommodate program correctness. If a kernel asks to receive five items and the buffer size is only allocated for two, the program cannot continue. RaftLib deals with this by detecting this condition with a monitoring thread, updated every $\delta \leftarrow 10\ \mu s$. When conditions dictate that the FIFO needs to be resized, it is done using lock-free exclusion and only under certain conditions (to maximize resizing efficiency). The resizing operation is most efficiently accomplished when the read position is less than the write pointer (i.e., the queue or ring-buffer is in a non-wrapped position). There are multiple conditions that could trigger a resize and they differ depending on the end of the queue under consideration. On the side writing to the queue, if the write process is blocked for a time period of $3 \times \delta$ then the queue is resized. On the read side, if the reading compute kernel requests more items than the queue has available then

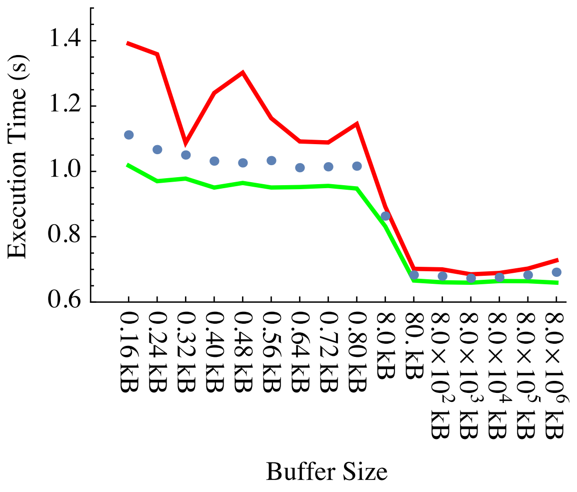


Figure 3.5: Queue sizes for a matrix multiply application, shown for an individual queue (all queues sized equally). The dots indicate the mean of each observation (each observation is a summary of 1k executions). The red and green lines indicate the 95^{th} and 5^{th} percentiles respectively. The execution time increases slowly with buffer sizes $\geq$ 8 MB, as well as becoming far more varied.

the queue is tagged for resizing. Temporary storage is provided on the receiver end to ensure the request is fulfilled as well as ensure that conditions for fast resizing are met. Further queueing optimizations are discussed in detail below and within subsequent chapters.

In addition to modeling individual pieces of the application (i.e., branch and bound search, etc.), RaftLib also can model the throughput of the entire application. Chapter 4 demonstrates the use of flow models to estimate the overall throughput of a streaming application. This procedure requires estimates of the non-blocking service rate of each compute kernel within the streaming system (techniques to do just that are described in Chapter 6). The flow-model approximation procedure can be combined with other well known optimization techniques such as simulated annealing or analytic decomposition [148, 149, 150] to continually optimize long-running high throughput streaming applications.

Performance monitoring is critical to selection of algorithms within an application. It is also central to the automated tuning and modeling that is part of RaftLib. As such the user has access to monitor useful things, such as queue size and current kernel configuration, as they are updated by the run-time. In addition to these, more exciting statistics such as mean queue occupancy, service rate (both instantaneous and time averaged), throughput, and queue occupancy histograms are available. The data collection process itself is optimized to reduce overhead and has been the subject of much research [116, 117] (see also Chapter 6).

Streaming systems can be modeled as queueing networks [120, 64, 19, 23]. Similar phenomena are observed in fork-join networks [17], supply chain management optimization [176] and operations optimization [202]. Each stream within the system is a buffer (which are modeled as queues). Sizes of buffers within the application can have a notable effect on application performance. Specifying buffers that are too small will create bottlenecks where otherwise none would exist. Conversely selecting buffer sizes that are arbitrarily large can decrease overall performance (through ancillary effects such as: more page-ins, cache over-runs, etc.). The effects of buffer sizing are shown empirically in Figure 3.5. RaftLib takes scheduling of compute kernels, allocation of buffers (queues), and resource mapping out of the user's hands. Once a compute mapping is defined, the run-time (through heuristics, machine learning, and/or mathematical modeling) attempts to keep the application performing optimally.

The queueing network that is a streaming application can be tuned via a stochastic queueing model. Doing so, however, often requires information, such as the service rate of each kernel, not typically available at run time (online). Stochastic models are desirable because they are much faster than the alternatives, e.g., branch-and-bound search, which require many memory reallocations. Complicating matters for online tuning of buffer size, many analytic methods used to require an understanding of the underlying service process distribution, not just it's mean. Both service rate and process distribution can be extremely difficult to determine online without affecting the behavior of the application (i.e., degrading application performance). In Chapter 6, we show that a heuristic approach can determine the service rate with relatively high accuracy and very low overhead. RaftLib incorporates this approach.

One often overlooked benefit of stream processing from the programmer perspective is that data "streams" can be contiguous in memory. Vectorized mathematical operations are a stalwart feature of high performance computation. RaftLib templates support auto-vectorization of mathematical operations directly on ports, inserting vectorized (SIMD) code where otherwise the compiler itself could not. Since templates are laid out at compile time, there is little extra overhead compared to hand coded SIMD operations, and since the operations occur directly from the port's memory there is no additional copy needed. In addition to vectorization, pre-fetching can also be improved by way of stream processing. Regularity of access (reads and writes to streams) can be improved via pre-fetch instructions. Currently

RaftLib pre-fetches elements on range operations, future work might extend this further. Processor architecture and topology dependent cache hints can improve performance over the processors' built in pre-fetch logic.

The "share-nothing" mantra of stream processing might introduce extra overhead, however it enables fairly easy programming of massively parallel systems. Each compute kernel can be easily duplicated on the same system, on different hardware across network links or even on GPGPU systems. As a research vehicle, RaftLib enables the study of stream processing communication and compute kernel placement. As a productivity tool, we are more interested in how few lines of code it takes to produce a result. Mentioned but not described has been the distributed nature of RaftLib. The capability to use TCP connections for many systems is clunky at best. With RaftLib there is no difference between a distributed and a non-distributed program from the perspective of the developer. A separate system called "oar" is a mesh of network clients that continually feed system information to each other. This information is provided to RaftLib in order to continuously optimize and monitor Raft kernels executing on multiple systems. The "oar" system also provides a means to remotely compile and execute kernels so that a user can have a simple compile and forget experience. Future work will see the full and complete integration of both TCP links and GPGPU kernels.

3.2.2 Authoring streaming applications

Next we consider the authoring of applications. We'll show some code segments to see how little code is needed to write a parallel algorithm and how familiar it can be to C++ programmers.

RaftLib views each compute kernel as a black-box at the level of a port interface. Once ports are defined by the user (or the runtime), the only observability that the run-time has is the interaction of the algorithm implementation inside the compute kernel with those ports. A new compute kernel is defined by extending `raft::kernel` as in Figure 3.2. Ports are the only means through which the kernel can access data from incoming or write to outgoing data "streams." Programmers building a kernel have a plethora of options to access streams from within the kernel. Figure 3.2 shows the simplest method (`pop`) to get data from the input stream, which is a return object that gives the user a reference to the head of the incoming queue (for variables `a` & `b`). A reference to the output queue is given by the `allocate_s` function. When `c` exits the calling scope, it is released to the outgoing queue. The return object from the `allocate_s` call (assigned to `c`) has associated signals accessible through the `sig` variable. There are multiple calls to perform push and pop style operations, each embodies some type of copy semantic (either zero copy or single copy), all provide a means to send or receive synchronous signals. There are also range operator equivalents for accessing more than one element. Synchronized signaling is implemented so that downstream kernels will receive the signal at the same time the corresponding data element is received (useful for things like end of file signals). Asynchronous signaling (i.e., immediately available to downstream kernels) is also available. Future implementations will utilize the asynchronous signaling pathway for global exception handling.

Arranging compute kernels into an application is one of the core functionalities of a stream processing system. RaftLib has an imperative mode of kernel connection via the `link` function. The `link` function call has the effect of assigning one output port of a given compute kernel to the input port of another compute kernel. A `map` object is defined in the `raft namespace` of which the `link` function is a member. Figure 3.3 shows our simple example application which takes two random number generating kernels, adds pairs of random numbers using the `sum` kernel and prints them.

When the user runs the `exe()` function of `map` object, the graph is first checked to ensure it is fully connected, then type checking is performed across each link. Before a link allocation type is selected (POSIX shared memory, heap allocated memory or TCP link), and each kernel is mapped to a resource. This could be pinning the thread or heavyweight process to a compute core, mapping the kernel to another compute node over a distributed system or even potentially a GPGPU. Once the link allocation types are selected, the run-time selects the narrowest convertible type for each link type and casts the types at each endpoint. Future versions could incorporate link data compression as well, further improving the cache-able data. Once memory is allocated for each link, a thread continuously monitors all the queues within the system and reallocates them as needed (either larger or smaller) to improve performance (either through branching up or instantaneous jumps in size through models selected by the process described in Chapter 7).

Streaming applications are often ideally suited for long running, data intensive applications such as big data processing or real-time data analytics. The conditions for these applications often change during the execution of a single run. Algorithms frequently use different optimizations based on differing inputs (e.g., sparse matrix vs. dense matrix multiply). The application can often benefit from additional resources or differing types of algorithms within the application to eliminate bottlenecks as they emerge. RaftLib gives the user the ability to specify synonymous kernel groupings that the run-time can swap out to optimize the computation. These can be kernels that are implemented for multiple hardware types, or can be differing algorithms. For instance, a version of the UNIX utility `grep` could be implemented with multiple search algorithms. Some of these algorithms require differing pieces, however they can all be expressed as a "search" kernel with common input and output ports.

Integration with legacy C++ code is one of our goals. As such, it is imperative that RaftLib work seamlessly with the C++ standard library functions. Figure 3.6 shows how a C++ container can be used directly as an input queue to a streaming graph, in parallel if out of order processing is allowed. Just as easily, a single value could be read in. Output integration is simple as well, standard library containers can receive the output of queues, or reduce streams to a single value through accumulation and reduction.

```
using ex_t = std::uint32_t;
/** data source  & receiver container **/
std::vector< ex_t > v,o;
ex_t i( 0 );
/** fill container **/
auto func( [&](){ return( i++ ); } );
while( i < 1000 ){ v.emplace_back( func() ); }
/** read from one kernel and write to another **/
map.link(  kernel::make< read_each< std::uint32_t > >(  v.begin(), v.end() ),
           kernel::make< write_each< std::uint32_t > >( std::back_inserter( o ) ) );
/** data is now copied to 'o' **/
```

Figure 3.6: Syntax for reading and writing to C++ standard library containers from raft::kernel objects. The `read_each` and `write_each` kernels are reading and writing on independent threads.

Copying of data is often an issue as well within stream processing systems. RaftLib provides a `for_each` kernel (Figure 3.7), which has behavior distinct from the `write_each` and `read_each` kernels. The `for_each` takes a pointer value and uses its memory space directly as a queue for downstream compute kernels. This is essentially a zero copy and enabling

```
int *arr = { 0, ..., N };
int val = 0;
auto &kernels(
map.link( kernel::make< for_each< int > >( arr, arr_length ),
          kernel::make< some_kernel< int > >() ) );
map.link( &kernels.getDst(),
          kernel::make< reduce< int, func /* reduct function */ > >( val ) );
/** val now has the result **/
```

Figure 3.7: Example of the `for_each` kernel, which is similar to the C++ standard library `for_each` function. The data from the given array is divided amongst the output queues using zero copy, minimizing data extraneous data movement.

behavior from a "streaming" application similar to that of an OpenMP [45] parallelized loop. Unlike the C++ standard library `for_each`, the RaftLib version provides an index to indicate position within the array for the start position. This enables the compute kernel reading the array to calculate the position within it. When this kernel is executed, it appears as a kernel only momentarily, essentially providing a data source for the downstream compute kernels to read. This kernel can also function as a splitting kernel, if needed it can arbitrate splitting of the statically allocated data chunks and partition them to newly cloned compute kernels.

Code verbosity is often an issue. Readily available in C++ is the declaration of a class or a template, when often what is wanted is the ability to pass a simple function and have it executed by the called function. Newer languages and C++11 have met this demand with lambda functions. RaftLib brings lambda compute kernels, which give the user the ability to declare a fully functional, independent kernel while freeing him/her from the cruft that would normally accompany such a declaration. Figure 3.8 demonstrates the syntax for a single output random number generator. The closure type of the lambda operator also allows for usage of the `static` keyword to maintain state within the function [52]. These kernels can be duplicated and distributed, however they do induce one complication if the user decides to capture external values by reference instead of by value, undefined behavior may result if the kernel is duplicated; especially across a TCP link (an issue we intend to resolve in subsequent versions of RaftLib).

```
using ex_t = std::uint32_t;
map.link( /** instantiate lambda kernel as source **/
   kernel::make< lambdak< ex_t > >( 0, 1, []( Port &input, Port &output )
      {
         auto out( output[ "0" ].template allocate_s< ex_t >() );
         (*out) = rand();
      } /** end lambda kernel **/ ) /** end make **/,
      /** instantiate print kernel as destination **/
   kernel::make< print< ex_t, ‘\n’ > >() );
```

Figure 3.8: Syntax for lambda kernel. The user specifies port types as template parameters to the kernel, in this example `std::uint32_t`. If a single type is provided as a template parameter, then all ports for this lambda kernel are assumed to have this type. If more than one template parameter is used, then the number of types must match the number of ports given by the first and second function parameters (input and output port count, respectively). The number of input ports is zero and the number of output ports is one for this example. Ports are named sequentially starting with zero. The third parameter is a lambda function which is called repeatedly by the runtime.

3.3 Benchmarking

Text search is used in a variety of applications. We will focus on the exact string matching problem which has been studied extensively. The stalwart of string matching applications (both exact and inexact) is the GNU version of the `grep` utility. It has been developed and optimized for 20+ years resulting in excellent single threaded exact string matching performance ($\sim$ 1.2 GB/s) on our test machine (see Table 3.1). To parallelize GNU `grep`, the GNU Parallel [184] utility is used to spread computation across one through 16 cores. Two differing text search algorithms will be tested and parallelized with RaftLib. One will utilize the Aho-Corasick [7] string matching algorithm which is quite good for multiple string patterns. The other will use the Boyer-Moore-Horspool algorithm [92] which is often much faster for single pattern matching. The realized application topology for both string matching algorithms implemented with RaftLib are conceptually similar to Figure 3.9, however the file read exists as an independent kernel only momentarily as a notional data source since the run-time utilizes zero copy, and the file is directly read into the in-bound queues of each `match` kernel.

Figure 3.10 shows code necessary to generate the application topology used to express both string matching algorithms using RaftLib. Not shown is the code to handle arguments,

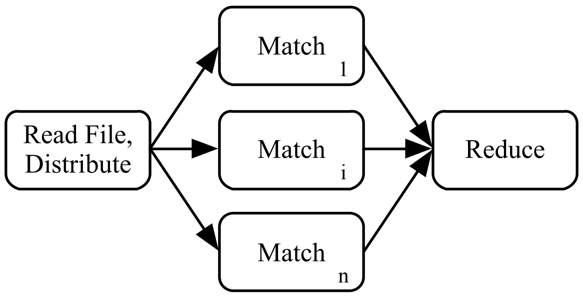


Figure 3.9: String matching stream topology for both Boyer-Moore-Horspool and Aho-Corasick algorithms. The first compute kernel (at left) reads the file and distributes the data. The second kernel labeled `Match` uses one or the other algorithms to find string matches within the streaming corpus. The matches are then streamed to the last kernel (at right) which combines them into a single data structure.

```
using strsearch = raft::search< raft::ahocorasick >;
std::vector< hit_t > total_hits;
auto kern_start( map.link< raft::out >( kernel::make< filereader >( file, offset ),
                                        kernel::make< strsearch >( search_term ) ) );
map.link< raft::out >( &kern_start.getDst(),
                       kernel::make<
                          write_each< match_t > >(
                             std::back_inserter( total_hits ) ) );
```

Figure 3.10: Implementation of the string matching application topology using RaftLib. The actual search kernel is instantiated by making a `search` kernel. The exact algorithm is chosen by specifying the desired algorithm as a template parameter to select the correct template specialization.

setup, etc. Note that there is no special action required to parallelize the algorithm. The `filereader` kernel takes the file name, it distributes the data from the file to each string matching kernel. The programmer can express the algorithm without having to worry about parallelizing it. The programmer simply focuses on the sequential algorithm. Traditional approaches to parallelization require the programmer to have knowledge of locks, synchronization, and often cache protocols to safely express a parallel algorithm. Even more exciting is that when using RaftLib, the same code can be run on multi-cores in a distributed network without the programmer having to do anything differently. The partitioner decides where to run which piece of the application and the online scheduler can make decisions to tune performance dynamically.

For comparison we contrast the performance of our implementations of Aho-Corasick and Boyer-Moore-Horspool against the GNU `grep` utility and a text matching application implemented using the Boyer-Moore algorithm implemented in Scala running on the popular Apache Spark framework. We'll use a single hardware platform with multiple cores and a Linux operating system (see Table 3.1).

We use version 2.20 of the GNU `grep` utility. In order to parallelize GNU `grep`, the GNU Parallel [184] application is used (version 2014.10.22), with the default settings. RaftLib (and all other applications/benchmarks used) is compiled using GNU GCC 4.8 with compiler flags "`-Ofast`." For this set of experiments, the maximum parallelism is capped to the number of cores available on the target machine. A RAM disk is used to store the text corpus to ensure that disk IO is not a limiting factor. The corpus to search is sourced from the post history of a popular programming site [178] which is $\sim$ 40 GB in size. The file is cut to 30 GB before searching. This cut is simply to afford the string matching algorithms the luxury of having physical memory equal to the entire corpus if required (although in practice none of the applications required near this amount). All timing is performed using the GNU `time` utility (version 1.7) except the Spark application, which uses its own timing utility.

Table 3.1: Summary of Benchmarking Hardware.

Processor	Cores	RAM	OS Version
Intel Xeon E5-2650	16	64 GB	Linux 2.6.32

Figure 3.11 shows the throughput (in GB/s) for all of the tested string matching applications, varying the utilized cores from one through 16. The performance of the GNU `grep` utility when single threaded is quite impressive. It handily beats all the other algorithms for single core performance (when not using GNU Parallel, as shown in the figure). Perfectly parallelized (assuming linear speedup) the GNU `grep` application could be capable of $\sim$ 16 GB/s. When parallelized with GNU Parallel however, that is not the case.

The performance of Apache Spark when given multiple cores is quite good. The speed-up is almost linear from a single core though 16 cores. The Aho-Corasick string matching algorithm using RaftLib performs almost as well, topping out at $\sim$ 1.5 GB/s to Apache Spark's $\sim$ 2.8 GB/s. RaftLib has the ability to quickly swap out algorithms during execution, this was disabled for this benchmark so we could more easily compare specific algorithms. Manually changing the algorithm RaftLib used to Boyer-Moore-Horspool, the performance

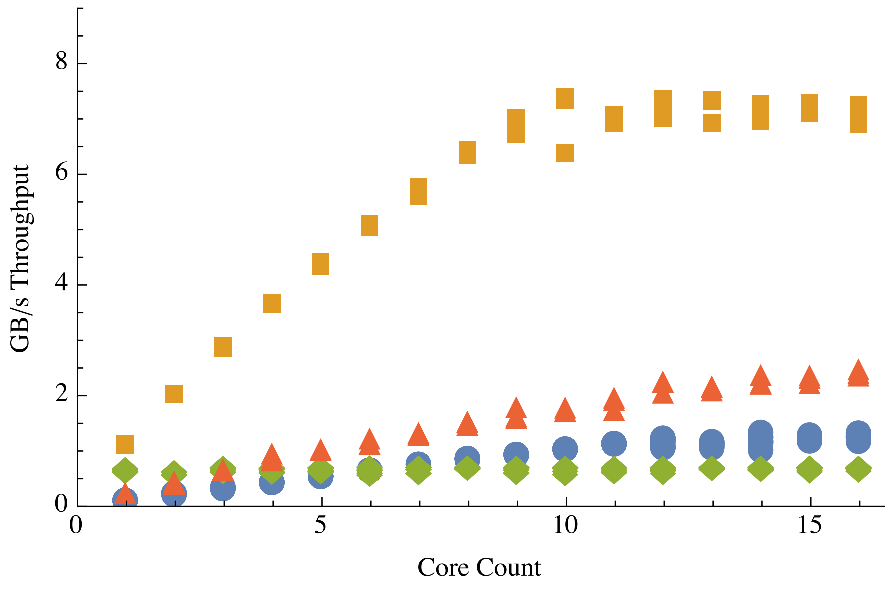


Figure 3.11: This figure shows the performance of each string matching application in GB/s by utilized cores. This is calculated using a 30 GB corpus searched on the hardware from Table 3.1. The green diamonds represent the GNU Parallel parallelized GNU `grep`. The red triangles represent Apache Spark. The blue circles and gold squares represent the Aho-Corasick and Boyer-Moore-Horspool text search algorithms, respectively, parallelized using RaftLib.

improved drastically. The speed-up from one through 10 cores is now linear, with the 30 GB file searched in ~ 4.1 s which gives it close to 8 GB/s throughput.

Overall the performance of the RaftLib Aho-Corasick string matching algorithm is quite comparable to the one implemented using the popular Apache Spark framework. The Boyer-Moore-Horspool however outperforms all the other algorithms tested. The change in performance when swapping algorithms indicates that the algorithm itself (Aho-Corasick) was the bottleneck. Once that bottleneck is removed we found that the memory system itself becomes the bottleneck. Future work with cache aware scheduling and pipeline pre-fetch could perhaps improve performance further by reducing memory latency. All in all the performance of RaftLib is quite good, comparable with one of the best current distributed processing frameworks (Apache Spark) and far better than the popular parallelizing utility GNU Parallel.

3.4 Concluding Remarks and What Follows

RaftLib has many features that enable a user to integrate fast and safe streaming execution within legacy C++ code. It provides interfaces similar to those found in the C++ standard library, which we hope will enable users to pick up how to use the library at a faster pace. We've also shown new ways to describe compute kernels, such as the "lambda" kernels which eliminates much of the "boiler-plate" code necessary to describe a full C++ class or template. What we've also described is a framework for massively parallel execution that is simple to use. The same code that executes locally can execute distributively with the integration of the "oar" network framework. No programming changes are necessary. This differs greatly from many current open source distributed programming frameworks.

What we've done with the RaftLib framework is lay a foundation for future research. How best to integrate stream processing with sequential computation is still an open question. Pragma methods such as OpenMP for loop parallelization work well for parallelizing loops, however they're far from ideal as programmers must fully understand how to use the available options in order to get the most out of OpenMP. RaftLib promises similar levels of parallelism that are automatically optimized by the run-time. Current and past works have demonstrated the viability of low-overhead instrumentation of vital online metrics such as non-blocking service rate and queue occupancy. Things like fast automatic model selection (e.g., Chapter 7), scheduling, and environmental adaptation must be researched and perfected in order for systems such as these to fully exploit the myriad of computational resources available today (multi-cores, vector processors, GPGPUs, etc.).

The RaftLib framework provides a platform for safe and fast parallel streaming execution within the C++ language. It serves as a productivity tool and a research vehicle for exploring integration and optimization issues. Despite the slow adoption rate of stream processing, we hope that the utilization of a widely used existing language (C++) serves as a catalyst to gain more than a niche user base. The subsequent chapters lay out the technologies necessary to make RaftLib possible, starting with techniques to quickly model the throughput of a potential streaming topology and hardware mapping.

Chapter 4

Modeling Streaming Applications

4.1 Introduction

In search of ever higher performance, computer architectures have diversified to include a wide variety of heterogeneous hardware such as traditional multi-core processors, field-programmable gate arrays (FPGAs) and general purpose graphics processing units (GPG-PUs). Even on a single die, multiple processor types are often present (e.g., ARM's big.LITTLE [87] and Xilinx's Zynq [54] platforms). Presented with multiple architectural platforms on which to run an application, developers need reliable and fast models to predict performance. One performance metric of interest to many "big-data" applications is overall throughput. This chapter explores a set of analytic models that are both computationally simple and widely applicable to applications that are formulated as directed acyclic graphs (i.e. they can be considered to be in the streaming data paradigm). Validation is performed across multiple heterogeneous resources, a pair of real streaming applications and multiple synthetic streaming applications.

Given a set of compute resources and a streaming application, how does an application developer model the overall throughput of the application for a specific hardware mapping? How does a developer determine the size of buffers to allocate based on a target (obtainable) throughput? For example, if an application developer is tasked with developing a streaming JPEG encode application as shown in Figure 4.1, how is that developer going to assign (map) the compute kernels to the available resources? There are several choices, every kernel labeled with SW (compiled software) can be mapped to a general multi-core processor, and those labeled with HW (synthesized hardware) can be mapped to an FPGA. The most obvious, albeit time consuming, approach is simply to do an exhaustive empirical measurement of all possible combinations and chose the best performing one. An alternative approach is to develop a model that reflects the changes in performance that result from alternative mappings, and search over the model space to yield a mapping. When a model of performance is combined with a partitioning algorithm like the ones pioneered by Kernighan-Lin [104] and later Sanchis [166], exhaustive search might not be needed. This alternative approach has the potential to be much faster than exhaustive empirical search. However, the quality of the final result is strongly influenced by the effectiveness of the model. The model's predictions should reasonably correspond to the actual application performance for this approach to be effective.

Performance models for multi-core and heterogeneous systems in general are nothing new. Various approaches exist in practice that use everything from execution histories [16] to the roofline model [197] to help decide how to place a compute kernel and how to modify it for maximum performance. This chapter focuses on the use of analytic performance models to analyze the obtainable overall throughput and the necessary buffering to obtain it. The technique used to model throughput is computationally efficient, with a polynomial time solution [82]. What follows in this chapter is a presentation of a computationally simple hybrid maximum flow / queueing network model that incorporates multiple hardware sharing models. The use of simple sharing models in no way bars or makes an claim to the compatibility of more complex sharing models, they are just not the subject of this chapter so the simplest ones available are chosen. Experimental results in Section 4.3 validate the proposed modeling approach concomitant with individual results for the resource sharing models.

4.1.1 Description

Given the throughput capacity into and out of each compute kernel within an application and the throughput achievable by each communications link, the model presented here calculates maximum data flow through the overall network. Using a constrained generalized maximum flow network the model determines maximum flow through an application topology given a set of constraints. Utilizing an $M/M/1$ queuing model, it also estimates the minimum required buffering capacity for each communication edge within the application. What follows is a description of the path from streaming application topology to flow network model, including a queueing network model. Model notation is summarized in Table 4.1.

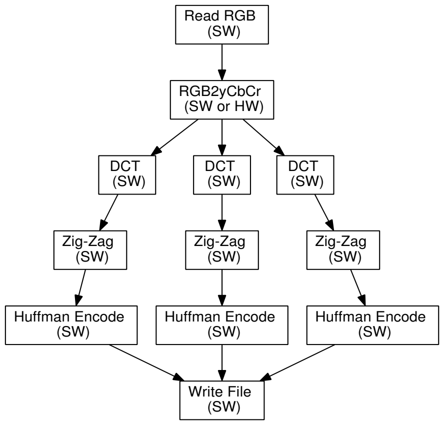
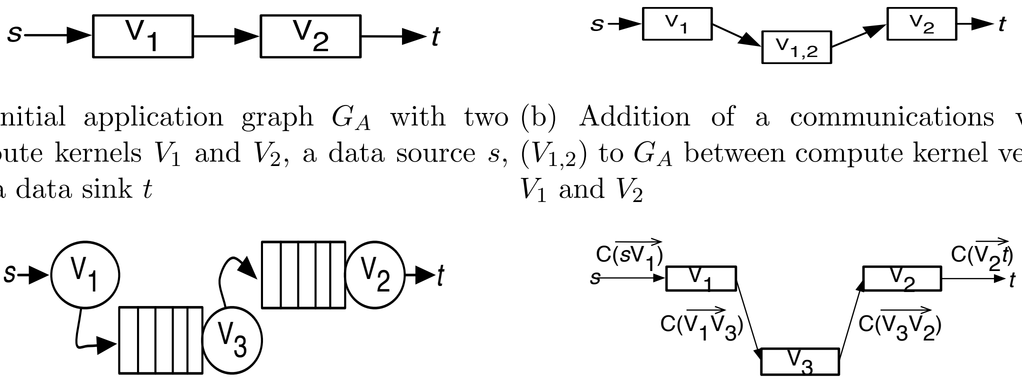


Figure 4.1: Application topology for JPEG encode [94] expressed as a streaming application. The SW and HW in parenthesis indicate an implementation is available on a multi-core processor and/or FPGA respectively.



(a) Initial application graph G_A with two compute kernels V_1 and V_2, a data source s, and a data sink t

(b) Addition of a communications vertex $(V_{1,2})$ to G_A between compute kernel vertices V_1 and V_2

(c) The queueing network G_Q that arises from the topology depicted in Figure 4.2b

(d) The overall flow graph G_F with capacities C at each edge

Figure 4.2: Stages of transition from application graph, to queueing network to flow model.

Notation	Description
V_i	vertex i
$\overrightarrow{V_iV_j}$	an edge between vertices i and j
$\mu(V_i)$	service rate at vertex i (in Bytes/s)
$\mu_s(V_i)$	shared service rate at vertex i (in Bytes/s)
$\lambda(V_i)$	arrival rate to queue for vertex i (in Bytes/s)
$\rho(V_i)$	utilization of server at vertex V_i
$R(\overrightarrow{V_iV_j})$	fraction of data outbound from V_i routed across $\overrightarrow{V_iV_j}$
$\gamma(V_i)$	gain function across vertex V_i
$C(\overrightarrow{V_iV_j})$	flow capacity of edge $\overrightarrow{V_iV_j}$ (in Bytes/s)
Γ	overall throughput (in Bytes/s)
$f(\overrightarrow{V_iV_j})$	flow along edge $\overrightarrow{V_iV_j}$ (in Bytes/s)
ϕ	constraint on ρ
$K(V_i)$	estimated buffering capacity associated with vertex V_i (in Bytes)

Table 4.1: The above notation is used to describe the model (where noted, the terminology is applicable to the queueing network, flow graph or both).

An application graph topology G_A (Figure 4.2a) is a connected directed graph consisting of each compute kernel within an application as a vertex V_i and every data-flow dependency (communications link) as an edge $\overrightarrow{V_iV_j}$. An application topology also defines a (pseudo-) data source s and sink t as start and end nodes. Since application topologies can have more than one actual data source and sink, the model inserts the node s with outbound links to all application kernels that do not yet have inbound edges and inserts the node t with inbound edges from all application kernels that do not yet have outbound edges. Nodes s and t are modeled has having infinite capacity so as to not influence the throughput achievable in the network.

The model views every communications link as a distinct resource with its own service rate. To model this behavior the application topology (G_A) is transformed by adding additional vertices for each communications link as shown in Figure 4.2b. The transformed application topology of Figure 4.2b can be directly modeled as a queueing network. The queueing network is defined as the directed graph G_Q (Figure 4.2c). Every application kernel is a queue and server pair in G_Q. Every communications link between compute kernels is also a queue and server pair in G_Q. As illustrated in Figure 4.2c, the nodes modeling communication links in the modified application graph have been renamed so as to simplify the notation. Each communications link could conceivably be reasoned about as being comprised of many sub-queues, however in this work an overall "virtual queue" subsuming the sub-queues will be assumed [117]. Formally G_Q is defined by the 4-tuple:

$$G_Q = (V_Q, E_Q, s \in V_Q, t \in V_Q)$$

where s is the source node and t is the termination (sink) node.

In a queueing network the two main parameters that characterize the performance of the network are $\lambda(V_i)$, the arrival rate of data at node V_i, and $\mu(V_i)$, the service rate at node V_i. For nodes in V_Q that represent compute kernels, their service rates are measured empirically, by either detaching the kernel from the application and measuring it in isolation or via the online approximation techniques outlined in Chapter 6. A compute kernel when detached from its queueing network is simply a single queue and server (the queue representing a data source). That single queue is assumed to have an infinite supply of data giving it non-blocking read behavior. Its outbound data port is assumed to always be empty such that outbound writes are also non-blocking. At equilibrium with no gain or loss a server's $\mu(V_i)$ is equal to its aggregate data ingest rate (with units of Bytes/s). The service rates of nodes in V_Q that represent communication links can be determined multiple ways: from the literature, hardware simulation or estimated using empirical measurement. The arrival rates $\lambda(V_i)$ will be derived from the flow model described below.

A flow graph is defined as a directed acyclic graph G_F (Figure 4.2d) where each server in the queueing network (Figure 4.2c) is represented as a vertex. G_F is constructed from G_Q by removing the queues on each edge $\overrightarrow{V_iV_j} \in G_Q$. This is reasonable since the queueing model represented here is actually a case of an open Jacksonian network [95, 96]. Formally the flow graph is defined as a 7-tuple:

$$G_F = (V_F, E_F, s, t, C, \gamma, R)$$

$$V_F = V_Q, \qquad E_F = E_Q$$

where $C : E_F \to \Re_+$ represents the flow capacity of each edge (determined as described below), and $\gamma : V_F \to \Re_+$ represents the data volume gain or loss associated with each node. It is defined as the ratio of the mean data volume out of a node relative to the mean data volume in. If $\gamma < 1$ then there is data loss in the node (e.g., data compression) and if $\gamma > 1$ there is gain in the node (e.g., data expansion). For nodes that represent compute kernels, these values will be determined empirically, and for nodes that represent communication links, $\gamma = 1$. For nodes with more than one outbound edge, $R : E_F \rightarrow (0, 1]$ represents the routing fraction associated with each outbound edge $\overrightarrow{V_iV_j}$ of node V_i. For nodes V_i with only one outbound edge $\overrightarrow{V_iV_j}$, $R(\overrightarrow{V_iV_j}) = 1$.

Given $\mu(V_i)$, $\gamma(V_i)$, and $R(\overrightarrow{V_iV_j})$ for each vertex and edge, the capacity C associated with each edge can be computed using Equation (4.1).

$$C(\overrightarrow{V_iV_j}) = \mu(V_i) \times \gamma(V_i) \times R(\overrightarrow{V_iV_j}) \tag{4.1}$$

Each edge in a flow graph is constrained by the capacity $C(\overrightarrow{V_iV_j})$. Note that the above makes the implicit assumption that each compute kernel has been mapped to a dedicated compute resource. This will be extended to reflect resource sharing in the section below.

To calculate the maximum stable throughput the model maximizes Γ (the overall throughput through the application) and f (the flow at every edge within the graph) subject to the following constraints:

$$\sum_{j|(i,j)\in E_F} f(\overrightarrow{V_iV_j}) - \sum_{j|(j,i)\in E_F} f(\overrightarrow{V_jV_i}) = \begin{cases} + & i = s \\ 0 & i = \text{circulation} \\ - & i = t \end{cases} \tag{4.2}$$

$$\gamma(\overrightarrow{V_iV_j}) = \frac{1}{\gamma(\overrightarrow{V_jV_i})} \tag{4.3}$$

$$f(\overrightarrow{V_iV_j}) \leq C(\overrightarrow{V_iV_j}) \tag{4.4}$$

$$\frac{f(\overrightarrow{V_iV_j})}{\sum_{x=1}^{N} f(\overrightarrow{V_iV_x})} = R(\overrightarrow{V_iV_j}) \tag{4.5}$$

Equation 4.2 states that flow must be conserved across all edges and that the only edges with positive or negative flow can be the s and t. Gain or loss as shown in Equation 4.3 is also conserved (application of the gain/loss approach to queueing networks is one contribution of this work). As in a standard maximum flow model, flow must be less than or equal to the capacity as shown in Equation 4.4. To maintain correct data routing, Equation 4.5 ensures that the volumes are maintained across each edge (another contribution of this work over prior methods).

To bound queue size, the model can be further constrained by ensuring a smaller $\rho = \lambda/\mu$ at each queueing station. This corresponds to maximizing Γ with the following additional constraint:

$$\rho(V_i) \leq \phi \tag{4.6}$$

For the results presented in this chapter, ϕ is set to 0.99998, however it could be any value ≤ 1. If equal to 1, then there will not be a queue size bound on the bottleneck node(s) (save for the special case of deterministic servers) since servers with non-deterministic processes modulating them having a utilization ≥ 1 result in an infinite queue length. A serendipitous property of the flow model is that ρ is implicitly constrained to 1, so reducing the rate is a simple pass through the graph. We also assume that the processes of each server modulating the arrival and service processes have a distribution with finite variance (i.e., infinite variance that could result in an infinite queue length is not allowed). The existence of un-bounded queues within the network does not preclude the usage of this modeling technique elsewhere within the network.

Once maximal values of $f(\overrightarrow{V_iV_j})$ have been calculated for every $\overrightarrow{V_iV_j} \in E_F$, these values can be used within the queueing model to determine the necessary buffering for the system at the calculated flow.

Our hypothesis is that the $M/M/1$ model gives an upper estimate of the queue occupancy, since we expect the actual service time distributions to have a lower coefficient of variation than an exponential distribution. An estimation of the buffering necessary at each queue is determined by solving for the queue occupancy K at a probability P_K that is close to zero as in Equation 4.7.

$$\begin{gathered} K(V_i) = \frac{\log(\frac{P_K}{1-\rho(V_i)})}{\log(\rho(V_i))} - 1 \\ \text{where } P_K = 10^{-7} \end{gathered} \tag{4.7}$$

The extent to which the assumptions made for the $M/M/1$ model hold true will be investigated in Section 4.3.

4.1.2 Sharing Models

Sharing of resources and resource contention is a function of several parameters. Schedulers are often involved, either from the operating system or built into the hardware. A resource such as an FPGA is typically not shared in time, but shared as a function of area. Diversity in the underlying behavior of sharing across platforms drives the complexity and specificity of sharing models. The models presented here are specific by necessity but intentionally simple. There is an associated noise with multi-core sharing (discussed in Chapter 7), however it is assumed within this simple model that the noise is insignificant.

For multi-core processors the sharing model is simply the service rate for a compute kernel executing in isolation divided by the number of kernels running on the same processor core (Equation 4.8).

$$\mu_s(V_i) = \mu(V_i)/n, \quad n = \text{ \# processes} \tag{4.8}$$

FPGAs are assumed to be share-able in area, but not temporally (although the concept of temporal sharing exists [170]). The sharing equation (Equation 4.9) reflects that by giving each compute kernel mapped to an FPGA its full μ until all available gates are exhausted (e.g., requiring more gates than physically exist on the device is considered an invalid configuration, and not considered).

$$\mu_s(V_i) = \mu(V_i) \times a_i \tag{4.9}$$

where $a_i = 1$ if $\sum_{i=1}^{N} Area_i \leq$ Available Area , else $a_i = 0$.

The Virtex-4 FPGAs used for empirical measurements given within this chapter communicate with multi-core processors over a PCI-X bus. The sharing model for this reflects a fair sharing policy on the part of the controller until the bandwidth limit is reached.

$$\mu_s(V_i) = \mu(V_i)/n, \quad n = \text{ \# communication links sharing bus} \tag{4.10}$$

4.1.3 Modeling Assumptions

The model presented above makes the following assumptions about the applications, graph topology and underlying hardware:

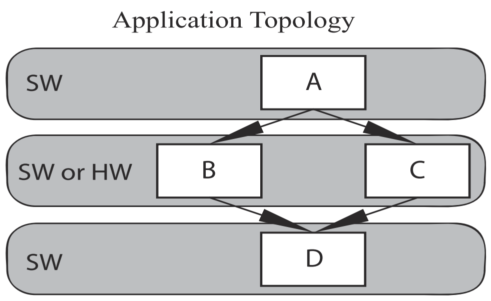


Figure 4.3: Example application topology

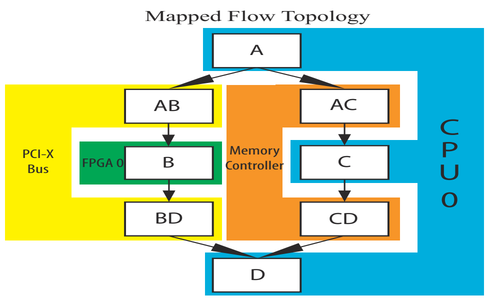


Figure 4.4: A mapping of the application topology in Figure 4.3

1. The application is assumed to in equilibrium: The streaming computation paradigm is typically used in application domains that require high-throughput, high volume computation. On initial start-up and termination non-steady state behavior is exhibited, however during the majority of the execution steady state behavior is typical.
2. The data volume into and out of each edge is measurable on the compute kernel in isolation (i.e., separated from the rest of the application topology.
3. Only non-blocking behavior exists: All compute nodes (servers) are allowed to process data as soon as it is present on its queue.
4. Data routing is independent of the state of the system: External signals don't drive a server to remove items from a queue, nor do they influence $R(\overrightarrow{V_iV_j})$.
5. All compute kernels are work conserving: When two compute kernels are mapped to the same resource, the work that is done by the compute kernel does not decrease.

4.1.4 Example

The approach (illustrated in Figures 4.3 to 4.7) begins with a streaming application whose data-flow topology is acyclic. It takes empirical measurements of each compute kernel through each in-edge and out-edge. The model uses these unshared, unconstrained measurements to calculate mean service rate μ, routing fraction R, and gain γ, associated with each kernel. These metrics are used in the generalized maximum flow model to calculate a maximum flow for the data-flow topology on a specific set of resources. The flows predicted

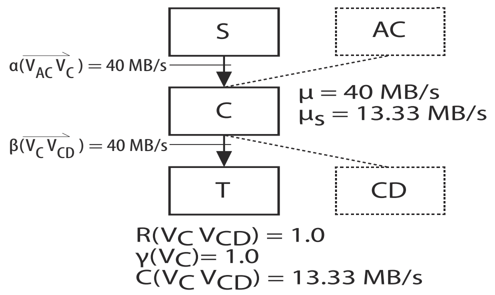


Figure 4.5: In order to measure the unshared throughput of compute kernel 'C', the kernel is taken out of its application network and given artificial data sources S and T for each in- and out-edge. The dotted lines show the edges where kernel 'C' would have connected in the application which are replaced by the solid lines from S and T. For this kernel the measured input rate for the edge α is 40 MB/s and the measured output rate at edge β is 40 MB/s. The routing fraction is 1.0 as there is only one out-edge and gain (γ) is also 1.0 since there is no gain or loss of data at this compute kernel. The output link capacity C is 13.33 MB/s after application of the sharing model in Equation 4.8.

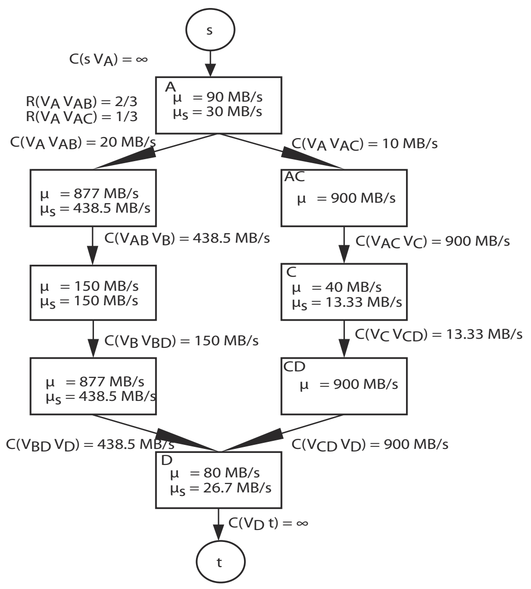


Figure 4.6: A complete flow graph from the application and mapping in Figure 4.4

Flow Solution

$f(S\,V_A)$	26.7 MB/s
$f(V_A V_{AB})$	17.84 MB/s
$f(V_A V_{AC})$	8.92 MB/s
$f(V_{AB} V_B)$	17.84 MB/s
$f(V_{AC} V_C)$	8.92 MB/s
$f(V_B V_{BD})$	17.84 MB/s
$f(V_C V_{CD})$	8.92 MB/s
$f(V_{BD} V_D)$	17.84 MB/s
$f(V_{CD} V_D)$	8.92 MB/s
$f(V_D T)$	26.7 MB/s

Figure 4.7: Solution to flow model in Figure 4.6

by the flow model are used directly in the $M/M/1$ queueing model to calculate necessary buffering capacity.

4.2 Model Evaluation Approach

In order to evaluate the model, two approaches are taken. First a pair of real applications are used: a JPEG encode application implemented to the ISO specification (and decomposed as shown in Figure 4.1) and a DES encrypt application. Second, a set of synthetic applications are generated using a widely used topology generator [62].

For each application, both real and synthetic, random mappings of application kernels to compute resources are generated and run on the hardware enumerated in Table 4.2. The subsections below describe the tools, hardware, and methods used to evaluate the modeling approach.

4.2.1 Tools

The Auto-Pipe development environment [72] is used for all experiments within this chapter. Auto-Pipe supports streaming data applications deployed on heterogeneous compute platforms. In order to make accurate measurements of queue occupancies and edge throughput, the TimeTrial [115] low-impact performance monitor is used. All applications and compute kernels (both real and synthetic) are expressed in combinations of C and VHDL and compiled with the GNU C compiler or synthesized with Synopsys Synplify Premier DP respectively.

4.2.2 Hardware

There are two distinct hardware platforms used for empirical testing. Each platform is referred to by the heading shown in Table 4.2.

Table 4.2: Hardware used for empirical measurement

Name	Machine 1	Machine 2
CPU	12 x 2.4GHz AMD Opteron	4 x 3.1GHz Intel Xeon E3
FPGA	2 x Virtex-4 LX100	None
RAM	32GB DDR2	8GB DDR3

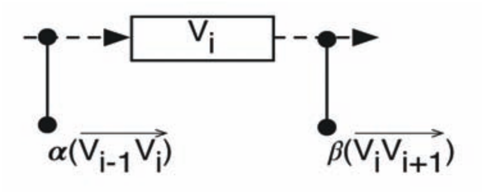

V_i
$\alpha(\overrightarrow{V_{i-1}V_i})$
$\beta(\overrightarrow{V_iV_{i+1}})$

Figure 4.8: Solid lines with dots indicate where TimeTrial measures throughput along an edge of an application

4.2.3 Empirical Testing

As detailed in Section 4.1.1, the model needs measurements of each compute kernel running on its assigned hardware as input. To accomplish this each compute kernel is instantiated in isolation and a test bench is produced by a test harness that provides high volume input to each input edge and consumes all data on each output edge. Throughput is measured using the TimeTrial measuring system and recorded. These measurements are designated α and β and are shown in Figure 4.8.

4.2.4 Selecting Compute Resources and Mapping Application Kernels

For a given application each compute kernel can be run on many potential resources. This is true for both the real and synthetic applications. Given a set of resources from Table 4.2, a subset (potentially all of them) must be selected for each run (Ω), this is done with a uniform random selection. Once the resource set Ω is selected, an application's compute kernels must be mapped to it. To map application kernels to Ω a random compute kernel (drawn uniformly from the set of kernels) is selected and assigned to $\omega \in \Omega$ (again, drawn uniformly from Ω). This process continues until each resource in Ω has one compute kernel mapped to it. The mapping algorithm then assigns compute kernels to resources by randomly walking the in- and out-edges of previously mapped compute kernels until all compute kernels are mapped. A constraint checking algorithm checks user provided constraints on resources while mapping to ensure that the finished kernel/hardware mapping will compile/synthesize and run (e.g., ensuring that the FPGA is not over-utilized). This process is intended not to generate optimal mappings, but rather to generate a range of reasonable mappings for the purpose of assessing the model.

4.2.5 Synthetic Benchmarks

Whenever the verification of a model is based principally on empirical evidence, a primary consideration is the extent to which the test sets used are truly representative of the overall universe of possibilities. That concern is addressed here through the use of several synthetically generated benchmarks. In order to produce synthetic applications, topologies are generated using the Task Graphs for Free (TGFF) tool [62].

During the topology generation process, the following parameters were used to control TGFF:

1. The number of compute kernels (nodes) in the application topology ranges from 1 to 80 with a uniform distribution.
2. The in-degree and out-degree of nodes is varied from 1 to 4, again uniformly distributed. Sources and sinks have only out- and in-edges respectively.

Once topologies are generate using TGFF, a test harness generates synthetic applications using the Auto-Pipe framework with the following parameters:

1. Mean execution time is set to 20 μs $\pm$ 10%. Execution time varies dynamically with an exponential distribution.
2. Input data volume for a vertex is statically set with a value chosen between 1 and 64 data bytes. The volume is distributed uniformly.
3. Edges in the graph are constrained so that data volumes are matched between in and out edges.

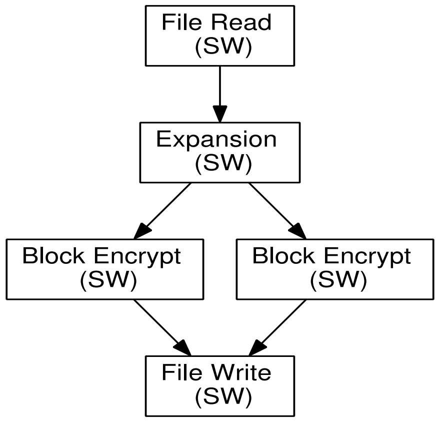


Figure 4.9: Application topology for the DES encryption algorithm expressed as a streaming application. All compute kernels are implemented in software.

4.2.6 Real Applications

The JPEG encode application as shown in Figure 4.1 is implemented according to the specifications in [94]. The DES encrypt application depicted in Figure 4.9 is implemented according to FIPS (46-3) standard published by NIST. The topology of each application is specified in the X language [72] which serves as input for the test harness. The harness takes the pre-coded compute kernel implementations and maps them to hardware resources in the same manner as the synthetic applications mentioned above.

4.3 Results

4.3.1 Processor Sharing Model

Under the processor sharing model (Section 4.1.2), when multiple compute kernels are mapped to a processor core, $\mu(V_i)$ is de-rated according to the number of kernels sharing a given core. A test application designed to run multiple processes on a single core is used to validate this model. Each process is synchronized to start concurrently with all the other processes within a single experiment (i.e., if 30 processes then all 30 processes are launched together). Each process runs 2 minutes according to the system wall-clock. Each time quantum is consumed by looping for 200 no-op instructions and incrementing a register

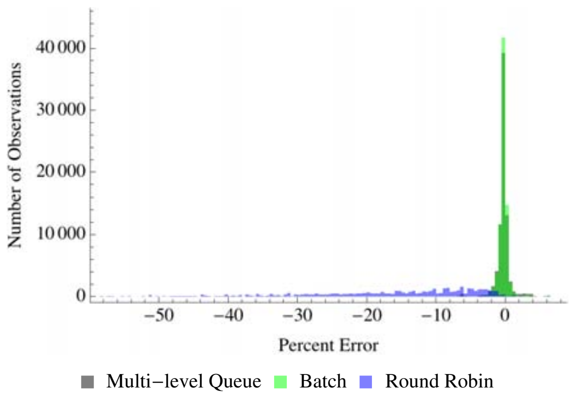


Figure 4.10: Percent error for processor sharing model from Equation 4.8 with three different scheduling algorithms (Multi-level queue, batch and round robin). All metrics are over 1 through 40 processes on one processor core. Model predicts executions per second. Error is calculated as $\frac{(\text{observed flow} - \text{modeled flow})}{\text{modeled flow}}$. R^2 values for each scheduler are .999854, .999952, and .766693 for multi-level queue, batch, and round robin respectively.

counter. Tests were run on both machines listed in Table 4.2. Three different scheduling algorithms (multi-level queue, batch and round robin) were chosen as they are representative of most modern systems [183].

In Figure 4.10 the processor sharing model validation percent error distribution is shown for the predicted executions per second. The overall model vs. observed fit is quite good. As expected the round robin scheduler resulted in more variation than the other two scheduling algorithms due to fixed quantum sizing. The fairest schedulers that closely match the assumptions the model makes are the multi-level queue and the batch scheduler. In the work that follows we constrain all experimentation to the multi-level queue scheduler.

4.3.2 The Flow Model

Validation of the flow model proceeds using the set of applications described in Section 4.2. Forty synthetic applications with 3 through 82 compute nodes were tested on Machine 1 (see Table 4.2). The results of flow predictions for each edge versus empirically measured flow are shown in Figure 4.11. Linear regression of the model and measured synthetic application results gives an R^2 value of .9999. The distribution of JPEG encode and DES encrypt application data is very similar to that of the synthetically generated graphs as shown in Figures 4.12 and 4.13.

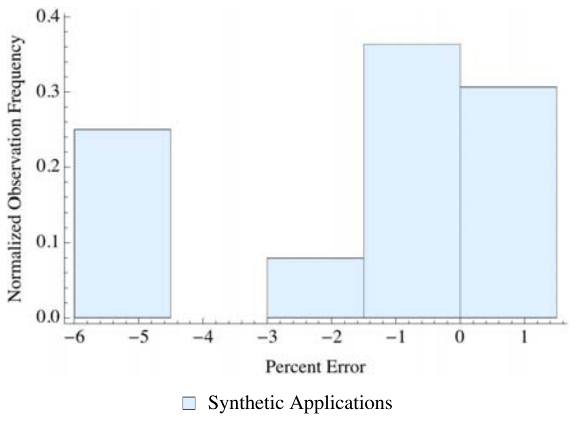


Figure 4.11: Percent error for gain/loss flow model for the synthetic application set, calculated as $\frac{\text{(observed flow}-\text{modeled flow)}}{\text{modeled flow}}$. Kernels executed on FPGA and multi-core CPUs.

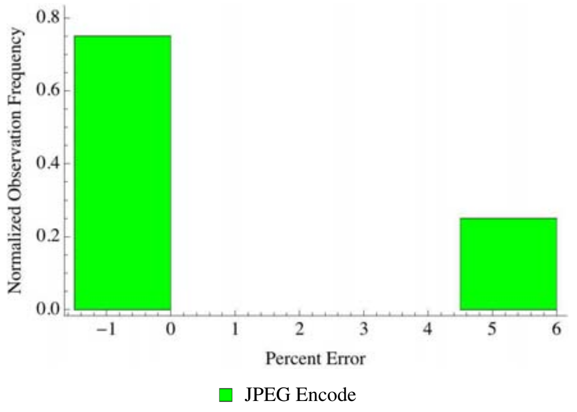


Figure 4.12: Percent error for gain/loss flow model for the JPEG encode application, calculated as $\frac{\text{(observed flow}-\text{modeled flow)}}{\text{modeled flow}}$.

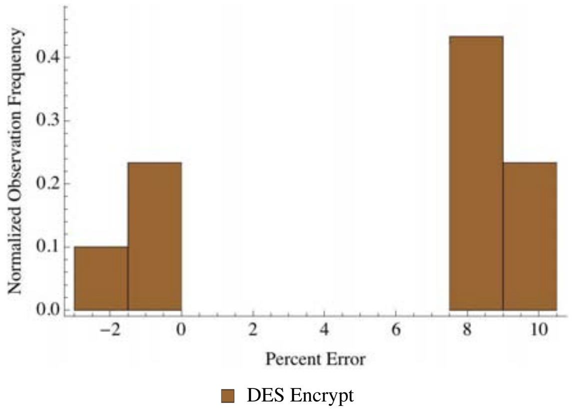


Figure 4.13: Percent error for gain/loss flow model for the DES encrypt application, calculated as $\frac{\text{(observed flow}-\text{modeled flow)}}{\text{modeled flow}}$.

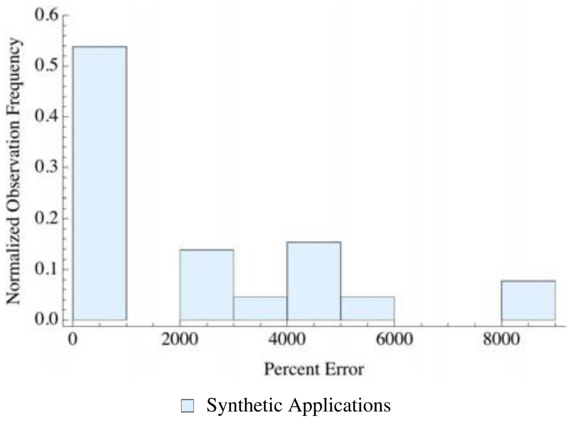


Figure 4.14: Synthetic application error for measured queue maximum occupancy vs. modeled queue maximum capacity. For all synthetic applications measured the modeled capacity is always greater. Empirically this shows that for this set of applications the $M/M/1$ queueing model provides a loose upper bound on buffering capacity.

Not shown is where the flow model can fail. Firstly, if any of the assumptions are violated, this model's results cannot be trusted. Second, as the number of processes on a single core increases, the error inherent in the simple model does as well. In our experiments we observed a strong correspondence between increasing percent error and the number of processes per core. Future work will investigate this relationship and perhaps explore the effectiveness of more complex sharing models.

4.3.3 The Queueing Model

The results for the synthetic, JPEG, and DES applications for an upper bound on queueing capacity are shown in Figures 4.14 , 4.15, and 4.16. These figures confirm that our model is conservative for estimating buffering capacity allocations. The modeling assumption is exponentially distributed arrival rates and service rates, while real service distributions are typically closer to deterministic (i.e., have a much lower coefficient of variation than an exponential), even if not fully deterministic. It is this distinction that yields conservative estimates for buffer requirements.

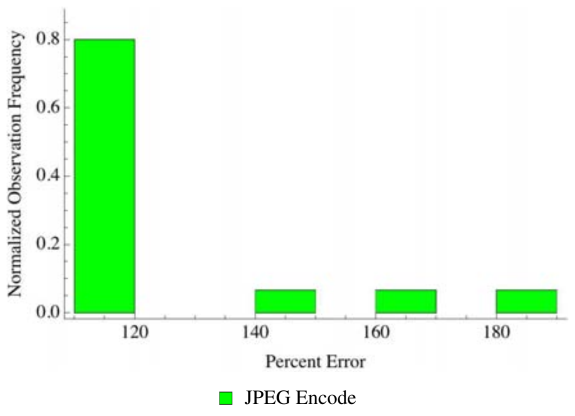


Figure 4.15: JPEG encode error for measured queue maximum occupancy vs. modeled queue maximum capacity. Three mappings are used across hardware and software.

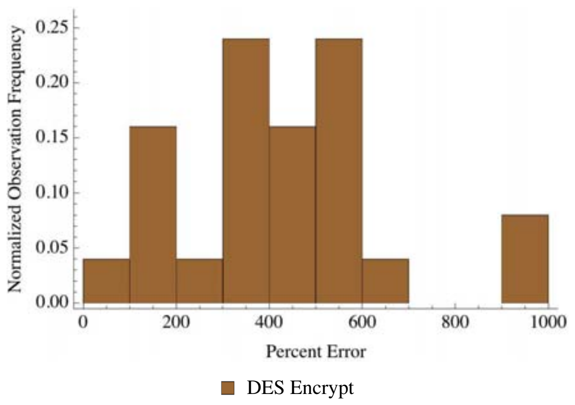


Figure 4.16: DES encrypt error for measured queue maximum occupancy vs. modeled queue maximum capacity. Four mappings are used on Machine 1, hardware on-chip encryption is not used.

4.4 Conclusions

With reconfigurable hardware, multi-core chips, general purpose graphics processors and other resources to choose from; application designers have a very difficult set of choices when selecting the best hardware for an application. A metric that is of particular interest to "big-data" applications is throughput. The analytic model presented in this chapter aims to provide an easy to use method for application developers to find the throughput for an application on a particular set of hardware resources while placing a conservative upper bound on queueing capacity necessary.

The model was tested using several synthetically generated applications, a JPEG encode application and a DES encrypt application. The empirical measurements show how the model performs under several conditions and how it can be used to solve for throughputs that are typically within 10% of reality and frequently much closer. This is quite impressive for a set of models that are explicitly trying to stay simple. A unique feature of the model presented is that it can be used across hardware and software platforms.

This chapter's evaluation primarily focused on offline optimization of streaming systems. Three hurdles prevented online optimization. The first hurdle, that of a computationally efficient model to explore the throughput of potential configurations was largely solved by the techniques laid out in this chapter. The next major hurdle is getting information while the application is executing, without destroying the performance characteristics that we expect of a well-running application. The last hurdle is how to trust a model for a hardware / software combination once we have enough information to use them.

While the flow model predictions were largely good, the subsequent queue occupancy predictions were often quite poor, drastically overestimating the necessary buffering. The techniques in the following chapters will describe methods to improve upon the understanding of how to apply these models and improve upon when they can be applied (online vs. statically).

Chapter 5

Best Case Execution Time Variation

Understanding the performance of software systems is often accomplished with the help of stochastic queueing models. As demonstrated within the previous chapter, these models don't always work as expected. The nominal assumptions verified within a controlled environment are not always transferable to real systems. The simple act of observation the behavior itself can change the observed behavior. One assumption directly influencing the performance of most queueing models is the execution time (service time) of each compute kernel and its distribution. Complete knowledge of the distribution (especially the worst case behavior) is generally futile for multi-core shared systems. Yet understanding it, however incompletely, is critical to selecting proper model formulations. Finding such a pattern also hints at the possibility of using automated pattern recognition to determine when and where such behavior could interfere with a modeling effort. When understanding complex phenomena, it is often the practice to find a useful bound. We contend that the minimal expected execution time variation of a system, or best case execution time variation (BCETV), is such a bound. By forecasting BCETV for a particular software and hardware combination, we hope to improve the *a priori* knowledge of a models' applicability. This chapter introduces the use of the mLevy distribution (derived from a Levy distribution) for characterizing the BCETV of short execution, compute bound kernels. A closed form expression for the probability density function, as well as its first and second moments are derived. The distributional assumptions and model are evaluated via empirical evaluation. Lastly, it is shown how this analytic distribution can be used when combined with other techniques to accept or reject a queueing model.

Several references simply assume that the distribution of a series of execution times should be Gaussian [97]. Other works (e.g., [134]) have shown some examples of successive execution

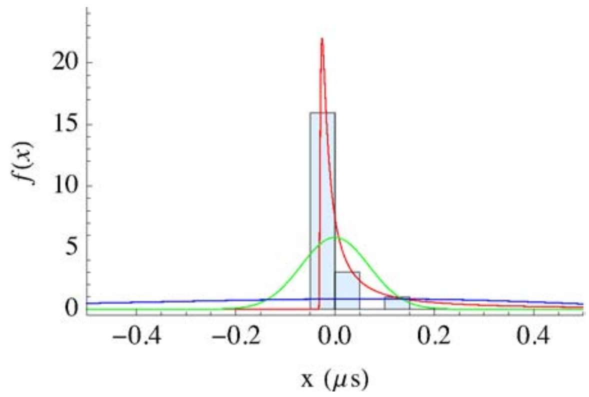
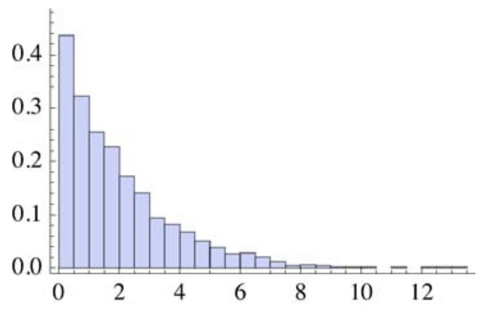


Figure 5.1: Histogram of the discrete PDF for a simple "no-op" workload execution time absolute error (light blue bars) in μs plotted against the PDFs of a fitted Gaussian distribution (green line), a Gumbel distribution (blue line) and the mLevy distribution (red line). Visually it is easy to see that our mLevy distribution is the best fit for this data set.

times that are not Gaussian with any high probability. Other phenomena such as worst case execution time have been modeled with the Gumbel distribution [67]. Empirically measured execution time noise for a minimal workload of "no-op" instructions (the difference between the nominal and measured execution times, plotted in Figure 5.1) exhibits a heavily skewed distribution. Simply assuming a Gaussian distribution (green line) overestimates the mass of one tail while underestimating the other. A Gumbel distribution (blue line) is arguably even worse. Some might posit that a Gamma distribution is a good fit, however the support exists only for $x \geq 0$ which fits neither reality or our use case as a noise model. Shifting the distribution is an option, however there is a better way. The mLevy distribution (red line, exact modifications to be discussed) is plotted against the same data, visually it is the best fit to the observed data.

Many performance models require details of the inner workings of the target processor [68]. When empirical evaluation is performed, often the results obtained are still uncertain. How well did the empirical evaluation sample the distribution of execution times? Even when detailed knowledge is assumed, or empirical evaluation is performed, there is still uncertainty in the values obtained. Causes of this execution time uncertainty can include cache behavior, interrupts, scheduling uncertainty as well as countless other factors. Distributional uncertainty can lead to poor stochastic model performance. Instead of focusing on the worst or even average case, our approach focuses on the best case and what this bound can do for the model decision making process. As an example, Figure 5.2a shows the distribution


(a) Exponential distribution ($\lambda = .5$)

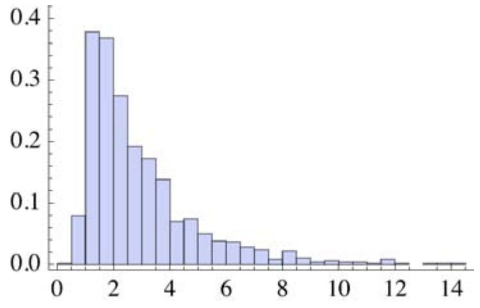

(b) Exponential distribution ($\lambda = .5$) with additive Gaussian noise

Figure 5.2: Stochastic models often make simplifying distributional assumptions about the modeled system. One common assumption is that of a Poisson arrival process (i.e., exponentially distributed inter-arrival times). This assumption is often violated by the "noise" that the hardware, operating system and environment impose upon the application. Figure 5.2a shows a nominal exponential distribution, while Figure 5.2b shows an example of a more realistic distribution containing Gaussian noise.

that a simple $M/M/k$ queueing model assumes for its inter-arrival distribution whereas Figure 5.2b might be closer to reality given a noisy system. One application of BCETV is to estimate how close a models' input assumptions will line up with reality assuming a best case variance. This could allow quick rejection of models whose assumptions are violated even in the best case.

BCETV is the minimum variation (error relative to the mean) which can be expected from any single observation of execution time. We assert that the minimal "no-op" workload can be used as a proxy for determining BCETV for short execution, compute bound kernels. In principle, these workloads should be quite deterministic in execution time, but clearly are not. We will show that the distribution of BCETV experienced by these workloads represents a reasonable lower bound "noise" model for nominal execution time. Utilizing empirical data, the Levy distribution is first truncated for finite support then redefined in terms of system parameters (i.e., processes per core and nominal execution time) to make it the mLevy. Evidence is provided that the mLevy distribution is a good match for BCETV, especially as the number of processes per core grows.

5.1 Methodology

The motivation to use a Levy distribution to model the best case execution time variation (BCETV) came from empirical observation. To be of use for modeling purposes however, several transformations must occur resulting in a modified distribution which we call mLevy. Ultimately we justify that decision by comparing model predictions to experimental observations. To that end we start by describing the process through which these data are collected. This is followed by the description of the mLevy distribution that we propose to use, and how to parameterize it.

5.1.1 Synthetic Workload

Our focus is the uncertainty in execution time of a running process due to factors other than the process itself. As such, we use an intentionally simple nominal workload so that the observed variation is due not to the application itself, but to other system related factors (e.g., operating system, hardware, etc.). Our nominal workload is the execution of a fixed number of null operations or "no-op" instructions [160]. Aside from no instructions at all, we assume that a null operation is the least taxing instruction. It follows from this logic that a series of null operations should present the most consistent execution time out of any real executable instruction sequence.

One aspect under study is how changing the nominal workload time changes the observed variation in actual execution times. In order to produce a workload of "no-op" instructions that is calibrated to a specific nominal execution time we use sequences of instructions of various lengths which are timed and then used as input for regression to produce an equation for the number of instructions to use for each nominal execution time. Calibration timing is performed while the timed process is assigned to a single core and executing with no other processes.

In theory any duration of workload could be created using this method, however in practice the file sizes become prohibitively large proportionate with the frequency of the processor and the desired running time (e.g., Platform A from Table 5.2 requires approximately 10 million"no-op" instructions for each second of execution time). Other approaches that reduce the file size could be used such as looping over a calibrated number of “no-op” instructions, however we’ve chosen to use the simpler aforementioned approach because it reduces the possibility of variation due to other factors, such as branching. Our method also assumes that cache pre-fetching will eliminate virtually all instruction cache misses which should then have no appreciable effect on the actual run time (which was verified for the experimental hardware through instrumentation). One concern with huge numbers of instructions is that translation lookaside buffer (TLB) misses might increase the observed variation. With TLB misses we would expect an increase in the overall observed variation with longer duration executions with a random pattern (dependent on other processes operating on the same core, TLB algorithm, etc.). As we will show below, this is not the case; more variation is observed for short execution times.

5.1.2 Hardware, Software, and Data Collection

To enable empirical data collection, a test harness was created that executes the synthetic “no-op” workloads while varying numbers of processes per core, nominal execution times, and execution platforms. As the synthetic workload processes are executing, the parameters in Table 5.1 are collected. In order to reduce the possibility that results gleaned from this study might be an artifact of a particular hardware platform or operating system, two different platforms are used as shown in Table 5.2 (two of platform A and seven of platform B). All platforms support a version of the Linux completely fair scheduler [129] which will be exclusively used during data collection.

Table 5.1: Experimental Parameters

Parameter	Symbol
Nominal Execution Time	t_N
# Processes per Core	p
Voluntary Context Swaps	v
Non-Voluntary Context Swaps	nv
Actual Execution Time	t_A
Execution Time Noise ($t_A - t_N$)	Δ

Each data point collected consists of the dimensions outlined in Table 5.1. Nominal execution times vary from $0.25\mu s$ through $3.7ms$ with observations at an interval of $0.25\mu s$ throughout the range. The number of workload processes per core varies from 1 through 20 processes.

Table 5.2: Hardware and Operating Systems

Label	Processor	Operating System
A	Intel E3 1220	Fedora 19, Linux Kernel v. 3.10.10
B	2 x AMD Opteron 2431	CentOS 5.9, Linux Kernel v. 3.0.27

Each sharing and nominal execution time pair is executed 1000 times to ensure a good distribution sample. The synthetic workloads are run on one of two of platform A or on one of seven of platform B from Table 5.2. In total 100+ million observations are made. Two factors limited the range of viable execution times: the lower bound on timer resolution (see below) and the memory needed to generate workloads of longer lengths (disk to store and physical memory to compile).

Generated data is divided into two sets. The first, a "training" set (of size 10^6) is segregated using uniform random sampling. The rest of the data is used for model evaluation and will be referred to as the "evaluation" set. We specifically want to judge the applicability of this noise model to multiple hardware types and operating systems using the same scheduler.

There has been much discussion about the best and most accurate way to time a section of code [34]. There are many methods including processor cycle counters and operating system "wall-clock" time. Given our reliance on empirical data for modeling and evaluation we feel it is important to cover how our timing measurements are made. In many cases, the use of a simple time stamp counter is effective assuming that the process will never migrate to another core. Another issue to consider is frequency scaling which can lead to wildly inaccurate timings when utilizing the processor cycle counter. To alleviate some of those concerns and provide a relatively universal timing interface we developed a system timer thread that utilizes the x86 time stamp counter instruction on a single reference core to update a user space timer. When a process or thread requests the current time, an in-lined function copies the current time struct which has two time references and it compares the two times. If they are the same then the calling code can be sure that the time has been fully updated and the function returns, if not the code loops until the values match. Frequency scaling is turned off for the time update thread.

This timing method has several advantages: (1) it is entirely in user space, (2) it is lock-free, and (3) it is monotonic even when the timed thread is shifted to a new core. Two concerns with this approach stem from the copying operation. How long does it take to copy the

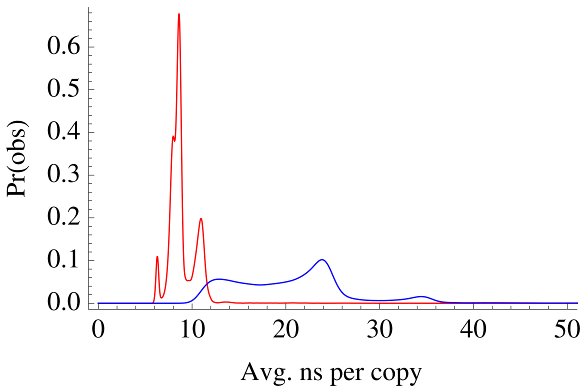


Figure 5.3: Smooth histogram of 10^6 data points each representing timed averages of 500 copy operations, first on the same NUMA node (red line) and then across different NUMA nodes (blue line). The performance of a copy on the same NUMA node seems to be much more consistent.

timer struct on a target system and what happens when there are multiple Non-uniform Memory Access (NUMA) nodes [113]? To test the latter of these concerns a benchmark was constructed to ascertain how long a copy operation takes when the copy is from the same NUMA node as the calling process and when the timer thread and requester thread are on differing NUMA nodes. The results of this are shown in Figure 5.3 for platform B from Table 5.2. What we've found is that reading memory allocated on a NUMA node other than the one closest to the time requesting process the access times can vary somewhat. Another related issue is cache line invalidation. Invalidates sent to cache lines storing the clock structure, and subsequent updates can add even more timing variation. To eliminate both issues, all subsequent experiments within this chapter use a single NUMA node (same socket with shared cache).

A common problem with highly accurate timing via software is determining what is ground truth. Short of an external atomic clock, there are only varying degrees of truth. In order to determine the precision and accuracy of our measurements, a standardized workload is created with a series of "no-op" instructions of varying lengths. Each "no-op" length is timed using either the x86 `rdtsc` instruction or the POSIX.1-2001 `clock_gettime()` function. Figure 5.4 shows the inter-quartile range (25^{th} to 75^{th} percentiles) difference of each timing measurement as a function of the length of the "no-op" instruction sequence, This plot informs us about the stability of the two timing methods. The system call to `clock_gettime()`

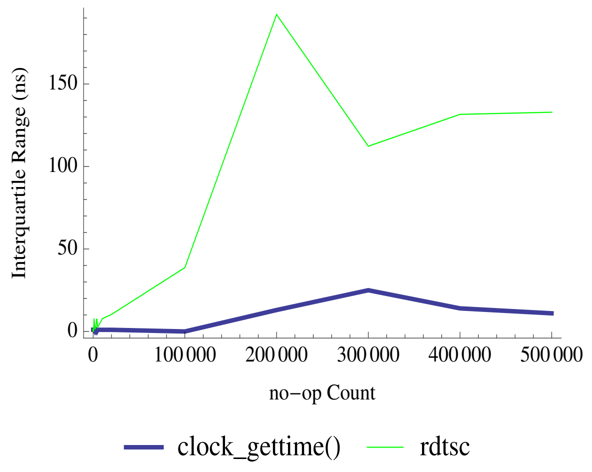


Figure 5.4: Interquartile range difference (IQRD $= 75^{th} - 25^{th}$) in nanoseconds for the times measured for each set of "no-op" instructions (number instructions listed on x-axis). Each instruction length was executed $10^6\times$ for each method. The IQR gives a visual representation to the stability of measurements for these two timing methods.

is more stable than the `rdtsc` instruction, especially for these small workloads. A hypothesis as to why it is more stable is that the measurement of actual workload time is small relative to the time it takes to perform a system call. To test this theory the timing methods themselves are timed by executing five hundred of each method (either the `rdtsc` x86 instruction or the `clock_gettime()` function) and using the average execution time of all five hundred to extrapolate the time to execute a single instruction. In this experiment the `rdtsc` instruction is used as the reference timer on platform A from Table 5.2. As expected (and shown in Figure 5.5) the system call to `clock_gettime()` takes almost $3\times$ as long on average compared to the x86 `rdtsc` instruction. For this reason, we exclusively use the `rdtsc` instruction for all empirical timing measurements in this work.

5.1.3 Distribution

Figure 5.1 provides a qualitative indication that a Levy distribution makes a good choice for modeling the noise present in execution times of a nominally fixed minimal workload (the proxy for BCETV). Quantitatively Table 5.3 summarizes the p-values for each distribution (higher is better), the table shows the minimum, maximum and mean values. The Levy distribution is the only distribution with greater than 10% of the data having a p-value $\geq .01$.

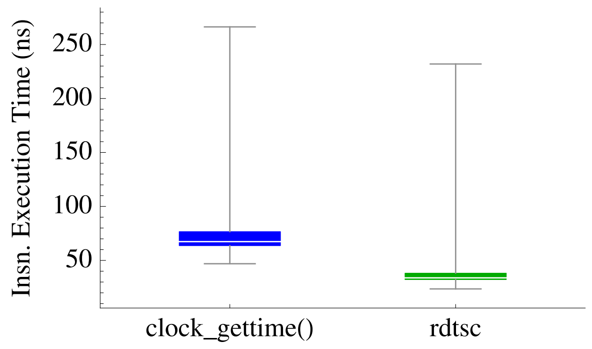


Figure 5.5: Box and whisker plot showing a speed comparison of the `rdtsc` x86 assembly instruction compared to a `clock_gettime()` call to the Linux real-time clock. The `rdtsc` instruction's $25^{th} - 75^{th}$ percentiles are almost identical at the nanosecond scale. The `clock_gettime()` function overall takes much more time (approximately $3\times$).

Table 5.3: Summary of Anderson-Darling Goodness of Fit Test

Distribution	Min	10^{th}	50^{th}	90^{th}	Max	Mean
Gaussian Distribution	0	1.17×10^{-15}	1.68×10^{-14}	3.0×10^{-5}	.719	.002
Levy Distribution	0	2.11×10^{-15}	2.15×10^{-14}	.038	.803	.025
Gumbel Distribution	0	8.93×10^{-16}	1.97×10^{-14}	6.39×10^{-06}	.357	.002
Cauchy Distribution	0	3.89×10^{-16}	1.34×10^{-14}	.002	.771	.009

Realized execution time is the sum of a nominal (mean) execution time and a noise term. If the nominal execution time is represented by a random variable N, and the noise is represented by a random variable V, then the realized execution time $R \sim N + V$. The goal of this work is to find a distribution to represent a lower bound for V which we term BCETV.

The Levy distribution [145] has a closed form probability density function (PDF, shown in Equation (5.1)), however in general it has no defined moments. Observations from the empirical data lead to a solution. Whereas the tail of the Levy distribution is infinite, the noise present within the real execution times is finite. The support for the empirical distribution, not surprisingly, is correlated with both the nominal length of execution and the number of processes assigned to a single compute core. This leads to the consideration of a Levy distribution that is truncated at a point represented by a new parameter Ω. The truncated Levy distribution is first defined using the truncation method of Equation (5.2) as Equation (5.3) where $F(\cdot)$ is the CDF of the PDF denoted by $f(\cdot)$. (Note: $\mathrm{erfc}(x)$ is the complement of the Error Function, $1-\mathrm{erf}(x)$, and E_i is the exponential integral function [2].) In order to make the equations more concise, $w = \frac{\beta}{2(\alpha-\Omega)}$ and $z = \frac{\beta}{2(x-\alpha)}$.

$$f_L(x;\alpha,\beta) = \frac{e^{-z}\,(2z)^{3/2}}{\sqrt{2\pi}\beta} \tag{5.1}$$

$$f_{mL}(x;\alpha,\beta,\Omega) = \frac{f_L(x;\alpha,\beta)}{F_L(\Omega;\alpha,\beta) - F_L(-\infty;\alpha,\beta)} \tag{5.2}$$

$$f_{mL}(x;\alpha,\beta,\Omega) = \frac{\sqrt{\beta}e^{-z}}{\sqrt{2\pi}(x-\alpha)^{3/2}\left(\mathrm{erfc}\sqrt{-w}\right)} \tag{5.3}$$

Restricting the use of the truncated Levy distribution, mL, to $x \leq \Omega$ and $x > \alpha$ leads to a closed form expression of the mean as shown in Equation (5.4). Lastly, a variance is also defined as Equation (5.5).

$$\mu_{mL}[\alpha,\beta,\Omega] = \frac{\beta\Gamma\left(-\frac{1}{2},-w\right)}{2\sqrt{\pi}\mathrm{erfc}\left(\sqrt{-w}\right)} + \alpha \tag{5.4}$$

$$\sigma^2_{mL}[\alpha,\beta,\Omega] = \frac{(\alpha-\Omega)^2\left(\frac{E_{\frac{5}{2}}(-w)\left(\sqrt{2\pi}\beta^{3/2}\mathrm{erfc}\left(\sqrt{-w}\right)+3(\Omega-\alpha)^{3/2}\left(4e^{w}-3E_{\frac{5}{2}}(-w)\right)\right)}{\sqrt{\Omega-\alpha}} + 4(\alpha-\Omega)e^{2w}\right)}{2\pi\beta\mathrm{erfc}^2\left(\sqrt{-w}\right)} \tag{5.5}$$

Our next task is to determine an appropriate parametrization of the truncated Levy distribution. We accomplish this task by fitting it to empirical measurements. The “training” data to be fit to, are sorted into groups W_{p,t_N} which are indexed by the number of processes sharing a core, p, and the nominal execution time, t_N (see Equation (5.6a)). Within each group, the execution time noise is computed for each observation as in Equation (5.6b).

$$W_{p,t_N} = \bigcup_i obs_i \in p, t_N \tag{5.6a}$$

$$\Delta = t_A - t_N \tag{5.6b}$$

Separately for each group W, Maximum Likelihood (ML) techniques are used to find the best parameters for a number of distributions, including the truncated Levy distribution that we are proposing. The quality of the distributions' fit to the empirical data is judged via an Anderson-Darling [11] goodness of fit test as shown in Table 5.3 (chosen because of the weight given to the tails of the distributions compared to other tests such as the Kolmogorov-Smirnov test [41]).

5.1.4 Parameterization and definition of mLevy

While the ML techniques used above can yield a parameterization for the truncated Levy distribution that is well matched to the data, in general ML techniques are quite computationally expensive and also require substantial support to be effective. An alternative is to redefine the truncated Levy distribution parameters, α, β, and Ω, in terms of a subset of the parameters in Table 5.4. This is the path we chose, which results in the mLevy distribution. The exact derivation and parameter selection is described in detail below.

The selection of parameters from Table 5.4 is reduced based on the intuition that the nominal execution time and number of processes sharing a core will have the largest impact on the true execution time. Given the design of the minimal compute kernel, it is expected (and confirmed) that there are zero voluntary context swaps allowing the variable to be discarded. A Pearson correlation coefficient between the target variables and the training data (Table 5.4) quantifies the intuition about the remaining parameters.

Table 5.4 summarizes the correlations within the training set between the execution time noise, Δ, and the other parameters. For the entire training set there is a weak correlation

Table 5.4: Correlation Between Target Predictors

	nv	t_N	p
Δ	.508	.771	-.0056

between the number of processes sharing a core and the execution time noise. There is a strong correlation between the nominal execution time and the noise. Not shown is the co-variance between the non-voluntary context swaps and the number of processes per core which implies a lack of independence. The models considered therefore consist only of the two independent parameters p and t_N.

Using simple linear regression to find coefficients for p and t_N that best fit the parameters for α, β and Ω found by ML, the relationships in Equation (5.7) are found with the following assumptions: $p \in \mathbb{Z} \ \wedge \ p > 1$ and $t_N \in \mathbb{R} \ \wedge \ t_N > 0$. To redefine the truncated Levy distribution as defined in Equation 5.3 to the mLevy distribution, several constraints must be added, namely: Equation (5.7a) is expected to have a negative range for the entire domain, Equation (5.7b) is positive for the entire domain and Equation (5.7c) is greater than α for the entire domain. A limitation of these equations is the range of data used to create them. It is expected that α will not continue to decrease as $t_N \to \infty$ and the β, Ω parameters probably have limitations as well; however these equations are supported through the range of data specified in Section 5.1.2.

$$\alpha = 4.75 \times 10^{-9} p - 0.220 t_N \tag{5.7a}$$

$$\beta = 4.19 \times 10^{-10} p + 0.007 t_N \tag{5.7b}$$

$$\Omega = 3.19 \times 10^{-6} p + 0.742 t_N \tag{5.7c}$$

Using Equations (5.7), which predict α, β and Ω based on p and t_N, the PDF and mean of the mLevy distribution can now be described in terms of p and t_N as shown in Equations (5.8) and (5.9), respectively. The variance of Equation (5.8) is a straightforward algebraic manipulation of Equation (5.5).

$$f_{mL}(x;p,t_N) = \frac{8.2\times10^{-6}(\sqrt{1p+1.6\times10^{7}t_N})e^{\frac{0.044p+6.98\times10^{5}t_N}{p-4.6\times10^{7}t_N-2.1\times10^{8}x}}}{(-4.8\times10^{-9}p+0.22t_N+x)^{3/2}\mathrm{erfc}\left(\sqrt{0.003\ -\frac{0.003p}{p+3.02\times10^{5}t_N}}\right)} \tag{5.8}$$

$$\mu_{mL}[p,t_N] = \frac{(1.2\times10^{-10}p+0.002t_N)\Gamma\left(-\frac{1}{2},0.003-\frac{0.003p}{p+3.02\times10^{5}t_N}\right)}{\mathrm{erfc}\left(\sqrt{0.003,-\frac{0.003p}{p+3.02\times10^{5}t_N}}\right)} + 4.8\times10^{-9}p-0.22t_N \tag{5.9}$$

5.2 Results

How well does the mLevy distribution approximate the actual BCETV observed while executing a nominally deterministic compute bound kernel? We will focus our evaluation on the PDF expressed in terms of processor sharing, p, and nominal execution time, t_N, presented above as Equation (5.8).

The Anderson-Darling (AD) goodness of fit test of Table 5.3 is, frankly, not very promising. Yet, we already know from Table 5.3 that the truncated Levy is the best out of the listed distributions used to model the training data. It is not at all surprising that our overall p-value when using AD is not very high ranging from 0 to 0.73. What is welcome news is that AD is not the only metric available, as it is relatively ineffective at identifying portions of the parameter space that have a good vs. a poor fit.

A second measure of how well the mLevy distribution fits empirical data is how well the moments match. When comparing the mean of the empirical data sets to that predicted by Equation (5.9), the differences are effectively below our ability to differentiate based on the techniques described in Section 5.1.2 (i.e., the difference is $\ll 10^{-12}s$). Comparing the variance for the mLevy vs. the empirical measurements results in an r-squared value of 0.69, which indicates a reasonable degree of correlation between model and data, though the alignment between the two is clearly not perfect.

While the above quantitative assessment of the mLevy distribution's match with empirical observation make it clear that the model is not perfect, we must keep in mind the fact that modelers can often exploit individual models that are far from perfect. Given prior use of models that are much more divergent from reality than our proposed mLevy distribution, there is the real potential for benefit from the ability to use a distribution that more closely matches empirical measurement than previous models.

We continue the assessment of how well the mLevy distribution characterizes the noise in observed execution times by presenting QQ-plots for three distributions relative to the empirical data (see Figure 5.6). The first column of plots is the mLevy distribution of Equation (5.8), the second column is a Gaussian distribution, and the third column is a Cauchy distribution. The latter two distributions are parameterized by fitting to the data using ML techniques. For each distribution, 4 distinct QQ-plots are shown, separating the processor sharing variable, p, into quartiles. The first (top) row represents the range $1 \leq p \leq 5$, the second row represents the range $6 \leq p \leq 10$, the third row represents the range $11 \leq p \leq 15$, and the fourth (bottom) row represents the range $16 \leq p \leq 20$.

First consider the results in Figure 5.6(g) and (j), which include the mLevy distribution and significant processor sharing. The model and the empirical data align well, the best evidence yet that the mLevy is a good execution time noise model. Next consider the results in Figure 5.6(a) and (d), which include the mLevy distribution and little processor sharing. In this case, there is reasonably good alignment at the low end of the range, but the empirical data has slightly less variation than the model at the high end of the range. Finally, note that the alignment between model and empirical data is noticeably worse for both the Gaussian and the Cauchy distributions across the entire range of p.

From the above we conclude that the mLevy distribution is a relatively good proxy for BCETV. The distribution of BCETV can in turn be used in many ways. To demonstrate the utility of BCETV, we explore the mean queue occupancy (MQO) of a single queue system when noise is added (as in Figure 5.2). The single queue system operates as two threads with one way communication that is designed to have an exponential inter-arrival and service time distribution (i.e., workload is dependent upon an exponential random number source). A simple model for MQO is the $M/M/1$ queueing model, it expects the inter-arrival times to be exponentially distributed. We posit that the farther from this distribution the actual system is, the greater the model's predictions will differ from empirical reality. The Kullback-Leibler (KL) divergence [112] is an information theoretic measure of the divergence between

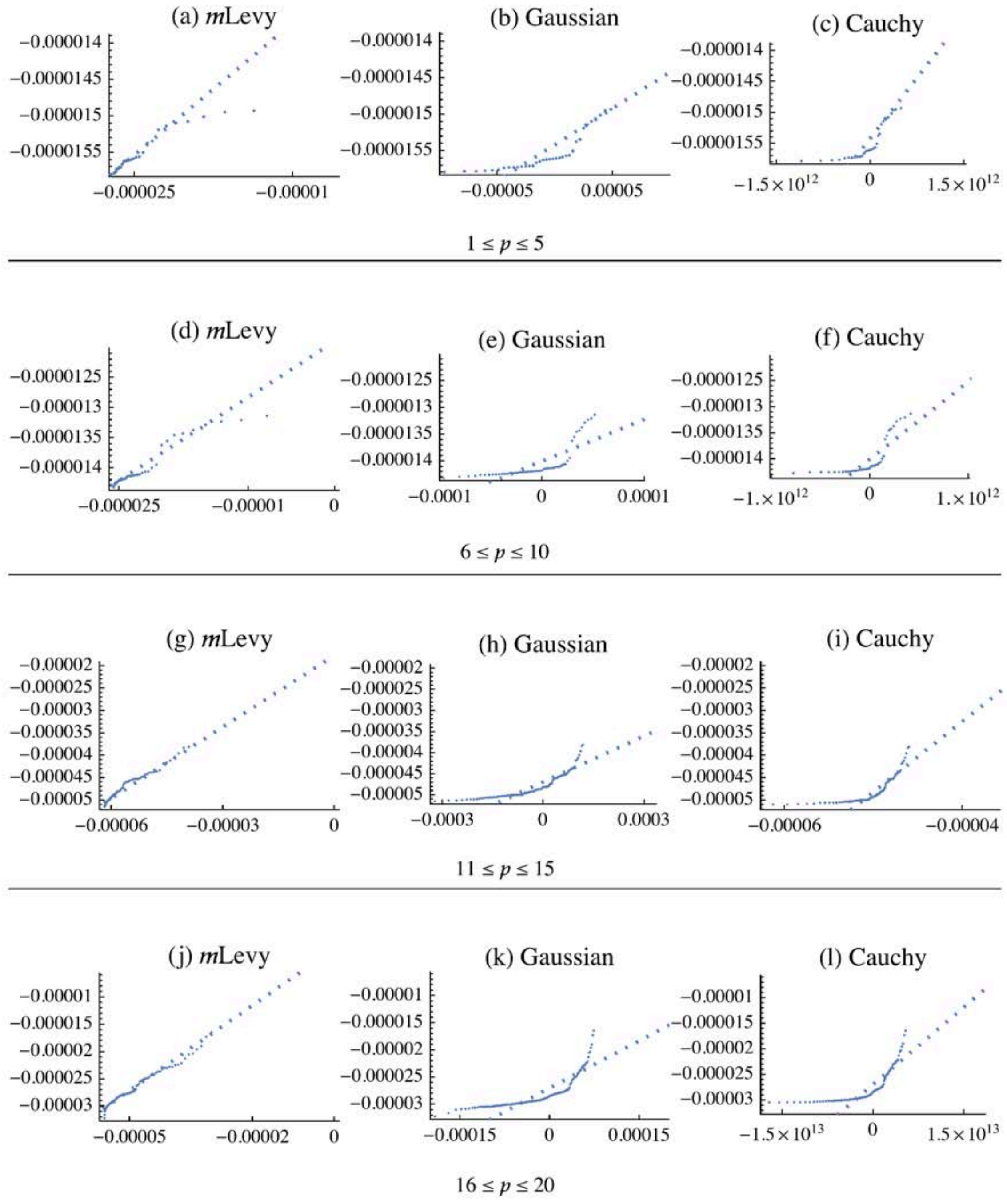


Figure 5.6: QQ-plots comparing empirical data (vertical axis) to the analytic distributions (horizontal axis) for the mLevy, Gaussian, and Cauchy distributions. The dashed line shows the ideal response.

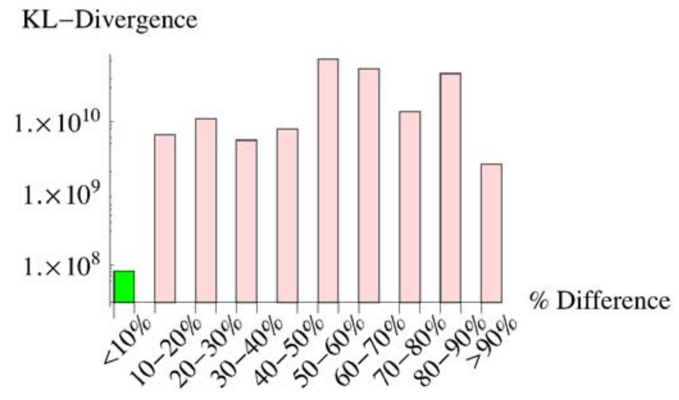


Figure 5.7: The y-axis shows the median KL divergence between the $M/M/1$'s expected exponential inter-arrival distribution and the lower bound predicted by convolving the exponential distribution with Equation 5.8. The x-axis is the percent difference between the mean queue occupancy predicted by an $M/M/1$ model and the actual measurements from a single queue system designed to have a perfectly exponential workload. The lowest KL divergence (green bar) is associated with more accurate predictions.

two distributions (zero being a perfect match). We are interested in how far the noised distribution (expected reality) as predicted by the convolution of the exponential distribution and Equation 5.8 differs from that expected by the $M/M/1$ MQO model. With a divergence of zero we should expect to find a very close match between modeled and experimental MQO. At higher divergences (the exact amount is an open question given the information theoretic metric) we don't expect the $M/M/1$ model to be very accurate. Figure 5.7 is a summary of median KL divergences (y-axis) separated by percent model accuracy (calculated as $\frac{|\text{modeled MQO} - \text{measured MQO}|}{\text{measured MQO}} \times 100$, x-axis) for $6,000+$ separate executions of the single queue system described above on the platforms shown in Table 5.2. It shows that lower KL divergence (green bar) between the expected exponential and that convolved with the BCETV distribution, is associated with more accurate MQO predictions. This implies that BCETV can be used as a predictor for model rejection (at least with a Markovian arrival process, and perhaps others).

5.3 Conclusions

This chapter demonstrated a noise model (based on the mLevy distribution) that appears to work far better than a simple Gaussian assumption, in fact far better than multiple other distributions that have been used by others. It has been shown to work for at least two differing platform types (see Table 5.2) using the same fair scheduling algorithm. We've derived the expressions for the PDF and the first two moments of the truncated Levy distribution that has finite support. Through empirical data collection, a model is derived that can be used to parameterize the truncated Levy distribution without resorting to computationally expensive parameter fitting. Using this model to redefine the truncated Levy distribution results in the mLevy distribution which is defined in terms of the number of processes per core and the expected execution time per firing.

In Figure 5.6 we showed how well the quantiles of the mLevy distribution match to the quantiles of the empirical data. We've also noted that the fit between the model and empirical data gets better as more processes are added per core. This is in keeping with our original assumption that a single process on a single core should exhibit its native distribution, in our case purely deterministic, or close to the nominal mean, t_N. The models demonstrated here are only validated over the range of empirical data that we've collected. For future work, we would like to extend the parameter estimators for $p > 20$ and higher nominal execution times t_N. This Chapter also assumes that each workload being modeled is homogeneous. An unbalanced workload (where one thread has higher service requirements than another), might experience slightly greater or lesser variation. In cases where the workloads under consideration are not correlated (i.e., no dependent execution) the total of the expected service times can be used with the mLevy and the fraction for each sub-workload could be used. This remains to be seen and has yet to be verified. For threads with similar service times, this limitation should not be an issue.

One concern with our approach is also one of its strengths, that it is based on wide empirical sampling. Is this noise model really applicable to multiple hardware types, or were our choices simply judicious? Could other parameters in addition to nominal execution time, t_N, and the number of processes per core, p, provide a better estimate on other platforms (e.g., alternative instruction sets). One potential application of this noise model is as a minimal expected noise for all workloads since the "no-op" loop itself is a minimal workload and therefore an approximate lower-bound on any real compute operation (with the limitations and assumptions discussed previously).

Before moving on to more advanced methods to accept or reject models for applications, one more critical piece is needed. If our modeling process is to be done while the application is executing then we need a method to approximate the non-blocking service rate of each kernel within a streaming system while it is executing. The next chapter will cover exactly what is meant by non-blocking service rate, other instrumentation methods and describe their implementation within the open source RaftLib framework (see Chapter 3).

Chapter 6

Dynamic Instrumentation

6.1 Introduction

Stream processing is a compute paradigm that promises safe and efficient parallelism. Modern big-data problems are often well suited for stream processing's throughput-oriented nature. Many small-data jobs, such as streaming images from a cellular phone are also well suited to stream processing. What both of these types of applications have in common is that their workloads are often very dynamic. This can be due to many factors such as: memory bandwidth utilization, functional unit usage, cache utilization, etc. Tuning the parameters of the application such as the number of parallel executing kernels, sizes of buffers between compute resources, location of buffer in non-uniform memory access systems, location of parallel executing compute kernels, and many others can drastically improve (or worsen if done poorly) the performance of long running computational tasks (short running tasks will likely see execution time increase due to the added overhead of dynamic adaptation). Most current techniques at instrumentation for streaming systems are either static, adapted from other types of systems (e.g., MPI), or are ad-hoc. This chapter describes our approach to dynamic instrumentation for stream processing systems; not only is it designed to be low overhead, but it can be turned on and off during execution.

Stream processing stitches together multiple compute kernels into a coherent single application. The efficiency of the resultant application is largely dependent on the ability of the runtime to manage data movement and locality of processing. All possible care is typically taken by library and language authors to minimize the actual copying of data from one kernel to the next and maximize cache line re-use with the end goal of minimizing what is frequently the most expensive part of a streaming computation: data movement (most expensive in terms of energy [27, 105] and time [200]). If the memory management unit energy is also considered this estimate grows even further. By managing buffer allocations efficiently we can cut energy usage. Going further, by managing the precise size and placement of buffers we can increase the utilization of each DRAM firing. In order to optimize buffer size and placement, models are needed. In order to do this while an application is running we also need efficient instrumentation to feed these models.

Most techniques to optimize communications links in streaming applications use queueing network models or network flow models. Ubiquitous to many of these models is the non-blocking service rate of each compute kernel. Classic approaches assume a stationary distribution. This carries the assumption that both the workload presented to the compute kernel and its environment are stable over time. One only has to look at the variety of data presented to any common application to realize that the assumption of a persistent homogeneous workload is naive. With the popularity of cloud computing we also have to assume that the environment an application is executing in can change at a moments notice, therefore we must build applications that can be resilient to perturbations in their execution environment. In this chapter we take a control-theoretic approach that assumes that there will be steady state behavior over some small fixed interval over-which actionable data can be collected and meaningful solutions fed back into the system.

Statistics such as mean queue occupancy, and even occupancy histograms are relatively easy to collect. Conceptually it is also simple to consider these as "dynamic" statistics, being able to be collected in realtime. The engineering difficulty collecting these statistics efficiently is data movement, which careful collection and compression has largely solved [116, 132, 139, 141]. These statistics are excellent measures for learning when an application is stressed, and where high queue utilization can point to bottlenecks within the streaming system. These statistics are not enough, however, to effectively model future behaviors of the system. Nor are they sufficient for many modeling purposes.

The contribution of this chapter is a method that enables online service rate approximation. The service rate, plainly defined, is how fast a compute kernel can process data. Queueing literature typically calls this the service rate (literally rate of servicing jobs or data). What is really interesting is how fast each compute kernel can process in isolation, data regardless of what is going on up or downstream. Picture an assembly line with three workers (convenient stick figure representation shown in Figure 6.1). Each worker can only perform work on jobs

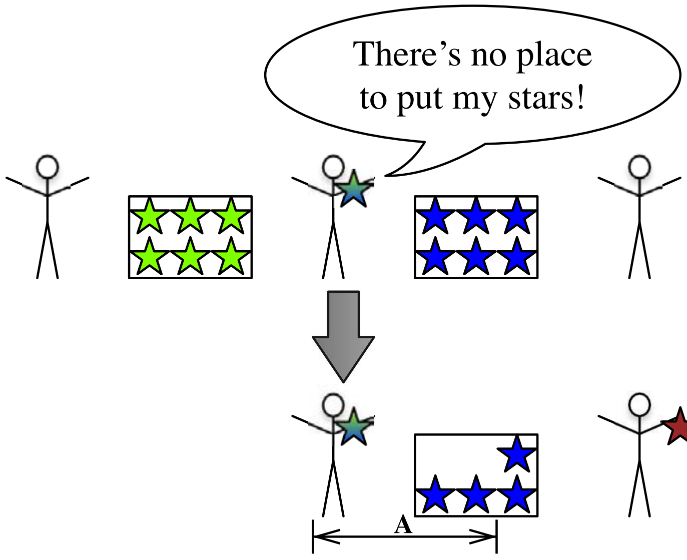


Figure 6.1: Factory workers in a row must wait till there is space to put their finished items (stars) before moving on to another star. Observing the middle worker, eventually the last one removes enough items from the bin and the middle worker is able to process stars over the period labeled "A." It is this period that leads to an online estimation of non-blocking service rate.

as they appear in their work queue (shown to the workers left). Each worker can also only perform work if there is a place to put work once it is complete (i.e., the outbound queue, shown to the workers right, must have room). How can we determine the rate at which the middle worker can process jobs, without being limited by the processing rates of the first and last workers in the line. This is exactly what we want to find out, and it is aptly named the non-blocking service rate since it is the rate where the middle worker isn't stopped or blocked by anyone. The key observation is that there are brief moments available when this condition exists, as in Figure 6.1, we want to observe the condition labeled A when the middle worker has room to put his stars. To show that this condition exists, even at the speeds with which modern computers execute, Figure 6.2 shows a simulation for a queue with a high utilization. The blue line (interpolated from discrete observations) represents queue occupancy, the red line overlay indicates the condition where the process modulating the arrival rate can be observed making non-blocking writes to the queue. This information is useful for several modeling and tuning tasks: feeding analytic queueing models, flow models and for making parallelization decisions.

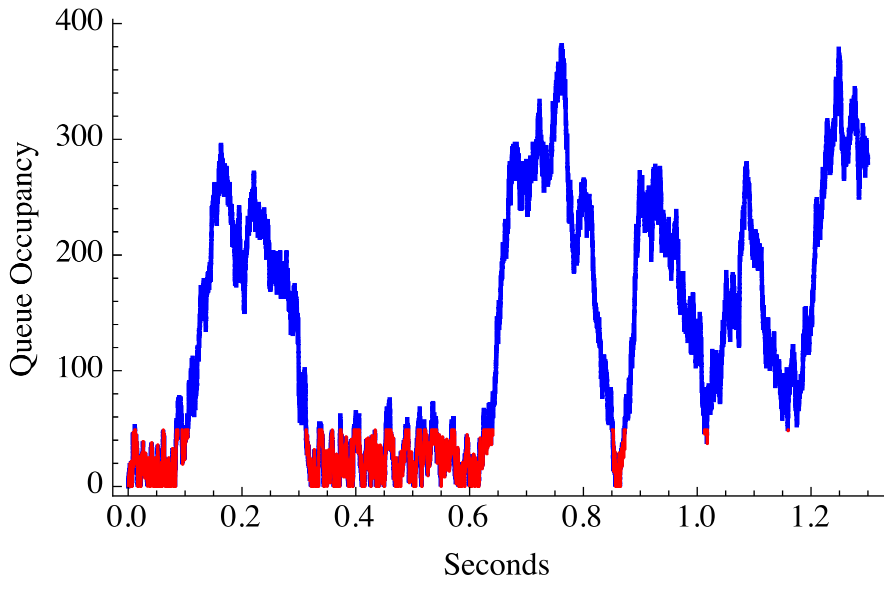


Figure 6.2: Figure 6.1 shows, at a macro scale, the situation that gives rise to the opportunity to observe non-blocked reads and writes while the application is executing. This figure shows those opportunities in red overlaid on a simulated $M/M/1$ queue separated from a Jaksonion network [95]. The arrival rate to this queue is 9.99 MB/s, the departure rate is 10 MB/s, resulting in a utilization ρ of .999.

An example of a simple streaming application that will be used throughout this chapter is shown in Figure 6.3. The kernel labeled as "A" produces output which is "streamed" to kernel "B" over the communication link labeled "Stream." These communication links are directed (one way). Strict stream processing semantics dictate that all of the state necessary for each kernel to operate is compartmentalized within that kernel, the only communication allowed utilizes the stream. State compartmentalization and subsequent one-way transmittal of state via streaming comes with increased communications between kernels. Increased communication comes with multiple costs depending on the application: increased latency, decreased throughput, higher energy usage. No matter what the cost function is, minimizing its result often involves optimizing the streams (queues and subsequent buffers) connecting individual compute kernels. The buffers forming the streams of the application can be viewed as a queueing network [19, 120]. It is this network that we want to optimize while the application is executing.

What follows is a brief description of the instrumentation within the RaftLib framework followed by the key contribution of this chapter which is low-overhead service rate determination.

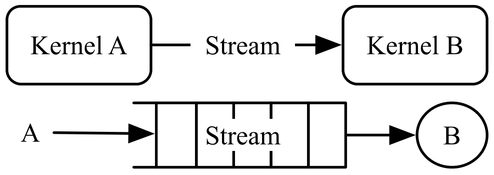


Figure 6.3: The simple streaming application at top has two compute Kernels A & B with a single stream connecting them. The corresponding queueing network is a single server B with a single queue fed by the arrival process generated by Kernel A.

6.2 Instrumentation Considerations

The simple act of observing a rate can change the behavior being observed. This phenomena is more obvious with large real world observations (e.g., observing animal behavior), however it is equally true for micro ones. Data observation might not make the data run away, however each observation requires non-zero perturbation to record it (e.g., a copy from the incremented register at some interval). Alternatively, saving every event under observation can quickly overwhelm the hardware and operating system. Trace files, even when compressed, can grow rapidly. Determining the service rate with trace data in a streaming fashion (saving none of it) might be possible, however it still increases traffic within the memory subsystem which is less than desirable in high performance applications. Concomitant to reducing communications overhead associated with monitoring is moving any computation associated with that instrumentation out of the application's critical path. To accomplish this, our instrumentation scheme (implemented within RaftLib) uses a separate monitoring thread. The reduction in overhead to the critical computation path comes at a cost: an increase in sensitivity due to timing precision and the probability of noise within any observations.

Figure 6.4 depicts the arrangement of the instrumentation system under consideration at a high level. A simplified streaming application with only two kernels is shown, connected by a single stream. Each kernel is depicted as executing on an independent thread. A monitor (depicted as an eye), performs all the instrumentation work, it executes on an independent thread as well. Each of these threads is scheduled by the streaming run-time and the operating system (either user space fiber threads or POSIX-compliant kernel threads, details of scheduling are discussed in Chapter 3). Each of these threads also could execute on independent processor cores or a single multiplexed core. Each abstraction layer has the potential to impart noise on any observations made by the monitor, the methods proposed here must deal with and operate in spite of this complexity.

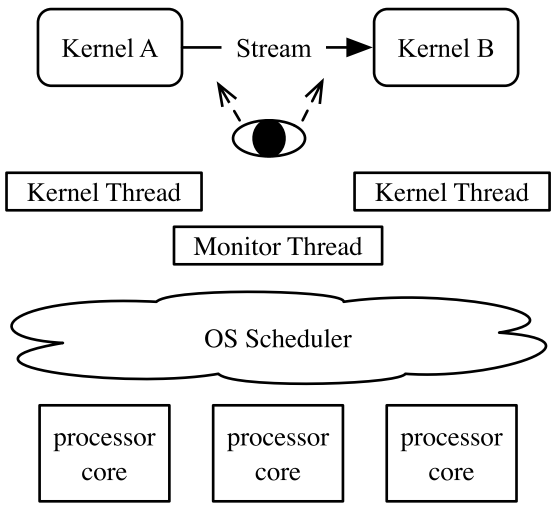


Figure 6.4: High level depiction of the abstraction layers coalesced around a simple streaming application with two compute kernels. An independent monitor thread serves to instrument the queue. Both the kernel threads and monitor threads are subject to the runtime and operating system (OS) scheduler.

As a result of the reliance on observing one the queue from another thread, accurate timing is absolutely necessary. It should be obvious that any variations in the denominator for things like rate calculations can have a skewing effect. The distributional characterization of best case execution time variation from Chapter 4 demonstrated that even nominally deterministic executions exhibit some non-deterministic behavior. This distribution is used as a minimum floor for our expected variation when executing on multi-core hardware. To get close to this minimal noise floor the timer is executed continuously on a real time thread which writes to shared memory. Another confounding factor on many multi-core machines that must be mitigated, is that of writing to multiple NUMA nodes. Even many non-explicit NUMA machines present NUMA-like behavior in the form of differing L3 latencies, these can also skew timing measurements if extreme accuracy is needed.

Figure 6.5, a duplicate of that from Chapter 4 shows the difference in latency of accesses to a shared timer region on NUMA machines with just two architectures. The risk in this case is slowing the critical path of the code being timed. To minimize this a timer *struct* is allocated on each NUMA node which is shared by threads executing on their respective nodes. This shifts the latency burden to the timer updating thread and not to the critical path. This also has the added advantage of the same latency being experienced by all timed threads and not just one making the effects easier to de-noise when required. The rest of the timer mechanism is identical to that described in Chapter 4.

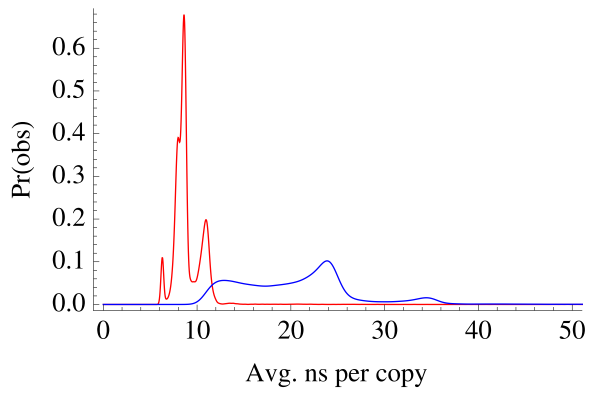


Figure 6.5: Smooth histogram of 10^6 data points each representing timed averages of 500 copy operations, first on the same NUMA node (red line) and then across different NUMA nodes (blue line). The performance of a copy on the same NUMA node seems to be much more consistent.

The period of observation (T) for the monitor thread is just as critical as an accurate time reference. Noise from the system and timing mechanism dominate for very small values of T making observations unusable [36]. The key observation from Chapter 4 with respect to noise is that some workload durations are less noisy (more repeatable) than others. The stable point changes based on the hardware, operating system, scheduler, and countless other factors. In Chapter 4 we examined but a small subset. Analytic derivation of the perfect sampling period is an exercise in futility, however a branch and bound search can find an acceptable T quickly enough. Figure 6.6 shows how much T varies through back to back samples as the length T is increased from very small (starting with the resolution of the timer) to very large. For detailed information on average timer latencies, see Chapter 4.

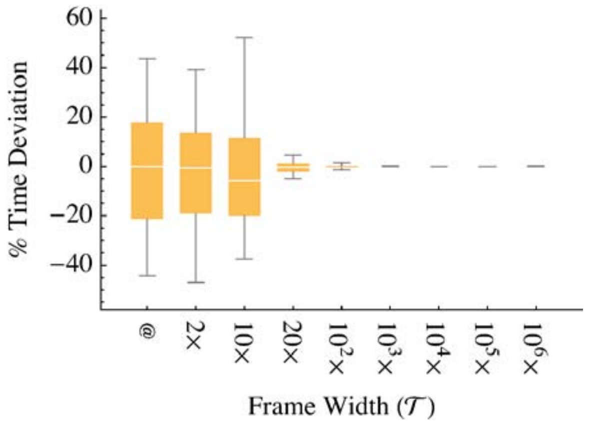


Figure 6.6: Observations of T variation using the timing mechanism of [21]. The @ symbol represents the minimum resolution of the timing mechanism ($\sim$ 300 ns for this example), subsequent box and whisker observations are the indicated multiple of @. The trend indicates that wider time frames (up to the approximate time quanta for the scheduler) give more stable values of T.

6.2.1 Throughput

The overall throughput of an application and the throughput at each link within an application is of great interest to application developers. It doesn't have quite the same power as the non-blocking service rate, but it is still quite useful. Once T is fixed, it is trivial to sample the number of writes into and reads out of a given queue. Using an iteratively updated mean, the throughput can be averaged over the execution of a program. Once a sufficiently stable throughput is calculated it is straightforward to zero the sum of reads or writes and the respective observation count and begin anew.

6.2.2 Queue Occupancy

The mean queue occupancy is also quite easy to calculate by sampling the size of the queue periodically over N periods. The mean queue occupancy is just the quotient of the sum of observed occupancies and N. Capturing more detailed data such as histograms of the queue occupancies are covered by techniques such as those developed within the TimeTrial framework of Lancaster et al. [115].

6.3 Service Rate

To minimize overall impact, the data necessary to estimate the service rate is split between the queue itself and the monitor thread, and is moved only when absolutely necessary. As depicted in Figure 6.4, the queue itself is now visible to three distinct threads: the monitor thread and the producer/consumer threads at either terminus of the queue. The only logic to consider within the queue itself is that necessary to tell the monitor thread if it has blocked and that necessary to increment a item counter as items are read from or written to the queue. The monitor thread reads these variables written by the run-time controlling the queue (work is actually performed by the producer or consumer threads). The monitor thread also resets or zeros the counter (which will be called tc from this point forward) and blocking boolean kept by the queue. In a non-locking (also non-atomic) operation, the monitor thread copies and zeros tc (the reader should consider the implications of the non-atomic copy and zero briefly). This mechanism has the advantage of being quite fast, however there are implications which must be dealt with.

The monitor thread samples at a fixed interval of time T which is the sampling period. When the monitor thread samples tc and the blocking boolean, it has no way of knowing if the server at either end only performed complete executions or partial ones. The only thing it can be certain of is that the data read are non-blocking if the boolean value is set appropriately. This means that tc can represent something less than the actual service rate. Also contained within the tc are data not-representative of average behavior; these include (list not exhaustive): caching effects, interrupts, memory contention, faults, etc.

The monitoring thread scheme samples at a fixed interval of time T which provides a relatively stable and monotonically increasing time reference with an average latency on the systems tested of approximately $50-300$ ns across all cores (includes latency for back to back x86 `rdtsc` instruction as well as serializing assembly instruction, e.g., `mfence`). Despite the luxury of a stable time reference, another trend complicates matters for the sampling of tc. As the service time decreases, the probability of observing a non-blocking queue transaction decreases as well.

Central to accurate estimation of service rate is observing non-blocking reads and writes by the server. While executing, the probability of observing a non-blocked read or write to a queue in general is very low, especially so for the edge cases of very high utilization

Table 6.1: Nomenclature used for queueing equations.

Symbol	Description
μ_s	mean service rate
ρ	server utilization
C	capacity of output queue
T	sampling period of monitor
k	items needed by server during T

and very low utilization. The equations below (a modification of the equations given by Kleinrock [109]) give the probability distribution for the simplified case where each server's process is Poisson, each data element is a single job, each job is the same size, and only a single in-bound and out-bound queue are considered (also known as an $M/M/1$ [103] queue; Table 6.1 lists variable definitions).

$$k = \lceil \mu_s T \rceil$$

$$Pr_{\mathrm{READ}}(T, \rho, \mu_s) = \rho^k$$

$$Pr_{\mathrm{WRITE}}(T, C, \rho, \mu_s) = \begin{cases} 1 - \rho^{C-k+1} & C \geq \mu_s T \\ 0 & C < \mu_s T \end{cases}$$

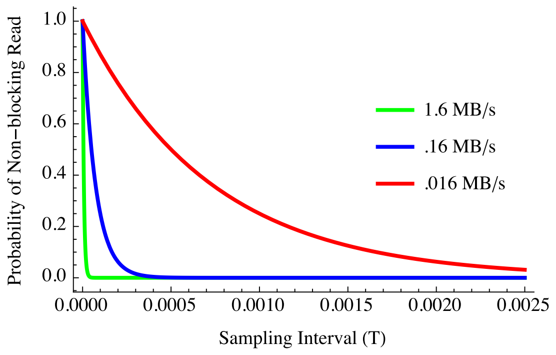


Figure 6.7: The probability (y-axis) of observing a non-blocking read given the observation period T (x-axis) in seconds. In general the faster the server or greater throughput the lower the probability of observing a non-blocking read from the queue.

In order to improve those odds there are some mechanisms that the run-time can implement. Given a full out-bound queue, resizing the queue provides a brief window over which to observe fully non-blocking behavior. Given an empty in-bound queue there are three implementable actions: (1) increasing the number in-bound servers feeding more arrivals to the queue, (2) changing execution hardware of the up-stream server can have the same effect as the aforementioned approach, (3) adjusting the scheduling frequency before observing full service rate in order to fill the in-bound queue. The benefits and implementations of these will be discussed after describing the theory behind the service rate approximation.

6.3.1 Online Service Rate Heuristic

Online estimation of service rate requires four basic steps: fixing a stable sampling period T, sampling only the correct states (expounded upon below), reducing and de-noising the data, then estimating the non-blocking service rate. The system has a finite number of states which are useful in estimating the non-blocking service rate. The most obvious states to ignore are those where the in-bound or out-bound queue is blocked (see Lancaster et al. [42, 118]). The others, as mentioned in Section 6.3, are data unrepresentative of the non-blocking service rate. Symbols used in this section are summarized in Table 6.2.

Table 6.2: Nomenclature used for Section 6.6.

Symbol	Description
T	sampling period
tc	sum of non-blocking reads during T
S	windowed set of items tc
S'	Gaussian filtered set of S
q	95^{th} quantile of S'
$\bar{q}$	population averaged q
d	bytes per data item

6.3.2 Sampling Period Determination

Each queue within a streaming application has its own monitor thread. As such, each T is queue specific, since each instrumented queue is in a slightly differing environment. An initial requirement is a stable time reference across all utilized cores. The timing method described in the previous section is employed. (our specific implementation uses the `x86` `rdtsc` instruction combined with the manufacturer recommended fence instruction (one of `mfence` or `lfence`), but any sufficiently high resolution time reference could be used). This provides a stable and monotonically increasing time reference whose latency on our test systems is approximately $50 - 300$ ns across the cores. Despite a relatively stable time reference, two trends complicate matters. First, as service time decreases, the probability of observing a non-blocking queue transaction decreases as well. Second, noise from the system and timing mechanism dominate for very small values of T making observations unusable.

Modern computing systems introduce some level of noise into the measurements [21]. Choosing a longer sampling period (T) reduces the impact of the noise. We wish, however, to observe kernel executions that are unimpeded by their environment (no blocking due to upstream or downstream effects). This goal stands in juxtaposition to noise reduction since shorter sampling periods increase the probability of observing non-blocked periods of execution.

Figure 6.6 shows how the empirically observed sampling period varies with desired sampling period, T, starting with the minimum latency (~ 300 ns) of back to back timing requests then iterating over multiples of that latency. The monitor thread tries to find the largest time period T (moving to the right in Figure 6.6) while minimizing observed queue blockage during the period. As is expected, the noise is less significant compared to the period as T increases. The implementation within RaftLib lengthens the period if: 1) no blockage occurred on the in-bound or out-bound buffer (with respect to a kernel) within the last k periods and (2) the realized period of the monitor was within ϵ of the current T over the last j periods (i.e., T was stable). Failure to meet these conditions results in the failure of our method i.e., we conclude that our approach will not result in sable service rate monitoring.

6.3.3 Service Rate Heuristic

Once a stable T has been determined, the next step is estimating the online service rate without having to store the entire data trace of all queueing transactions. The head and tail of each queue store counts of non-blocking transactions, tc, as well as the size of each item copied, d. The transaction count, tc, requires very little overhead since it is simply a counter. The item size is typically constant for any given queue. The instrumentation thread samples tc from the head and tail every T seconds. There are many factors that can slow down a kernel in the context of a full execution, only a few can make it appear to execute faster (see Figure 6.8), these factors will have to be dealt with in order to accurately

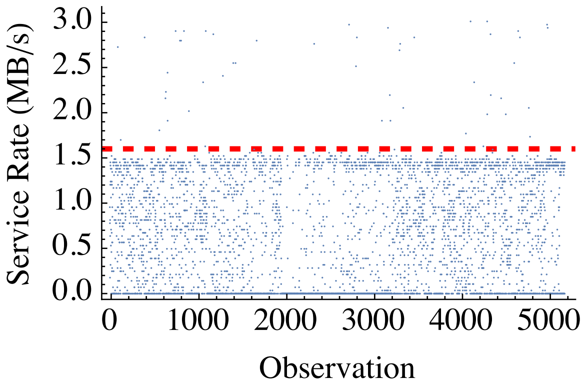


Figure 6.8: Direct observations of the service rate, using the logic of [42], for a nominally fixed rate micro-benchmark kernel. The x-axis is the increasing observation index with time, the y-axis represents the actual data rate observed at each sample point. The red dashed line is the nominal service rate.

estimate the service rate. To do this we will use an estimate of the maximum, well-behaved tc to estimate the service rate of interest. (well-behaved is articulated below). For simplicity, the discussion that follows will consider only actions that occur at the head of the queue with the understanding that equivalent actions can occur at the tail as well.

Algorithm 1 summarizes the process described below. This description presumes an implementation of a streaming mean and standard deviation (see Chan et al. [44] and Welford [195]) through the $updateStats()$, $updateMeanQ()$ and $resetStats()$ methods. The mechanics of the $QConverged()$ function are described later. Within the description the reader should note that unlike graphics applications, padding is not used for the filter so that the filter radii is width $2 \times$ radius smaller than the data window.

While sampling tc, the timing thread creates a list S, ordered by entry time (implemented as a first-in first-out queue). S is maintained as a sliding window of size w. If S is of sufficient size, it is expected that the set S tends toward a Gaussian distribution ($\mathcal{N}(\mu_S, \sigma_S)$), as it is a list of sums of non-blocking transactions. S, however contains many data elements that are not necessarily indicative of non-blocking service rates. These elements arise from the following conditions: (1) the monitor thread observed only a partial firing of the server (i.e., the server had the capability to remove j items from the queue but only $< j$ items were evident when retrieving tc); (2) the monitor thread clears the queue's current value of tc during a firing (i.e., the counter maintaining tc is non-locking because locking it introduces

```
$stream \leftarrow tc$;
$output \leftarrow$ output stream;
$S \leftarrow \{\}$;
while $True$ do
    $tc_{current} \leftarrow pop(stream)$;
    $S' \leftarrow \{\}$;
    for $i \leftarrow gauss_{radius}$, $i < |window| - gauss_{radius}$,$i++$ do
        $val \leftarrow$ Dot( $S[i - gauss_{radius};;i + gauss_{radius}]$,$GaussianFilter$ );
        push( $S'$, $val$ );
    end
    $\mu_{S'} \leftarrow$Mean($S'$);
    $\sigma_{S'} \leftarrow$StandardDeviation( $S'$);
    $q \leftarrow$NQuantileFunction( $\mu_{S'}$, $\sigma_{S'}$, .95 );
    updateStats( $q$ ); if  $QConverged()$ then
        push( $output$, getMeanQ() );
        resetStats();
    end
end
```

Algorithm 1: Service rate heuristic.

delay); (3) outlier conditions as discussed in the introduction to this Section which are not indicative of normal behavior.

Filters are frequently used in signal processing applications to de-noise data sets. In general, a filter is a convolution between two distributions so that the response is a combination of both functions. The underlying distribution of S without outliers tends towards a Gaussian, therefore a Gaussian discrete filter is used to shape the data in S so that it is sufficiently well-behaved (de-noised) for estimating the maximum. The filtered data make up the set S'. Equation 6.2 describes the kernel, where $x \leftarrow [-2, 2]$ is the index with respect to the center. Through experimentation, a radius of two was selected as providing the best balance of fast computation and smoothing effect.

$$GaussianFilterKernel(x) \leftarrow \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}} \tag{6.2}$$

Once filtered, S' is used to estimate the maximum. Since we must still account for outliers, rather than explicitly use the maximum, we estimate the maximum via the 95^{th} quantile of S'. This is a reasonable approximation given that: once filtered, S' even more closely has a Gaussian distribution than S and a quantile is more robust to outliers than the sample maximum. Operationally, we use the sample mean, $\widehat{\mu_{S'}}$, and standard deviation, $\widehat{\sigma_{S'}}$, to estimate $\mathcal{N}(\mu_{S'}, \sigma_{S'})$, and the quantile is of course

$$q = \widehat{\mu_{S'}} + 1.64485\ \widehat{\sigma_{S'}}. \tag{6.3}$$

Direct utilization of q is sufficient for some purposes, however it is only valid for the time period comprising the window over which it was collected $p \leftarrow T \times w$. Subsequent sets S_i' update $\mu_{S'}$ and $\sigma_{S'}$ resulting in frequent new values (e.g., Figure 6.9). Stability is gained by using the streaming mean of successive values of q_i (as shown in Figure 6.9). Where $\bar{q}$ is the averaged, estimated maximum non-blocking transaction count tc, assuming only one queue for simplicity, the service rate is simply $\frac{\bar{q} \times d}{T}$. This, however also assumes that the underlying distribution generating tc is at least stable over the observation period. As with all online estimates, $\bar{q}$ becomes more stable with more observations (e.g., Figure 6.10).

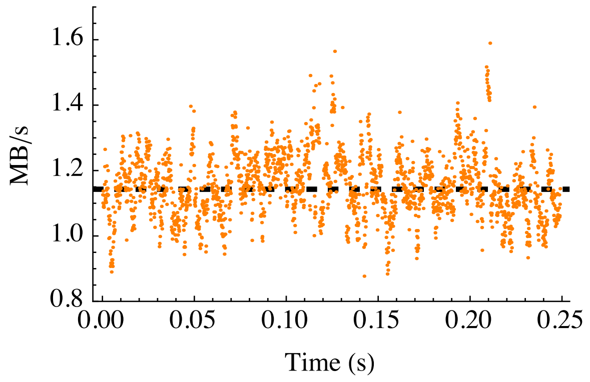


Figure 6.9: Plot of the values of q with increasing time. Each value of q is the result of a computation of Equation 6.3. The dashed line across the y-axis represents the set or expected service rate.

Convergence of $\bar{q}$ to a "stable" value is expected after a sufficiently large number of observations. In practice, with μs-level sampling, convergence is rarely an issue. Determining when $\bar{q}$ is stable is accomplished by observing σ of $\bar{q}$. Minimizing the standard deviation is equivalent to minimizing the error of $\bar{q}$. With a finite number of samples, it is unlikely that $\sigma(\bar{q})$ will ever equal to zero, however observing the rate of change of the error term to a given tolerance is roughly equivalent to minimizing the error of the approximation given the conditions for weak convergence by Lyapunov [33]. To accomplish this in a streaming

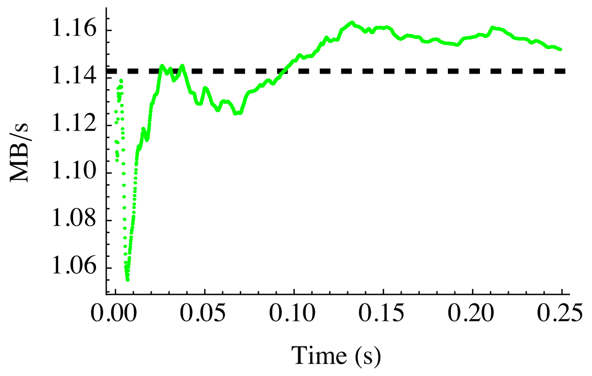


Figure 6.10: An example of convergence of $\bar{q}$ with increasing time. Data is from a single queue tandem server micro-benchmark, observing the departure rate from the queue to the server with the set service rate marked as a dashed line.

manner, a similar approach to that taken previously is used with differing filters to approximate the relative rate of change over the window. A discrete Gaussian filter with a radius of one is followed by a Laplacian filter with discretized values (in practice, one combined filter is used). This type of filter is widely used in image edge detection. Here, we are utilizing to minimize the standard deviation; essentially the filter gives a quantitative metric for the rate of change of surrounding values. The kernel is given in Equation 6.4 with $x \leftarrow [-1, 1]$ and $\sigma \leftarrow \frac{1}{2}$. The minimum and maximum of the filtered $\sigma(\bar{q})$ are kept over a window $w \leftarrow 16$ where convergence is judged by these values all being within some tolerance (ours is set to 5×10^{-7}).

$$LaplacianGaussian(x) \leftarrow \frac{x^2 e^{-\frac{x^2}{2\sigma^2}}}{\sqrt{2\pi}\sigma^5} - \frac{e^{-\frac{x^2}{2\sigma^2}}}{\sqrt{2\pi}\sigma^3} \tag{6.4}$$

An example of a stable and converged $\bar{q}$ is shown in Figure 6.11, where the data plot is of the dual filtered $\sigma(\bar{q})$ and the vertical line is the point of convergence. The time scale on the x-axis is the same as that of Figure 6.10 so that the stability point on Figure 6.11 matches that of Figure 6.10.

Once convergence is achieved, it is a simple matter to restart the process described above, and begin the search again. Figure 6.12 shows a sample run where the average service rates are known (solid blue y-axis grid lines). The x-axis grid lines (dashed vertical lines)

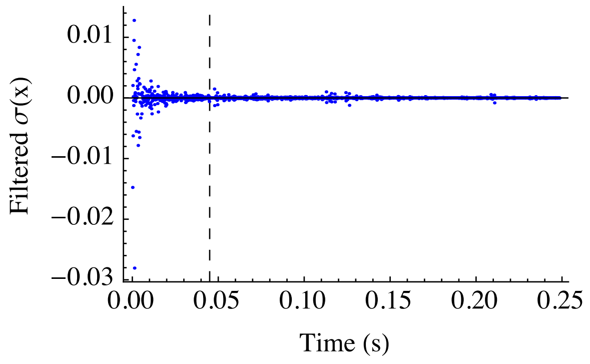


Figure 6.11: Plot of the filtered standard deviation of $\bar{q}$, the point of convergence is indicated by the vertical dashed line.

show points of convergence to stable solutions after subsequent restarts of the approximation algorithm.

Changes in $\bar{q}$ are assumed to mean a change in the process distribution governing tc.

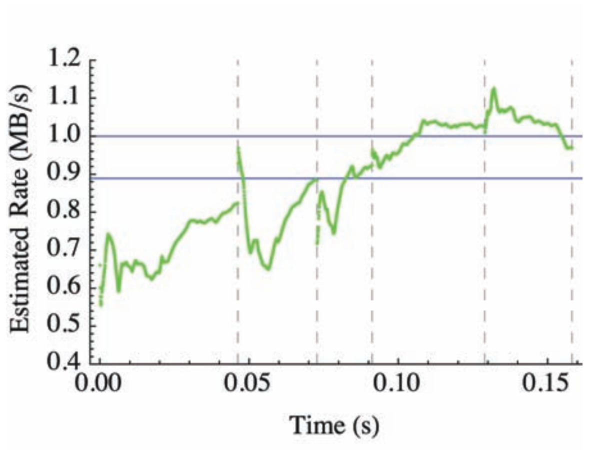


Figure 6.12: Example of $\bar{q}$ capturing two distinct service rates during execution of a micro-benchmark. The instrumentation captures the departure rate from a single queue to a compute kernel.

6.4 Evaluation

6.4.1 Infrastructure

The hardware used for all empirical evaluation is listed in Table 6.3. All code is compiled with the "-O2" compiler flag using the GNU GCC compiler (version 4.8.3).

In order to assess our method over a wide range of conditions, a simple micro-benchmark consisting of two threads connected by a lock-free queue is used. This is the same topology illustrated in Figure 6.3. Each thread consists of a while loop that consumes a fixed amount of time in order to simulate work with a known service rate. The amount of work, or service-rate, is generated using a random number generator sourced from the GNU Scientific Library [75]. The service rates of kernels within the micro-benchmark are limited to approximately $\sim$ 8 MB/s due to the overhead of the random number generator and the size of the output item (8 bytes). Service time distributions are set as either exponential or deterministic. Parameterization of the distributions is selected using a pseudorandom number source. The exact parameterization range and distribution are noted where applicable within the results section.

Table 6.3: Summary of hardware used for empirical evaluation

Platform	Processor	OS	Main Memory
1	2 $\times$ AMD Opteron 6136	Linux 2.6.32	64 GB
2	2 $\times$ Intel E5-2650	Linux 2.6.32	64 GB
3	2 $\times$ Intel Xeon X5472	Darwin 13.4.0	32 GB
4	2 $\times$ Six-Core AMD Opteron 2435	Linux 3.10.37	32 GB
5	Intel Xeon CPU E3-1225	Linux 3.13.9	8 GB

The streaming framework used is the RaftLib C++ template library (see Chapter 3).

6.5 Applications

In addition to the micro-benchmarks described above, two full streaming applications are also explored. The first, matrix multiply, is a synchronous data flow application that is expected to have relatively stable service rates. The second is a string search application that has variable rates. Ground truth service rates for each kernel are determined by executing each kernel offline and measuring the rates individually using a large resident memory data source (constructed for each kernel) and ignoring the write pointers so that it simulates an infinite output buffer.

6.5.1 Matrix Multiply

Matrix multiplication is central to many computing tasks. Implemented here is a simple dense matrix multiply ($C = AB$) where the multiplication of matrices A and B are broken into multiple dot-product operations. The dot-product operation is executed as a compute kernel with the matrix rows and columns streamed to it. This kernel can be duplicated n times (see Figure 6.13). The result is then streamed to a reducer kernel (at right) which re-forms the output matrix C. This application differs from the micro-benchmarks in that it uses real data read from disk and performs multiple operations on it. As with the micro-benchmarks, it has the advantage of having a continuous output stream from both the matrix read and dot-product operations.

The data set used for the matrix multiply is a $10,000 \times 10,000$ matrix of single precision floating point numbers produced by a uniform random number generator.

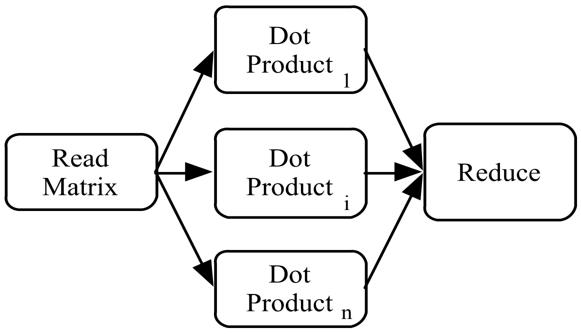


Figure 6.13: Matrix multiply application. The first kernel reads both matrices to be multiplied and streams the data to an arbitrary (n) number of dot product kernels. The final kernel reduces the input from the dot to a multiplied matrix.

6.5.2 Rabin-Karp String Search

The Rabin-Karp [101] algorithm is classically used to search a text for a set of patterns. It utilizes a "rolling hash" function to efficiently recompute the hash of the text being searched as it is streamed in. The implementation divides the text to be searched with an overlap of $m - 1$ (for a pattern length of m), so that a match at the end of one pattern will not result in a duplicate match on the next segment. The output of the rolling hash function is the byte position within the text of the match. The output of the rolling hash kernel is variable (dependent on the number of matches), for model selection testing purposes the input data will be specially constructed in order to produce a regular steady state output. The next kernel verifies the match from the rolling hash to ensure hash collisions don't cause spurious matches. The verification matching kernel can be duplicated up to j times. The final kernel simply reduces the output from the verification kernel(s), returning the byte position of each match (see Figure 6.14). The corpus consists of 2 GB of the string, "foobar."

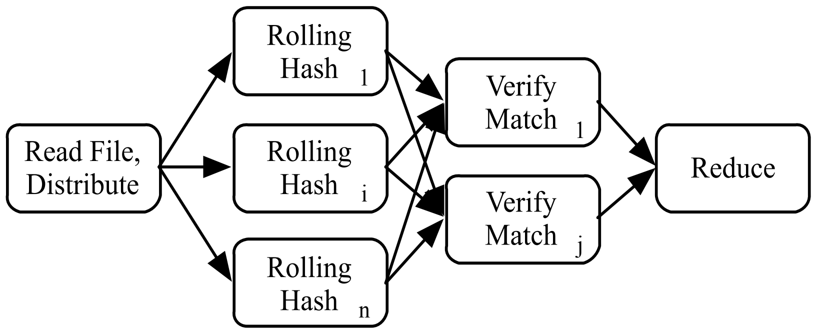


Figure 6.14: Rabin-Karp matching algorithm. The first compute kernel (at left) reads the file to be searched, hashes the patterns to search and distributes the data to n "rolling-hash" kernel(s). Next are $j, j \leq n$ verification kernel(s) to guard against matches due to hash collision. The final kernel (at right) is a reducer which consolidates all the results.

6.6 Validation

The methods that we have described are designed to enable online service rate determination. Just how well do these methods work in real systems while they are executing? In order to evaluate this quantitatively, several sets of micro-benchmarks and real applications are instrumented to determine the mean service rate of a given server. We start with two sets of micro-benchmarks, the first having a stationary distribution (with a fixed mean) and the second having a bi-modal distribution that shifts its mean halfway through its execution.

Each micro-benchmark is constructed with the configuration depicted in Figure 6.3 and executed with a fixed arrival process distribution. The service rate of Kernel B is varied for each execution of the micro-benchmark from (0.8 MB/s $\rightarrow \sim$8 MB/s). The results comprise 1800 executions in total. The departure rate from the queue is instrumented to observe the service rate of Kernel B. The goal is to find the service rate of this kernel without *a priori* knowledge of the actual rate (which we are setting for this controlled experiment). Figure 6.15 is a histogram of the percent difference between the service rate estimated via our method and the "set" filtered rate.

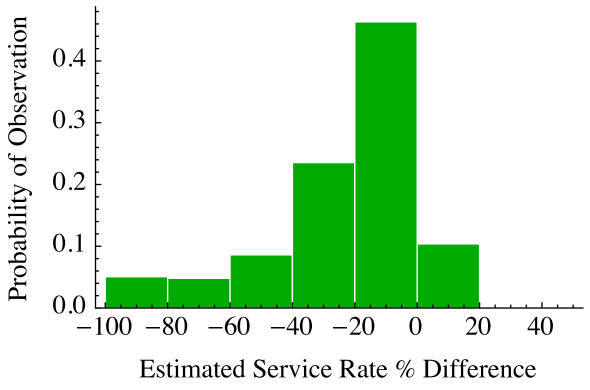


Figure 6.15: Histogram of the probability of estimating the service rate of Kernel B from Figure 6.3. Each execution is a data point, with the percent difference calculated as $(\frac{(\text{observed rate}-\text{set rate})}{\text{set rate}}) \times 100$. Not plotted are four outliers to right of the plotted data which are greater than 1000% difference, which is not unexpected given the probabilistic nature of our heuristic.

We see in this histogram that generally the correspondence between estimated service rate and ideal service rate is reasonably good. We expect divergence since these rates are determined while the application is executing, not the full execution time average rate. When it errs, the estimate is typically low, which is consistent with previous empirical data, in which actual realized execution times are typically longer than nominal 5. The majority of the results are within 20% of nominal in any case. The computational environment of any given kernel can change from moment to moment. We simulate environment change by moving the mean of the distribution halfway through execution of Kernel B (with reference to the

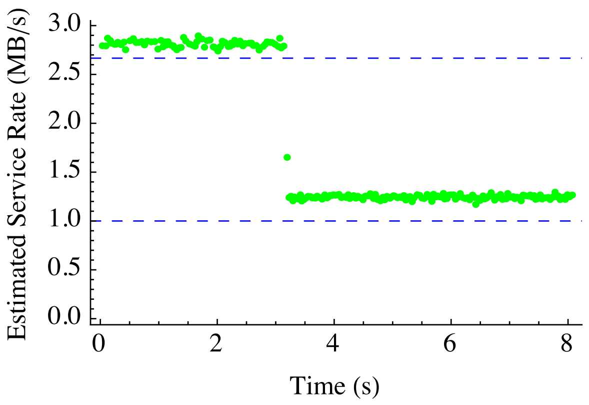


Figure 6.16: Depiction of the ideal (drawn from empirical data) of the instrumentation's ability to estimate the service rate while the application is executing. Each dot represents the converged service rate estimate (y-axis). The top and bottom dashed lines represent the first and second phases as verified by manual measurement in isolation.

number of data elements sent). We are interested in whether our instrumentation can detect this change, potentially enabling many online optimizations. An example with a wide switch in service rate is shown in Figure 6.16 where the first phase is at $\sim$2.66 MB/s and the second is much lower at $\sim$1 MB/s. Not all examples are so clear cut.

In order to classify the dual phase results into categories, a percent difference (20%) from the manually determined rates for each phase is used. Approximately 14.7% of the data had nominal service rate shifts that were known to be less than the 20% criteria specified. Figure 6.17 shows the effectiveness of our technique in categorizing the distinct execution phases of the micro-benchmarks. The rightmost graph shows the categorizations for low ρ, and the leftmost graph shows the categorizations for high ρ. Here, we make two observations. First, the system correctly detects both phases more effectively in high utilization conditions, which are the conditions under which correct classification is likely to be more important. Second, the classification errors that are made are all conservative. That is, it is correctly detecting the final condition of the kernel, indicative of a conservative settling period for rate estimation.

It is well understood that a server with sufficient data on it's input queue should be able to proceed with processing (assuming no other complicating factors). Therefore one trend that we expect to see is an improvement in the approximation for higher server utilizations. In

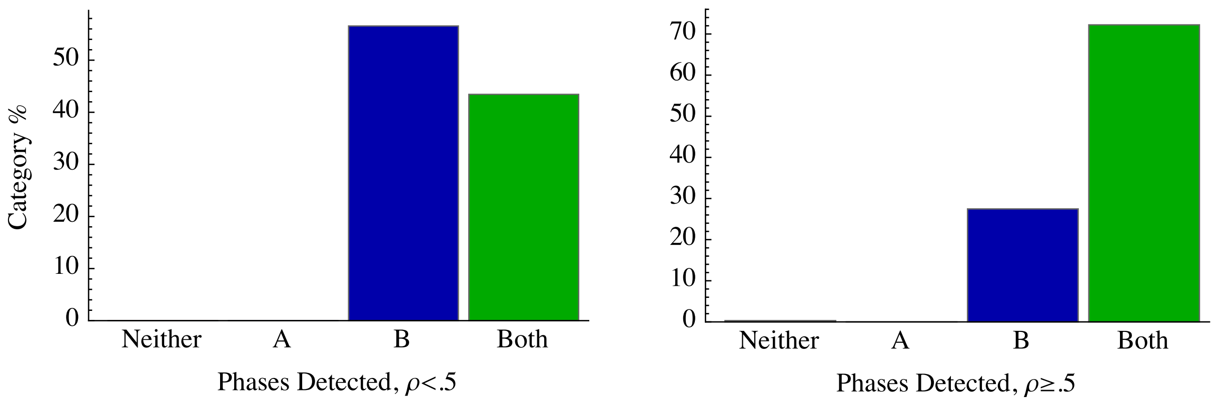


Figure 6.17: Data from a dual-phase micro-benchmark that generates two distinct service rate phases separated by server utilization, ρ, and then by correct classification at each phase (as "Neither," "A," "B," or "Both"), which represent the heuristic finding none, only the first phase, only the second phase or both, respectively.

addition, servers that are more highly utilized typically have a much more profound impact on the performance of the application as a whole (e.g., they are dramatically more likely to be throughput bottlenecks in the overall data flow).

Overall, the heuristic did quite well. Looking at the single phase data, only four of the micro-benchmark results were extremely off. The dual phase data were also fairly good, the heuristic failed to find either phase in only 0.24% of the instances. The real test of any instrumentation is how well it can handle situations beyond those that are carefully controlled. The only variable that is within the users' control is that of data set selection. Notably these applications are not limited to the slower service rates of the micro-benchmark applications but are dependent on the mechanics of the application. The matrix multiply application is executed on platform 2 from Table 6.3 with the number of parallel dot-products set to five. Only the reduce kernel is instrumented (see Figure 6.13) as the dot-products would be rather easy given the high data rates inherent in transmitting an entire row by copy. The ground-truth service rate realized by each queue (the total service rate being a combination of rates from each input-queue) are determined by the method described in Section 6.5.1. Overall the results are not quite as clean as those of the microbenchmark, but that is expected given the chosen kernel has an extremely low ρ. A majority (63%) are within the range of measurements observed during manual estimation removing each kernel from the system and manually measuring data rates at each input port).

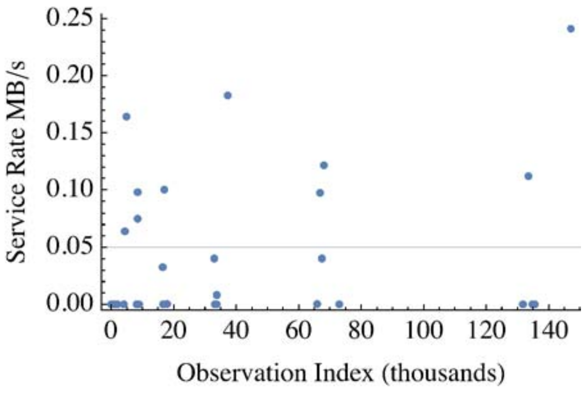


Figure 6.18: Plot of the trace for the instrumented partial service rate of the reduce kernel (the full rate being the sum of all rates for each in-bound queue, assuming PASTA [88]). The manually determined rate for this experimental setup ranged from 0.05 MB/s to 0.43 MB/s. Overall, a majority of the results ∼63% are within this range.

Similar to the results for the matrix-multiply application, the results for the Rabin-Karp application are also relatively good (recall that these are rates taken at points over the course of execution). The application is executed on platform 2 from Table 6.3 with the number of matching kernels fixed at four and the number of verification kernels fixed to two. Figure 6.19 shows the online service rate by convergence point each data point represents a converged estimate of the service rate (potentially multiple convergences for a single application execution). Instrumented is a single queue arriving to the verify block from the hash kernel. Again, we've intentionally picked a case where the ρ is very low, which is very difficult for the instrumentation to find a non-blocking read from the queue. In total, only $\sim 35\%$ of estimates are within the range observed when manually measuring service rate, although most of the data points are fairly close. This highlights the limitations of our approach. If the non-blocking reads are not observed then the rate simply cannot be determined with too much accuracy.

Low overhead instrumentation should be exactly that, low overhead. This means that there should be little, if any, impact on the execution of the application itself. Low impact also means that the system executing both the application and instrumentation should see as little increase in overhead as possible. Given that our system utilizes a separate monitor thread,

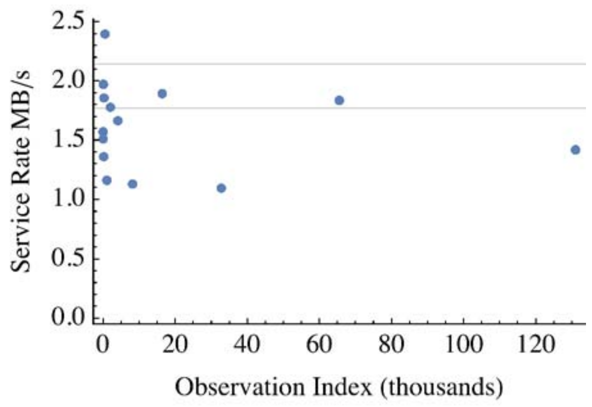


Figure 6.19: Plot of the converged estimates of service rate for a single queue within the Rabin-Karp string matching application. The utilization of this server is less than 0.1 meaning that the queue is almost always empty which leads to less opportunity for recording non-blocking reads from the queue.

this could be a concern. Using the single queue micro-benchmark, the impact was measured with instrumentation and without instrumentation. Using the GNU `time` command over dozens of executions, the average impact is only 1 - 2%. Impact to the system overall was equally minimal, load average increased only a small amount (by 0.1 on average).

6.7 Conclusions

We demonstrate an algorithm for approximating the online service rate of kernels in a streaming system. In streaming systems that exhibit filtering, the heuristic presented here can also be used to detect non-blocking departure rates which can inform a runtime of routing decisions made by the kernel as well as the amount of filtering currently exhibited. Overall our methodology works quite well. When the heuristic fails, it usually fails knowingly (e.g., no convergence is reached or non-blocking reads were not observed, methods for determining failure are examined in Chapter 7). The first task, before falling back to a state where service rate is not used is to try the strategies enumerated to improve the probability of successful approximation.

It has been validated using micro-benchmarks and two full streaming applications. While evidence has been shown for the estimation of the service rate's central moment and its variance, efficient methods also exist for streaming computation of higher moments [153]. Using the method of moments along with some simple classification, it should be clear that online distribution selection can be performed using the techniques described within this work as a basis, then extending them to include higher moment estimation. RaftLib currently supports the methods described here. Future work, and extensions to RaftLib's instrumentation system, will include higher moment estimation.

Parallelization decisions can easily benefit from the information that this method provides. Instead of relying on static (compile time) information, decisions can now be made with up-to-date data improving optimality of the execution. Related, but not shown here, is the ability of this process to instrument streams entering or exiting a TCP stack. It is assumed that there should be no difference in monitoring user-space queues feeding data into a TCP link. An open question is exactly how best to synchronize the ingress and egress transaction data.

In conclusion, we've demonstrated a probabilistic heuristic that under most conditions can estimate the service rate of compute kernels executing within a streaming system while that application is executing. It has been demonstrated to be effective using micro-benchmarks and two full applications. The next chapter examines some methods to perform online stochastic model selection for stream processing systems. Using these methods, we also show that when our instrumentation unknowingly fails, that a properly trained statistical model can tell us.

Chapter 7

Model Selection

When do you trust a model is a fundamental question. With modern computing systems, there are countless abstraction layers between what is observed and what we think we are modeling. Choices must be made in the modeling process, one of the most important is finding the right abstraction level to model a phenomena. Ignoring intermediate layers of abstraction, however means that assumptions are being made which may or may not be met. More disconcerting are assumptions that were never considered, but should have been. Chapter 4 describes a network flow model which works quite well for the applications tested. Chapter 4 also described the application of some very simple queueing models, through careful offline testing, all the modeling assumptions were verified. What actually happened was less than stellar performance because the environment changed at run-time. What we want is a way to tell when these types of events are probable. Even more useful would be a way to divine this while an application is executing (or online). This chapter discusses the application of two machine learning techniques: Artificial Neural Networks (ANN) and Support Vector Machines (SVM), for the purpose of assessing the usability of performance models.

Machine learning methods are great at predicting patterns, indeed another name for the field is pattern recognition. What patterns can our ML processes hope to find within the data? To find this pattern we launched an almost exhaustive search. We focused on finding a noise pattern that was repeatable, predictable, and useful for prediction. Furthermore, we decided to focus on an easier target than predicting the worst case deviations, and focus on the best possible case for noise within a compute system. After tilting at dozens of windmills, our quest was successful. Chapter 4 describes this noise distribution, its application and how it can be used to predict *a priori* if a particular stochastic queueing model can be applied.

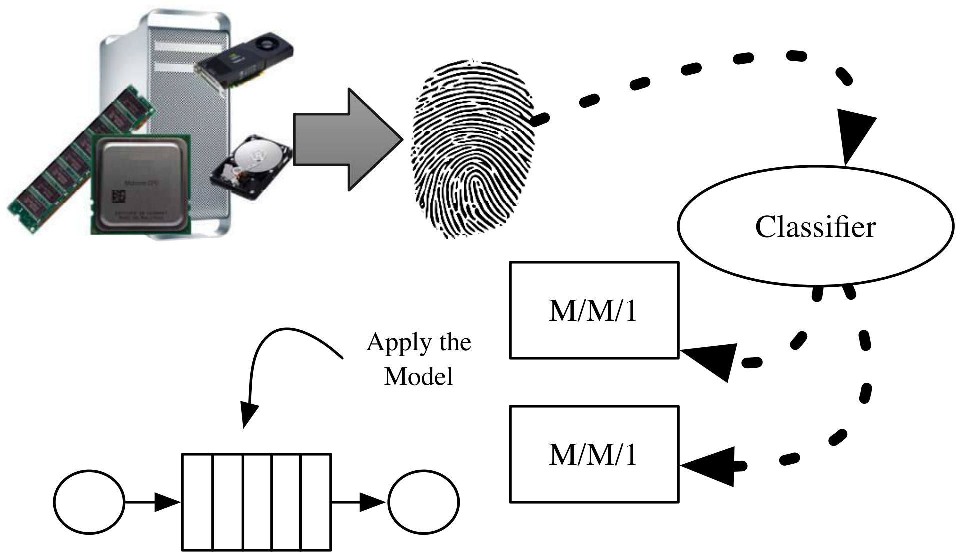


Figure 7.1: In order to automatically tune systems, models must be selected to estimate performance. In this chapter, we show that characteristics extracted from computer systems, and the applications running on them can be combined to create a "fingerprint" which is classified as belonging to one model class or another.

The modified Levy distribution described late in Chapter 4, is shown to be a good approximation of the best case execution time variation. This is the best case, the least noise we can expect from a given system. We also demonstrated how it could be used to accept or reject a queueing model using Kulback-Leibler divergence. This is, however, expensive for non-normal distributions. Chapter 7 looks at ways to improve our understanding of where we can and cannot use a model by "finger-printing" compute systems with features that are extracted from the target system. These fingerprints are used to train a machine learning process to decide which model to use. We posit that with online tuning, there are simply too many parameters to consider when attempting to model a modern computer system fast enough to return actionable results, so we turn to pattern recognition and automated classification. Figure 7.1 shows the process at a high level, the following sections will delve deeper.

7.1 Stochastic Models and Streaming Applications

Successful application of a stochastic queueing model often requires knowledge of many factors that are unknowable without extensive application and hardware characterization. Extensive characterization, is quite expensive (both in time and effort) when considering streaming applications of any appreciable size (or even standard applications [107]). Complicating matters further is that each streaming application could require that multiple models be selected in order to fully model its performance; each with its own assumptions and parameters that must be quantified before use. Even when modeling assumptions are verified offline, often they are broken by unpredictable behavior (as was the case in Chapter 4) that can occur during execution. This chapter introduces a machine learning method for classifying the reliability of stochastic queueing models for stream processing systems.

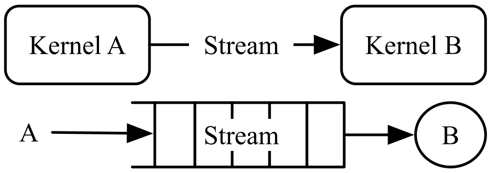


Figure 7.2: The top image is an example of a simple streaming system with two compute kernels (labeled A & B). The bottom image is the resulting queue with arrival process A (emanating from compute kernel A) and server B. For more complex systems this becomes a queueing network.

Streams allocated within a streaming application can be modeled as a stochastic queueing network for which there are well understood relationships between input arrival rates, computational service rates, queue occupancies, etc., in the steady state. Some streaming systems can even re-allocate buffers during execution to tune the size based on the environment. Each resizing operation takes precious time, so model based approaches are preferred. Hand selection of performance models for these applications is clearly impractical (hundreds of queues, each with a unique environment). Offline modeling attempts are often thwarted by dynamic characteristics present within the system that were not included in the model. This chapter outlines what is perhaps an easier route. Utilizing features easily extracted from a system along with a streaming approximation of non-blocking service rate to form a unique system fingerprint, we show that two differing machine learning mechanisms: Support Vector Machine (SVM) and Artificial Neural Networks (ANN) can identify where a model can and cannot be used. First the main case of identifying where a stochastic model can and cannot be used is examined. Second, we examine the trustworthiness of the online service rate instrumentation described in Chapter 6. Results are shown that demonstrate that both machine learning methods are generalizable to multiple operating systems and hardware types.

7.2 Stochastic Queueing Model Selection

7.2.1 Methodology

For most stream processing systems (including RaftLib) the queues between compute kernels are directly implied as a result of application construction. The stochastic mean queue occupancy models under consideration are intended for systems at steady state. In order to have some form of steady state, consider only a brief moment of time. It is assumed that most systems will have at least a small moment of steady state followed by a shift. Integrated over time this might appear to be a non-steady state, but zooming into a small window of time reveals more stable behavior. Applications whose behavior is too erratic to have any steady state behavior over any period of time are not good candidates for these models or this method.

Figure 7.3 illustrates the set of features which are used to train our machine learning models. One of the most important features used within our models is the service rate of the up and downstream kernels. Methods to divine service rates are discussed at length within Chapter 6. Architectural features of a specific platform such as cache size are used as features, all can be obtained with very low overhead (and often statically). All of our features can be found using either literature search or directly by querying the hardware. Platforms where this information is unknown are avoided, however a surfeit of such platforms exists (see Table 7.1). Processor specific performance counters (from open source tools such as PAPI [139]) could have been used, and this is left to future work. It is quite possible that additional features could improve specificity or generalizability. Working under the assumption that accurate determination of service rate distribution (through moment matching or other methods) is typically expensive and should be avoided for real-time application control, it is not used to train our models (we do show that giving the SVM the approximate distribution as well improves accuracy greatly).

Implicit within most stochastic queueing models (save for the circumstance of a deterministic queue) is that $\rho < 1$ to obtain a finite queue. It is expected that either ML process should be able to find this relationship based upon the training process. It is shown in Section 7.2.4 that this is indeed the case. If deciding on a queueing model were as simple as selecting one class for $\rho \geq 1$ and another for $\rho < 1$ then the methods described in this chapter would be relatively inconsequential. However we also assume that the ML process is not explicitly told what the actual service process distributions are of the compute kernels modulating data arrival and service so this relationship is not quite so binary. It is also shown in the results that training the SVM with broader distributions slightly decreases the overall classification accuracy while increasing the generalizability of the trained SVM. It is assumed that a similar relationship holds for the ANN approach.

Once trained, the parameters are supplied to a machine learning process (either SVM or ANN) which will label each parameter combination as being “usable” or “not” for the stochastic queuing model (in our case the $M/D/1$ and $M/M/1$ models) for which the ML process is trained. The results for both the SVM and ANN are evaluated based on frequency of Type 1 and Type 2 errors. The worst case for our selection process is positively identifying a model for a particular queue for which it wasn’t suited. In this case we use a model erroneously and tune to false specifications. On the other end of the spectrum are false negatives, where we reject a model. In the worst case for this scenario, the fall back of branch and bound search in many cases can be used. False positives should be driven to a minimum. A few false negatives are acceptable. First we’ll cover the hardware and software techniques utilized to produce a fingerprint for classification, followed by discussion of the support vector machine, then the artificial neural network.

7.2.2 Support Vector Machine

A SVM is a method to separate a multi-dimensional set of data into two classes by a separating hyperplane. It works by maximizing the margin between the hyperplane and the support vectors closest to the plane. The theory behind these are covered by relevant texts

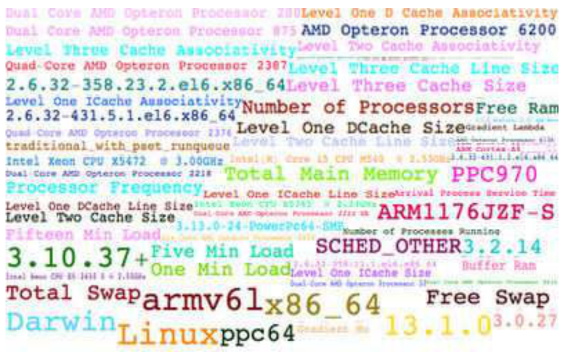

Level One D Cache Associativity
AMD Opteron Processor 6200
Level Three Cache Associativity
Level Two Cache Associativity
Quad-Core AMD Opteron Processor 2387
Level Three Cache Line Size
2.6.32-358.23.2.el6.x86_64
Level Three Cache Size
Level One ICache Associativity
2.6.32-431.5.1.el6.x86_64
Number of Processors
Free Ram
Level One DCache Size
Gradient Lambda
Quad-Core AMD Opteron Processor 2376
traditional_with_pset_runqueue
Level Two Cache Line Size
Intel Xeon CPU X5472 @ 3.00GHz
Dual Core AMD Opteron Processor 2218
Total Main Memory
PPC970
Processor Frequency
Level One ICache Line Size
Arrival Process Service Time
Level One DCache Line Size
Level Two Cache Size
ARM1176JZF-S
3.13.0-24-PowerPC64-SMP
Number of Processes Running
Fifteen Min Load
SCHED_OTHER
3.2.14
3.10.37+
Five Min Load
One Min Load
Buffer Ram
Level One ICache Size
Total Swap
armv61
x86_64
Free Swap
Darwin
Linux
ppc64
13.1.0
3.0.27

Figure 7.3: Word cloud depicting features used for machine learning process with the font size representing the significance of the feature as determined by [49].

on the subject [53, 191]. An SVM labels an observation with a learned class label based on the solution to Equation (7.1) [29, 51] (the dual form is given, $\mathbf{e}$ is a vector of ones of length l, Q is an $l \times l$ matrix defined by $Q_{i,j} \leftarrow y_i y_j K(x_i, x_j)$ K is a kernel function, specific symbolic names match those of [46]). A common parameter selected to optimize the performance of the SVM is the penalty parameter, C, discussed further in Section 7.2.4.

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \frac{1}{2}\boldsymbol{\alpha}^T \mathbf{Q} \boldsymbol{\alpha} - \mathbf{e}^T \boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \leq \boldsymbol{\alpha}_i \leq C, i = 1, \ldots, l, \end{aligned} \tag{7.1}$$

$$K(x, y) = e^{-\gamma ||x-y||^2}, y > 0. \tag{7.2}$$

A Radial Basis Function (RBF, [171], Equation (7.2)) is used to map attributes to features. The parameter γ is optimized separately in order to maximize the performance of the SVM/Kernel combination. In order to train and test the SVM a set of micro-benchmarks and full benchmark streaming applications (described below) are used. All are authored using the RaftLib library in C++ and compiled using `g++` using the `-O1` optimization flag.

A micro-benchmark (with the topology shown in Figure 7.2) has the advantage of a user specified service distribution for both compute kernels `A` and `B`. A synthetic workload for each compute kernel is composed of a simple busy-wait loop whose looping is dependent on a random number generator (either Exponential, Gaussian, Deterministic, or a mixture of multiple random distributions are produced). Simple workloads similar to those used within the real applications also constitute up to 5% of the micro-benchmark loop workloads. Data exiting the servers are limited to one 8-byte item per firing. A dense matrix multiply and Rabin-Karp string search application are used as described in Chapter 6.

7.2.3 Data Collection & Hardware

Using benchmarking the applications enumerated above, we were able to collect a variety of features from each platform using a myriad of methods ranging from system calls through architecture-specific methods. Service rate is also used, which is approximated online via methods described in Chapter 6. The number of features which coalesce to form a fingerprint prohibit their complete enumeration, however some of the more pertinent ones include: service rate, instruction set architecture, cache hierarchy sizes, operating system (OS) and version, scheduler, and main memory available (further enumerated in Figure 7.3).

To collect mean queue occupancy, a separate monitor thread is used for each queue to sample the occupancy over the course of the application. For both real and synthetic applications, the service times of micro-benchmark compute kernels are verified via monitoring the arrival and departure rate of data from each kernel with a non-blocking infinite queue (implemented by ignoring the read and write pointers) in addition to the online measurements (double verification). All timing is performed using the POSIX.1-2001 `clock_gettime()` function with a real time system clock using the setup described in Chapter 6. The CPU time stamp counter could have also been used, however the system call is more portable across the varied architecture types used during testing.

Relying on measurements from only one hardware type or operating system would undoubtedly bias any classification algorithm. To reduce the chance of bias for one particular platform, empirical data are collected from platforms with the processors and operating systems listed in Table 7.1. For all tests either the Linux or Apple OS X versions of the completely fair scheduler are used. To unbias the results further, task parallel sections of each application are replicated varying numbers of times (up to $2x$ the number of physical processor cores available). Application kernels are run "un-pinned." That is, the compute core which each executes on is assigned by the operating system and not by the user (although the core from which timing is determined is "pinned". Presumably more stable queueing model classification could be obtained by "pinning" each compute kernel to dedicated cores, however

Table 7.1: Summary of processor types and operating systems used for both the micro-benchmark and application data collection.

Platform	Processor Type	OS	Kernel Version
P_1	Intel Xeon CPU E5-2650	Linux	2.6.32
P_2	Quad-Core AMD Opteron 2376	Linux	2.6.32
P_3	Intel Xeon X5472	Darwin (OS X)	13.1.0
P_4	Dual-Core AMD Opteron 2218	Linux	2.6.32
P_5	ARM1176JZF-S	Linux	3.10.37
P_6	Dual-Core AMD Opteron 2222 SE	Linux	3.0.27
P_7	IBM Power PC 970	Linux	3.13.0
P_8	Six-Core AMD Opteron 2431	Linux	3.0.27
P_9	Intel Xeon E5345	Linux	2.6.32
P_{10}	Intel Xeon CPU E3-1225	Linux	3.13.9
P_{11}	Dual Core AMD Opteron 875	Linux	2.6.32
P_{12}	AMD Opteron 6136	Linux	2.6.32
P_{13}	ARM Cortex-A9	Linux	3.3.0
P_{14}	Intel Core i5 M540	Darwin (OS X)	13.1.0
P_{15}	AMD Opteron 6272	Linux	2.6.32
P_{16}	Six-Core AMD Opteron 2435	Linux	3.0.27
P_{17}	Dual Core AMD Opteron 280	Linux	2.6.32
P_{18}	Quad-Core AMD Opteron 2387	Linux	2.6.32
P_{19}	Dual-Core AMD Opteron 2220	Linux	2.6.32
P_{20}	Dual-Core AMD Opteron 8214	Linux	2.6.32

this is not a realistic environment for many platforms which have no provision for locking a thread to a particular core.

Micro-benchmark data are collected from all of the platforms in Table 7.1, Matrix multiply and Rabin-Karp Search data are collected from platforms 2, 8, 10, and 15. In all, approximately 45,000 observations were made for the micro-benchmark application. This data is divided using a uniform random process into two sets with a 20/80 split. The set with 20% of the data is used for training the ML method and the 80% is set aside as a testing set. To give an idea of the range with which the ML methods are trained, the micro-benchmark training set has the following specifications: approximately 8,200 observations, server utilization ranges from close to zero to greater than one and distributions vary widely (a randomized mix of Gaussian, deterministic, and the model's expected exponential distribution as well as some mixture distributions). For each of the other two applications, the SVM trained exclusively on the training micro-benchmark data (same training set as above) is used.

7.2.4 SVM and Training

Before the SVM can be trained as to which set of attributes to assign to a class, a label must be provided. The two classes, "use" and "don't use" are encoded as a binary one and zero respectively. The SVM is trained to identify one stochastic model at a time (i.e., either "use" or "don't use" for $M/M/1$ or $M/D/1$ but not both at the same time). In order to label the data-set as to which queueing model to use, a fixed difference is used. If the actual observed queue occupancy is within $n \leftarrow 5$ items, then the model is deemed acceptable otherwise false. A percentage based function for l shows a similar trend. After sampled mean queue occupancy is used for labeling purposes, it is removed from the data set presented to the SVM.

Feature selection is a very hot topic of research [86]. There are several methods that could be used including (but not limited to) Pearson correlation coefficients, Fisher information criterion score [49], Kolmogorov-Smirnov statistic [47]. Our selected feature set has a total of 35 linearly independent variables. The rest of the features exhibit weak non-linear dependence between variables. Extensive cross-validation followed by evaluating the Fisher information criterion score showed that the training data relied extensively on 67 of our candidate features. Most notably the variables that indicated the type of processor, operating system kernel version and cache size ranked highest followed closely by amount of main memory and total number of processes on the system. During the training phase we noted that despite the Fisher information criteria results, the additional 9 features provided a significant increase in correct classification, therefore all 76 are used as opposed to the reduced set selected via statistical feature selection.

For all data sets (and all attributes contained in each set) the values are linearly scaled in the range $[-1000, 1000]$ (see [185]). This causes a slight loss of information, however it does prevent extreme values from biasing the training process and reduces the precision necessary for the representation. Once all the data are scaled, there are a few SVM specific parameters that must be optimized in order to maximize classification performance (γ and C). We use a branch and bound search for the best parameters for both the RBF Kernel ($\gamma \leftarrow 4$) and for the penalty parameter ($C \leftarrow 32768$). The branch and bound search is performed by training and cross-validating the SVM using various values of γ and C for the training data set discussed above. The SVM framework utilized in this work is sourced from LIBSVM [46].

SVM Results

To evaluate how effective a SVM is for model reliability classification we'll compare the class label predicted by the SVM compared to that of ground truth as determined by the labeling process. If the queueing model is usable and the predicted class is "use" then we have a true positive (TP). Consequently the rest of the error types true negative (TN), false positive (FP) and false negative (FN) follow this pattern.

The micro-benchmark data ($\text{Micro}_{\text{test}}$) consists of queues whose servers have widely varying distributions and server utilizations. As enumerated in Figure 7.4, the SVM correctly predicts (TP or TN) 88.1% of the test instances for the $M/M/1$ model and 83.4% for the $M/D/1$ model. Overall these results are quite good compared to manual selection [19]. Not only do these results improve upon manual mean queue occupancy predictions, they are actually faster since the user doesn't have to evaluate the service time and arrival process distributions, and they can be done online while the application is executing.

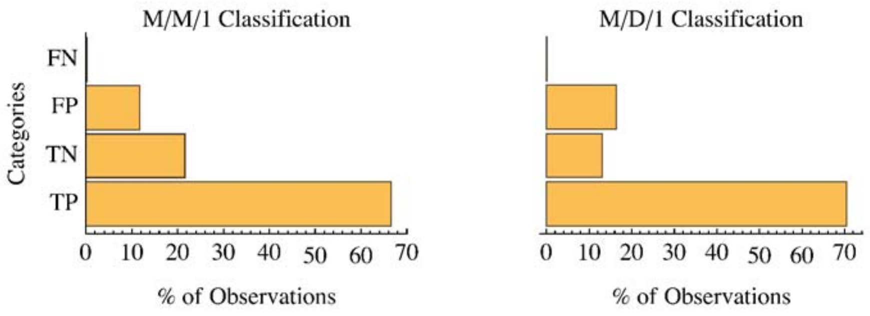


Figure 7.4: Summary of overall classification rate by error category. In general the correct classification is quite high $TP + TN > 83\%$ in all cases.

Server utilization (ρ) is a classic and simple test to divine if a mean queue length model is suitable. At high ρ it is assumed that the $M/M/1$ and $M/D/1$ models can diverge widely from reality. It is therefore assumed that our SVM should be able to discern this intuition from its training without being given the logic via human intervention. Figure 7.5 shows a box and whisker plot for the error types separated by ρ. As expected the middle ρ ranges offer the most true positive results. Also expected is the correlation between high ρ and true negatives. Slightly unexpected was the relationship between ρ and false positives.

Directly addressing the performance and confidence of the SVM is the probability of class assignment. Given the high numbers of TP and TN it would be useful to know how confident

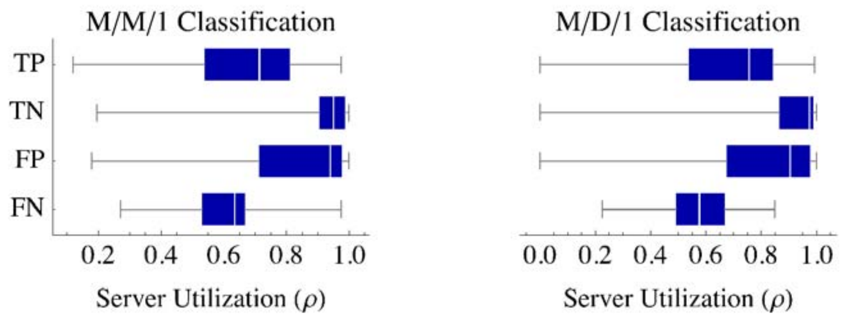


Figure 7.5: Summary of true positive (TP), true negative (TN), false positive (FP), false negative (FN) classifications for the $M/M/1$ (left) and $M/D/1$ (right) queueing models for the microbenchmark's single queue by server utilization ρ demonstrating empirically that the SVM can recognize the instability of these models at high ρ.

the SVM is in placing each of these feature sets into a category. Probability estimates are not directly provided by the SVM, however there are a variety of methods which can generate a probability of class assignment [199]. We use the median class assignment probability for each error category as it is a bit more robust to outliers than the mean. For the $M/M/1$ model we have the following median probabilities: TP = 99.5%, TN = 99.9%, FP = 62.4% and FN = 99.8%. The last number must be taken with caution given that there are only 79 observations in the FN category for $M/M/1$. For the $M/M/1$ FP it is good to see that these were low probability classifications on average, perhaps with more training and refinement these might be reduced. For the $M/D/1$ classification, probabilities mirror those of the $M/M/1$: TP=95.9%, TN=95.8%, FP=50.9%, FN=85.3%. The same qualification applies to the $M/D/1$ trained SVM for the FN probabilities as the FN category only contains 39 examples. Calculating probabilities is expensive relative to simply training the SVM and using it. It could however lead to a way to reduce the number of false positives. Placing a limit of $p = .65$ for positive classification reduces false positives by an additional 95% for the micro-benchmark data. Post processing based on probability has the benefit of moving this method from slightly conservative to very conservative if high precision is required, albeit at a slight increase in computational cost.

The full application results are consistent with those of the micro-benchmark applications. Each application is run with a varying number of compute kernels with its queue occupancies sampled. Table 7.2 breaks the application results into categories by model and application. Due to the processor configuration and high data rates with this application all examples are tested with a high server utilization. One trend that is not surprising is the lack of true positives within Table 7.2. The application as designed has very high throughput, consequently all servers are highly utilized. In these cases (ρ close to 1), it is expected that neither of these models is usable. As is the case for the micro-benchmark data, the overall correct classification rates are high for both applications and models tested.

Table 7.2: % SVM classification rate for application data.

Application	Model	TP	TN	FP	FN	Correct Classification
Matrix Multiply	$M/M/1$	17.1%	75.2%	5.4%	2.4%	92.3%
Matrix Multiply	$M/D/1$	5.4%	83.9%	4.6%	6.1%	89.3%
Rabin-Karp	$M/M/1$	0.0%	86.0%	14.0%	0.0%	86.0%
Rabin-Karp	$M/D/1$	0.0%	87.4%	12.6%	0.0%	87.4%

One potential pitfall of this method is the training process. What would happen if the model is trained with too few distributions and configurations. To test this a set of the training data from a single distribution (the exponential) is labeled in order to train another SVM explicitly for the $M/M/1$ model. We then apply this to two differing test sets. The first is data drawn from an exponential distribution and the second is data drawn from many distributions (training data is excluded from all test sets). The resulting classification rates are shown in Table 7.3. Two trends are apparent: specifically training with a single distribution increases the accuracy when attempting to classify for only the distribution for which the model was trained, and conversely lack of training diversity increases the frequency of false positives when attempting use the SVM to classify models with distributional assumptions that it have not been trained for. Unlike the false positives seen in prior sets, these are high confidence predictions that post processing for classification probability will not significantly improve. One thing is clear, training with as many service rate distributions as possible and as many configurations tends to improve the generalizability of the SVM for our application.

Table 7.3: % for SVM predictions with SVM trained only with servers having an exponential distribution and tested as indicated.

Dist.	# obs.	Model	TP	TN	FP	FN	Correct Classification
exp.	3249	$M/M/1$	53.0%	31.2%	15.7%	0.1%	84.2%
many	6297	$M/M/1$	55.8%	0.0%	44.2%	0.0%	55.8%

Our method is currently only applicable online to queues with sufficient local steady state behavior for the instrumentation in Chapter 6 to converge or to applications will well characterized steady state workloads. To show what happens when a local steady state is not reached we will use the Rabin-Karp string searching kernel and change the packet to be extremely large proportionate to the size of the data set. This results in fewer data packets sent from the source kernel to the "rolling-hash" kernel and no steady state. The resulting observed queue occupancies are much lower than what is calculated by either queueing model. Applying an $M/M/1$ mean queue occupancy model to this application will still result in a queue which is sized for the mean potential occupancy. Table 7.4 shows the result of attempting to evaluate the SVM against a queue that has not reached steady state. As a consequence of the streaming streaming service rate approximation method, it is knowable when the application has reached at least a local steady state and this condition can generally be avoided.

Table 7.4: % for SVM evaluated against a Rabin-Karp string search algorithm that has not reached steady state.

Model	#*obs*	TP	TN	FP	FN	Correct Classification
$M/M/1$	120	21.6%	41.5%	20.7%	16.2%	63.1%
$M/D/1$	120	11.1%	44.4%	44.4%	0.0%	55.5%

7.2.5 Artificial Neural Network (ANN)

An artificial neural network is a method of using a collection of nodes (mathematical functions) which inter-connect to mimic what is thought to be the functioning of a biological neural network [26]. ANN(s) have been used extensively in pattern recognition of all types, and once trained have often been more effective at finding patterns than the previously mentioned SVM approach [102]. The other advantage of an ANN is that they are very amenable to development in hardware [48], and as such might present an avenue for extremely low overhead decision making if said hardware is ever incorporated into main stream SoCs.

Due to the complexity of the ANN's under consideration, it is relatively difficult to describe the network fully in concrete terms, however here are the relevant specs for our model: 76 surface nodes, 2 hidden layers of 228 nodes each, and two output nodes (one for each class). Our activation function (for both stochastic models) is a linear rectifier function (a.k.a., the softplus function [66]). The L1 and L2 regularizers are 0 and .1 respectively.

ANN and Training

Given the success of the SVM, what we really want to show is that the ANN is at least equivalent to the SVM for this application, and that the results garnered from the SVM approach aren't simply some artifact of the experimental setup. The ANN is trained from the same microbenchmark data pool as the SVM. The data are re-partitioned using a uniform random process into training and testing sets. The training set contains 20% of the data whereas the testing set contains the rest. Once partitioned, numerical data are scaled to a range of $[-1000, 1000]$, just as the SVM data.

ANN Results

Figure 7.6 shows classification accuracy for our neural network for the two stochastic models. The overall accuracy for the ANN trained for the $M/M/1$ model is 80.8%, whereas the $M/D/1$ sits at approximately 79.2%. Neither of these results is quite as good as that of the SVM, however much time can be spent optimizing the parameters and error functions of the neural net (as well as the activation functions) which was not taken in our case. The SVM our ANN is compared to had been highly optimized for it's expected task (optimized parameters, not over-trained).

What is surprising with the ANN are the relatively high numbers of false positives for both types of stochastic model. For the Neural Network to be truly effective, these would have to be diminished. False negatives are an okay condition, the end result of which is a slightly longer but conservative optimization process. False positives could lead to a mis-allocated buffer, resulting in sub-optimal performance, mis-managed resources or both.

7.3 Conclusions & Future Work

We have shown a working example of the use of a SVM to classify a stochastic queuing model's reliability for a particular queue within a streaming application that is usable online. This same concept was tried again with a differing classification approach, an artificial neural network. Both methods worked quite well, and once trained are fast to classify their targets.

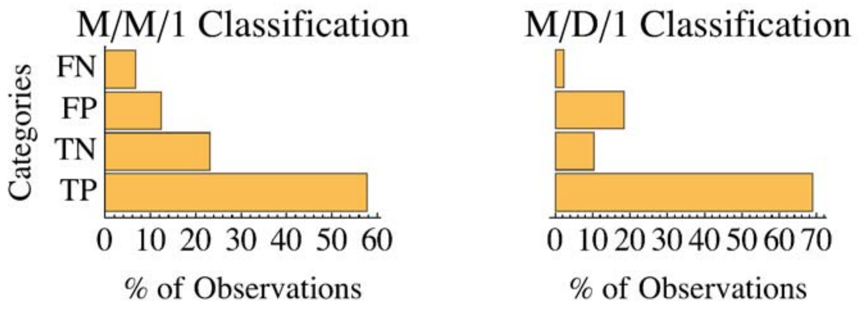


Figure 7.6: Overall the neural network results are quite good considering they were not extensively tuned. NOTE: 'TP' = True Positive, 'FP' = False Positive, and the rest follow. Also note that the training and testing sets for both are not over the same hardware sets since each training and testing process uses randomly selected training and hold out sets.

Both methods enable fast online modeling and re-optimization of stream processing systems (with the caveat that the ANN is slightly worse with more false positives). Across multiple hardware types, operating systems, and applications we've shown that hardware and software fingerprints can be used to classify the reliability of a stochastic queueing model.

This work chose to largely ignore the actual distribution of each compute kernel. What would happen if we knew the underlying distribution of the service and arrival process for each compute kernel in the system? Manually determining the distributions of each compute kernel and retraining the SVM with this knowledge for the $M/M/1$ model we arrive at a 96.6% correct classification rate. This works just as well for the $M/D/1$ model where we observed 96.4% of the queues being correctly classified as either "use" or "don't use." It should be noted that in neither case was the SVM given knowledge that the model it was receiving was for one distribution or another. One obvious path for future work is faster and lower overhead process distribution estimation. Mathematically this can be done with the method of moments, what is left is an engineering challenge.

Empirical data could also be seen as a weakness of our approach since it is obviously finite in its coverage of the combinatorial plethora of possible configurations. We trained our SVM and ANN using as wide a variety of configurations as possible, however the permutations of possible application configurations are quite high. Other combinations of applications could provide slightly differing results. Our choices of attributes is limited to what the hardware and operating system could provide. Omniscient knowledge of the system would obviously be helpful, it is possible that future platforms will provide more robust identification and monitoring features which could improve the training and classification process.

In conclusion we have demonstrated an automated way to classify the reliability of stochastic queueing models for streaming systems. We have shown that it can be done, and that in many cases it works quite well for the applications and configurations tested.

Chapter 8

Online Tuning: Putting It All Together

Throughout the course of this dissertation we've shown several methods that enable online tuning of stream processing systems. Specifically we've looked at allocating buffers within a streaming system. We've shown methods to approximate the throughput through a queueing network using network flow models. We came up with a new closed form distribution to model the best case execution time (service time) variation, and demonstrated how to apply that variation to deciding whether or not to use a particular stochastic queueing model. Next we gave techniques that enable low overhead service rate determination while an application is executing and in the last chapter it was shown that several machine learning techniques can successfully determine when a model is or is not reliable using fingerprints extracted from the system. In this chapter we will conclude by applying several of the techniques elaborated on in other chapters, comparing RaftLib's adaptive behavior to an optimal found through complete enumeration.

8.1 Online Modeling of Streaming Systems

The one certainty in life besides death is change. Things will change around us, whether that be the whether or more drastic phenomena. This is true for applications that we run on computers. The beautiful abstractions that we've built our computational houses upon invite change. Much of it is hidden, dynamic, and some of it relatively random. What we want are robust applications that can roll with the punches that they receive; gracefully reacting and adapting to the bumps that they encounter. That is the main point of our online modeling efforts, online modeling so that our run-time can make the best decisions for our applications as possible with as little overhead as possible. We'll show that our online efforts can come close to the results produced by complete enumeration of the design space. I've mentioned that environments change for applications. In the next few paragraphs I'll outline some of the environmental stimuli, why they're important, and a few real examples as to why they matter. I'll give a few reasons behind why these are not fleeting examples, but ones that will continue and multiply as technology advances. Lastly I'll give a few examples as to how RaftLib can adapt based on the technologies that we've developed within this thesis.

Increasingly architectures have multiple types of functional units. The big.LITTLE architecture from ARM, and the shared floating point unit architecture of AMD are but two notable examples. These carry with them the promise of power savings and efficiency but they also present challenges for streaming applications. Namely that the apparent service rates for each kernel can change based on contention for the floating point units (AMD architecture) or if the kernel is migrated (involuntarily) from a robust A57 to a more efficient A53 compute core (ARM architecture). These two examples aren't some fleeting trend, these heterogeneous architectures are arriving from multiple vendors in response to tighter budgets and energy efficiency requirements. Similar to this is dynamic voltage frequency scaling (DVFS) which can change performance characteristics of a core, which in turn changes the apparent service rate. DVFS is common to almost every modern multi-core processor.

Changes in workload within a system are common. On cloud systems this is the norm, clusters are not producing unless they are computing. With each scheduled job on a system the amount of compute time available to each thread (assuming fair scheduling) is lessened. This results in an effective lessening of the service rate for the kernel that experiences a smaller time quanta each time it is scheduled. The same behavior applies to other finite resources as well such as physical memory, cache lines (which can be polluted by other processes) and even things like load/store queues can become bottlenecks. Freeing up memory for other tasks, as well as providing adequate buffering capacity within each queue is critical to the performance of the application (smaller queues decrease the memory footprint, increasing cache utilization whereas larger buffers can improve throughput).

8.1.1 Why is online tuning important?

Workload characterization is hard [37]. The characterization takes time, it is expensive and it is critical to the current state of the art tuning methods [63]. Characterizations are unfortunately ephemeral, valid only for the architecture on which the application was characterized. Moving from one architecture to another requires more characterization. The expense doesn't stop at re-characterization, it is also in re-factoring. Once the applications are characterized, a code re-write is often in order to take advantage of new hardware. The vision of RaftLib is to create a tuning system that enables a programmer to write kernels once (focusing only on the kernel) then optimize the parallel network as much as possible once written. No offline characterization is needed, the user simply hits go and given enough time the runtime adapts to the hardware it is given.

8.1.2 Adaption Types

RaftLib is a multi-dimensional adapter. It can adapt by changing buffer sizes (up or down), modify placement of compute kernels, and duplicate compute kernels while executing. In this subsection I'll discuss the highlights of each and brief methodology for those techniques not explicitly highlighted in other sections so that the results shown will have context.

Multiple buffers exist within a streaming application (whether explicit or implicit), queueing behavior naturally results from the interaction of multiple asynchronous threads/kernels. Each buffer has some optimal sizing that will allow the sender and receiver to execute at some optimal performance (for some definition of optimality, and with the performance gains arising from reduced blocking which occurs with bursty transmissions). RaftLib uses branch and bound search and model based techniques to find an approximately optimal buffer while the application is executing, not just once but continually until the application is finished executing. RaftLib adapts buffers through a combined monitoring effort. Queues are monitored for mean occupancy, if it is above k percent for n observations then the queue is resized. If service rate information is unavailable, then a branching approach is taken as the default. Conversely if queues are over-provisioned the opposite action is taken and the queue is downsized. If service rate information is available, then a SVM is utilized to choose a model. If either model is available then we select the fastest to compute (determined via static profiling and saved as a profile for each machine when RaftLib is setup). If no model is usable, then the fallback is again branch and bound. There is of course an engineering limit imposed on the buffers, which is set to avoid impossible memory allocations (i.e., above user limits, or above that allowed by virtual and physical memory) which for our adaptation experiments is set to 16 MB.

The duplication of compute resources is determined independently (separably) from buffer sizing. This simplifies the solution and exposition considerably (in reality duplication and sizing are inextricably linked, and the independent solution to duplication leads to an automatic correction of the buffer capacity, i.e., buffers that were once always full now have a lesser arrival rate). Not being the primary subject of this thesis, the logic for duplication is quite simple. If the service rate monitoring is available for the downstream kernel, then it is used, otherwise indirect measures such as queue occupancy are used. If the queues downstream are 50% full for k monitoring cycles then the upstream kernel is duplicated if the run-time determines that it is duplicable. If service rate information is available (i.e. turned on and recently converged) then a flow model analysis tells the runtime if any additional gains can be made through duplication.

Obviously duplication might result in a run-time wanting to move the kernel to a new processor. The following is only applicable to operating systems which allow thread "pinning" such as Linux/Unix variants. Assigning kernels to processors is a partitioning problem, discussed in Chapter 3. I've largely side-stepped the partitioning problem since in and of itself, is a large, rich subject. RaftLib uses a very simplistic partitioning scheme based on minimizing the communications distance between nodes (partition provided by the Scotch partitioning framework). The partition Scotch returns maps which kernels are assigned to each compute core. This however is only done offline (for the initial partition). Partitioning using this method is rather slow compared to the time in which we have to make an assignment decision, therefore when duplicating and assigning only neighboring cores are considered. Online duplication decisions are based on flow model results, with the theoretical duplicate core placed in the flow model as having an identical service rate to its clone (with appropriate sharing model applied as described in Chapter 4). Movement of extant kernels is also possible, however the logic that determines core assignment is biased against moving. In our experiments the bias is set at a 15% probability of moving based on a uniform random number generator. This is a simple engineering solution to reduce the probability of frequent core migrations, which would otherwise have a deleterious effect on cache utilization.

8.1.3 Evaluation Methodology

The hardware used for all empirical evaluation is listed in Table 6.3. All code is compiled with the "-O2" compiler flag using the Clang compiler (version 3.5). For test applications, we will use the Boyer-Moore-Horspool string searching algorithm (fully described in Chapter 3). Both have the desired characteristic in that they can produce very high throughput rates in string search and workloads that are variable depending on the composition of the corpus being searched. The corpus searched is the set of all posts from a popular programming help site [178] (data set is cut to exactly 23 GB).

In order to judge how well our online adaptation works, we need a baseline for the application on the specified hardware. To do this a complete enumeration is done with the following limits:

1. number of threads is limited to the number of cores on each machine
2. buffer size for each buffer is capped to 4096 elements of each type, the step size is 16
3. core selection for each thread is selected by a uniform random process

.

8.1.4 Adaption Results

Figure 8.1 shows the baseline for a fairly complete enumeration $1 - 14$ cores (note: 1 core masked for interrupts not used and one core was reserved for receiving timing data and handling I/O calls to the OS so as not to disturb the measurements). Complete enumeration found the highest throughput of 10.93 GB/s with no statistically significant gain moving from 10 threads to 11. Zooming in on the tail of Figure 8.1, the step pattern of the larger graph is seen very clearly. this is the increment in thread count, with the slow curve representing the increase in buffer size. The buffer size for this throughput varied between 3088 and 4096 elements whereas the placement favored the string searching kernels being very close to the data source and the reducer being further away which is what would be expected from cooperative caching.

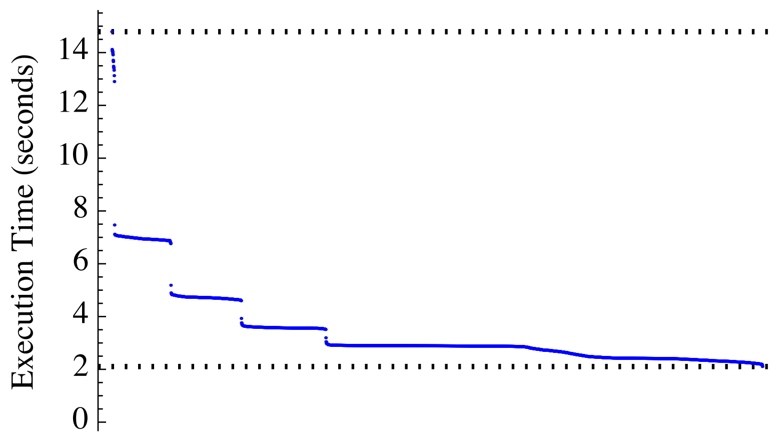


Figure 8.1: Complete enumeration of the Boyer-Moore-Horspool string search (with the caveats enumerated in the text). The min execution time (lower dotted line) is found at 2.105 s, the max at 14.790 s (upper dotted line) representing a maximum throughput for the data set size of 10.93 GB/s and minimum of 1.55 GB/s.

The enumeration result represents a slight improvement in the performance from the earlier benchmarking in Chapter 3 of RaftLib's dynamic adaptation, in fact this shows that for this platform, RaftLib is able achieve 73% of fully optimal throughput. These earlier experiments, however left the kernels "un-pinned" and did not allow auto-parallelization since a comparison against Apache Spark was the goal (per thread performance). It is expected that starting with a reduced level of parallelism and increasing it will hurt performance, while starting with a relatively optimal partition and keeping close to it will improve it.

The results of RaftLib dynamically adapting (starting with only three compute kernels at the beginning of execution) and scaling up to 11 matching kernels (along with the data source and join kernels, cloning events shown in timeline in Figure 8.3) are fairly good overall. Queue sizing events are likewise plotted on a timeline in Figure 8.4, resultant queue sizes fell in the range 2880 elements through 64128 while starting at a uniform 64 elements per queue. The overall execution time for this example is 2.92 s which results in a throughput with this data-set of 7.96 GB/s. This is 72% of optimal throughput which is quite good considering the topology started off identical to the 1.55 GB/s topology timed as part of the complete enumeration. The auto-parallelization of kernels, at least in this limited case, only decreased performance by 1% compared to the statically parallel execution of the same algorithm described in Chapter 3. It is highly probable (given the probabilities of lock contention for adding, and allocating new kernels), that more overhead could be associate with larger graphs.

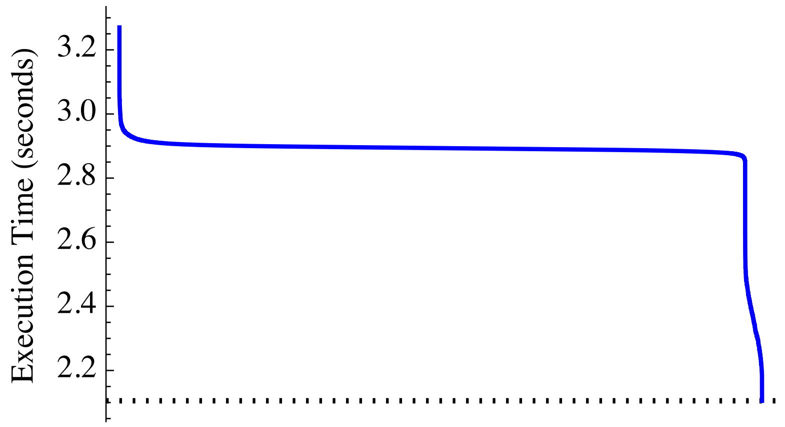


Figure 8.2: Zoomed in tail of Figure 8.1 showing the long flat step between thread counts. At each thread count increase there is a precipitous drop followed by a slow curving increase as buffer sizes are increased and performance improves gradually. The last drop is followed by a no improvement with increased thread counts for this particular architecture.

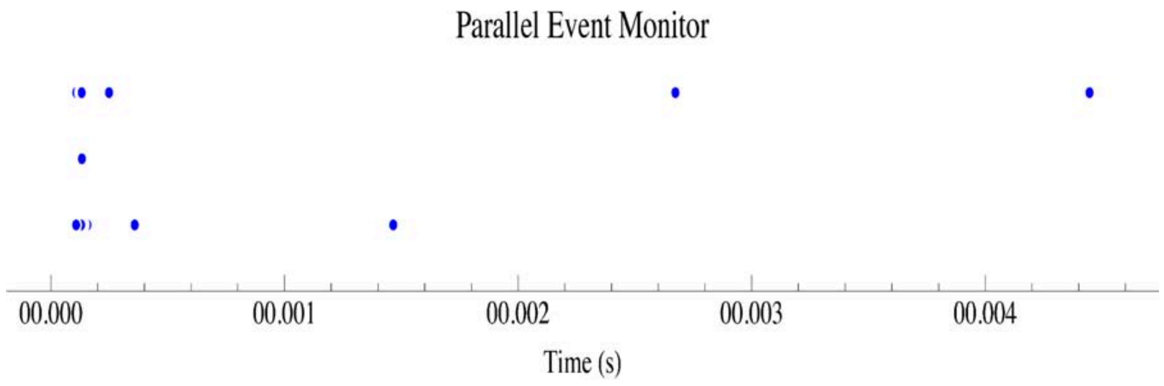


Figure 8.3: Parallel event monitor thread, which is responsible for duplication and core assignment, recorded 11 total kernel duplication events. Most were quite early on, but by 4 ms the parallelism level is unchanged, even though the monitor thread is still running and monitoring the system

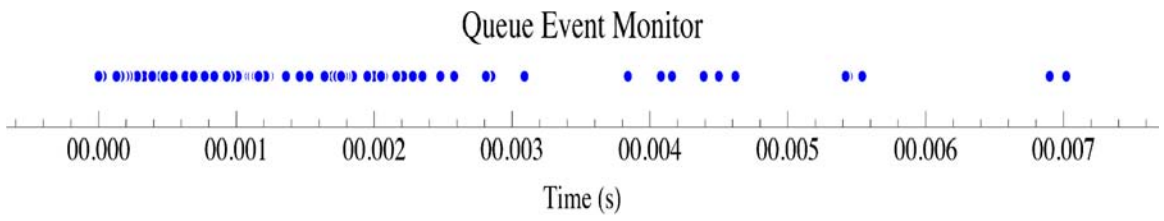


Figure 8.4: Queue resizing events for optimal run of Boyer-Moore-Horspool, queue sizes ranged from 2880 elements through 64128 elements. After the end of the timeline, the monitor thread is still running but no further resizing events were recorded

Outside of this exemplar, multiple executions of RaftLib optimized Boyer-Moore-Horspool fell in the range of 2.92 s through 4.24 s.

8.2 Conclusions

In this chapter we demonstrated that the technologies described in previous chapters can work together to form a coherent, adaptive system. The instrumentation, modeling, model selection, and run-time complement each other. We showed that RaftLib can adapt to within 28% of optimal, completely while executing. Searching for the optimal configuration (despite limiting the available knobs) took just over a week (executing each configuration 100 x to ensure statistically sound time averages). While empirical and subject to the whimsies of probability, this is definitely a success.

Chapter 9

Conclusions and Future Work

In this thesis we described a new stream processing library (RaftLib) which enables integration of classic stream processing within legacy C++ environments. With this library came the promise of an auto-tuned execution on multi-core processor cores. To live up to this promise we explored techniques for fast and "accurate-enough" modeling. In this section we summarize the contributions of each chapter, discuss how they fit together, and lastly conclude with a section on where future work may lie.

9.1 Conclusions

In Chapter 1 we promised a streaming system that could be tuned without any user intervention save for authoring, compiling, and executing. In Chapter 3 we described the streaming framework, some implementation details, and a benchmark comparing it to another leading streaming frameworks. Overall, RaftLib compared quite favorably in performance. Leading up to that performance, however, were many models, methods and techniques that were left to subsequent chapters. First, some models needed to describe the performance (throughput) of streaming applications had to be developed (see Chapter 4).

In Chapter 4 we described an extension of techniques first described in Operations Research literature (the use of flow models to approximate throughput through a queueing network). Our primary contribution was the extension of these prior works by utilizing a gain/loss flow model and our unique contribution was the addition of routing constraints which requires one additional pass through the graph to arrive at an appropriate throughput. We demonstrated through empirical evaluation that our method does indeed work well, however we found our estimates of buffer capacity wanting. The estimates were generally sufficient to maintain maximum throughput, however they were generally huge. This wastes much space and after a point begins to stifle performance. In other cases the buffers were too small, which also restricts performance as blocking increases. To remedy this we began to dissect what went wrong in Chapter 5.

The worst case execution time problem has been studied for decades (see Chapters 2 and 5 for discussion). We chose to focus on the variation in the best case execution time, which we hoped would lead to a pattern which could be used to forecast if our characterizations made offline would hold in a dynamic online environment (see Chapter 5). We found that a Levy distribution fit the pattern quite well, however it also has infinite support (and no defined moments in general). To remedy this we defined a truncation based on millions of empirical execution observations. We further re-defined the distribution in terms of threads assigned per core and the expected service time of each thread assigned. This redefinition led to the $m$levy distribution, which is usable without expensive parameter fitting. We also demonstrated that the mLevy can be used to accept or reject a queueing model with offline analysis. This analysis, however was quite expensive. With the end goal being online modeling and tuning, faster methods are needed. First however, we needed some idea of the non-blocking service rate for each compute kernel. To do this we need to developed new instrumentation techniques.

In Chapter 6 we describe a way to approximate the non-blocking service rate of each kernel within a streaming system while the application is executing (online). We show that in many cases it is possible to get quite accurate estimates, and in the cases where convergence is impossible we suggest multiple techniques that can improve the estimates. Another contribution of this approach, over prior ones is that the instrumentation itself can be turned on and off dynamically, which when combined with control optimizations could reduce overhead even further. In addition to a mean and variance (which are provided via our instrumentation) it is also possible to estimate higher moments (now an engineering effort). With higher moments, the run-time can estimate the process distribution while executing. With a method to divine service rates in an online manner, now we need a faster method than those used in Chapter 5 to accept or reject performance models as usable. To do this we turned to machine learning.

The pattern of variation discovered in Chapter 5 was used with a metric called KL-divergence to assess at what point the distribution expected by the queueing model diverged too much from the distribution observed dynamically during execution. This method is extremely slow. Machine learning methods are very good at finding patterns and at classification when a pattern exists (such a pattern is suggested by Chapter 5). Using both support vector machine and neural network models, we demonstrated that machine learning approaches (when trained offline on a plethora of data) could indeed identify when a performance model is appropriately used. These methods are fairly quick to execute and classify (offline trained model used for online classification). The support vector machine approach (being the most accurate of the methods assessed) is incorporated into the RaftLib framework.

In Chapter 3 we ended with a set of benchmark experiments that demonstrated the performance of RaftLib using a range of thread counts against other popular frameworks. These experiments disabled thread-pinning and also auto-parallelization (so that we could compare exact thread to thread performance vs. the other frameworks). In Chapter 8 we put everything together. We start by executing an application that looks like a tandem queueing network, which quickly expands to absorb available computing power, moves the threads to locally optimal positions, and adjusts queue sizes for maximum throughput using model based calculations and bounding search as a fall-back methodology. Not surprisingly, the performance was slightly less than that shown in Chapter 3 (where the thread counts are selected for RaftLib), however it is 72% of optimal performance (determined through complete enumeration of the design space), and 50% in the worst outlier case. This is quite good for a system that tunes itself, with sub-second decisions and no human interaction other than authoring, compiling, and executing.

9.2 Future Work

This dissertation described some methods that reduce the cost of re-factoring and tuning performant parallel code. The attained tuning levels are excellent for a first start, however they are nowhere near the 99% of optimal performance attainment that a high performance computing center would expect. Not to trivialize the accomplishments enumerated in the previous section, but there are many research avenues that could potentially improve this number. First is optimizing the control loops so that the overhead of performing online optimization can be reduced further. Second, integrating cache aware buffer allocations, and more optimal pre-fetching could improve performance. Third, improvements in language semantics that enable the compiler to make better optimization choices about where to place data could go a step towards improving the online optimization process before we ever get online.

Most modern systems (both commodity and special purpose) are non-uniform in processor type and memory access (also known as heterogeneous systems). Extracting performance from heterogeneous systems requires deep understanding of the interactions between hardware and software. Extracting performance is accomplished in two very different ways based on the user type. Advanced users require control, they have oracle knowledge of the application, and they want to use it. Novice users on the other hand often abuse advanced features to their detriment. Given the proliferation of massively parallel, heterogeneous hardware I would like to target the novice / intermediate level programmer with better languages for more efficient parallel programming. Looking past legacy code, I think there are better modalities on the horizon. How do we quantify which is the best? As a scientist I want to quantify what makes a language or programming modality "good." Currently decisions about language/library semantics, and indeed adoption are largely aesthetically based (including those within RaftLib). I want to explore other things like crowd sourcing to quantify the "best" semantic constructs based upon some criteria (e.g., intuitiveness). Another avenue of research is complete language construction via crowd sourcing. New linguistic designs have the advantage of a clean slate design. How should parallel execution be specified (if at all)?

We've talked about buffer allocations within this dissertation as having a fixed size from moment to moment. How about compressed buffers? Much has been done on memory compression research, but very little on buffer compression in performance sensitive streaming applications. An interesting research question is if you can combine an online model based approach to determine if buffer compression is useful and when it is not. Can compressed buffers improve memory performance by reducing the overall amount of traffic? Concomitant to compression is reduction in memory traffic. Advances in compute near memory [80] have the potential to reduce memory traffic by making contiguous otherwise unwieldy access strides (e.g., convolutions and graph analytics). Giving these types of operations more efficient memory footprints might improve the adoption of stream processing for more than just classically streaming workloads.

While we've taken great pains to validate each portion of this work through empirical evaluation, any empirical validation is subject to change with differences in hardware/software systems. Future work might be to expand the training selection for our models to include more hardware types, scheduler types, and operating systems. In addition to more training, another addition from the work done in Chapter 7 could be to perform full model selection based on a set of models instead of simply assessing reliability.

References

[1] Marwen Abbes, Foutse Khomh, Y-G Guéhéneuc, and Giuliano Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 181–190. IEEE, 2011.

[2] Milton Abramowitz and Irene A Stegun. *Handbook of Mathematical Functions: with Formulas, Graphs, and Mathematical Tables.* Courier Dover Publications, 2012.

[3] William B. Ackerman. Data flow languages. *Computer*, 15(2):15–25, 1982.

[4] Vikram Adve, Alan Carle, Elana Granston, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, John Mellor-Crummey, Scott Warren, and Chau-Wen Tseng. Requirements for data-parallel programming environments. Technical report, DTIC Document, 1994.

[5] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. April: a processor architecture for multiprocessing. In *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*, pages 104–114. IEEE, 1990.

[6] Kunal Agrawal, Jeremy Fineman, and Jordyn Maglalang. Cache-conscious scheduling of streaming pipelines on parallel machines with private caches. In *To appear in the Proceedings of the IEEE International Conference on High Performance Computing (HiPC)*, December 2014.

[7] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[8] G Alon, Dirk P Kroese, Tal Raviv, and Reuven Y Rubinstein. Application of the cross-entropy method to the buffer allocation problem in a simulation-based environment. *Annals of Operations Research*, 134(1):137–151, 2005.

[9] Venkat Anantharam. The optimal buffer allocation problem. *Information Theory, IEEE Transactions on*, 35(4):721–725, 1989.

[10] Venkat Anantharam and Ayalvadi J. Ganesh. Correctness within a constant of an optimal buffer allocation rule of thumb. *Information Theory, IEEE Transactions on*, 40(3):871–882, 1994.

[11] Ted W Anderson and David A Darling. Asymptotic theory of certain" goodness of fit" criteria based on stochastic processes. *The Annals of Mathematical Statistics*, pages 193–212, 1952.

[12] Péter Arató, Sándor Juhász, Zoltán Ádám Mann, András Orbán, and Dávid Papp. Hardware-software partitioning in embedded system design. In *IEEE International Symposium on Intelligent Signal Processing*, pages 197–202. IEEE, 2003.

[13] Dorian C Arnold, Henri Casanova, and Jack Dongarra. Innovations of the NetSolve grid computing system. *Concurrency and Computation: Practice and Experience*, 14(13-15):1457–1479, 2002.

[14] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalape&ntilde;o jvm. *SIGPLAN Not.*, 35(10):47–65, October 2000.

[15] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.

[16] C. Augonnet, S. Thibault, and R. Namyst. Automatic calibration of performance models on heterogeneous multicore architectures. In *Proc. of Euro-Par 2009–Parallel Processing Workshops*, pages 56–65, 2010.

[17] François Baccelli and Zhen Liu. On the stability condition of a precedence-based queueing discipline. *Advances in Applied Probability*, pages 883–898, 1989.

[18] Magdalena Balazinska, Hari Balakrishnan, Samuel R Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 33(1):3, 2008.

[19] Jonathan C. Beard and Roger D. Chamberlain. Analysis of a simple approach to modeling performance for streaming data applications. In *Proc. of IEEE Int'l Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 345–349, August 2013.

[20] Jonathan C. Beard and Roger D. Chamberlain. Use of simple analytic performance models of streaming data applications deployed on diverse architectures. In *Proc. of Int'l Symp. on Performance Analysis of Systems and Software*, pages 138–139, April 2013.

[21] Jonathan C. Beard and Roger D. Chamberlain. Use of a Levy distribution for modeling best case execution time variation. In A. Horváth and K. Wolter, editors, *Computer Performance Engineering*, volume 8721 of *Lecture Notes in Computer Science*, pages 74–88. Springer International Publishing, September 2014.

[22] Jonathan C. Beard and Roger D. Chamberlain. Run time approximation of non-blocking service rates for streaming systems. In *Proceedings of the 17th IEEE International Conference on High Performance and Communications*, pages 792–797. IEEE, August 2015.

[23] Jonathan C. Beard, Cooper Epstein, and Roger D. Chamberlain. Automated reliability classification of queueing models for streaming computation using support vector machines. In *Proceedings of the 6th ACM/SPEC international conference on Performance engineering*, ICPE '15, New York, NY, USA, January 2015. ACM.

[24] Jonathan C. Beard, Cooper Epstein, and RogerD. Chamberlain. Online automated reliability classification of queueing models for streaming processing using support vector machines. In Jesper Larsson Trff, Sascha Hunold, and Francesco Versaci, editors, *Euro-Par 2015: Parallel Processing*, volume 9233 of *Lecture Notes in Computer Science*, pages 82–93. Springer Berlin Heidelberg, 2015.

[25] Jonathan C. Beard, Peng Li, and Roger D. Chamberlain. Raftlib: A C++ template library for high performance stream parallel processing. In *Proceedings of Programming Models and Applications on Multicores and Manycores*, PMAM'15, New York, NY, USA, February 2015. ACM.

[26] Christopher M Bishop. *Pattern recognition and machine learning.* springer, 2006.

[27] Shekhar Borkar. Exascale computing a fact or a fiction?, May 2013. Keynote Speech, 27th International Parallel & Distributed Processing Symposium.

[28] Jeffrey Bosboom, Sumanaruban Rajadurai, Weng-Fai Wong, and Saman Amarasinghe. StreamJIT: A commensal compiler for high-performance stream programming. In *Proc. of ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 177–195. ACM, 2014.

[29] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proc. of 5th Workshop on Computational Learning Theory*, pages 144–152, 1992.

[30] George EP Box and Norman Richard Draper. *Empirical model-building and response surfaces*, volume 424. Wiley New York, 1987.

[31] Gert Brettlecker, Heiko Schuldt, and Hans-Jörg Schek. Efficient and coordinated checkpointing for reliable distributed data stream management. In *Advances in Databases and Information Systems*, pages 296–312. Springer, 2006.

[32] Eric A Brewer. High-level optimization via automated statistical modeling. In *ACM SIGPLAN Notices*, volume 30, pages 80–91. ACM, 1995.

[33] Bruce M Brown et al. Martingale central limit theorems. *The Annals of Mathematical Statistics*, 42(1):59–66, 1971.

[34] Randal Bryant and David Richard O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.

[35] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. on Graphics*, 23(3):777–786, 2004.

[36] JA Cadzow and HR Martens. *Discrete-Time and Computer Control Systems*. Englewood Cliffs: Prentice-Hall, 1970.

[37] Maria Calzarossa and Giuseppe Serazzi. Workload characterization: A survey. *Proceedings of the IEEE*, 81(8):1136–1150, 1993.

[38] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[39] T.L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. on Software Engineering*, 14(2):141–154, 1988.

[40] Amit Chakrabarti, Graham Cormode, and Andrew McGregor. Robust lower bounds for communication and stream computation. In *Proc. of 40th ACM Symposium on Theory of Computing*, pages 641–650. ACM, 2008.

[41] IM Chakravarty, JD Roy, and RG Laha. *Handbook of Methods of Applied Statistics*. McGraw-Hill, 1967.

[42] Roger D. Chamberlain and Joseph M. Lancaster. Better languages for more effective designing. In *Proc. of Int'l Conf. on Engineering of Reconfigurable Systems & Algorithms*, July 2010.

[43] Roger D. Chamberlain, Joseph M. Lancaster, and Ron K. Cytron. Visions for application development on hybrid computing systems. *Parallel Comput.*, 34(4-5):201–216, May 2008.

[44] Tony F Chan, Gene H Golub, and Randall J LeVeque. Algorithms for computing the sample variance: Analysis and recommendations. *The American Statistician*, 37(3):242–247, 1983.

[45] Rohit Chandra. *Parallel Programming in OpenMP*. Morgan Kaufmann, 2001.

[46] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[47] Olivier Chapelle, Vladimir Vapnik, Olivier Bousquet, and Sayan Mukherjee. Choosing multiple parameters for support vector machines. *Machine Learning*, 46(1-3):131–159, 2002.

[48] Tianshi Chen, Shijin Zhang, Shaoli Liu, Zidong Du, Tao Luo, Yuan Gao, Junjie Liu, Dongsheng Wang, Chengyong Wu, Ninghui Sun, et al. A small-footprint accelerator for large-scale neural networks. *ACM Transactions on Computer Systems (TOCS)*, 33(2):6, 2015.

[49] Yi-Wei Chen and Chih-Jen Lin. Combining SVMs with various feature selection strategies. In *Feature Extraction*, pages 315–324. Springer, 2006.

[50] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Computing Surveys (csuR)*, 34(2):171–210, 2002.

[51] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[52] Working Draft, Standard for Programming Language C++. http://goo.gl/JIOjsU. Accessed Ocbober 2014.

[53] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods.* Cambridge University Press, Cambridge, UK, 2000.

[54] Louise H Crockett, Ross A Elliot, Martin A Enderwitz, and Robert W Stewart. *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc.* Strathclyde Academic Media, 2014.

[55] Frederico RB Cruz, Anderson Ribeiro Duarte, and Tom Van Woensel. Buffer allocation in general single-server queueing networks. *Computers & Operations Research*, 35(11):3581–3598, 2008.

[56] David E Culler. Dataflow architectures. Technical report, DTIC Document, 1986.

[57] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.

[58] G De Michell and Rajesh K Gupta. Hardware/software co-design. *Proceedings of the IEEE*, 85(3):349–365, 1997.

[59] Richard H Deane and Colin L Moodie. A dispatching methodology for balancing workload assignments in a job shop production facility. *AIIE Transactions*, 4(4):277–283, 1972.

[60] Jack B Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[61] Jack Bonnell Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980.

[62] R.P. Dick et al. TGFF: Task graphs for free. In *Proc. of 6th Int'l Workshop on Hardware/Software Codesign*, pages 97–101, 1998.

[63] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.C. Andre, D. Barkai, J.Y. Berthou, T. Boku, B. Braunschweig, et al. The International Exascale Software Project Roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60, 2011.

[64] Rahav Dor, Joseph M. Lancaster, Mark A. Franklin, Jeremy Buhler, and Roger D. Chamberlain. Using queuing theory to model streaming applications. In *Proc. of 2010 Symposium on Application Accelerators in High Performance Computing*, July 2010.

[65] Rahav Dor, Joseph M. Lancaster, Mark A. Franklin, Jeremy Buhler, and Roger D. Chamberlain. Using queuing theory to model streaming applications. In *Proc. of Symp. on Application Accelerators in High Performance Computing*, July 2010.

[66] Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. Incorporating second-order functional knowledge for better option pricing. *Advances in Neural Information Processing Systems*, pages 472–478, 2001.

[67] Stewart Edgar and Alan Burns. Statistical analysis of WCET for scheduling. In *Proc. of 22nd IEEE Real-Time Systems Symposium*, pages 215–224, 2001.

[68] Jakob Engblom and Andreas Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proc. of 6th Int'l Conf. on Real-Time Computing Systems and Applications*, pages 88–95, 1999.

[69] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proc. of 38th Int'l Symp. on Computer Architecture*, pages 365–376, 2011.

[70] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. Graph distances in the streaming model: The value of space. In *Proc. of 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 745–754, Philadelphia, PA, USA, 2005. SIAM.

[71] LR Ford and DR Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, New Jersey, USA, 1962.

[72] M.A. Franklin, E.J. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer. Auto-Pipe and the X language: A pipeline design tool and description language. In *Proc. of Int'l Parallel and Distributed Processing Symp.*, April 2006.

[73] D.R. Fulkerson and G.B. Dantzig. Computation of maximal flows in networks. *Naval Research Logistics Quarterly*, 2(4):277–283, 1955.

[74] INO Fumihiko, Shinta Nakagawa, and Kenichi Hagihara. GPU-Chariot: A programming framework for stream applications running on multi-GPU systems. *IEICE Transactions on Information and Systems*, 96(12):2604–2616, 2013.

[75] M. Galassi, B. Gough, G. Jungman, J. Theiler, J. Davies, M. Booth, and F. Rossi. The GNU scientific library reference manual, 2007. *URL http://www. gnu. org/software/gsl.*

[76] Michael R. Garey and David S. Johnson. *Computers and Intractability*, volume 174. Freeman San Francisco, CA, 1979.

[77] Hugh G Gauch Jr. Model selection and validation for yield trials with interaction. *Biometrics*, pages 705–715, 1988.

[78] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[79] GNU. grep. http://www.gnu.org/software/grep/. Accessed Ocbober 2014.

[80] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The terasys massively parallel pim array. *Computer*, 28(4):23–31, 1995.

[81] A.V. Goldberg, S.A. Plotkin, and É. Tardos. Combinatorial algorithms for the generalized circulation problem. In *Proc. of 29th Symp. on Foundations of Computer Science*, pages 432–443, 1988.

[82] D. Goldfarb, Z. Jin, and J.B. Orlin. Polynomial-time highest-gain augmenting path algorithms for the generalized circulation problem. *Mathematics of Operations Research*, 22(4):793–802, 1997.

[83] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.

[84] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. S-Net: A typed stream processing language. In *Proc. of 18th Int'l Symp. on Implementation and Application of Functional Languages*, pages 81–97, 2006.

[85] Manish Gupta and Prithviraj Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *Parallel and Distributed Systems, IEEE Transactions on*, 3(2):179–193, 1992.

[86] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182, 2003.

[87] Marcus Hähnel and Hermann Härtig. Heterogeneity by the numbers: A study of the odroid xu+ e big. little platform. In *Proceedings of the 6th USENIX conference on Power-Aware Computing and Systems*, pages 3–3. USENIX Association, 2014.

[88] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action.* Cambridge University Press, Cambridge, UK, 2013.

[89] Wim Heirman, Trevor Carlson, and Lieven Eeckhout. Sniper: scalable and accurate parallel multi-core simulation. In *8th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES-2012)*, pages 91–94. High-Performance and Embedded Architecture and Compilation Network of Excellence (HiPEAC), 2012.

[90] Lorin Hochstein, Jeffrey Carver, Forrest Shull, Sima Asgari, Victor Basili, Jeffrey K Hollingsworth, and Marvin V Zelkowitz. Parallel programmer productivity: A case study of novice parallel programmers. In *Proc. of ACM/IEEE Supercomputing Conference*, pages 35–35. IEEE, 2005.

[91] Urs Hölzle. *Adaptive optimization for SELF: reconciling high performance with exploratory programming.* PhD thesis, Stanford University, 1995.

[92] R Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980.

[93] Jeong-Hyon Hwang, Ying Xing, Ugur Çetintemel, and Stan Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 176–185. IEEE, 2007.

[94] International Telegraph and Telephone Consultative Committee et al. Information technology-digital compression and coding of continuous-tone still images-requirements and guidelines. *Rec. T*, 81, 1992.

[95] J.R. Jackson. Networks of waiting lines. *Operations Research*, 5(4):518–521, 1957.

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

[97] Raj Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, 1991.

[98] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High-Performance Programming.* Elsevier Science, 2013.

[99] W.S. Jewell. Optimal flow through networks with gains. *Operations Research*, pages 476–499, 1962.

[100] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *ACM SigPlan Notices*, volume 29, pages 171–185. ACM, 1994.

[101] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[102] David A Kelly. Neural networks for handwriting recognition. In *Aerospace Sensing*, pages 143–154. International Society for Optics and Photonics, 1992.

[103] M.G. Kendall and W.R. Buckland. *A Dictionary of Statistical Terms.* Edinburgh and London: Published for the International Statistical Institute by Oliver & Boyd, Ltd., 1957.

[104] Brian W Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *Bell system technical journal*, 49(2):291–307, 1970.

[105] Gokcen Kestor, Roberto Gioiosa, Darren J Kerbyson, and Adolfy Hoisie. Quantifying the energy cost of data movement in scientific applications. In *2013 IEEE international symposium on workload characterization (IISWC)*, May 2013.

[106] Brucek Khailany, William J Dally, Ujval J Kapasi, Peter Mattson, Jinyung Namkoong, John D Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, Mar 2001.

[107] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. Workload characterization and prediction in the cloud: A multiple time series approach. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 1287–1294. IEEE, 2012.

[108] Toru Kisuki, P Knijnenburg, M OBoyle, and H Wijshoff. Iterative compilation in program optimization. In *Proc. CPC10 (Compilers for Parallel Computers)*, pages 35–44. Citeseer, 2000.

[109] L. Kleinrock. *Queueing Systems. Volume 1: Theory.* Wiley-Interscience, 1975.

[110] Kathleen Knobe and CD Offner. Compiling to tstreams, a new model of parallel computation. Technical report, Technical report, 2005.

[111] Jay Kreps. Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). http://goo.gl/OtztI4. Accessed September 2014.

[112] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The Annals of Mathematical Statistics*, pages 79–86, 1951.

[113] Christoph Lameter. NUMA (Non-Uniform Memory Access): An overview. *Queue*, 11(7):40:40–40:51, July 2013.

[114] Christopher Lameter. Shoot first and stop the OS noise. In *Proceedings of the Linux Symposium*, July 2009.

[115] Joseph M. Lancaster et al. TimeTrial: A low-impact performance profiler for streaming data applications. In *Proc. of Int'l Conf. on Application-specific Systems, Architectures and Processors*, pages 69–76, September 2011.

[116] Joseph M. Lancaster, E. F. Berkley Shands, Jeremy D. Buhler, and Roger D. Chamberlain. TimeTrial: A low-impact performance profiler for streaming data applications. In *Proc. IEEE Int'l Conf. on Application-specific Systems, Architectures and Processors*, September 2011.

[117] Joseph M. Lancaster, Joseph G. Wingbermuehle, Jonathan C. Beard, and Roger D. Chamberlain. Crossing boundaries in TimeTrial: Monitoring communications across architecturally diverse computing platforms. In *Proc. 9th IEEE/IFIP Int'l Conf. Embedded and Ubiquitous Computing*, pages 280–287, October 2011.

[118] Joseph M. Lancaster, Joseph G. Wingbermuehle, and Roger D. Chamberlain. Asking for performance: Exploiting developer intuition to guide instrumentation with TimeTrial. In *Proc. 13th Int'l Conf. High Performance Computing and Communications*, pages 321–330, September 2011.

[119] Maurice Landry, Jean-Louis Malouin, and Muhittin Oral. Model validation in operations research. *European Journal of Operational Research*, 14(3):207–220, 1983.

[120] Stephen S. Lavenberg. A perspective on queueing models of computer performance. *Performance Evaluation*, 10(1):53–76, 1989.

[121] J.Y. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet.* Springer-Verlag, 2001.

[122] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[123] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. *Proc. IEEE*, 75(9), 1987.

[124] Hongsik Lee, Dong Nguyen, and Jongeun Lee. Optimizing stream program performance on cgra-based systems. In *Proceedings of the 52nd Annual Design Automation Conference*, page 110. ACM, 2015.

[125] Charles E Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[126] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D. Chamberlain. Deadlock avoidance for streaming computations with filtering. In *ACM Symp. on Parallelism in Algorithms and Architectures*, jun 2010.

[127] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D. Chamberlain. Adding data parallelism to streaming pipelines for throughput optimization. In *Proc. of IEEE Int'l Conf. on High Performance Computing*, 2013.

[128] Peng Li, Jonathan Beard, and Jeremy Buhler. Deadlock-free buffer configuration for stream computing. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 164–169. ACM, 2015.

[129] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. *ACM SIGPLAN Notices*, 44(4):65, 2009.

[130] Weiguo Liu, B. Schmidt, G. Voss, and W. Muller-Wittig. Streaming algorithms for biological sequence alignment on GPUs. *IEEE Trans. on Parallel and Distributed Systems*, 18(9):1270–1281, Sept 2007.

[131] Scott Lloyd and Maya Gokhale. In-memory data rearrangement for irregular, data-intensive computing. *Computer*, 48(8):18–25, Aug 2015.

[132] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM Sigplan Notices*, 40(6):190–200, 2005.

[133] J. Manyika, McKinsey Global Institute, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, and A.H. Byers. *Big Data: The Next Frontier for Innovation, Competition, and Productivity.* McKinsey Global Institute, 2011.

[134] Abdelhafid Mazouz, S-A-A Touati, and Denis Barthou. Study of variations of native program execution times on multi-core architectures. In *Proc. of Int'l Conf. on Complex, Intelligent and Software Intensive Systems*, pages 919–924, 2010.

[135] James R McGraw. Data-flow computing: the VAL language. *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, 1982.

[136] Leo A Meyerovich and Ariel S Rabkin. Empirical analysis of programming language adoption. In *Proc. of ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, pages 1–18. ACM, 2013.

[137] Barton P Miller, Mark D Callaghan, Jonathan M Cargille, Jeffrey K Hollingsworth, R Bruce Irvin, Karen L Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[138] Helmuth Moltke. *Militärische werke*, volume 3. ES Mittler und sohn, 1893.

[139] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[140] Brad A Myers. Separating application code from toolkits: eliminating the spaghetti of call-backs. In *Proc. of 4th ACM Symposium on User Interface Software and Technology*, pages 211–220. ACM, 1991.

[141] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[142] Sun Developer Network. The java hotspot performance engine architecture. *Sun Microsystem*, 2007.

[143] Marcel F. Neuts. *Matrix-Geometric Solutions in Stochastic Models: An Algorithmic Approach*. John Hopkins Univ. Press, 1981.

[144] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *Distributed Shared Memory-Concepts and Systems*, pages 42–50, 1991.

[145] John Nolan. *Stable Distributions: Models for Heavy-tailed Data*. Birkhauser, 2003.

[146] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. Exploring the potential of heterogeneous von neumann/dataflow execution models. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 298–310. ACM, 2015.

[147] John D Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E Lefohn, and Timothy J Purcell. A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, volume 26, pages 80–113. Wiley Online Library, 2007.

[148] Shobana Padmanabhan, Yixin Chen, and Roger D. Chamberlain. Optimal design-space exploration of streaming applications. In *Proc. IEEE Int'l Conf. Application-specific Systems, Architectures and Processors*, pages 227–230, September 2011.

[149] Shobana Padmanabhan, Yixin Chen, and Roger D. Chamberlain. Convexity in non-convex optimizations of streaming applications. In *Proc. of 18th IEEE Int'l Conf. on Parallel and Distributed Systems*, pages 668–675, December 2012.

[150] Shobana Padmanabhan, Yixin Chen, and Roger D. Chamberlain. Unchaining in design-space optimization of streaming applications. In *Proc. of Workshop on Data-Flow Execution Models for Extreme Scale Computing*, September 2013.

[151] Gregory Michael Papadopoulos. *Implementation of a general purpose dataflow multiprocessor*. PhD thesis, Massachusetts Institute of Technology, 1988.

[152] Shyam Parekh and Jean Walrand. A quick simulation method for excessive backlogs in networks of queues. *Automatic Control, IEEE Transactions on*, 34(1):54–66, 1989.

[153] Philippe Pébay. Formulas for robust, one-pass parallel computation of covariances and arbitrary-order statistical moments. *Sandia Report SAND2008-6212, Sandia National Laboratories*, 2008.

[154] Oliver Pell and Oskar Mencer. Surviving the end of frequency scaling with reconfigurable dataflow computing. *ACM SIGARCH Computer Architecture News*, 39(4):60–65, 2011.

[155] F Perry. Sneak peek: Google cloud dataflow, a cloud-native data processing service. *URL: http://googlecloudplatform. blogspot. com/2014/06/sneak-peek-googlecloud-dataflow-a-cloud-native-dataprocessing-service. html*, 2014.

[156] B. Pourbabai, JPC Blanc, and FA Van der Duyn Schouten. Optimizing flow rates in a queueing network with side constraints. *European Journal of Operational Research*, 88(3):586–591, 1996.

[157] I PRESENT. Cramming more components onto integrated circuits. *Readings in computer architecture*, page 56, 2000.

[158] Peter Puschner and Alan Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2):115–128, 2000.

[159] RaftLib. http://www.raftlib.io. Accessed August 2015.

[160] A. Ralston, E.D. Reilly, and D. Hemmendinger. *Encyclopedia of Computer Science.* John Wiley and Sons Ltd., Chichester, UK, 2003.

[161] Colin R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* John Wiley & Sons, Inc., New York, NY, USA, 1993.

[162] James Reinders. *Intel Threading Building Blocks: Outfitting C++ For Multi-core Processor Parallelism.* O'Reilly Media, Inc., 2007.

[163] Secretary Rumsfeld Press Conference at NATO Headquarters, Brussels, Belgium. http://www.defense.gov/Transcripts/Transcript.aspx?TranscriptID=3490, 2002.

[164] Samza. http://samza.incubator.apache.org. Accessed November 2014.

[165] Daniel Sanchez and Christos Kozyrakis. Zsim: fast and accurate microarchitectural simulation of thousand-core systems. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 475–486. ACM, 2013.

[166] Laura A Sanchis. Multiple-way network partitioning. *Computers, IEEE Transactions on*, 38(1):62–81, 1989.

[167] Robert G Sargent. Validation of simulation models. In *Proceedings of the 11th conference on Winter simulation-Volume 2*, pages 497–503. IEEE Press, 1979.

[168] Robert G Sargent. Verification and validation of simulation models. In *Proceedings of the 37th conference on Winter simulation*, pages 130–143. winter simulation conference, 2005.

[169] Robert R Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, 1997.

[170] Graham Schelle, Jamison Collins, Ethan Schuchman, Perrry Wang, Xiang Zou, Gautham Chinya, Ralf Plate, Thorsten Mattner, Franz Olbrich, Per Hammarlund, et al. Intel nehalem processor core made fpga synthesizable. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 3–12. ACM, 2010.

[171] Bernhard Schölkopf and Alexander J Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* MIT Press, Cambridge, MA, 2002.

[172] P.J. Schweitzer. Maximum throughput in finite-capacity open queueing networks with product-form solutions. *Management Science*, pages 217–223, 1977.

[173] Mehul A Shah, Joseph M Hellerstein, and Eric Brewer. Highly available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 827–838. ACM, 2004.

[174] J. Shalf, D. Quinlan, and C. Janssen. Rethinking hardware-software codesign for exascale systems. *Computer*, 44(11):22–30, 2011.

[175] Sameer S Shende and Allen D Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[176] David Simchi-Levi. *Designing and managing the supply chain.* Mcgraw-Hill College, 2005.

[177] J MacGregor Smith. Properties and performance modelling of finite buffer m/g/1/k networks. *Computers & Operations Research*, 38(4):740–754, 2011.

[178] Stack Exchange Data Dump. http://goo.gl/PBgYvwz. Accessed November 2014.

[179] William J Stewart. *Probability, Markov Chains, Queues, and Simulation: The Mathematical Basis of Performance Modeling.* Princeton University Press, Princeton, NJ, 2009.

[180] H.S. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. on Software Engineering*, 3(1):85–93, 1977.

[181] Storm: Distributed and fault-tolerant realtime computation. https://storm.apache.org. Accessed November 2014.

[182] Harold H Szu and Ralph L Hartley. Nonconvex optimization by fast simulated annealing. *Proceedings of the IEEE*, 75(11):1538–1540, 1987.

[183] A.S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 2007.

[184] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[185] David MJ Tax and Robert PW Duin. Support vector data description. *Machine Learning*, 54(1):45–66, 2004.

[186] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Proc. of 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 365–376. ACM, 2010.

[187] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In R. Horspool, editor, *Proc. of Int'l Conf. on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 49–84. Springer International Publishing, 2002.

[188] TIOBE Programming Community index. http://goo.gl/7oPQ94. Accessed Ocbober 2014.

[189] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[190] Eric J. Tyson, James Buckley, Mark A. Franklin, and Roger D. Chamberlain. Acceleration of atmospheric Cherenkov telescope signal processing to real-time speed with the Auto-Pipe design system. *Nuclear Inst. and Methods in Physics Research A*, 585(2):474–479, October 2008.

[191] Vladimir Naumovich Vapnik and Vlamimir Vapnik. *Statistical Learning Theory*, volume 2. John Wiley & Sons, New York, NY, 1998.

[192] John Villasenor and William H Mangione-Smith. Configurable computing. *Scientific American*, 276(6):54–9, 1997.

[193] Francois-Marie Arouet Voltaire. Candide. 1759. *Trans, by H. Morley. London: George Routledge*, 1991.

[194] Richard Vuduc, James W Demmel, and Jeff Bilmes. Statistical models for automatic performance tuning. In *Computational ScienceICCS 2001*, pages 117–126. Springer, 2001.

[195] BP Welford. Note on a method for calculating corrected sums of squares and products. *Technometrics*, 4(3):419–420, 1962.

[196] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1):3–35, 2001.

[197] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[198] Joseph G. Wingbermuehle, Roger D. Chamberlain, and Ron K. Cytron. ScalaPipe: A streaming application generator. In *Proc. Symp. on Application Accelerators in High-Performance Computing*, July 2012.

[199] Ting-Fan Wu, Chih-Jen Lin, and Ruby C Weng. Probability estimates for multi-class classification by pairwise coupling. *The Journal of Machine Learning Research*, 5:975–1005, 2004.

[200] Wm A Wulf and Sally A McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[201] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, 1998.

[202] Mehmet Mutlu Yenisey. A flow-network approach for equilibrium of material requirements planning. *International journal of production economics*, 102(2):317–332, 2006.

[203] Zhe Zhang, Yu Gu, Fan Ye, Hao Yang, Minkyong Kim, Hui Lei, and Zhen Liu. A hybrid approach to high availability in stream processing systems. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 138–148. IEEE, 2010.

Vita

Jonathan Curtis Beard

Degrees	Ph.D. Computer Science Washington University in St. Louis August 2015
	M.S. Bioinformatics The Johns Hopkins University May 2010
	B.S. Biological Sciences Louisiana State University December 2005
	B.A. International Studies, Central Asian Focus Louisiana State University December 2005
Professional Societies	Association for Computing Machinery Institute of Electrical and Electronics Engineers Computer Society Upsilon Pi Epsilon
Publications	Joseph M. Lancaster, Joseph G. Wingbermuehle, **Jonathan C. Beard**, and Roger D. Chamberlain. Crossing boundaries in TimeTrial: Monitoring Communications Across Architecturally Diverse Computing Platforms. In the Proceedings of the 9th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing, pages 280-287, October 2011
	Jonathan C. Beard and Roger D. Chamberlain. Use of Simple Analytic Performance Models of Streaming Data Applications Deployed on Diverse Architectures. In the Proceedings of the 2013 International Symposium on the Performance Analysis of Systems and Software, pages 138-139, April 2013

Jonathan C. Beard and Roger D. Chamberlain. Analysis of a Simple Approach to Modeling Performance for Streaming Data Applications. In Proceedings of the 21st IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, pages 345-349, August 2013

Jonathan C. Beard and Roger D. Chamberlain. Use of a Levy Distribution for Modeling Best Case Execution Time Variation. In Computer Performance Engineering, Volume 8721 of Lecture Notes in Computer Science, pages 74-88. A. Horvàth and K. Wolter, editors, Springer International Publishing, September 2014

Jonathan C. Beard, Cooper Epstein, and Roger D. Chamberlain. Automated Reliability Classification of Queueing Models for Streaming Computation Using Support Vector Machines. In the Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, ICPE 15, New York, NY, USA, pages 325-328, January 2015. ACM

Jonathan C. Beard, Peng Li, and Roger D. Chamberlain. Raftlib: A C++ Template Library for High Performance Stream Parallel Processing. In the Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'15. New York, NY, USA, pages 96-105, February 2015. ACM

Peng Li, **Jonathan Beard**, and Jeremy Buhler. Deadlock-free Buffer Configuration for Stream Computing. In the Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'15. New York, NY, USA, pages 164-169, February 2015, ACM

Jonathan C. Beard and Roger D. Chamberlain. Runtime Approximation of Non-blocking Service Rates for Streaming Systems. In the Proceedings of the 17th IEEE International Conference on High Performance and Communications, pages 792-797, August 2015

Jonathan C. Beard, Cooper Epstein, and Roger D. Chamberlain. Online Automated Reliability Classification of Queueing Models for

Streaming Processing using Support Vector Machines. In the Proceedings of the 21st International European Conference on Parallel and Distributed Computing, Euro-Par'15, Volume 9233 of Lecture Notes in Computer Science, pages 82-93. Jesper Larsson Träff, Sascha Hunold, and Francesco Versaci, editors. Springer Berlin Heidelberg, August 2015

August 2015