Software Approaches to Manage Resource Tradeoffs of Power and Energy Constrained Applications

by

Ramy Medhat

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2017

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Miroslav Pajic Assistant Professor, Duke University Department of Electrical and Computer Engineering
Supervisors:	Sebastian Fischmeister Associate Professor, University of Waterloo Department of Electrical and Computer Engineering
	Borzoo Bonakdarpour Assistant Professor, McMaster University Department of Computing and Software
Internal Member:	Catherine Gebotys Professor, University of Waterloo Department of Electrical and Computer Engineering
Internal Member:	Rodolfo Pellizzoni Associate Professor, University of Waterloo Department of Electrical and Computer Engineering
Internal-External Member:	Samer El-Kiswany Assistant Professor, University of Waterloo David R. Cheriton School of Computer Science

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

In what follows is a list of publications which I have co-authored and used their content and ideas in this dissertation. For each publication, I present a list of my contributions.

The use of the content, from the listed publications, in this dissertation has been approved by all co-authors.

1. Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. Power-efficient Multiple Producer-Consumer. In *Proceedings of the IEEE 28th International Symposium on Parallel & Distributed Processing (IPDPS)*, Phoenix, USA, 2014 [96].

    - Designed and executed the experimental study.
    - Designed the problem formalization.
    - Designed the algorithm.
    - Implemented the algorithm.
    - Designed and executed the experiments.
    - Wrote a significant portion of the article.

2. Ramy Medhat, Deepak Kumar, Borzoo Bonakdarpour, and Sebastian Fischmeister. Sacrificing a Little Space Can Significantly Improve Monitoring of Time-sensitive Cyber-physical Systems. In *Proceedings of the ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS)*, Berlin, Germany, 2014 [99].

    - Designed the problem formalization.
    - Designed the algorithm.
    - Implemented the algorithm.
    - Designed and co-executed the experiments.
    - Wrote a significant portion of the article.

3. Ramy Medhat, Borzoo Bonakdarpour, Deepak Kumar, and Sebastian Fischmeister. Runtime Monitoring of Cyber-Physical Systems Under Timing and Memory Constraints. In *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, issue 4, pp. 79, 2015 [97].

    - Designed the problem formalization.

- Designed the algorithm.
- Implemented the algorithm.
- Designed and co-executed the experiments.
- Wrote a significant portion of the article.

4. Ramy Medhat, Mike Lam, Barry Rountree, Borzoo Bonakdarpour, Sebastian Fischmeister. Managing the Performance/Error Tradeoff of Floating-point Intensive Applications. In *International Conference on Embedded Software (EMSOFT)*, Seoul, South Korea, 2017 [98].

   - Designed the tool.
   - Implemented the tool.
   - Designed and executed the experiments.
   - Wrote a significant portion of the article.

5. Ramy Medhat, Shelby Funk, and Barry Rountree. Scalable Performance Bounding under Multiple Constrained Renewable Resources. In *5th International Workshop on Energy Efficient Supercomputing (E2SC)*, Colorado, USA, 2017 [100].

   - Designed the multi-resource model.
   - Implemented the model.
   - Designed and executed the experiments.
   - Wrote a significant portion of the article.

6. Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. Energy-efficient Multiple Producer-Consumer. Submitted to *IEEE Transactions on Parallel and Distributed Systems (TPDS)*.

   - Designed and executed the experimental study.
   - Designed the problem formalization.
   - Designed the algorithm.
   - Implemented the algorithm.
   - Designed and executed the experiments.
   - Wrote a significant portion of the article.

Abstract

Power and energy efficiency have become an increasingly important design metric for a wide spectrum of computing devices. Battery efficiency, which requires a mixture of energy and power efficiency, is exceedingly important especially since there have been no groundbreaking advances in battery capacity recently. The need for energy and power efficiency stretches from small embedded devices to portable computers to large scale data centers. The projected future of computing demand, referred to as exascale computing, demands that researchers find ways to perform exaFLOPs of computation at a power bound much lower than would be required by simply scaling today's standards.

There is a large body of work on power and energy efficiency for a wide range of applications and at different levels of abstraction. However, there is a lack of work studying the nuances of different tradeoffs that arise when operating under a power/energy budget. Moreover, there is no work on constructing a generalized model of applications running under power/energy constraints, which allows the designer to optimize their resource consumption, be it power, energy, time, bandwidth, or space. There is need for an efficient model that can provide bounds on the optimality of an application's resource consumption, becoming a basis against which online resource management heuristics can be measured.

In this thesis, we tackle the problem of managing resource tradeoffs of power/energy constrained applications. We begin by studying the nuances of power/energy tradeoffs with the response time and throughput of stream processing applications. We then study the power performance tradeoff of batch processing applications to identify a power configuration that maximizes performance under a power bound. Next, we study the tradeoff of power/energy with network bandwidth and precision. Finally, we study how to combine tradeoffs into a generalized model of applications running under resource constraints.

The work in this thesis presents detailed studies of the power/energy tradeoff with response time, throughput, performance, network bandwidth, and precision of stream and batch processing applications. To that end, we present an adaptive algorithm that manages stream processing tradeoffs of response time and throughput at the CPU level. At the task-level, we present an online heuristic that adaptively distributes bounded power in a cluster to improve performance, as well as an offline approach to optimally bound performance. We demonstrate how power can be used to reduce bandwidth bottlenecks and extend our offline approach to model bandwidth tradeoffs. Moreover, we present a tool that identifies parts of a program that can be downgraded in precision with minimal impact on accuracy, and maximal impact on energy consumption. Finally, we combine all the above tradeoffs into a flexible model that is efficient to solve and allows for bounding and/or optimizing the consumption of different resources.

Acknowledgements

I would like to thank my supervisors, Professor Sebastian Fischmeister and Professor Borzoo Bonakdarpour for their continuous support and guidance. They have taught me many things, most important of which is how to think about a problem thoroughly and how to improve the problem statement during several rounds of cross-examination. I am grateful for their effort during my studies and could not have asked for more supportive and encouraging advisors.

I would like to thank Professor Barry Rountree for his tremendous support, providing me with space to explore my ideas and collaborate with amazing researchers. I would like to thank Professor Mike Lam for the fruitful collaboration and his consistent availability for brainstorming and feedback. I would also like to thank Professor Shelby Funk for her support, encouragement, and time to have many whiteboard brainstorming sessions. I would also like to thank my committee members: Professor Miroslac Pajic, Professor Rodolfo Pellizzoni, Professor Catherine Gebotys, and Professor Samer Al-Kiswany for taking the time and effort to participate in my examination committee and provide me with valuable feedback.

I would also like to thank my colleagues in the Real-Time Embedded Systems group: Hany Kashif, Samaneh Navabpour, Deepak Kumar, Wallace Wu, Jean-Christophe Petkovic and David Shin.

I cannot express enough gratitude to my parents, Manal and Medhat, for their continuous support every step of the way. They have pushed me forward at every juncture of my life, and without them I would not be writing a PhD thesis acknowledgement. I would like to thank my sister Yara for always believing in her brother and pushing me to strive for improvement.

Finally, I would like to thank my wife, Menna. Thank you for your support, encouragement, and belief. I would also like to thank our son, Mourad, for making my life brighter.

Dedication

To my parents, Manal and Medhat.
To my sister, Yara.
To my wife, Menna.
To my son, Mourad.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Importance of Power and Energy Efficiency

Designing low-power computing system architectures has been an active area of research in the recent years, partly due to the increasing cost of energy as well as the high demands on producing and manufacturing environment-friendly devices. While the former is an explicit financial cost-benefit issue, the latter is attributed to green computing. However, with the recent explosion in mobile computing and incredible popularity of smart-phones and tablet computers, energy efficiency in software products has become a prime concern in application design and development and in fact as important as energy-optimal hardware chips.

Energy efficiency plays an important role in large-scale data centers. This can be easily observed in the growing size of data centers that serve internet-scale applications. Currently, such data centers consume 1.3% of the global energy supply, at a cost of $4.5 billion. This percentage is expected to rise to 8% by 2020 [79]. In fact, power and cooling are the largest cost of a data center. For example, a facility consisting of 30,000 square feet and consuming 10MW, for instance, requires an accompanying cooling system that costs from $2-$5 million [106], and the yearly cost of running this cooling infrastructure can reach up to $4-$8 million [138]. The rise in energy costs has become so prevalent that the cost of electricity for fours years in a data center is approaching the cost of setting up a new data center [12].

Power efficiency does not always imply energy efficiency. Power efficiency is sometimes the objective regardless of energy efficiency. With the increasing urgency to reach exascale

computing, governments must inevitably impose a constraint on the power consumption of data centers [15]. Power efficiency constraints also apply to embedded systems. Certain components are constrained to draw power allowed by the battery installed in the system, thus for instance forcing designers to reduce the clock speed of processors. A lower peak power also extends the lifetime of the battery [135] which is a target for consumer electronics. Thus, system designers often target energy efficiency as well as power constraints.

1.2 Resource Tradeoffs and the Status Quo

Multiple resources affect and are affected by power and energy consumption. Time is a resource that is often associated with power consumption. Higher power consumption leads to lower execution time or lower response time. Lower precision of floating point arithmetic reduces energy consumption through faster performance at the cost of accuracy [39]. Higher usage of network bandwidth implies higher power consumption [25]. Moreover, there are subtle tradeoffs that allow power to control resource utilization. For instance, in some cases, parallel tasks can be staggered using power bounds to reduce network bottlenecks, preventing these tasks from sending data at the same time, causing a bottleneck [100].

The tradeoffs between various resources and power and energy are non-trivial to manage, especially in dynamic heterogeneous systems. Identifying the optimal configuration of resources to meet a system's constraints and objectives is a difficult problem. This problem becomes more complicated when we consider the efficiency of the proposed solution. An energy management system running on a battery powered device should be conscious of its own energy footprint, refraining from using costly CPU-intensive algorithms. Moreover, the energy management system should respond to changes in the environment under timing constraints, which is often the case in embedded systems. On the other side of the spectrum, Big data and HPC applications could deploy tens of thousands of tasks. The different characteristics of tasks, nodes, and network infrastructure grossly complicate the tradeoff management of such massively parallel systems.

There is a large body of work on managing the performance energy tradeoff. The work ranges from low-level control at the CPU level [37, 70, 101, 196] to high level control at the task level across multiple nodes [62, 92, 141, 142, 169]. There has been extensive work studying the bandwidth energy tradeoff from the perspective of network design [25, 89, 140]. Moreover, various articles demonstrated the impact of precision on performance as well as energy [39, 41, 116, 117].

Most of the literature tackles the intricacies of specific tradeoffs and present sophisticated optimizations that impact various application areas. However, the current literature

lacks the following in hierarchical order:

- The current literature lacks specific in depth studies on the energy/power tradeoff against specific aspects of resource use. For instance, the tradeoff between energy and response time and throughput of low-level concurrency primitives, as opposed to the extensively studied energy / performance tradeoff.
- The industry's focus on green computing allowed for advanced controls on the power and energy consumption of computing systems in the form of hardware power bounding and drivers for DVFS (Dynamic Voltage and Frequency Scaling) and DPMS (Dynamic Power Management System). Such controls are not robustly available for other resources such as bandwidth and precision. The current literature lacks studies on how to use such power controls to indirectly manage other resources.
- Finally, the current literature lacks work on a generalized model of systems with respect to energy / power and all resources that affect or are affected by energy / power consumption.

The above gaps in the literature culminate to the lack of a generalized system resource model that is centered around power and energy consumption. A model that allows capturing nuanced tradeoffs allows the designer to better optimize performance. By exploring resources that can be controlled using power and energy, a designer can have more control over a wider array of system resources. A model that accounts for a wide array of resources and their interactions allows the designer to optimize systems to their full extent while maintaining granular control over specific resource tradeoffs.

1.3 Thesis Statement

In this thesis, we validate the following hypothesis: as opposed to hardware based approaches, exclusively software-based approaches can provide a control mechanism with which engineers can manage the physical resource tradeoffs of their applications running in a power and/or energy constrained setting.

1.4 Research Problem

Thus, the high level research question we pose here is how multiple resource tradeoffs can be managed in an environment where power and energy are constrained. More concretely,

can we construct a model that captures multiple resource tradeoffs with power and energy and allows for constraining and/or optimizing the consumption of different resources.

In this thesis we tackle the above research problem by setting the following high level objectives:

- Study the tradeoff of power / energy versus performance at different levels of abstraction in order to construct a generalized model.
- Study how power / energy constraints can affect the consumption of resources that are difficult to control explicitly.
- Construct a model that captures the resource tradeoffs involving power and/or energy.

1.5 Overview of the Proposed Approach

The thesis begins by studying the nuances of the tradeoff between energy and performance at the level of DVFS and DPMS for stream processing applications. We then move onto a higher task-level abstraction and study the tradeoff of power and performance for batch processing applications. Next, we study how bandwidth can be treated as a resource that can be indirectly managed by power controls. Next, we study how we can manage the tradeoff between precision and power. Finally, we combine the tradeoffs we studied above with other well-established tradeoffs to construct a flexible general model that allows for optimizing and/or constraining resource consumption.

1.5.1 Energy versus Response Time

We begin in Chapter 2 by studying the intricacies of the energy / performance tradeoff of a general concurrency problem: namely the producer consumer problem. The producer consumer problem models a wide range of systems, from sensor processing to web servers to parallel stream processing in Big Data applications. Studying the energy profile of different implementations of this problem gives us insight into the impact of different synchronization primitives on energy consumption with respect to CPU core wakeups and frequency residency. Then, we develop an algorithm to manage the tradeoff between energy consumption and consumer response time. Furthermore, we extend the algorithm to manage the tradeoff between energy and memory footprint and demonstrate that intelligently timed memory allocations can significantly reduce energy consumption.

1.5.2 Power versus Performance

In Chapter 3, we study the power / performance tradeoff at a higher level of abstraction, working towards our end goal of a general model. To that end, we study the collective power consumption of a cluster of machines performing batch computation. While it is well investigated that more power almost always means better performance, we attempt to design an algorithm to maximize performance under a cluster power bound. We refer to this as the power distribution problem (how to distribute power optimally across the cluster). The problem becomes complicated when there is diversity in both the task load and the node power profiles. First, we introduce an ILP to minimize the makespan of a set of tasks with dependencies under a power bound. Next, we design an online heuristic to reduce the makespan by dynamically and strategically transferring power from one node to another to help long running tasks finish sooner. We demonstrate the effectiveness of the ILP and the heuristic on a well known set of parallel processing benchmarks.

1.5.3 Power, Energy and Bandwidth Tradeoffs

In Chapter 4, we extend the work in Chapter 3 in two ways. First, we study the relationship between power and bandwidth. More concretely, we study how power can be used to create artificial staggering in order to alleviate bandwidth bottlenecks. We demonstrate that millisecond staggering in computation tasks can significantly reduce total communication time, even in over-provisioned network infrastructures. Our results indicate that in stressed networks, staggering can be a significant improvement to execution time. Second, we develop a linear programming model that performs power distribution as well as bandwidth and energy distribution. The linear programming model is significantly more scalable than the ILP in Chapter 3. We demonstrate how the LP can scale to solve larger problems where there are tens of thousands of variables.

1.5.4 Power, Energy and Precision

In Chapter 5, we study the final tradeoff in the thesis: power versus precision. Prior work has demonstrated that programs that use single-precision floating point execute faster than their double-precision counterparts, and thus consume less energy. However, there is a loss in accuracy due to rounding error. We explore whether there is a tradeoff where only part of the program uses single-precision, and this part dictates the energy savings and the loss in accuracy. We refer to control points within the tradeoff as precision levels. Furthermore, we develop a tool that analyses a program that uses double-precision floating point,

and identifies functions in the program that are most recommended to be downgraded to single-precision. A highly recommended function is one that saves a significant amount of energy while having minimal impact on accuracy. We demonstrate how our tool allows the developer to manage the energy precision tradeoff by applying the tool on a well-known computer vision application. We demonstrate that the developer can select multiple control points where accuracy is traded for energy and performance. With our tool, we identify one function in the program that can reduce energy consumption by 16% with zero impact on accuracy in our test data set.

1.5.5 Generalized Model

By Chapter 5 we have already studied the tradeoff of power and/or energy with response time, throughput, execution time, bandwidth, and precision. In Chapter 6, we treat all the aforementioned factors as resources, whether renewable (e.g. power, bandwidth, throughput) or non-renewable (energy, time, accuracy). Using this abstraction, we can construct a generalized model that captures all tradeoffs of modelled resources. To that end, we modify the linear program in Chapter 4 to model a network of producers and consumers extending the abstraction of Chapter 2 where all the tradeoffs of the previous chapters are captured. In this model, a node in the network can be both a consumer of data received on its incoming edges and a producer of data on its outgoing edges. We abstract the precision of a node as a quality level at which the node performs its computation. This abstraction captures precision levels as well as algorithm alternatives which can consume varying amounts of resources. Our model allows the system designer to optimize for a specific resource while bounding any / all of the other resources, and is extensible to model more resources that have not been studied in this thesis.

1.6 Organization

The thesis is organized as follows: Chapter 2 presents the tradeoff of energy versus response time. Chapter 3 presents the tradeoff of power versus performance. Chapter 4 presents the tradeoff of power and bandwidth. Chapter 5 presents the tradeoff of energy and precision. Chapter 6 presents the generalized model that captures all tradeoffs. Chapter 7 presents the literature review, and finally Chapter 8 concludes the thesis.

Chapter 2

Energy versus Response Time

The majority of classic concurrency control algorithms were designed in an era when energy efficiency was not an important dimension in algorithm design. Concurrency control algorithms are applied to solve a wide range of problems from kernel-level primitives in operating systems to networking devices and web services. These primitives and services are constantly and heavily invoked in any computing system and by a larger scale in networking devices and data centers. Thus, even a small change in their energy spectrum can make a huge impact on overall energy consumption for long periods of time.

This chapter focuses on the classic *producer-consumer* problem, which models a wide range of stream processing applications. First, we study the energy profile of a set of existing producer-consumer algorithms. In particular, we present evidence that although the functional goal of these algorithms are the same, these implementations behave drastically differently with respect to energy consumption. Then, we present a dynamic algorithm for the multiple producer-consumer problem, where consumers in a multicore system use learning mechanisms to predict the rate of production, and effectively utilize this prediction to attempt to *latch onto* previously scheduled CPU wake-ups. Such group latching increases the idle time between consumer activations resulting in more CPU idle time and, hence, lower average CPU frequency, which in turn reduces energy consumption. We enable consumers to dynamically reserve more pre-allocated memory in cases where the production rate is too high. Consumers may compete for the extra space and dynamically release it when it is no longer needed. Our experiments show that our algorithm provides a 38% decrease in energy consumption versus a mainstream semaphore-based producer-consumer implementation when running 10 parallel consumers. We validate our algorithm with a set of thorough experiments on varying parameters of scalability. Finally, we present our recommendations on when our algorithm is most beneficial.

2.1 Introduction

Classic algorithms in computer science are heavily used in virtually any computing system ranging from web services and networking devices to device drivers and operating systems kernels. However, these algorithms were designed in an era when energy efficiency was not an important dimension in algorithm design. For example, Dijkstra's shortest path algorithm fails in the context of energy-optimal routing problems, as simply evaluating edge costs as energy values does not work [145]. Thus, we argue that many of such classic algorithms need to be re-visited and re-designed, so that on top their functional requirements, energy constraints are treated as a first-class objective as well. Some of these algorithms are applied in such a high capacity that even small improvements in their energy consumption behavior may have a huge impact in the energy profile of large-scale systems and mobile devices in long periods of time.

Producer-consumer is a classic problem in concurrent computing, where two processes, the *producer* and the *consumer* share a common bounded-size memory buffer as a queue. The multiple producer-consumer problem extends this setup by having multiple producers and consumers using the same buffer. In this chapter, we study the more general case of a set of producer-consumer *islands* where each island has its own buffer, and its own producers and consumers. We abstract each island as a single producer-consumer pair associated with a separate buffer. This abstraction helps simplify our online analysis and reduction of energy consumption without loss of generality. In that setup, we focus on the interaction between multiple producer-consumer pairs, and in turn their combined energy consumption. The multiple producer-consumer pairs problem applies to a multitude of real-world scenarios in many systems around us, specifically stream processing applications. Examples include:

- *Operating systems primitives.* Such primitives provide developers with high-level system calls to read and consume data received from I/O devices, e.g., in device drivers. In this setting, each device is a producer, providing data to its respective consumer, i.e. the user application.
- *Web servers.* HTTP requests produced by web browsers are stored in separate buffers per web application, that are consumed and processed by threads in each application's thread pool.
- *Runtime monitoring.* In runtime monitoring, different events produced by the environment or internal system processes are consumed and processed by multiple runtime monitors with separate buffers.

- *Networking.* In most networking devices (e.g., routers), data packets received from the network need to be removed and processed from internal buffers of the device and routed to different destinations. Buffers separate subnets, Virtual LANs, or QoS levels.

In this chapter, we propose a novel energy-efficient algorithm for the multiple producer-consumer pair problem for multicore systems, where each consumer is associated with one and only one producer. To the best of our knowledge, this is the first instance of such an algorithm. To better understand the vital contributing factors to the energy consumption behavior of the problem, we first conducted a study to analyze the energy profile of existing popular implementations of the producer-consumer problem. The implementations in our analysis consist of a yield-based algorithm, two algorithms based on synchronization data structures (i.e., semaphores and mutexes), and two algorithms that employ batch processing. We observed that these implementations behave drastically differently with respect to energy consumption. While the yield algorithm is the worst in energy efficiency due to high CPU utilization, the algorithms, where the consumer processes data items in batches are the most energy-efficient due to a lower average CPU frequency. In particular, batch processing results in 75% reduction in energy compared to yield and 32% compared to the semaphore-based implementations. This is validated by a strong positive correlation between average CPU frequency and energy consumption. Such a dramatic shift in energy profile clearly motivates the need for designing an energy-aware solution for the producer-consumer problem.

Roughly speaking, our proposed algorithm exploits bounded-time dynamic batch processing. It interprets time as a track with periodic slots. To increase the idle time between wakeups, it dynamically constitutes track slots, so consumers can latch on and exploit a CPU wakeup in groups. Given a set of cores, since each core may host a set of consumers, a *core manager* component targets aligning consumers to the slots in that core's track. The core manager is responsible for managing the slot allocations on the track of its respective core. Consumers are designed so that they can dynamically predict production rate of data items to compute and request appropriate latching time. Furthermore, consumers may lend each other buffer space, so that a consumer dealing with a producer with high production rate can continue latching on other consumers and not cause new wakeups.

Our approach can well-adapt to systems that already employ dynamic voltage and frequency scaling (DVFS). In particular, our design

1. can work with state-of-the-art DVFS schemes by leveraging their energy savings which focus on optimum choices of CPU frequency. Dynamic batch processing bene-

fits from such frequency manipulations such that a batch consumption job consumes minimum energy.

2. reduces energy savings further by reducing the number of CPU wakeups. Thus, a lower number of wakeups coupled with efficient CPU frequency manipulations results in an improved energy consumption.

To demonstrate the effectiveness of our approach, we conduct our experiments over a wide range of varying parameters and explore the strengths and weaknesses of using dynamic batch processing. This allows us to provide a recommendation as to what systems are better suited to employ dynamic batch processing. We argue that our dynamic batch processing approach is particularly beneficial in two application areas:

1. According to a Google study [11], web servers are rarely completely idle and seldom operate near their maximum utilization, instead operating most of the time at between 10 and 50 percent of their maximum utilization levels. Moreover, the CPU contributes to more than 50% of Google server power consumption. In such servers, our approach results in longer CPU idle periods, which saves a great deal of energy. This energy saving comes at a cost in terms of response time, which is significantly reduced when using batch processing. However, our approach provides controls with which a user can tune the energy saving versus response time.

2. Data-processing-intensive applications are becoming more widespread every day. In such applications, throughput is more important than individual item response time. Our experiments show that our approach results in higher throughput while saving a considerable amount of energy. This makes dynamic batch processing significantly beneficial for applications that can fit in a producer-consumer model.

We validate our claims by conducting thorough experiments using a synthetic queueing theory based arrival pattern, as well as a data set of a web server incoming HTTP requests log [5]. Our results show that our algorithm can lower energy consumption by 51% compared to a semaphore-based implementation of producer-consumer when running 10 parallel consumers. In comparison to a simple batch processing approach, our algorithm provides up to a 15% improvement in energy consumption. We experiment with varying buffer sizes, number of parallel consumers, as well as data item inter-arrival period. The objective of these experiments is to study the scalability of the proposed algorithm and identify its strengths and weaknesses. To that end, we present our results and recommendations on when it is most appropriate to use our dynamic batch processing approach. Our

experimental results demonstrate the tradeoffs between response time and energy savings. The results suggest that the selection of consumer implementation should depend on the required maximum response time and the expected mean inter-arrival period.

Organization. The rest of the chapter is organized as follows. In Section 2.2, we describe the background concepts on CPU power states. Section 2.3 presents our findings on energy profile of various implementations of the producer-consumer problem. We formally state the energy optimization objective for the multiple producer-consumer problem in Section 2.4. Our energy-efficient solution to multiple producer-consumer is described in Section 2.5, while Section 2.6 analyzes the results of experiments. Finally, we make concluding remarks in Section 2.7.

2.2 Background

Power management technologies approach energy/power efficiency from different perspectives:

- *Static power management* (SPM) simplifies the power management problem by providing support for low-power modes at the hardware level. A system can statically transition to the low-power modes on demand. An example of this is a cell phone going into idle mode when it is locked, or a sensor periodically sleeping at a predefined period.
- *Dynamic power management* (DPM) employs dynamic techniques at run time that determine which power state the system should be in. DPM uses different techniques to infer whether or not a transition to a more efficient state is worthwhile, and, which efficient state to transition to.

In DPM, hardware with scalable power consumption is combined with management software to achieve improved efficiency. Hardware support comes in multiple flavors, e.g., Dynamic Voltage Scaling (DVS) and Dynamic Frequency Scaling (DFS). DVS scales the voltage at which the CPU operates, and thus controls its energy consumption. This is based on the basic Watt's law

$$P = V \cdot I$$

where V is the voltage, I is the current, and P is the power. DVS is becoming more prominent in disk drives, memory banks, and network cards [45]. DFS scales the frequency

at which the CPU operates, such that when a high demand occurs, the frequency is raised to meet that demand, at a higher energy cost. When the CPU utilization drops, so does the operating frequency, causing a decrease in energy/power consumption. This is because dynamic power is calculated by

$$P_d = C \cdot V^2 \cdot f$$

where C is the capacitance switched per cycle, V is the voltage, and f is the current CPU frequency. DVS and DFS are often combined into DVFS, where both techniques are used to scale CPU power consumption. CPUs generally support a predefined set of frequency/voltage combinations performance states, known as P-states. These states define the performance of the CPU in terms of power and throughput.

A relatively different approach to power saving is utilizing CPU C-states. C-states are modes at which the CPU operates, differing mainly in their power consumption. This is achieved by turning off parts of the CPU that are needlessly consuming energy. This may include gradually turning off internal CPU clocks, cache, the bus interface, and even decreasing the CPU voltage (DVS). C-states generally start at C0 which indicates the CPU is fully active, and gradually increases the number (C1, C2, ...) until the idle state or in some cases the hibernate state.

Race-to-Idle is a well-known energy saving concept based on the premise that it is more energy efficient to execute the task at hand faster (a higher P-state, which indicates a higher CPU frequency) and then go to idle mode (i.e., a deeper C-state, which is a deeper CPU idle state), versus running the task at a lower speed resulting in less idle time. Race-to-idle is based on the fact that idle power is significantly less than active power even at a low frequency. Furthermore, recent CPU chipsets such as the Intel Haswell are even more optimized to save a significant amount of power in idle mode. This indicates that hardware manufacturers are moving towards approaches that attempt to increase CPU residency in deeper C-states. This is especially useful in the context of *core parking*, where specific cores are put in a deep sleep state, reducing their energy consumption significantly.

Although race-to-idle is a valid approach, in the context of the producer-consumer problem - where items are arriving at a certain rate and not in bulk - race-to-idle may not be the most appropriate strategy. If the items are arriving in such a way that denies the CPU any actual idle time, voltage scaling will detect that there is a consistent load on the CPU and

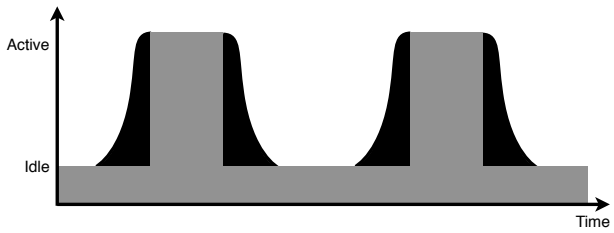


Figure 2.1: Overhead due to waking up and idling the CPU.

thus increase its frequency. Effectively, the
CPU attempts to race-to-idle but never gets a chance to go to idle, and this results in huge energy waste, due to "racing" most of the time without any chance to "rest". This is in addition to the energy wasted due to idling and waking up frequently. In other words, a certain delay must occur in order for idle mode to be advantageous. Figure 2.1 illustrates that more contiguous idle time is more efficient. Thus, a valid energy saving strategy is to increase the contiguousness of idle periods. This approach should be combined with race-to-idle to ensure that a power management strategy targets more idle time with minimum wasted power due to idle-active and active-idle transitions.

2.3 Producer-Consumer Energy Profile

In this section, we present experimental evidence showing that different implementations of a widely used concurrency control algorithm exhibit drastically different energy consumption profiles.

2.3.1 Producer-Consumer Implementations

The *producer-consumer* problem is a classic multi-process synchronization problem, where a *producer* process produces data items and places them in a memory buffer, and, a *consumer* process consumes the items from the same memory buffer. Since these processes work concurrently, they need to synchronize to prevent deadlocks and race conditions. Most of the implementations we study in this section rely on the use of a circular buffer. The advantage of a circular buffer is that reading from it and writing into it involves two different pointers, and, thus, alleviates the need to put a single counter in a critical section to avoid concurrency issues.

We study the following implementations:

- **Yield.** This implementation uses sched_yield within a spinlock to yield the CPU if the buffer is full/empty.
- **Mutexes and conditional variables (Mutex).** This implementation uses pthread mutex to ensure mutually exclusive concurrent access to a buffer. We use conditional variables to signal when data is available for the consumer and when space is available for the producer.

- **Semaphores (Sem).** This implementation uses two semaphores used for synchronizing emptiness and fullness of the buffer.
- **Batch processing (BP).** This implementation is similar to the semaphores implementation, except that the consumer waits until the buffer is full and then processes all items in the buffer in one batch.
- **Periodic batch processing (PBP).** This implementation is similar to the batch processing implementation, except that the consumer processes the batch within fixed time intervals (using the usleep() system call) instead of whenever the buffer gets filled. The period for this experiment is 1ms.

2.3.2 Experimental Settings

We study the energy consumption of the different producer-consumer implementations using two methods: PowerTop[1] and RAPL (Running Average Power Limit)[2].

PowerTop is a popular Linux tool that uses CPU performance counters to estimate the power consumption of all running processes in the system. We use PowerTop to measure the number of wakeups per second that a process causes, and the percentage of CPU usage that the process consumes. The unit for CPU usage in PowerTop is microseconds per second, meaning the number of microseconds the process spends executing every second.

RAPL is an interface that provides monitoring and controlling power consumption of specific Intel CPUs. Starting from the second generation Core i7 processor, codenamed Sandy Bridge, RAPL can be used to monitor energy consumption by reading machine specific registers (MSRs). We use the Linux RAPL driver over PAPI to measure energy consumption of each experiment. Our test machine uses an Intel Core i7 Sandy Bridge processor.

Finally, each implementation of producer-consumer is tested using two datasets. First, a synthetic dataset based on an $M/M/1/\mathbb{B}$ queue [159]. This denotes that the production and processing times are drawn from exponential distributions (M/M), there is 1 consumer (1 consumer for every producer), and the items are buffered in a buffer of size $\mathbb{B}$. Second, a real-life dataset based on web server incoming HTTP requests log [5]. We use a portion of the log that covers web requests received over 40 days. Each experiment constitutes running 5 producers/consumers in parallel until the full dataset items are consumed. In the case of

[1] https://01.org/powertop/
[2] https://01.org/blogs/tlcounts/2014/running-average-power-limit--rapl

the web server data, the log is divided equally such that each producer simulates 8 days of logs. All producers run on one core, and all consumers run on another core. Since cores can be idled separately, this allows us to demonstrate the impact of consumers synchronization on energy consumption regardless of the behavior of producers, which is not in our control. We execute 50 replicates of each experiment for statistical confidence. 95% confidence intervals are calculated for all measurements. We measure five metrics in each experiment:

- **Energy (joules).** The number of joules consumed by the system when the respective implementation is executed.
- **Wakeups/s.** The number of CPU wakeups per second due to the respective implementation.
- **Usage (ms/s).** The number of milliseconds out of every second that the CPU spends executing the respective implementation.
- **Core idle percentage.** The percentage of idle time that each CPU core spends during the respective implementation.
- **Core frequency percentage.** The percentage of time the CPU spends operating at each possible frequency while executing the respective implementation.

2.3.3 Experimental Results

Sanity Checks

We perform the following set of sanity checks to ensure our experimental setup is valid:

- We execute a test with a busy waiting multithreaded program running on two cores of the processor, and we ensure that no experiment reaches the energy consumption found in that implementation.
- We execute a test where no background processes are running except kernel tasks, and we measure energy. We ensure that the energy consumed in this experiment is less than any other experiment we run.
- We measure the statistical confidence interval to ensure that our conclusions are not based on outliers.

Energy consumption of producer-consumer implementations

Figure 2.2 shows the energy consumption of each of the five producer-consumer implementations. The energy results are highly consistent, as apparent in the small error bars. The Yield implementation consumes the most energy since the CPU is mostly active during its execution. On average, both CPU cores are idle only 1% of the time when executing Yield. Mutex and Sem have similar energy consumption profiles, using 50% of the energy consumed by Yield. BP reduces energy consumption further down to 35% of Yield, and 68% of Sem. Finally, PBP is consistently the most energy efficient implementation consuming 32% of Yield and 62% of Sem. This indicates that batch-processing-based implementations can improve upon the most widely used implementation today significantly.

Understanding root causes of energy profile

Next, we study the CPU usage and number of wakeups in each implementation (see Figure 2.3). Yield has the lowest number of wakeups which is expected since the CPU is active most of the time. However, the CPU usage of Yield is significantly less than all other implementations. This is counter-intuitive given the high energy consumption of Yield. Upon studying the percentage of time spent in each available CPU frequency, Yield spends on average 99% of the time at the highest CPU frequency of 3.4GHz. Intel SpeedStep dynamic voltage scaling recognizes that the CPU is active most of the time and upgrades its frequency to the maximum.

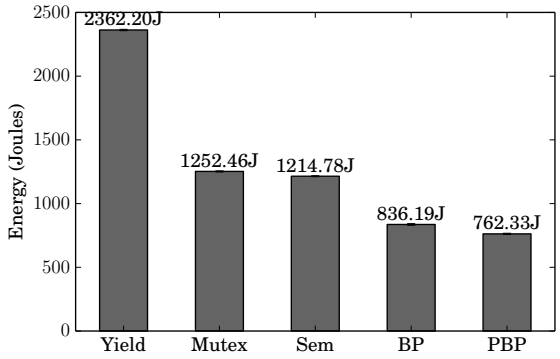


Figure 2.2: Energy consumption for all five implementations when running the $M/M/1/\mathbb{B}$ dataset.

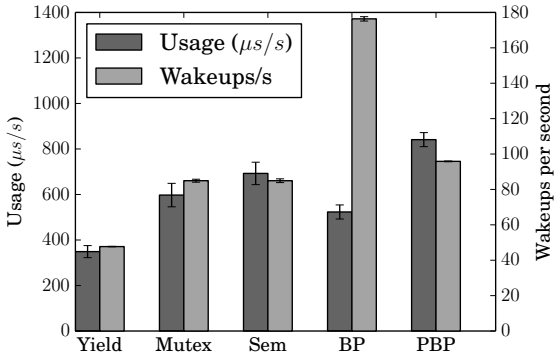


Figure 2.3: Wakeups/s versus usage $\mu s/s$ for all five implementations when running the $M/M/1/\mathbb{B}$ dataset.

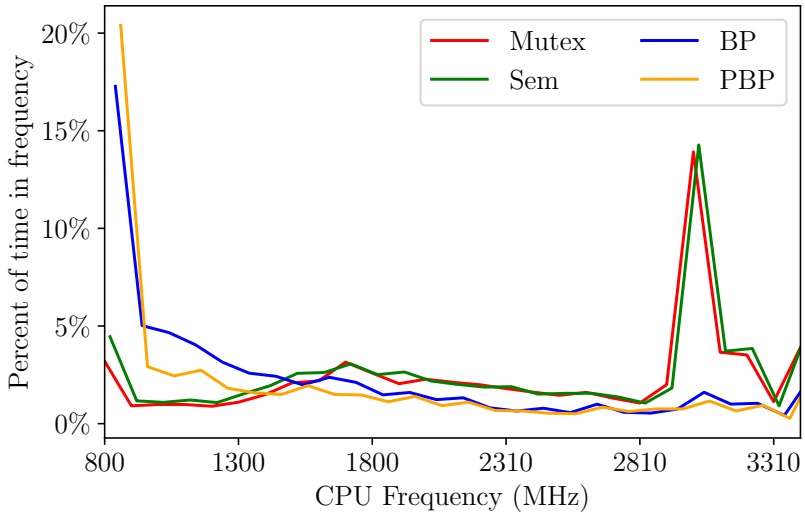


Figure 2.4: Percentage of time the CPU spends operating at different frequencies for Mutex, Sem, BP, and PBP when running the M/M/1/c dataset.

The number of wakeups of Mutex and Sem is similar, as is their energy consumption. BP has a significantly high number of wakeups, which can be explained by the consumers waiting for the buffer to be filled. PBP reaches a compromise by using a periodic activation of consumers, which reduces the number of wakeups by approximately 50%. This decrease in the number of wakeups results in an increase in usage since the CPU has no chance to go to sleep too often.

The key to understanding the energy consumption of the different implementations is the distribution of time spent in different CPU frequencies. As mentioned earlier, Yield spends 99% of its time at the maximum CPU frequency. Figure 2.4 shows the percent of time spent in different CPU frequencies of the remaining four implementations. We omitted Yield from Figure 2.4 to make it more readable for the remaining implementations. Notice that Mutex and Sem have low percentages when residing in lower frequencies, and a spike at 3GHz. This explains their relatively high energy consumption compared to the batch-based implementations. BP on the other hand spends most of the time at the lowest CPU frequency of 800MHz (approx. 18%). PBP spends even less time at frequencies higher than 800MHz.

These results are interesting since the initial assumption is that a batch-based implementation will have a lower energy consumption due to the reduced number of wakeups.

However, results show that batch-based behavior actually leverages DVFS effectively to increase the percentage of time the CPU operates at lower frequencies. A simple DPM scheme demotes the CPU C-state gradually by checking CPU activity periodically. Implementations such as Mutex and Sem activate the CPU too frequently and, thus, the DPM has no chance to demote the C-state. On the other hand, Intel SpeedStep increases CPU frequency gradually to match CPU activity, which causes it to operate mostly at high frequencies in these implementations.

Batch-based implementations such as BP and PBP give DPM time to demote the CPU c-states. This idling of the CPU prevents Intel SpeedStep from promoting the CPU frequency, and thus most of the time is spent at the lowest CPU frequency. PBP has a lower energy consumption versus BP due to the following reasons:

- CPU spends less time in higher frequencies compared to BP (see Figure 2.4). This is because BP only wakes up to a completely full buffer, which increases the processing load resulting in the CPU climbing up to higher frequencies. However, the periodicity of PBP prevents this from happening, since it frequently wakes up to consume a small number of items, and the CPU does not have a chance to climb up to higher frequencies.
- The number of wakeups of PBP is less than BP.
- Due to periodically consuming items, PBP finishes processing the dataset slightly faster.

These conclusions are further supported by analyzing the correlation between the average CPU frequency and energy/power consumption. This correlation is calculated across all 250 experiments, and checked for pearson correlation assumptions. The result is a 0.70 correlation between average CPU frequency (weighted average) and both energy and power, indicating a strong positive correlation. The correlation between the number of wakeups per second and energy/power is -0.71, indicating a strong negative correlation. We explore this further by studying the correlation between energy consumption and the percent of the time spent at frequencies above 3GHz, which has a higher value of 0.86, indicating the significance of residency at higher frequencies on energy/power. Between batch processing implementations, the significant factor is the number of wakeups, which is 0.65 correlated to energy/power. Thus, the large difference in energy consumption between batch implementations and all others is due to the lower average frequency which is caused by the higher number of wakeups, while the small difference between both batch

implementations is mainly due to the difference in the number of wakeups, since these implementations don't differ significantly in CPU frequency distribution.

Batch processing has its drawbacks, mainly the latency in responding to items. Mutex and Sem implementations have much lower latency. However, when energy efficiency is a main concern, a batch-based implementation with a bounded latency can provide a energy-efficient and acceptable solution. We study this in further detail in the upcoming sections.

In summary, the lesson learned from this simple study of a fundamental problem is the following:

> *We believe that the reduction level in energy consumption observed in our experiments strongly justify the pressing need to re-visit the design and implementation of classic algorithms to make them energy efficient.*

2.4 Formal Problem Description

In this section we formally present the optimization problem for reducing energy consumption in multiple producer-consumer pairs.

2.4.1 Approach

As explained in Section 2.3, the number of wakeups and the average CPU frequency have a significant effect on energy consumption. Batch processing implementations leverage longer periods of idle in increasing the number of CPU wakeups. This results in the CPU waking up to lower frequencies instead of promoting its frequency due to continuous load and lack of idle time. Thus, a logical objective is to force the CPU to sleep longer between wakeups. However, such an approach requires incorporating the DPM and DVFS mechanisms in the underlying system to determine the best period to sleep in order to force the CPU to go into a deeper idle state and consequently reduce average CPU frequency. In this chapter, we attempt to formulate the problem in a way that is independent of the underlying DPM and DVFS mechanism implemented in the system. First, we assume a simple power model where there are only two CPU states: active and idle. The CPU immediately goes to idle when a sleep or lock is called. For instance, in such an abstract system, a producer will sleep after producing an item, and wake up to produce the next item. A semaphore based consumer will wake up whenever a ticket is available to consume, and will sleep while

there are no tickets available in the semaphore. Thus, the objective becomes *reducing* the number of wakeups. This implies longer idle periods between wakeups. If we reapply this back to our actual system, longer idle periods results in a higher chance that the CPU actually gets demoted to idle, as opposed to shorter idle periods where the CPU has no chance to go to sleep. This results in a higher number of physical wakeups/sec and a lower average CPU frequency.

2.4.2 System Assumptions

We begin by stating the assumptions on which the problem is based:

- *Multicore system.* The system we attempt to optimize for energy is a multicore system, which supports core parking.
- *Abstraction of DVFS and DPM.* For generality, we abstract the operations of DVFS and DPM in managing CPU frequency and idle state respectively.
- *Multiple producer-consumers.* The system hosts a set of producer-consumer pairs where each consumer has its own buffer.
- *Independent producer rates.* Each producer produces data elements at its own non-linear and non-constant rate, independent of other producers.
- *Maximum response latencies.* Each consumer defines the maximum time allowed for a data item to be buffered and not processed. Any data item must be processed before or at the maximum response latency.
- *Consumer isolation.* Consumers are isolated on dedicated cores. This assumption is to isolate the effect of background processes that can potentially wakeup the core on which the consumer is executing. We also assume producers are either processes on separate cores or external events, such that they do not interfere with the consumers and their dedicated cores.

2.4.3 The Optimization Problem

We now formalize the multiple producer-consumer energy efficiency problem for a multicore system. Since we assume cores support parking, and can be idled and frequency scaled

separately, we will focus the rest of the problem on one core and extend trivially to all cores of the system.

A set of producers $P = \{p_1, p_2, \ldots, p_M\}$ produce data items at their independent varying rates. Each producer p_i, $1 \leq i \leq M$, produces the data items $\mathcal{D}_i = d_{i,1}, d_{i,2}, \ldots, d_{i,N_i}$, where N_i is the total number of items produced by producer p_i. Each data item is a tuple defined as follows:

$$d_{i,j} = \langle \upsilon_{i,j}, \pi_{i,j} \rangle$$

where $\upsilon_{i,j}$ is the time stamp at which producer p_i produces data item $d_{i,j}$, and $\pi_{i,j}$ is the computational load of consuming $d_{i,j}$ in terms of CPU cycles.

We assume pairs of producers and consumers. Let there be a set $C = \{c_1, c_2, \ldots, c_M\}$ of consumers running on a single core. In case the system has multiple cores, we assume that each core serves a fixed set of consumers (i.e, consumers do not change core). Each consumer c_i, $1 \leq i \leq M$, consumes items produced by producer p_i. For each consumer c_i, let the activation times of the consumer be the total order set $\mathcal{T}_i = \{t_{i,1}, t_{i,2}, \ldots, t_{i,k_i}\}$, where k_i is the number of activations of the consumer. The value of k_i depends on the time of the activations and the number of data items buffered in between. If the consumer does not employ batch processing (such as Sem), then $k_i = N_i$, meaning the consumer will be activated every time it receives a data item. If it uses simple batch processing and has a buffer of size 10, then $k_i = \lceil N_i/10 \rceil$.

We now define four functions that return activation times, and buffered and processed data items:

- **Activation times.** Let $\delta_i(t)$ be a function that returns the last activation of consumer i before time t:

$$\delta_i(t) = \max\{t_{i,j} \mid t_{i,j} < t\}$$

- **Buffered items.** We define function $\gamma_i(t)$ to denote the buffered data items at time t (items that have not been consumed yet):

$$\gamma_i(t) = \{d_{i,j} \mid \delta_i(t) < \upsilon_{i,j} \leq t\}$$

  Thus, if the buffer size is $\mathbb{B}$, then the following condition must hold to prevent buffer overruns:

$$\forall i : \forall j : \gamma_i(t_{i,j}) \leq \mathbb{B} \tag{2.1}$$

- **Processed items.** Let $\beta(d_{i,j})$ be a function that returns the time at which data item $d_{i,j}$ is processed by consumer i:

$$\beta(d_{i,j}) = \min\{t_{i,j} \mid t_{i,j} \geq \upsilon_{i,j}\}$$

Thus, the following condition must hold to maintain a maximum response time $\mathbb{M}$:

$$\forall i : \forall j : \beta(d_{i,j}) - \upsilon_{i,j} \leq \mathbb{M} \tag{2.2}$$

where $\mathbb{M}$ is the upper bound on response time. Let A be the set of the union of all activation times $t_{i,j}$ for every consumer c_i running on a single core.

$$A = \bigcup_{i \in [1,M]} \mathcal{T}_i$$

We will refer to A as the set of *core activations*. Thus, this set is of the form $A = \{a_1, a_2, \ldots, a_L\}$, where each a_l is a core activation time. Let $\lambda(a_l)$ be a function that returns all the data items that are buffered by all consumers at core activation a_l:

$$\lambda(a_l) = \bigcup_{i \in [1,M]} \gamma_i(a_l)$$

Next, we define functions that help calculate the energy consumption given an abstract DPM and DVFS:

- **Wakeup energy.** Let us define

  $$\epsilon_\omega(t_p, f_s) \rightarrow \langle E, f_e \rangle$$

  as a function that calculates the energy consumed by the CPU when no items are being processed for a duration of t_p, given that the CPU is initially operating at frequency f_s. If the DPM decides to idle the CPU core during time t_p, the following costs will apply:

  - The energy cost associated with the power-down of the CPU core (which is initially running at frequency f_s).
  - The energy cost of an idle CPU for a portion of t_p.
  - The wakeup energy of the CPU at the end of period t_p.

  The function will return the total energy consumed (denoted $\epsilon_\omega(t_p, f_s) \cdot E$) and the frequency of the CPU at the end of t_p (denoted $\epsilon_\omega(t_p, f_s) \cdot f_e$).

  If the period is too short to go to idle, and the DPM decides to keep the CPU active, then there is no energy consumed due to idling in period t_p, and the returned energy is that of an active CPU for the duration of t_p.

- **Consumption energy.** Let us define the following function:

$$\epsilon_\vartheta(D, f_s) \rightarrow \langle E, t_c, f_e \rangle$$

This function receives a set of data items D and the initial CPU frequency f_s. The function returns a tuple of the energy E consumed to process these items, the time t_c to consume them, and the CPU frequency f_e at the end of consumption. This function represents how DVFS would manage the energy consumption and completion time of a set of data items. We assume the CPU core does not go to idle while consuming items in D since all items are already buffered and ready for processing. This assumption isolates DPM from DVFS, such that $\epsilon_\vartheta(D)$ is solely responsible for the energy consumption of the CPU in stretches of time where it is active, without DPM intervention.

Now, we define the functions that calculate the energy consumption between every two consecutive core activations in A. Let us start with the very first activation; i.e., a_1. The CPU is assumed to be idle before a_1, and, hence, we need to calculate the first wakeup energy:

$$\omega(a_1) = \epsilon_\omega(a_1, 0)$$

where $\omega(a_1)$ denotes the tuple of energy consumed to wake the CPU up for the first time (E) and the CPU frequency at a_1 (i.e., f_e). Hence, the first wakeup energy is $\omega(a_1) \cdot E$.

Next, we need to calculate the energy of consuming items buffered at a_1. Let $\vartheta(a_1)$ be defined as follows:

$$\vartheta(a_1) = \epsilon_\vartheta(\lambda(a_1), \omega(a_1) \cdot f_e)$$

where $\vartheta(a_1)$ denotes the tuple of (1) energy consumed to process buffered items at a_1 (i.e., E), (2) the final frequency at the end of consumption (i.e, f_e), and (3) the consumption time (i.e, t_c). Hence, the energy of consuming items buffered at a_1 is $\vartheta(a_1) \cdot E$. Now, the definition of ϑ can be generalized as follows:

$$\vartheta(a_l) = \epsilon_\vartheta(\lambda(a_l), \omega(a_l) \cdot f_e)$$

where $\omega(a_l) \cdot f_e$ is the final frequency of the (possibly) idle period right before a_l.

For all the subsequent activations, wakeup energy is calculated recursively as follows:

$$\omega(a_l) = \epsilon_\omega(a_l - a_{l-1} - \vartheta(a_{l-1}) \cdot t_c, \vartheta(a_{l-1}) \cdot f_e) \tag{2.3}$$

Thus, the activation wakeup energy ω of core activation a_l is the energy consumed within the inactive period between a_{l-1} and a_l. This period is calculated as the time between a_{l-1} and a_l minus the time spent consuming the items that were buffered at a_{l-1}.

Finally, our ultimate objective is to minimize the energy consumption of the CPU core given all possible activations A. Our optimization function is defined as follows:

$$\min_{A} \left\{ \sum_{l=1}^{L} \vartheta(a_l) \cdot E + \omega(a_l) \cdot E \right\} \tag{2.4}$$

Thus, the optimization objective to construct a wakeup pattern for all consumers running in the CPU core such that the set of core activation times A minimizes the total energy consumption.

2.5 Energy-aware Multiple Producer-Consumer Algorithm

This section presents the design of our algorithm to solve the multiple producer-consumer problem presented in Section 2.4. The optimization Objective 2.4 will yield an optimal energy consumption given retroactive knowledge of all arriving items. To design an online algorithm, we incorporate the insight gained in Section 2.3 with the formulation in Section 2.4. Roughly speaking, our algorithm interprets time as a *track* with periodic *slots*. Consumers latch onto core activations scheduled at slot boundaries to process their respective buffered items. This results in an increase of the length of idle time between consecutive core activations. Hence, the design attempts to minimize energy by targeting the two components of the objective in Formula 2.4:

- overall wakeup energy $\sum_{l=1}^{L} \omega(a_l) \cdot E$ is reduced by extending the period of idle time between core activations.
- consumption energy $\sum_{l=1}^{L} \vartheta(a_l) \cdot E$ is reduced due to longer idle periods which result in lower initial frequencies $\omega(a_l) \cdot f_e$.

To that end, we introduce a *core manager* component that targets aligning consumers to the slots in that core's track. The core manager is responsible for managing the slot allocations on the track of its respective core. Consumers are designed so that they can predict the production rate of data items to compute an appropriate latching time.

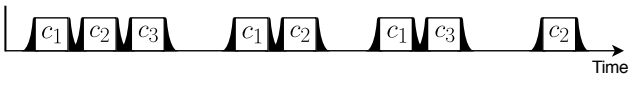
In the rest of this section, we describe our technique and design choices for core managers and consumers in Subsections 2.5.1, 2.5.3, and 2.5.2, respectively.

2.5.1 Contiguous Consumer Activations

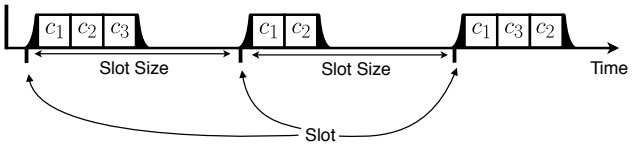
As established in Section 2.3, longer time between consumer activations results in the CPU going into deeper sleep, and hence, running at a lower frequency at the next consumer activation. This results in a lower average CPU frequency, and in turn, lower energy consumption. Recall that in Section 2.3, we identified batch processing as a more energy-efficient implementation of the producer-consumer problem. Now, consider three consumers: c_1, c_2, and c_3 that are invoked when their respective buffer is full. Since the amount of time it takes to fill the buffer depends upon the rate of the data item production, a possible activation pattern of these three consumers could be as shown in Figure 2.5a. The DPM can handle the consumer activations in the figure in one of two ways:

- The DPM can put the CPU to sleep after each consumer finishes consumption, resulting in an energy and time overhead.
- The DPM does not idle the CPU due to the shortness of the inactive period between consumer activations, and DVFS increases the CPU frequency assuming a continuous load, resulting in significantly higher energy consumption.

Our algorithm attempts to group consumer activations dynamically. We begin with interpreting time as a *track* with periodic *slots*. In our case, this is denoted as the *slot size* Δ. The default slot size is equal to the minimum of all maximum acceptable response latencies defined by the producer-consumer pairs. Figure 2.5b presents this idea. Observe that upon grouping, the number of wakeups is reduced to three, and the idle time between slot activations is longer. This gives DPM a chance to idle the CPU, which in turn wakes up to a lower frequency and so on. This example illustrates the potential impact of grouping on energy consumption. Let the timestamps of the start of these slots be the set $S =$



(a) Uncontrolled wakeups of multiple consumers.



(b) Aligned wakeups with consumer latching.

Figure 2.5: Uncontrolled vs. aligned wakeups of 3 consumers A, B, and C.

$\{s_1, s_2, s_3, \ldots \infty\}$ where $s_m - s_{m-1} = \Delta$. The initial objective of the algorithm is to ensure that all core activations are aligned to the slots:

$$\forall l : \ a_l \in S \tag{2.5}$$

While this constraint may result in a suboptimal core activation pattern, it simplifies the problem in the context of an online algorithm.

2.5.2 Consumer Design

The consumer is responsible for predicting the rate of items produced by the producer based on the recent past. We use a *moving average estimation* to determine the upcoming rate of items.

$$\hat{r}_{i,j+1} = \frac{\sum_{j'=j-h+1}^{j} r_{i,j'}}{h}$$

where i is the index of the consumer, j is the index of the consumer activation, $r_{i,j'}$ is the rate recorded at activation j', and

h is the number of previously recorded rates used by the moving average to estimate the future rate $\hat{r}_{i,j+1}$. Any rate $r_{i,j}$ is calculated as follows for consumer c_i:

$$r_{i,j} = \frac{|\gamma_i \left(t_{i,j-1}, t_{i,j}\right)|}{t_{i,j} - t_{i,j-1}}$$

where γ_i is defined in Equation 2.1. The reason for selecting the moving average is the simplicity of its calculation, imposing very low overhead on the processing involved, which is a desirable characteristic when attempting to minimize energy consumption.

2.5.3 Core Manager Design

The fundamental part of the proposed solution is the design of the core manager. The core manager performs the following steps:

1. Upon a scheduled wakeup, it looks up the registered consumers for the current slot, and activates them. This can be achieved by, for instance, signalling a semaphore. The consumers update their predicted rate in their individual data structures and begin processing items normally.

2. After activating all consumers, the core manager calculates how to distribute the shared buffer space such that consumers experiencing high load receive extra space to accommodate for the increase. Resizing is performed once every q core manager invocations. This is detailed in Subsection 2.5.3.

3. Next, the core manager reads the predicted rate of each consumer, and according to the new buffer size, makes a reservation to minimize the number of CPU wakeups. This is detailed in Subsection 2.5.3.

4. Finally, the core manager determines the next slot to wake up. Note that this does not necessarily have to be at time $s_m + \Delta$, where s_m is the current slot. The core manager will schedule the next slot with *at least one reservation*, thus, ensuring that the CPU is not activated needlessly. This results in a longer idle period, and thus has a direct impact on reducing wakeup energy and an indirect impact on reducing consumption energy (Formula 2.4).

It is worth noting that the core manager does not use a significant amount of memory in storing the reservations, since it only needs to maintain the set of reservations in the near future. Past reservations are replaced and future reservations are limited to only the next activation of every consumer.

Dynamic Buffer Resizing

Consider the case where the predicted rate of items is too high to be accommodated within one slot. This implies that a buffer overflow may occur before the closest slot triggers. Such a scenario motivates our idea on implementing a dynamic buffer resizing solution.

We previously introduced $\mathbb{B}$ as the size of the buffer used by each producer-consumer pair. To allow dynamic buffer resizing, we divide the buffer into a guaranteed portion $\mathbb{B}_0$ and a dynamic portion $\mathbb{B}_d$ such that $\mathbb{B} = \mathbb{B}_0 + \mathbb{B}_d$. Thus, the total amount of dynamic buffer space is $\mathbb{B}_d \times M$, where M is the number of consumers.

A consumer can relinquish part of the dynamic portion of its buffer to other consumers that are experiencing a high load. Thus we define $B_{i,m}$ as the actual reserved dynamic buffer size of consumer i at slot m. Thus, at slot m, the total buffer size available for consumer i is $\mathbb{B}_0 + B_{i,m} \leq \mathbb{B}$.

A consumer can be forced to run before its reserved slot to avoid a buffer overflow. This would occur if the producer is attempting to write an element for which there is no

space, and the consumer is still sleeping. The consumer is then immediately activated to empty the buffer and provide space for the producer. We refer to this forced wake up as a *buffer trigger*.

Dynamic resizing works as follows: after the core manager wakes up the scheduled / forced consumers, it calculates the number of times each consumer was buffer triggered during the last q invocations. It then redistributes the dynamic buffer space proportionally according to the number of buffer triggers. Consumers with more buffer triggers receive more dynamic space. Note that the core manager only redistributes space used by the consumers it activated, so as to not interfere with sleeping consumers awaiting activation.

The shared buffer space requires synchronization since it is being used across consumers. This synchronization is similar to that used in the multiple producer consumer problem with a single shared buffer.

The core manager blocks each producer-consumer pair while the update is in progress using mutexes. There might be a case where a buffer is shrinking and a producer has already written to an item beyond the newly shrunk size. Consequently, another producer which now has a bigger buffer cannot fully utilize it since there is no space left. This problem will be remedied once the consumers starts pulling items out of the buffer, and will not arise again until the next q^{th} invocation of the core manager, at which new resizing will apply.

Reservation

The process of selecting a slot to reserve is based on minimizing the cost function ρ over the set of possible slots:

$$\rho_i\left(s_{m'}\right) = \frac{\omega\left(s_{m'}\right) \cdot E + \epsilon_\vartheta(\hat{r}_{i,j+1} \times \left(s_{m'} - s_m\right), \omega\left(s_{m'}\right) \cdot f_e) \cdot E}{\hat{r}_{i,j+1} \times \left(s_{m'} - s_m\right)} \tag{2.6}$$

where s'_m is the slot being evaluated and s_m is the current slot. The value of $\hat{r}_{i,j+1} \times \left(s_{m'} - s_m\right)$ is used to calculate the number of data items predicted to have been buffered in slot $s_{m'}$, given that $t_{i,j} = s_m$ (the last activation of the consumer was at slot time s_m). The cost function ρ is normalized to represent the cost per data item. This gives perspective on the tradeoff between latching on a slot with a low predicted number of items versus reserving a new slot with a high predicted number of items.

Recall that the size of the buffer that the consumer reads from is now $\mathbb{B}_0 + B_{i,m}$. Thus, given that the current time (slot) is s_m and the predicted rate is $\hat{r}_{i,j+1}$, the time expected

to fill the buffer is

$$s_m + \frac{\mathbb{B}_0 + B_{i,m}}{\hat{r}_{i,j+1}}$$

To provide response time guarantees, the actual time we consider as the time the buffer is filled is the following:

$$t_f = s_m + \min\{\frac{\mathbb{B}_0 + B_{i,m}}{\hat{r}_{i,j+1}}, T_i\}$$

where T_i is the maximum response time allowed for consumer i.

The core manager starts evaluation at the latest slot to occur before t_f and backtracks until it is impossible to find a slot with lower ρ. If a slot has higher ρ than its predecessor, then it is safe to assume that no better slots can be found by further backtracking. Using a helper function in the core manager that backtracks to the next slot *with reservations*, the backtracking process only consumes one iteration and is, hence, a lightweight operation taking constant time and energy.

2.6 Implementation and Experimental Results

This section presents the results of the experiments to compare our algorithm with other standard implementations of the multiple producer-consumer problem. The experimental settings are similar to the ones presented in Subsection 2.3.2, with the addition that the core manager runs on a separate core, resulting in a total of 3 cores used.

2.6.1 Experimental Parameters

The experiments are based on executing producer-consumer pairs in parallel. We evaluate three implementations: Sem and BP discussed in Section 2.3, and our proposed algorithm in Section 2.5, periodic batch processing with latching (PBPL). We chose Sem because it is the most widely used implementation and also the most energy efficient out of common producer-consumer implementations. We chose BP because it is the simplest form of batch processing and is our frame of reference in terms of trivial batch processing implementations. To that end, we run two sets of experiments:

- $M/M/1/\mathbb{B}$*:* Each producer-consumer pair use an $M/M/1/\mathbb{B}$ process to generate / service requests. We experiment with different arrival rates by varying the λ parameter of the exponential distributions in the $M/M/1/\mathbb{B}$ process in an increasing

fashion, such that the mean inter-arrival period ranges from $10\mu s$ to $10ms$. We use a constant service time rate parameter and $\mathbb{B}$ indicates the buffer size we use.

- *Web server log data:* In these experiments the producers use the web server log data set mentioned in Section 2.3 with different phase shifts to create more variation among the producers and their production rates.

Furthermore, we experiment with the following parameters:

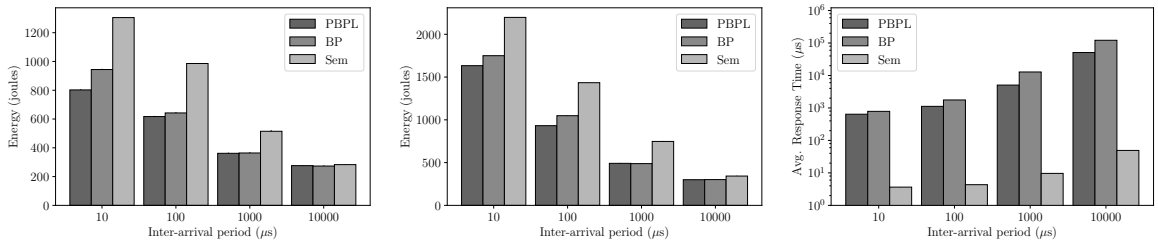
- *Number of consumers:* We experiment with 10, 20, and 30 producer-consumer pairs.
- *Buffer size:* For the batch processing based implementations (BP and PBPL), we experiment with three different buffer sizes: 25, 50, and 100. Due to space limitations, the next sections report results only from buffer size 25, and we refer to results of the other buffer sizes in Section 2.6.7.
- *Maximum response time:* For our implementation (PBPL), we experiment with different response time upper bounds, namely: 10μs, 100μs, 1ms, and 10ms.

We run each experiment for a duration of 100 seconds and measure the energy consumed during that period given an infinite supply of data items to consume.

2.6.2 Experimental Metrics

The following is the set of metrics measured for the executed experiments:

- **Energy.** The energy consumption in joules.
- **Number of wakeups per second.** The number of wakeups per second measured by PowerTop.
- **Average response time.** The average response time of consuming items, measured from the time the item is produced till the time it is consumed.
- **Number of buffer triggers.** The number of times a consumer is activated because the buffer is full.
- **Throughput.** The number of items consumed per second.
- **CPU frequency percentage.** The percentage of time spent by the CPU at different frequencies.



(a) Energy / 10 consumers (b) Energy / 30 consumers (c) Avg. Response time / 10 consumers

Figure 2.6: Plots of energy consumption and average response time at various mean inter-arrival periods of the $M/M/1/\mathbb{B}$ queue.

2.6.3 Energy Consumption Results

$M/M/1/\mathbb{B}$ Process

Figure 2.6a shows the energy consumption of the three compared implementations at exponentially increasing mean inter-arrival periods. The results in this figure are based on 10 concurrent consumers running on the same core and sharing a single core manager. As can be seen in the figure, PBPL reduces energy consumption of the popular semaphore-based implementation (Sem) by 38%, and is 16% better than a basic buffered implementation. That is in the case of a short inter-arrival period of 10μs. The advantage decreases as the mean inter-arrival period increases, with PBPL and BP approaching each other at $1ms$ and approaching sem at $10ms$. This is expected since idle power dominates experiments with a large inter-arrival period, since we are measuring each trial within a window of 100 seconds.

Figure 2.6b show the energy consumption of the three implementations when 30 consumers are running in parallel. The reduction in energy drops to 26% when 30 consumers run in parallel, which is due to the high contention accompanied by running a larger number of consumers in parallel with a short inter-arrival period.

Web Server Dataset Input

We observe similar energy savings when running the producers using the web server dataset that we used for our study in Section 2.3 (see Figure 2.7). PBPL reduces energy consumption

by 33%, 42%, and 45% when 10, 20, and 30 consumers are running in parallel, respectively. It also improves upon BP by up to 7.5%.

Energy consumption results from both sets of experiments demonstrate a consistent advantage in using the PBPL implementation. PBPL reduces energy consumption significantly and consistently for systems that experience high loads, whether due to high concurrency or short inter-arrival period.

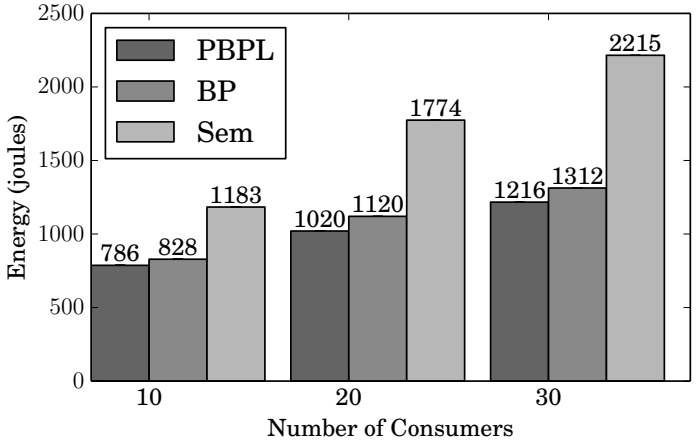


Figure 2.7: A plot of energy consumption of all 3 implementations running 10, 20, and 30 concurrent producer-consumer pairs. These experiments are based on the web server dataset.

The explanation for the reduced energy consumption of PBPL and BP is similar to our findings in the study in Section 2.3. The batch-based invocation of consumers in both implementations results in a lower average CPU frequency, since the CPU has a chance to switch to idle. When the CPU wakes up, it starts operating at the lowest available frequency and gradually increases. This is in contrast to Sem which does not transition to idle that often and, hence, the CPU DVFS mechanism increases the frequency due to the continued arrival of items to process. The overhead of context switching for each element is magnified as processing overhead, resulting in an average frequency of 1.8GHz for Sem versus an average of 1.2GHz for BP.

Figure 2.8 shows the frequency distribution of all three implementations when running 10 parallel consumers. Sem spends approximately 45% of its time at 3GHz, while BP and PBPL spend approximately 30% of their time at 800MHz, the lowest available frequency. PBPL outperforms BP due to the periodicity of its wakeup pattern. This allows it to mitigate the high loads more uniformly, resulting in a lower average CPU frequency of 1.1GHz. This is visible in Figure 2.8 where PBPL spends more time at 800MHz and less time at every higher frequency. Moreover, the latching behavior of consumers allows the load to be more uniform, and reduces the number of wakeups per second by 30% on average.

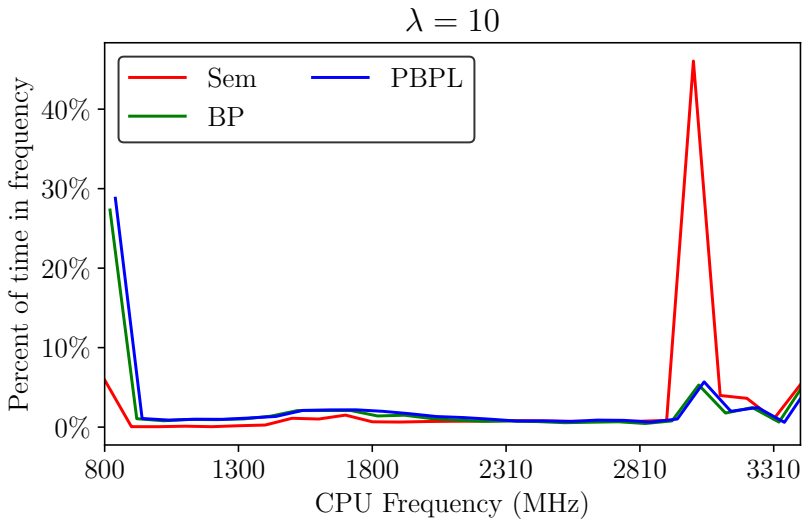


Figure 2.8: Percentage of time the CPU spends operating at different frequencies for Sem, BP, and PBPL when running the $M/M/1/\mathbb{B}$ dataset.

2.6.4 Average Response Time Results

As expected, the average response time of batch-processing-based implementations is significantly higher than that of a standard semaphore implementation. Figure 2.6c shows the response times of experiments based on $M/M/1/\mathbb{B}$ arrival pattern with varying mean inter-arrival periods. The figure shows that both BP and PBPL are orders of magnitude slower than Sem in terms of average response time. Due to the prediction and latching mechanism of PBPL, it has a response time ranging from 82% to 42% of the response time of BP. This is a significant advantage of PBPL, since it can provide better energy savings than BP while reducing response time by half.

PBPL allows the user to control the maximum allowed latency, thus allowing the user to control the tradeoff between energy savings and average response time. To that end, we run a set of experiments that demonstrate the energy savings of PBPL versus Sem when different maximum latency settings are applied. Figure 2.9 shows the different average response times of PBPL when different latency settings are applied. As can be seen in the figure, a maximum response time setting of any value less than the mean inter-arrival period causes a spike in the energy consumption of PBPL, due to the complicated logic of prediction and latching, and the overhead of setting timers whose period is shorter than the mean inter-arrival rate. Energy savings begin to manifest when the maximum

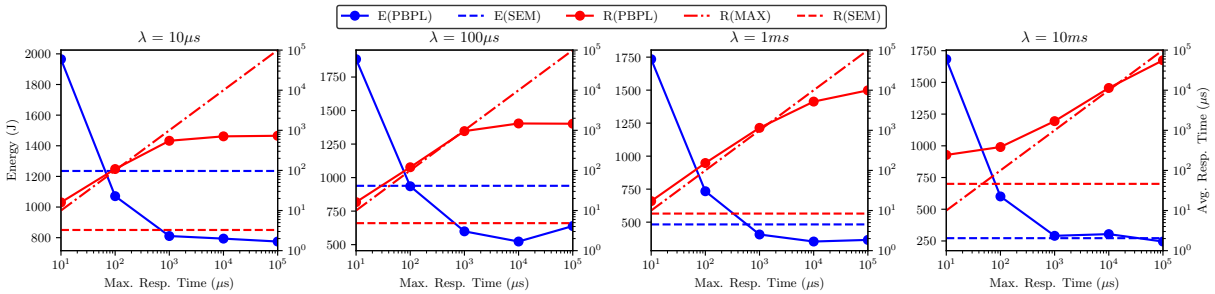


Figure 2.9: A plot of the energy and average response time of the three implementations when running 10 concurrent producer-consumer pairs at different latency presets (Max. Resp. Time). The experiments are executed at various mean inter-arrival periods of the $M/M/1/\mathbb{B}$ queue.

response time is set to the inter-arrival period or higher. This can be easily mitigated by selectively turning PBPL on or off based on the detected mean inter-arrival rate and configured maximum response time. This will result in a system of energy consumption equal to the minimum of PBPL and Sem.

2.6.5 Throughput Results

Next, we study throughput. Interestingly, while the average response time of batch-based implementations is orders of magnitude slower than that of the semaphore implementations, the throughput of batch-based implementations is consistently equal or higher. Figure 2.10 demonstrates the throughput recorded for the $M/M/1/\mathbb{B}$ experiments running 30 parallel consumers. As can be seen, both BP and PBPL outperform Sem at all inter-arrival periods. The improvement in throughput reaches 1.58x in case of an inter-arrival period of $10\mu s$. This

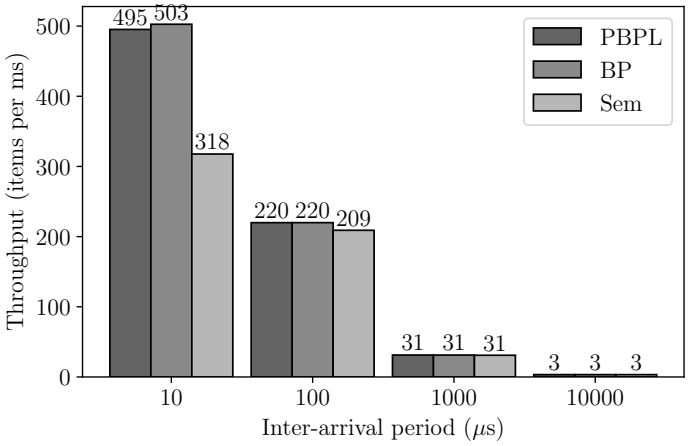


Figure 2.10: A plot of the throughput of all 3 implementations running 10 concurrent producer-consumer pairs at various mean inter-arrival periods of the $M/M/1/\mathbb{B}$ queue.

improvement diminishes at larger inter-arrival periods. The cause for this behavior is that when the inter-arrival period is significantly short the parallel producers and consumers become contentious. This results in a significant increase in context switching overhead in Sem, which causes the producers to block too often waiting for an empty slot in the circular buffer. Frequent blocking results in thrashing between the producers and consumers threads. This thrashing is reduced when using BP and PBPL since they process items in bulk, resulting in less context switching, and higher throughput.

2.6.6 Buffer Triggers

A buffer trigger indicates the CPU was activated before its scheduled slot. This counteracts the benefits of PBPL by increasing the total number of CPU wakeups. We introduced dynamic buffer resizing to reduce the number of buffer triggers. According to our $M/M/1/\mathbb{B}$ experiments, dynamic buffer resizing reduces the number of buffer triggers by approximately 50%. This translates into an energy saving of 10% compared to a run without dynamic buffer resizing when running 10 consumers at an inter-arrival period of $10\mu s$. The energy benefits diminish as shown in Figure 2.6a due to the decreased number of wakeups altogether.

2.6.7 Discussion

We validated the effectiveness and efficiency of our algorithm by conducting a set of experiments. Our experiments demonstrate the following advantages of using the proposed approach:

- Our proposed approach (PBPL) can reduce energy consumption by 38% compared to a standard semaphore implementation of producer-consumer. It can reduce energy consumption by 16% compared to a simple batch-processing implementation.
- PBPL consistently has lower response time than BP, in some cases reaching 42% of the response time of BP.
- PBPL has a higher throughput than a semaphore based implementation in most cases, reaching 1.58x in a highly contentious setting.

More importantly, the experiments help identify the circumstances where the proposed approach is beneficial, and where it should not be used.

- If the required maximum response time is too stringent (less than 1ms in our experiments), then PBPL will fail to provide meaningful energy savings. In fact, if transiently that is the case, the system should fall back to a semaphore based implementation.
- Energy savings are diminished when the inter-arrival period is longer than 10ms.
- PBPL reduces energy consumption and increases throughput when the system is contentious, either due to a high number of consumers or a short inter-arrival period, or both.
- If the inter-arrival period is too short or the number of consumers is too high, PBPL fails to provide any advantages, since the system is almost fully utilized all the time.
- The gap between PBPL and BP in terms of energy is dimished when the buffer size increases.

Another aspect that can influence energy savings is service time. In this chapter, we have shown the impact of synchronization strategies - in isolation - on energy consumption. It is important to note that if service time is considered, the impact of PBPL will be affected. As service time increases, the CPU is active for a longer time while servicing requests. This results in increases in CPU frequency, and consequently, energy consumption. Applications where consumers are CPU-bound are less likely to benefit from our concurrency based energy savings, since energy consumption is dominated by consumers processing data items. Applications that perform limited computation on data items or serve static content can benefit from PBPL. Examples are web servers serving static content, runtime monitors updating state, or networking buffers passing on requests without any involved processing.

We have also experimented with larger buffer sizes (50 and 100). Similar to our conclusions regarding contentiousness of inter-arrival periods, smaller buffer sizes magnify energy savings. Large buffer sizes result in a relaxed system that does not transition from idle to active very frequently, and hence the energy savings are diminished.

The above findings help identify the system where PBPL is most suitable. If the system is contentious with strict constraints on throughput rather than response time, PBPL is a suitable choice to reduce energy consumption. The situations where PBPL increases energy consumption can be easily detected and conditioned to run an implementation such as Sem.

2.7 Summary

In this chapter, we proposed a novel energy-efficient algorithm for the multiple producer-consumer problem for multicore systems, where each consumer is associated with one and only one producer. To our knowledge, this is the first instance of such an algorithm. Our approach is based on dynamic periodic batch processing, such that consumers process a set of items and let the CPU switch to idle state, hence, saving energy. Consumers make prediction about the rate of incoming data items and group themselves together. This results in two energy reducing consequences: (1) the number of wakeups is significantly reduced versus trivial batch processing, and (2) high loads are mitigated and spread out uniformly preventing DVFS mechanisms from promoting the CPU frequency, resulting in the CPU spending most of its time at lower frequencies.

We observed that our algorithm can lower energy consumption by 51% compared to a mutex/semaphore implementation when running 10 parallel consumers. In fact, it provides up to 15% improvement over the batch processing implementation in our study. We also observe that our algorithm excels with the increase in the number of consumers, making it scalable and robust.

Chapter 3

Power versus Performance

The power performance tradeoff has taken a new form with the increasing importance of exascale computing. Limited provisioning of power stations is enforcing a bound on the power consumption of data centers. Thus, performance optimization under power constraints is becoming a major design issue for large scale data centers. This chapter tackles the problem of performance optimization for distributed applications operating under a power constraint. We focus on a subset of distributed applications where nodes encounter multiple synchronization points and exhibit inter-node dependency. We abstract this structure into a dependency graph, and leverage the asymmetry in execution time of parallel tasks on different nodes by using *power redistribution*. Power redistribution dynamically detects that a node is *blocked* (i.e. waiting for other nodes), and reduces its power bound so that it can increase the *blocking* nodes power bound while still maintaining the same total power consumption. To that end, we present an *online* heuristic that dynamically redistributes power at run time. We implement the heuristic as an MPI wrapper that infers dependency and redistributes power online. To quantify the accuracy of the heuristic, we compare it to an *offline* solution based on integer linear programming (ILP) that optimally distributes power. In our experiments, the heuristic shows significant reductions in total execution time of a set of parallel benchmarks with speedup reaching 1.8 times.

3.1 Introduction

Heterogeneous clusters are becoming favorable as opposed to homogeneous clusters [137], due to their lower operational cost and easier and cheaper upgrades. However, improving

performance of heterogeneous clusters when operating at a power constraint is a challenging task due to two reasons: (1) heterogeneous clusters are often over-provisioned, such that the cluster power bound can be lower than the maximum total power consumption of all nodes running at maximum frequency, and (2) when running a distributed application with data dependency, nodes will often block waiting for other nodes to continue. The latter will result in cluster power being under-utilized, which translates into increased total execution time. Thus, there is need for a dynamic method to intelligently distribute power among nodes according to their inter-dependency. Extensive work has studied the trade-off between power efficiency and delay [58, 62], as well as work that targets reducing execution time, which results in reduced energy consumption [141]. However, there is little work on dynamically optimizing power distribution of distributed applications with inter-node dependency given a cluster power bound.

With this motivation, in this chapter, we focus on designing a technique that attempts to maximize the performance of distributed applications running on a heterogeneous cluster, while satisfying a given power bound. To achieve this goal, we introduce *power redistribution*. Power redistribution is an online method of dynamically changing node power bound according to the application's behavior. For instance, a node will becomes idle when it completes its task and blocks waiting for other nodes. Such a node is called a *blocked* node, which waits for *blocking* nodes to continue execution. A power redistribution controller can dynamically *redistribute* the power budget from the blocked node to the *blocking* nodes. Thus, such a controller will set the blocked node to run at a power bound lower than its original bound. The controller then uses the gained power budget to increase the power bound of blocking nodes, which results in the blocking nodes finishing their tasks sooner.

To illustrate how power redistribution works, consider a cluster consisting of two identical nodes. The cluster power bound is 10 watts. Each node uses *ondemand* DVFS, yet is configured to never exceed 5 watts, thus honoring the cluster power bound. The nodes run a computationally intensive task and consume power close to 5 watts. Assume that the first node finishes its task, and its power consumption drops to 2 watts. However, since the power bound on the second node is static, we have to wait for the node to finish its task. An opportunistic power redistribution heuristic will detect this scenario and *transfer* the power budget from node 1 to node 2, setting the maximum for node 1 to 2 watts, and 8 watts for node 2. This allows node 2 to finish its task sooner, thus reducing the total execution time.

This chapter introduces the following:

1. We introduce an online heuristic to distribute power in a cluster running a parallel

application. We construct a power distribution controller that is responsible for distributing power dynamically while honoring a cluster power bound (Figure 3.1). The controller can detect when a node is blocked, and it determines how much power is gained by idling the blocked node. The controller then distributes the gained power to blocking nodes using short network commands.

2. To make the approach more versatile for MPI applications, we build an MPI wrapper that communicates with the power distribution controller, making power redistribution transparent to the application developer. A significant advantage of the proposed solution is that it is designed to improve performance of distributed applications with minimal requirements. Simply by linking existing code to the proposed MPI wrapper and connecting a low-end embedded board hosting the power distribution controller, a cluster can immediately benefit from dynamic power redistribution. Our approach utilizes DVFS drivers by configuring the maximum frequency that a DVFS scheme can use, thus allowing DVFS to operate normally while honoring the node power bound.

3. As a reference to evaluate our heuristic, we build an ILP (Integer Linear Programming) model that is capable of producing an optimal assignment of power to synchronization blocks of the program. The ILP model is used for comparison to evaluate the speedup results of the power redistribution heuristic.

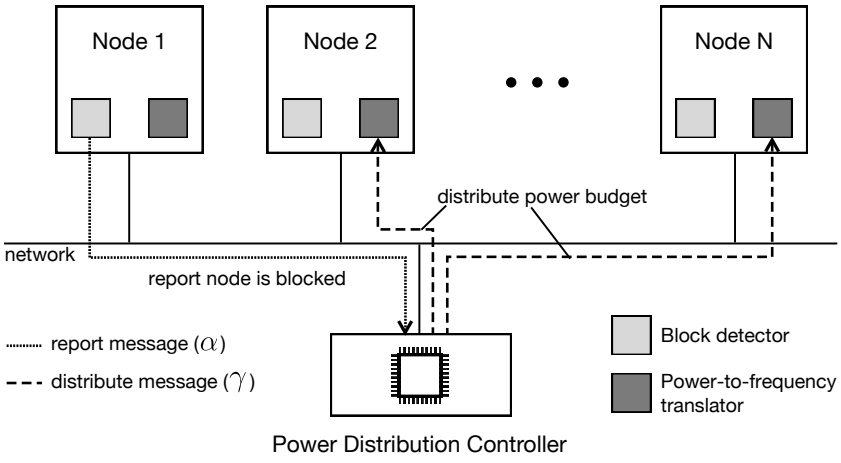


Figure 3.1: Block diagram of an HPC cluster equipped with power distribution.

To validate our approach, we conduct a set of simulations and actual experiments. First, we simulate a simple parallel program running using equal-share (all nodes have the same

power bound), ILP based, and heuristic based power distribution schemes. The simulation demonstrates the correlation between the variability in execution time and speedup. The improvement in execution time is promising, reaching a speedup of 2.5 times for ILP-based distribution and 2.0 time achieved by our online heuristic. To demonstrate the applicability of the approach to homogeneous clusters, we run the simulation where all parallel tasks consume the same amount of time, and all nodes are identical. The speedup in this case is an encouraging 2.0 for optimal distribution (ILP), and 1.64 for the online heuristic-based distribution.

We then experiment on a physical environment using a 12-node cluster where nodes vary in CPU, operating system, and manufacturer. We implement our MPI wrapper that detects changes to node state and communicates it with a power distribution controller that executes our online heuristic. Using different benchmarks from the NAS benchmark suite, we demonstrate that our online heuristic can produce speedups up to 1.8 times, while the ILP based solution reaches 1.86 times. We also draw conclusions on the type of MPI program that would benefit most from our online power redistribution heuristic.

Organization: The rest of the chapter is organized as follows. Section 3.2 introduces a motivating example to demonstrate our approach. Section 3.3 outlines a formal definition of the power distribution problem. Section 3.4 details the design of our online heuristic. Section 3.5 presents an algorithm to achieve the optimal solution. Section 3.6 presents the simulation results. Section 3.7 presents the MPI specific implementation and the experimental results. Finally, we summarize the chapter in Section 3.8.

3.2 Motivation

This section outlines a motivating example to demonstrate the existence of an opportunity to optimize performance by redistributing power. Listing 3.1 shows an abridged version of the `rank` function in the Integer Sort benchmark of the NAS Benchmark Suite [10]. As can be seen in the code, there are four main blocks of computation: The first block spans lines 2 to 7. This is followed by a blocking collective operation at line 8. This sequence continues until the last block spanning lines 17 to 19.

Figure 3.2 illustrates a possible execution of this function on a 3-node cluster. In this cluster, a cluster power bound is enforced, and this results in a power cap assigned per node, which sets a maximum frequency that DVFS can use. Thus, as shown in the figure, all blocks consume at most 33% of the cluster power bound. It is possible and quite frequent that some nodes finish execution of a block of computation earlier than others,

yet a blocking operation forces them to wait. This can be the result of, for instance, using heterogenous nodes, differences in workload, or nodes executing in different execution paths. This is demonstrated in Figure 3.2 by the dark grey blocks in the figure, which we denote as *blackouts*. Naturally, execution cannot proceed until all nodes arrive at the *barrier*. This also applies to node-to-node send and receive operations.

Our research hypothesis is that an intelligent distribution of power can reduce these blackout periods, resulting in reduction of total execution time. This is demonstrated in Figure 3.3. The thickness of a block indicates how much power it is allowed to consume. The percentage of total power each block consumes is shown inside the block. Blocks that consume a relatively short time in Figure 3.2, such as the first block in node 2, operate at a lower power cap in Figure 3.3 (using at most 23% of total power versus the original 33%). An optimum solution is capable of eradicating all blackouts, and minimizing those that are unavoidable (such as a ring send/receive). This results in a reduced total execution time within the cluster power bound.

```
1  void rank( int iteration ) {
2      for(i=0;i<NUM_BUCKETS+TEST_ARRAY_SIZE;i++) {
3      // Computation
4      }
5      for( i=0; i<TEST_ARRAY_SIZE; i++ ) {
6      // Computation
7      }
8      MPI_Allreduce( ··· );
9      for( i=0, j=0; i<NUM_BUCKETS; i++ ) {
10     // Computation
11     }
12     MPI_Alltoall( ··· );
13     for( i=1; i<comm_size; i++ ) {
14     // Computation
15     }
16     MPI_Alltoallv( ··· );
17     for( i=0; i<TEST_ARRAY_SIZE; i++ ) {
18     // Computation
19     }
20 }
```

Listing 3.1: Abridged rank method in the NPB IS benchmark

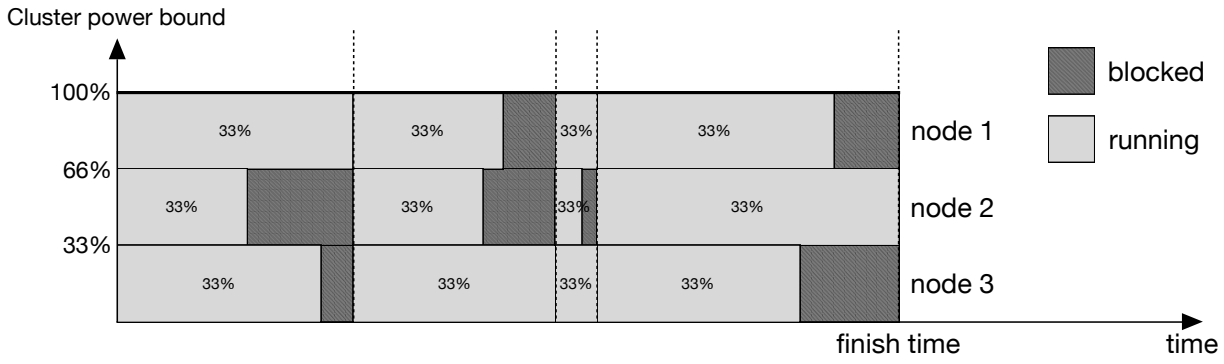


Figure 3.2: A possible execution of rank on 3 nodes.

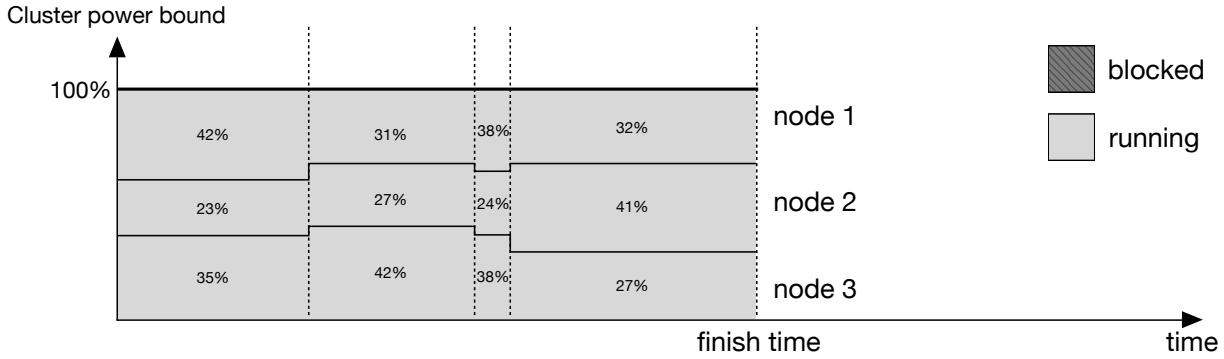


Figure 3.3: An optimum execution of rank on 3 nodes using power redistribution.

3.3 Formal Problem Description

This section presents a formal description of the power distribution problem. Consider a set $\mathcal{N} = \{N_1, N_2, \ldots, N_n\}$ of nodes in a parallel computing cluster that is required to satisfy a power bound $\mathbb{P}$. Each node runs a single instance of a parallel program. We model the execution of the program instance on a node as a sequence of tasks:

$$\mathcal{J}_i = \langle J_{i,1} J_{i,2} J_{i,j} \cdots \rangle$$

where $J_{i,j}$ is the j^{th} task on node i. A task represents a block of execution of the program instance on a single node that, once started, can be completed independently and without communication with other nodes. A task is defined as the following tuple:

$$J_{i,j} = \langle \tau(J_{i,j}, P), \theta(J_{i,j}), \pi(J_{i,j}) \rangle$$

where

- $\tau(J_{i,j}, P)$ is a function that encodes the execution time of task j on node i operating under power bound P, which directly enforces a maximum frequency that the CPU of node i can utilize.
- $\theta(J_{i,j})$ is a function that encodes the dependency of one task on a set of preoccuring tasks with the condition that it does not depend on multiple tasks in any other node. Such behavior can be expressed indirectly by chaining dependency. Naturally, in the serial execution of a program instance on node i, every task j is dependent on its predecessor $j-1$. That is, $J_{i,j-1} \in \theta(J_{i,j})$. This implies that the execution of task j cannot begin unless task $j-1$ is completed.
- $\pi(J_{i,j})$ denotes the power bound that node i should honour during execution of task j.

Our objective is to determine the mapping π of all tasks on all nodes to their power bounds ($\pi(J_{i,j})$) such that:

1. The dependency of tasks θ is not violated;
2. The cluster power bound $\mathbb{P}$ is not exceeded, and
3. The total execution time is minimized.

3.3.1 Task Dependency Graph

In order to calculate the total execution time, we construct a *task dependency graph*.

Task dependency graph A *task dependency graph* D is a directed acyclic graph, where vertices represent tasks $J_{i,j}$ and directed edges represent the dependency relation as described by θ, such that if $(J_{i,j}, J_{i',j'})$ is a directed edge of D, then $J_{i,j} \in \theta(J(i', j'))$. ∎

The task dependency graph is acyclic since a cycle will indicate circular dependency of tasks, which is impossible to occur since a task cannot be dependent on itself, directly or indirectly.

3.3.2 Total execution time

The total execution time of a program is the time taken to execute the critical path of the program. Let an *initial task* (J^I) be a task that does not depend on any other task to begin execution. A *final task* (J^F) is a task on which no other task depends. An *execution path* ρ is a path from an initial task to a final task ($J^I \rightsquigarrow J^F$) in a task dependency graph. The function $\epsilon(\rho)$ denotes the execution time of path ρ. Let a task dependency graph D contain a set of execution paths $\varrho_D = \{\rho_1, \rho_2, \rho_k, \cdots\}$. The critical path is the path in ϱ_D with the longest execution time.

3.3.3 Example of a Task Dependency Graph

To demonstrate how a task dependency graph is constructed and how it is used to determine the total execution time of a parallel program, in this section, we introduce a simple MPI program as our running example throughout the chapter. The program performs a set of commonplace MPI operations. The code in Listing 3.2 demonstrates an MPI program that goes through 3 steps:

1. Broadcasts a message from the root node.
2. Sends a message between nodes in a ring.
3. Performs a reduction on a variable.

Assume this program runs in a cluster of 3 nodes. Based on the steps mentioned earlier, nodes will execute the following tasks:

- $J_{\_,1}$: represents lines 2-11. This is applicable to all nodes.
- $J_{0,2}$ represents lines 13-21. However, $J_{1,2}$ and $J_{2,2}$ represent lines 13-25.
- $J_{0,3}$ represents line 22, while the other two nodes represent line 27.
- $J_{\_,4}$ represents lines 30-31.
- $J_{\_,5}$ represents line 32.

```
1  void main(int argc, char *argv[]) {
2      int msg1, msg2, msg3;
3      int rank, size, next, prev;
4      MPI_Init(&argc, &argv);
5      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
6      MPI_Comm_size(MPI_COMM_WORLD, &size);
7
8      if (rank == 0)
9          msg1 = 10;
10
11     MPI_BCast(&msg1,1,MPI_INT,0,MPI_COMM_WORLD);
12
13     next = (rank + 1) % size;
14     prev = (rank + size - 1) % size;
15
16     if (rank == 0)
17         msg2 = size;
18
19     if (rank == 0) {
20         msg2--;
21         MPI_Send(&msg2, 1, MPI_INT, next, rank, MPI_COMM_WORLD);
22         MPI_Recv(&msg2, 1, MPI_INT, prev, prev, MPI_COMM_WORLD);
23     }
24     else {
25         MPI_Recv(&msg2, 1, MPI_INT, prev, prev, MPI_COMM_WORLD);
26         msg2--;
27         MPI_Send(&msg2, 1, MPI_INT, next, rank, MPI_COMM_WORLD);
28     }
29
30     msg2 = rank;
31     MPI_Reduce(&msg2, &msg3, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);
32     MPI_Finalize();
33 }
```

Listing 3.2: A simple MPI program.

In total, there are 15 tasks in the system. Figure 3.4 presents the dependency graph of the program with some hypothetical task execution times. These execution times are a result of applying the same power bound $\mathcal{P}$ on every node in the cluster, which we denote as the *nominal power bound*. The nominal power bound $\mathcal{P}$ is equal to $\mathbb{P}/N$, simply distributing the cluster power bound equally among all nodes in the system. Every block in the figure represents a task, which can be identified by its column, indicating to what node the task belongs to, and its row, indicating the index of that task in the sequence of

node tasks. The nominal execution time of each task (that is $\tau(J_{i,j}, \mathcal{P})$) is indicated by the number inside the block. The arrows represent dependency among nodes. As can be seen in the figure, the longest execution path starts with $J_{2,1}$ and proceeds along the dashed lines. Hence, the total execution time is 19 time units.

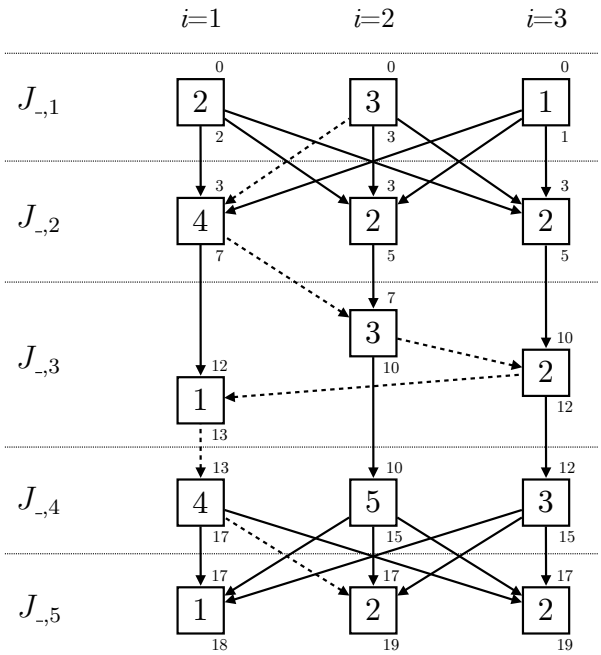


Figure 3.4: Dependency graph of the program in Listing 3.2.

Now, let us validate that the longest execution path is indeed indicative of the total execution time:

- Execution starts at task 0 in all nodes, which is a block of code that ends with a call to MPI_BCast. A broadcast operation is an implicit barrier, and, hence, no node can proceed unless all $J_{\_,1}$ tasks are completed. This is visualized by connecting every $J_{\_,2}$ task with every $J_{\_,1}$ task.

- Since $J_{2,1}$ takes the longest time, all $J_{\_,2}$ start after 3 time units. This is indicated by the superscript of these blocks in the figure.

- $J_{2,2}$, which ends with a call to MPI_Recv, will block execution until node 1 completes $J_{1,2}$ which ends with a call to MPI_Send. Thus, $J_{2,3}$ is dependent on both its predecessor $J_{2,2}$ and $J_{1,2}$ which will send a message. The consequence of this dependency

is that $J_{2,3}$ executes at 7 time units, which is the maximum of the completion times of $J_{2,2}$ and $J_{1,2}$, as indicated by the subscript of the respective blocks.

- If we follow this process we can determine the completion time of all nodes, indicated by the subscript of the final tasks $J^F_{\_,5}$, after which the program terminates. The last tasks to complete are $J_{2,5}$ and $J_{3,5}$, which finish after 19 time units.

As shown in Figure 3.4, the execution time of the longest path is the time at which all nodes in the cluster finish execution.

3.4 Online Heuristic for Power Redistribution

This section introduces the design of an online heuristic that dynamically distributes power. We approach the design of the heuristic with the following set of objectives:

- The optimum solution introduced in the previous section requires the knowledge of the execution time of every task at every CPU frequency. This is not realistically available for a running application, and, hence, obtaining an optimum online solution is not possible. Thus, our objective is to build an algorithm that receives realistic input and can make online decisions.
- Making power distribution decisions must incur minimal overhead; i.e., in a thrashing-free manner.
- Design the algorithm to be lightweight, executable on non-sophisticated power-efficient hardware.

The heuristic targets HPC clusters composed of heterogenous nodes. Figure 3.1 illustrates the structure of such a cluster, and introduces the following components:

- **Block detector.** The block detector is responsible for detecting when a node becomes blocked, awaiting some input from one or more other nodes. It is also responsible for detecting when the node becomes active again. it reports these changes in state to the power distribution controller. The block detector is further explained in Subsection 3.4.1.

- **Power distribution controller.** The power distribution controller receives messages whenever a node is blocked or unblocked. Since the blocked node will transition to idle, the total power consumption of the cluster will drop. This will provide a *power budget* that can be distributed to other nodes. The power distribution controller makes a decision on how to distribute the power budget on *running* nodes. The decision procedure is the core of our online heuristic, which is explained in detail in Subsection 3.4.2.

- **Power-to-frequency translator.** This component receives the distribute message and translates the power bound dictated by the power distribution controller to a CPU frequency. It selects the maximum CPU frequency that can satisfy the power bound in the message and forces the node to operate at that frequency.

3.4.1 Block Detector

As shown in Figure 3.1, the block detector sends a report message to the power distribution controller whenever a change in the node state is detected. A report message is a tuple $\alpha = \langle s, i, B, p_g \rangle$ where

- s is the state of the node, whether Blocked or Running.
- i is the index of the node from which the report message originated.
- B is a set of node indices that are causing node i to be blocked. If $s =$ Running, B becomes an empty set.
- p_g is the power gained by blocking node i, which is calculated as follows: $p_g = p_{f_c} - p_s$ where f_c is the CPU frequency before the block is encountered, p_{f_c} is the power consumed by running the CPU at frequency f_c, and p_s is the idle power.

To compute p_g, we require that each node hosts a lookup table mapping CPU frequency to power. This is obtained by executing a simple benchmark that loads the CPU 100% at each frequency, and records the power consumption. However, if a node hosts a multicore CPU and is executing multiple tasks concurrently, one per core, the power gain becomes the current power minus the power consumed when one less core is active. Hence, we require that the lookup table includes the power consumption of the node at each available frequency *and* at every possible number of active cores. For instance, a quad-core CPU

that supports 10 different frequencies will result in a lookup table of 40 entries. An entry will be identified by (1) the number of active cores in parallel (e.g., $1 - 4$ for quad-core CPUs), and (2) the CPU frequency.

To formally define this, let $p_{m,f}$ be the power consumption of the node when m cores are active and the CPU frequency is f. Let m_c and f_c be the number of active cores and the CPU frequency before the block is encountered on a task executing in one core. In that case, p_g is calculated as follows:

$$p_g = p_{(m_c-1,f_c)} - p_s \tag{3.1}$$

3.4.2 Online Heuristic Design

Algorithm 1 details the logic behind the power distribution heuristic. The heuristic is initialized with a cluster power bound $\mathbb{P}$ watts and an empty (online) dependency graph $G = (V, E)$. The following steps detail the operation of the heuristic:

1. The function ProcessMessage is invoked whenever the power distribution manager receives a report message α from any node in the cluster (line 4).

2. Lines 5-11 in function ProcessMessage update the online dependency graph using the received message. ProcessMessage creates a vertex for the node if it does not already exit, and connects the vertex to other vertices representing nodes that are blocking the sender node.

3. Lines 12-15 calculate the available power budget by adding the power gain p_g of all blocked nodes in the graph.

4. The function then calls RankGraph which calculates the priority of a node based on the number of other nodes that it is blocking. A node of rank 0 has no incoming edges, and hence is not blocking any node. A node of rank 1 blocks one other node, and so on.

5. Finally, function DistributePower is responsible for distributing the power budget over running nodes. If a node is blocking two other nodes, it will receive twice the amount of power received by a node that is blocking only one other node. This strategy allows the system to gradually increase the power bound of the older blocking nodes, since every time any node is blocked, the older blocking node receives a portion of the power gain.

Algorithm 1 Power distribution online heuristic.

```
1: INPUT: Cluster power bound ℙ, number of nodes n
2: declare G = (V, E), initially V = E = {}                ▷ Online Dependency Graph
3: declare p_o = ℙ/n
4: function ProcessMessage(α)
5:     if α.i ∉ V then
6:         v ← AddVertex(V, α.i)
7:     else
8:         v ← V[α.i]
9:     v.s ← α.s
10:    v.p_g ← α.p_g
11:    UpdateEdges(G, v, α.B)                  ▷ Update connections of v using α
12:    declare ε = 0                                          ▷ Power budget
13:    for u ∈ V do
14:        if u.s = Blocked then
15:            ε ← ε + u.p_g
16:    t ← RankGraph
17:    DistributePower(ε, t)

18: function UpdateEdges(G, v, B)
19:    ClearOutgoingEdges(v)
20:    for u ∈ B do
21:        AddEdge(E, v, u)

22: function RankGraph
23:    t ← 0
24:    for u ∈ V do
25:        if u.s = Running then
26:            u.r ← |{e = (a, b) ∈ E | e.b = u}|
27:            t ← t + u.r                                ▷ Sum of all ranks
28:    return t

29: function DistributePower(ε, t)
30:    for u ∈ V do
31:        if u.s = Running then
32:            p'_b = p_o + ε × u.r/t
33:            if u.p_b ≠ p'_b then
34:                u.p_b = p'_b
35:                γ ← (u.i, u.p_b)
36:                SendPowerBound(γ)
```

3.5 Optimal Solution

This section presents a method based on integer linear programming (ILP) to obtain the optimal solution for the power distribution problem. We, in particular, develop this method, so we have a reference of goodness for our online algorithm in Section 3.4. The ILP solution is based on the following assumptions:

- A task is a block of execution followed by a communication/synchronization primitive.
- The power bound of a node cannot be changed while the task is executing.

While these assumptions limit the optimality of the solution, in the sense that a better solution can be found if for instance the power bound can be changed mid task, these assumptions allow for a reasonably sized model, by breaking down a program to well-defined synchronization points. For future work we attempt to extend the ILP model to waive these assumptions.

In order to limit the number of variables in the ILP instance, we design an algorithm that establishes potential interleavings among tasks executing in different nodes. This algorithm is similar to real-time scheduling algorithms for task dependency on multiprocessors, with the added dimensions of task power bounds and variable execution times. The following subsection introduces the *Task Concurrency Optimization* algorithm.

3.5.1 Task Concurrency Optimization Algorithm

The task concurrency optimization algorithm determines which tasks can execute concurrently without violating the dependency structure encoded in the task dependency graph. Since our objective is to reduce the length of blackouts, we can make an abstraction and avoid exploration of all possible interleavings in a parallel program. First, we begin by introducing the following definitions.

Task Max-Depth The max-depth δ of a task J in a task dependency graph D is the length of the longest path that starts from an initial task and ends with task J. That is

$$\delta(J) = \max\{l \mid \rho(l) = J \ \wedge \ \rho \in \varrho_D\}$$

Task Depth Range The depth range Δ of a task J in a task dependency graph D is an integer interval defined as follows:

$$\Delta(J) = [\delta(J), \beta(J) - 1]$$

where $\delta(J)$ is the max-depth of task J and the start of the interval, and $\beta(J)$ is the minimum max-depth of all J's children, that is the set of tasks that are dependent on J:

$$\beta(J) = \min \{\delta(J') \mid J \in \theta(J')\}$$

where J' is a task in the task dependency graph. ∎

Let us clarify the use of these definitions by referring to our earlier example in Listing 3.2, and the respective task dependency graph in Figure 3.4. Table 3.1 shows the max-depths of all the tasks in the graph. Note that the max-depth is affected by the ring of sends/receives in task 3 across all 3 nodes. Figure 3.5 visualizes how max-depths map to concurrency in execution. The dark grey blocks represent blackout periods, which should be optimally eradicated to reduce total execution time.

Table 3.1: Max-depths of tasks in Figure 3.4.

	Node 1	Node 2	Node 3
Task 1	0	0	0
Task 2	1	1	1
Task 3	4	2	3
Task 4	5	3	4
Task 5	6	6	6

Table 3.2 shows the depth ranges of all tasks in the graph. The depth ranges allow us to revisit the task concurrency in Figure 3.5 to produce an assignment that exhibits less blackouts. Figure 3.6 demonstrates applying depth ranges to determine optimum task concurrency. Observe task $J_{3,2}$ in the figure, which represents the code executed after the MPI_BCast and before the MPI_Recv called by the third node. The next block of code to be executed by node 3 requires that node 2 sends a message ($J_{2,3}$). This implies that the execution of $J_{3,2}$ can be *stretched* beyond its max-depth level until node 2 sends a message. *Stretching* a task in this manner implies allowing it to operate at a lower power level, thus enabling a higher power cap for other nodes in the cluster.

Thus, Figure 3.6 shows that utilizing depth ranges helps remove blackout periods in execution, except for unavoidable blackouts such as a message ring as shown in our example.

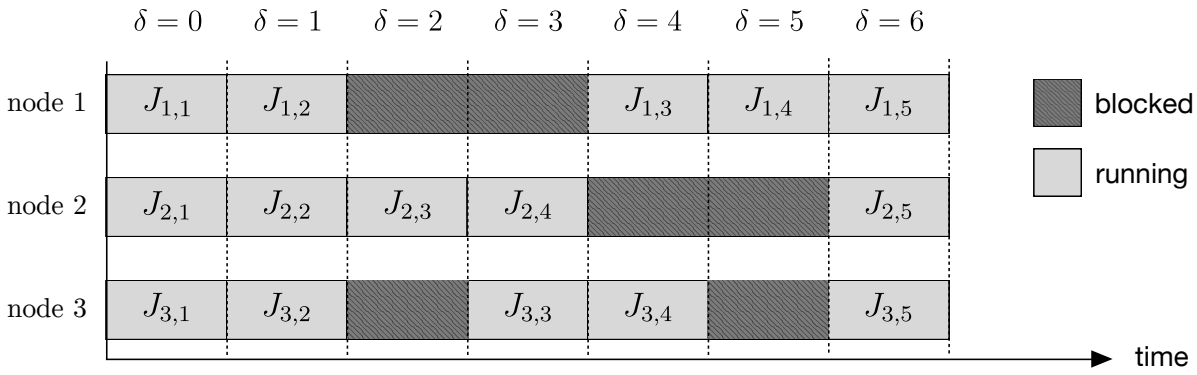


Figure 3.5: Task concurrency as indicated by task max-depths.

Table 3.2: Depth ranges of tasks in Figure 3.4.

	Node 1	Node 2	Node 3
Task 1	[0,0]	[0,0]	[0,0]
Task 2	[1,1]	[1,1]	[1,2]
Task 3	[4,4]	[2,2]	[3,3]
Task 4	[5,5]	[3,5]	[4,5]
Task 5	[6,6]	[6,6]	[6,6]

The algorithm is simple to implement, since calculating the max-depths of tasks in the dependency graph is a straight-forward node traversal, and the complexity is $O(E)$, where E is the number of edges in the graph. Finding max-depth in the case of the task dependency graph is thus linear in the size of the graph. Likewise, computing depth ranges requires iterating over the outgoing edges of every task in the graph, resulting in similar complexity.

3.5.2 ILP Instance

This section introduces the ILP instance used to find the optimum power bound assignment π, which assigns a power bound to every task in the dependency graph. Refer to Figure 3.6, which shows how depth ranges reduce blackouts. Note that the figure does not represent the actual execution times of all tasks. For instance, the execution time of tasks $J_{\_,1}$ in Figure 3.4 are 2, 3, and 1, respectively. This implies a blackout will exist at node 1 between 2 and 3 time units, and at node 3 between 1 and 3 time units. Figure 3.7 illustrates how

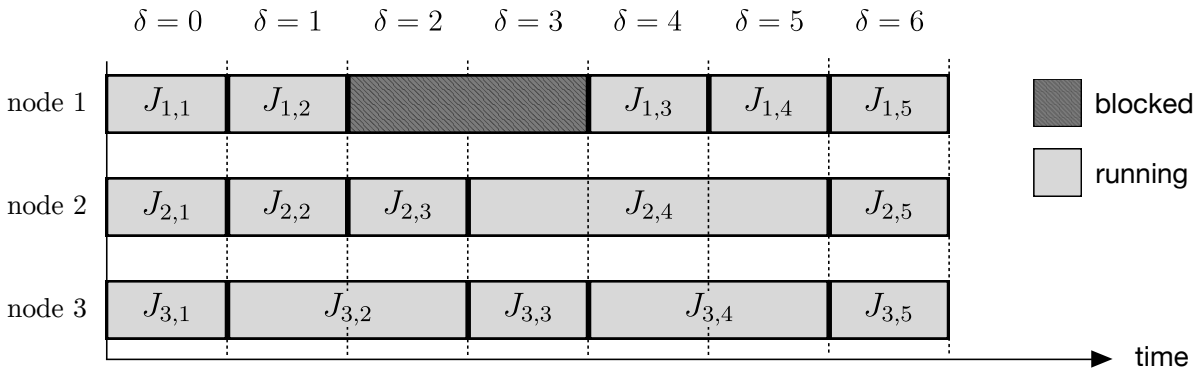


Figure 3.6: Task concurrency after applying depth ranges.

such blackouts would occur.

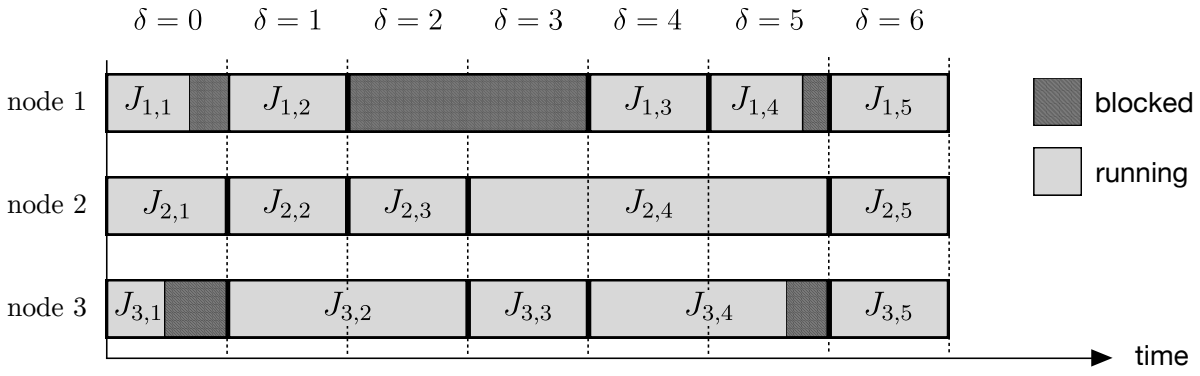


Figure 3.7: Blackouts in execution due to difference in execution time.

Solving the ILP instance produces a set of power-bound assignments to tasks that minimizes these blackout periods. The ILP instance is detailed as follows.

Variables. The ILP instance abstracts the range of power bounds that can be applied to a certain node into a finite set of power bounds that map to operating frequencies of the node's CPU. This is a reasonable abstraction since any CPU supports a finite set of operating frequencies, and we can hypothetically determine the power ceiling of the node when operating at every respective frequency. Hence, we introduce the following variables:

- *(Task-to-power-bound-assignment $x_{j,b}$)* This is a binary variable that indicates whether

task j is assigned to power bound b. In a dependency graph consisting of 10 tasks, where each task can operate at 5 different power bounds, the ILP model will contain 50 task-to-power-bound-assignment variables.

- *(Maximum execution time t)* This variable represents the maximum time it takes any node in the system to finish execution.

Constraints. The model employs 3 types of constraints.

- *(Unique power bound assignment)* These constraints ensure that no task is assigned to two different power bounds. There is one such constraint per task in the graph.

$$\forall j : \sum_{b} x_{j,b} = 1$$

- *(Cluster power bound enforcement)* These constraints enforce the power bound $\mathbb{P}$ on the entire cluster. Generating these constraints relies on the output of the task concurrency optimization algorithm (refer to Figure 3.6). Every depth level column indicates which tasks will execute concurrently. For instance, task $J_{3,2}$ is concurrent with $\{J_{2,2}, J_{1,2}\}$ at depth level $\delta = 1$. It is also concurrent with $J_{2,3}$ at depth level $\delta = 2$. Hence, there is one power bound enforcement constraint per depth level in the graph:

$$\forall \delta : \sum_{\delta_j} x_{j,b} \times b \leq \mathbb{P}$$

where δ_j is a set that contains any task for which δ is within its depth range:

$$\delta_j = \{J \mid \delta \in \Delta(J)\}$$

- *(Maximum execution time)* These constraints ensure that no node executes beyond the maximum execution time variable t, which is the variable to be minimized.

$$\forall i : \sum_{j \in \mathcal{J}_i} \sum_{b} x_{j,b} \times \tau(j,b) \leq t$$

where i indicates the node, $\mathcal{J}_i$ is the sequence of tasks in that node, j is a task in that sequence, and $\tau(j,b)$ is the execution time of task j under power bound b.

The total number of constraints in the model is the result of the following formula:

$$\sum_i |\mathcal{J}_i| + \max_J\{\delta(J)\} + n$$

where n is the number of nodes.

Optimization objective. Finally, the objective of the model is to minimize the maximum execution time: min t

3.6 Simulation results

To validate the intuition behind our model, the ILP solution, and the online heuristic, we implement a simulator to calculate the total execution time of an MPI program. The simulator is initialized with the following:

- a text file detailing the task dependency graph,
- a cluster power bound, and
- the type of simulation: *Equal-share*, *ILP*, or *Heuristic*.

The *Equal-share* simulation assigns equal power bounds to all nodes in the cluster. The *ILP* simulation first solves the power assignment problem for an optimal (or nearly optimal due abstractions) solution, and then simulates execution using the resulting task-to-power-assignments. The *Heuristic* simulation applies the online power distribution algorithm.

Figure 3.8 shows the results of simulating the dependency graph in Figure 3.4. The power-to-frequency lookup values, as well as the execution time of tasks at different CPU frequencies are measured on an Arndale Exynos 5410 ARM board. The results indicate that the ILP-based solution excels at the lower power bounds, producing a 2.5 speedup versus equal-share. The heuristic also produces a significant speedup of 2 at lower power bounds. The improvement for both ILP-based solution and the heuristic decreases until it matches the execution time of equal-share as the power bound is relaxed. This is expected since at a relaxed power bound the nodes are already operating at their maximum frequencies.

These results are based on the assigned execution times in the task dependency graph in Figure 3.4, which are completely synthetic. To add some notion of ground truth to the

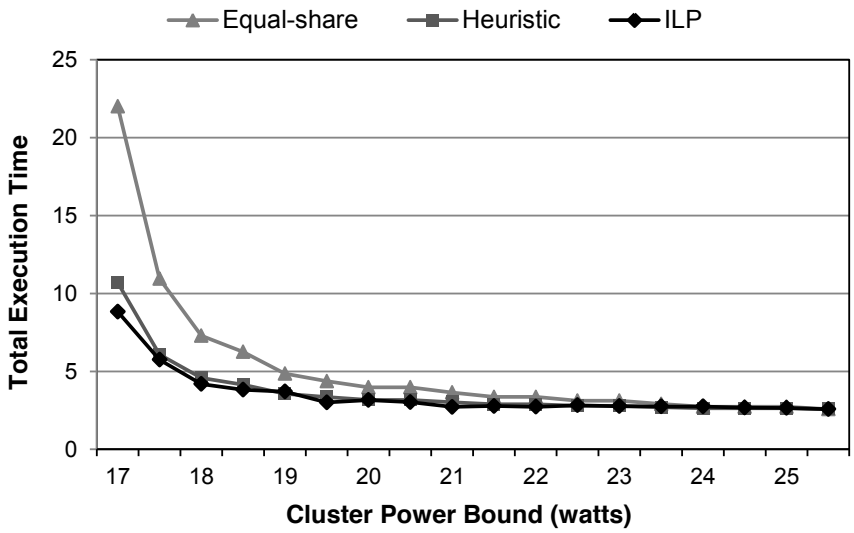


Figure 3.8: Simulation results of the dependency graph in Figure 3.4.

simulations, we rerun the simulation given that the execution times of all tasks is the same. Hence, no bias exists that would favor a power distribution alternative to equal-share. In such a case, the ILP-based solution still outperforms equal-share at lower power bounds, producing a speedup of 2, while the heuristic speedup is 1.64. The improvement comes from the fact that the ring communication pattern forces blocking in the equal-share distribution, even when the execution times of tasks is the same. This is improved significantly by applying the *stretching* of tasks across multiple depth levels, and distributing power optimally on running nodes.

In light of these results, we construct a set of experiments based on the same dependency graph, yet varying in execution times. We quantify the variation in execution times using the standard deviation of execution times of individual tasks. Hence, the experiments present synthesized execution times to target specific standard deviations. The standard deviation starts at 0 and increases till 6, given a mean of 10 time units. Figure 3.9 illustrates the speedup gained by running the heuristic and the ILP solution, given the minimum possible cluster power bound. The figure shows a trend of increasing speedup as the variation increases, which confirms our intuition that our algorithms excel when execution times exhibit more variability. Yet, at high variability, speedup becomes unstable since it is heavily dependent on the specific execution times assigned to tasks.

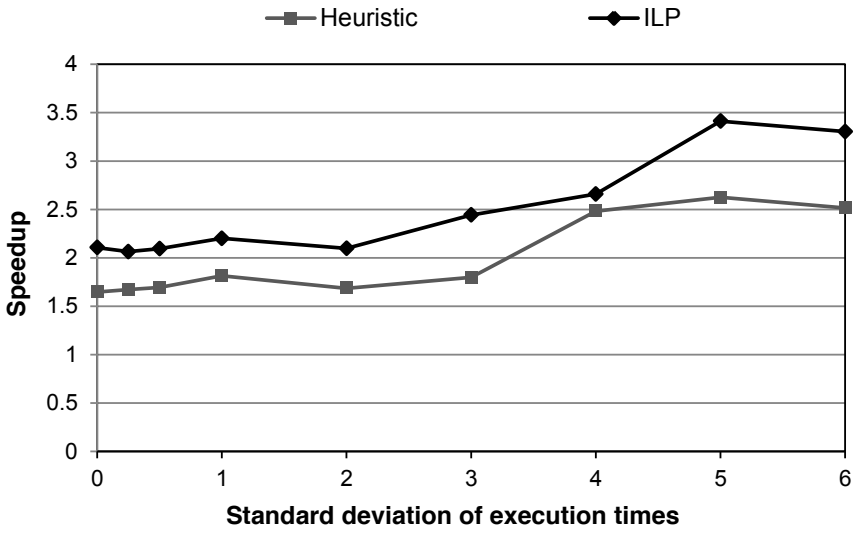


Figure 3.9: Simulation results using different standard deviations of execution times.

3.7 Implementation and Experimental Results

3.7.1 Implementation

The model proposed in this chapter can generally map to clusters that exhibit task dependency. MPI is a widely used interface for managing synchronization and communication among tasks. To demonstrate how our model operates in real-life environments, we implement an MPI wrapper with underlying logic to perform the functionality of the Block Detector (see section 3.4.1). The power distribution controller is implemented as a standalone lightweight UDP server that receives report messages and responds with distribute messages.

MPI wrapper

The MPI wrapper is designed to intercept MPI calls and deduce whether the node will be blocked or unblocked. Currently the wrapper supports MPI_Send, MPI_Recv, MPI_BCast, MPI_Wait, MPI_Scatter, MPI_Reduce, and MPI_AlltoAll.

Report Manager

A report manager is responsible for buffering report messages to the power distribution controller to avoid thrashing the CPU with frequency changes. It applies a timeout mechanism that uses the breakeven solution to the popular ski-rental problem [76]. Figure 3.10 demonstrates how the timeout is calculated for a hypothetical MPI_BCast call.

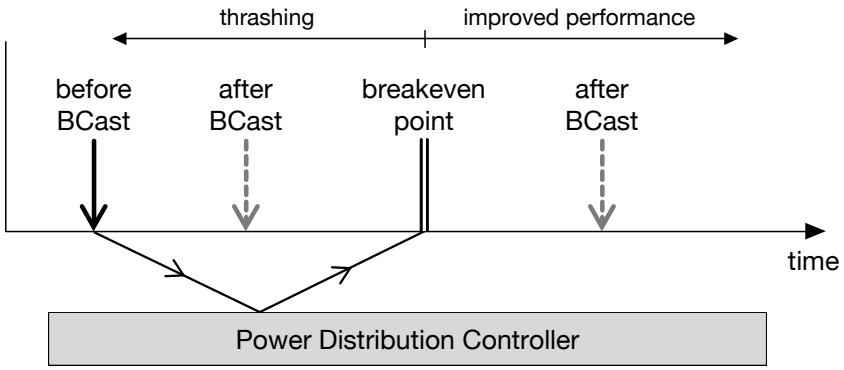


Figure 3.10: The breakeven point at which the report manager checks the buffer.

The wrapper is missing implementations for MPI_IRecv and other asynchronous functions. Also, it lacks support for multiple communicators, which would require a hierarchical approach to power distribution.

3.7.2 Experimental setup

To validate the heuristic in practice, we design a set of extensive experiments to study the performance of the heuristic in different circumstances. We attempt to cover multiple use cases by varying the hardware setup and the software benchmark.

Cluster setup

The use of embedded boards to build clusters is rapidly becoming a trend. Boards such as Raspberry Pi, BeagleBone, etc. are orders of magnitude cheaper than large-scale Intel-based racks. Embedded boards are also much less power hungry and require significantly less power for cooling. Recent work in [34] and [121] demonstrates the use of some of these boards in constructing an HPC cluster. In fact, ARM has recently gained traction in the

HPC domain as a power-efficient contender to Intel [21,134]. Thus, building heterogeneous clusters of cheap embedded nodes drastically reduces the operational cost, yet it introduces challenges in managing power within such diverse clusters.

To that end, we setup a 12 node heterogeneous cluster of ARM based boards. The nodes are as follows:

- Arndale Exynos 5410, hosting a dual-core 1.7GHz A15 CPU.
- odroid-x, hosting a quad-core 1.8GHz A15 CPU.
- 2 BeagleBone boards, hosting a AM335x 720MHz ARM Cortex-A8 CPU.
- 2 BeagleBone Black boards, hosting a AM335x 1GHz ARM Cortex-A8 CPU.
- 6 Raspberry Pi boards, hosting a 700MHz ARM CPU.

Heterogeneity also extends to the OS installed on the nodes. The Arndale board runs linaro ubuntu trusty (14.04), the odroid runs linaro ubuntu raring (13.04), the the BeagleBone boards run Debian, and the Raspberry Pi runs Raspbian. This selection of varying manufacturer, CPU capabilities, and OS and kernel versions mimics what would be available at a larger scale in heterogenous clusters. All nodes use MPICH 1.4.1, and are connected to an Extech 380803 Power Analyzer that measures their collective power consumption.

Benchmarks

We run 3 benchmarks in the NAS Parallel Benchmark suite (NPB). The benchmarks are as follows:

- **IS**. An integer sort benchmark that is memory intensive.
- **EP**. Embarrassingly parallel benchmark that is CPU intensive.
- **CG**. The conjugate gradient benchmark that is communication intensive.

The benchmarks present a diverse set of applications that demonstrate the behavior of the heuristic when the load is dominantly CPU bound, memory bound, or communication bound. For every benchmark, we run a standard problem size as defined by NPB (size class A).

Experimental Factors

The experiments are based on combinations of the following experimental factors:

Active nodes. To demonstrate the scalability of the heuristic in increasing cluster sizes of 2, 4, 6, 8, 10, and 12 nodes. The setups are: 2 Raspberry Pi, 4 Raspberry Pi, 6 Raspberry Pi, 6 Raspberry Pi plus 2 BeagleBone Black, then adding 2 BeagleBone, and then finally adding Arndale and odroid for a total of 12 nodes.

Power distribution mode. Every experiment is executed with one of the following power distribution schemes in place:

- **Median**. Each node receives its median power level. This simple approach ensures that every node is operating at a medium level, and it naturally satisfies the power bound which is the sum of medians. We chose median over equal-share since our cluster has nodes with non-intersecting power spectrums, and thus a median distribution is a more reasonable approach versus a mean (equal-share) that is biased by extremes.
- **Heuristic (Online)**. Our heuristic is responsible for managing power distribution within the power bound.
- **ILP (Offline)**. The ILP solution is used to determine the power level to assign to every task in every node.

Finally, every experiment is executed 3 times to ensure that the results are not outliers.

3.7.3 Experimental Results

Figure 3.11 shows the results of executing the EP benchmark on the different cluster configurations. The first three sets of columns represent results from a homogeneous cluster of Raspberry Pi boards. As can be seen, the heuristic manages to produces some speedup yet it is not as prominent as the latter three columns. This is attributed to the homogeneity of the cluster, since all nodes are identical and are executing the same workload. However, there is still variance in execution time among identical nodes. The heuristic distributes power within this period of variance to the lagging nodes. This results in a speedup up to 1.16x. As for ILP, it has surprisingly lower speedup than the heuristic. Upon deeper inspection, we discovered that the heuristic has an advantage over ILP: it can change a task's frequency mid-execution. This allows it to achieve a tighter asymptotic power

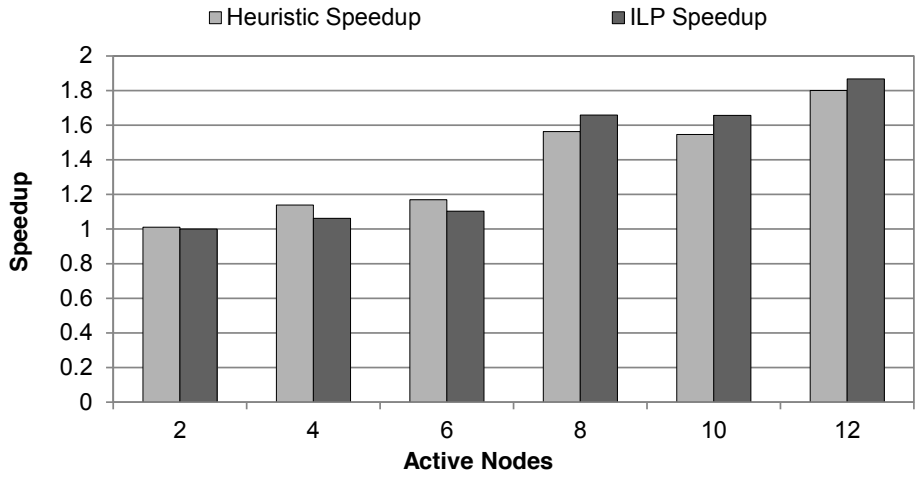


Figure 3.11: Heuristic speedup versus ILP speedup of the EP benchmark.

consumption relative to the power bound. It is only apparent when the number of nodes is low and the nodes are homogeneous, making this subtle difference the major deciding factor.

The latter three columns in Figure 3.11 demonstrate an increasing trend in both heuristic and ILP speedup. The addition of 2 BeagleBone Blacks in the 8 node configuration causes a strong rise in speedup, since these boards are more powerful than Raspberry Pis, running at 1000MHz. At 10 nodes, speedup takes a slight dip due to the addition of 2 weak BeagleBones. Finally, the addition of the multicore Arndale and odroid causes a strong rise in the speedup once again.

Figure 3.12 presents the heuristic speedup for both the CG and IS benchmarks. Note that the ILP speedup has been omitted since these benchmarks have a high number of synchronization points, and therefore the ILP problem becomes extremely large (tens of thousands of variables) and finding a solution can take days. Both benchmarks can only execute on a number of nodes that is a power of 2. The results of the heuristic with respect to the CG benchmark show a fluctuating speedup of approximately 1.0. This lack of improvement is due to the high volume of communication in the CG benchmark, and the sparsity of long running CPU bound tasks. The heuristic actually causes a speed-down at 4 active nodes, yet it is a negligible value. This suggests that the heuristic is better suited for CPU bound applications and has minimal negative impact on communication heavy applications. The heuristic performs better in the IS benchmark, yet since it also involves frequent communication and lacks the long running heavy computation of EP, the speedup is not as impressive as in EP.

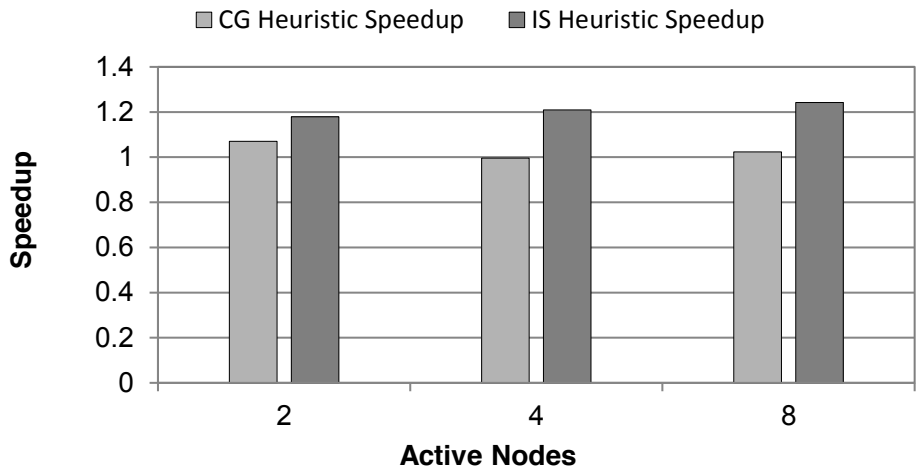


Figure 3.12: Heuristic speedup of the CG and IS benchmarks.

3.8 Summary

In this chapter, we tackled the problem of optimizing the performance of distributed programs on heterogeneous embedded clusters subject to a power bound. There is little work on this problem in the literature, but we argue that given energy constraints of data centers and the increasing demand for computing power, we are in pressing need to address the problem. We then introduced a formulation of the power distribution problem, and an online heuristic that detects when a node is blocked and, subsequently, redistributes its power to other nodes based on a ranking algorithm. We then presented an ILP-based solution to obtain the optimal job-to-power-bound assignment, which we use to evaluate the performance of our heuristic. We validated the approach using simulation and actual experiments. Our online heuristic produces a speedup of up to a factor of 1.8, specially in CPU bound programs, while it is ineffective in communication intensive applications.

Chapter 4

Power, Energy and Bandwidth Tradeoffs

In the age of exascale computing, it is crucial to provide the best possible performance under power constraints. A major part of this optimization is managing power and bandwidth intelligently in a cluster to maximize performance. There are significant improvements in the power efficiency of HPC runtimes, yet no work has explored our ability to use power as a control mechanism for mitigating bandwidth bottlenecks. In this chapter, we explore using power as a means to artificially stagger parallel tasks to avoid bandwidth bottlenecks. Moreover, we construct a model based on network flow principles that allows us to efficiently bound the makespan of the parallel program by optimizing power and bandwidth distribution. We demonstrate that our model is extensible and supports both renewable and non-renewable resources.

4.1 Introduction

In the previous chapter we introduced the power distribution problem. The problem is essentially identifying an optimal assignment of task power bounds to minimize the execution time of the the entire program (modelled as a task graph). We then introduced an integer program to provide a theoretical bound on the performance, which is synonymous with the minimum possible execution time. Unfortunately, the ILP is not scalable enough to support real-life deployments of parallel programs.

In this chapter we broaden the scope of the problem and provide a much simpler method of establishing a theoretical bound on performance. Rather than focusing solely on power

as the scarce resource, we use *network flow* algorithms to bound performance across any combination of *renewable resources*. Non-renewable resources such as energy, processors and disk space endure over time: if some amount of disk space is unused at a given moment, it can be saved and used at a later time. Renewable resources such as power and bandwidth are only useful in the moment: they cannot be banked to be used later. If a processor is capable of running at 120 Watts but has only enough work to consume 60 Watts, the extra watts cannot be "saved" for later.

While renewable resources cannot be moved in time, they can often be scheduled in space. Where multiple nodes share one communication link, temporal bandwidth limits can be established to optimize throughput or execution time. Where a job-wide power bound exists, power can likewise be scheduled so that the critical path of execution runs as fast as possible and no other path becomes longer than critical.

Because this problem is one of temporal scheduling we must work under a constraint that the sum of any renewable resource in the system never exceed the total amount of that resource available. Running under an MPI model of communication, however, does not provide a complete ordering of tasks. As with Lamport's logical clocks, node synchronization via message passing can establish a before-and-after temporal relationship, as can the ordering of tasks on a single processor. The usefulness of network flow lies in the fact that, so long as renewable resources are scheduled at MPI communication calls, we are guaranteed to never arrive at an infeasible solution. While the theoretical optimal may require finer-grained scheduling, the network flow heuristic in practice gets us very close to the optimal without having to resort to computationally infeasible ILP approaches. And because we are not relying on power-specific features, we are able to create near-optimal schedules given constraints on multiple renewable resources.

The rest of the chapter is organized as follows: Section 4.2 introduces the problem statement. Section 4.3 presents the proposed linear programming model. Section 4.4 demonstrates how to use the model to represent MPI programs. Section 4.5 illustrates an example of an MPI program modelled and optimized using our LP. Section 4.7 presents our experiments on benchmarking the LP, and Section 4.8 explores the practical improvement of communication time when using an LP suggested resource distribution. Finally, Section 4.9 summarizes our approach.

4.2 Problem Statement

Much attention has been paid to the problem of scheduling under a power bound, but power is not the only constrained resource in computing systems. In this work, we aim to

schedule tasks under dual constraints – namely, power and network bandwidth. Specifically, the scheduler we consider must determine not only *when* a task executes, but also determine how much power and network bandwidth will be allocated to the task. The aim is to have a scheduler that determines these things in a manner that the job completes as quickly as possible.

4.2.1 Resource-Constrained Project Scheduling

The *resource-constrained project scheduling problem* (RCPSP) addresses the issue of assigning tasks to one or more resources with the aim of meeting some stated objective. All the resources have limited availability. The challenge is to meet the objective without ever using more resources than are available. This problem has been extensively studied for over 50 years (see, e.g., [35, 36]). Variations of this problem include the following [182]:

- With or without preemption,
- With or without precedence constraints,
- Discrete time or continuous time,
- Constant or variable resource availability, and
- Single-mode or multi-mode.

The first four variations are self-explanatory. In single-mode systems, tasks' behavior is constant, while in multi-mode the behavior (e.g., the runtime) can change in different modes.

Typically, the objective function is to minimize the makespan – i.e., the completion time of the last task to complete execution. Other, non-standard, objective functions may be used instead [182]. For example, maximizing cash flow, minimizing resource consumption, and maximizing the smoothness of the resource usage are possibilities for alternative objective functions.

This work addresses minimizing the makespan with constrained power and constrained network bandwidth. We consider a multi-mode problem, as runtimes change with changes in power. We assume the bounds are constant and that tasks' resource usage remains constant over time. The time model is continuous, meaning the scheduler can be invoked at any time, not just at discrete time intervals. Finally, tasks run without preemptions and precedence constraints are present in the system.

4.3 Linear Programming Model

4.3.1 The Model

First we define two types of tasks: a computation task t^c and a network communication task t^n. We define a computation task $t^c = \langle C, P, S \rangle$ as a tuple where $C : W \to T$ is a function that maps a power bound in domain W (watts) to the duration of the task if running at the given power bound. P is the set of predecessor tasks and S is the set of successor tasks. A network communication task is defined as $t^n = \langle N, P, S \rangle$ where $N : B \to T$ is a function that maps a bandwidth bound in domain B (bps) to the duration of the task if running at the given bandwidth bound.

Next, we define the task graph as a directed acyclic graph $G = (V, E)$ where V is the set of tasks (computation and communication), and E is a set of edges represented as ordered pairs of vertices such that

$$E = \{(x, y) \mid y \in S_x\}$$

where S_x is the set of successors of task x. E^+ is the transitive closure of edges E, such that

$$E^+ = \{(x, y) \mid x \rightsquigarrow y\}$$

indicating there is a path from x to y. To incorporate nodes reachable by transitive closure in the predecessors and successors of a task, we define the following:

$$P_t^+ = \{t' \mid (t', t) \in E^+\}$$

$$S_t^+ = \{t' \mid (t, t') \in E^+\}$$

where P_t^+ is the set of predecessors of task t including both direct and transitive closure predecessors. S_t^+ is the set of successors of task t including both direct and transitive closure successors.

As mentioned earlier, we use the multi-commodity network flow model to solve our problem, where the commodities are power and network bandwidth. For each edge there is a flow of power and bandwidth, denoted as $\mathcal{F}^{\rho}_{(x,y)}$ and $\mathcal{F}^{\beta}_{(x,y)}$ where (x, y) is an edge from task x to task y in E^+. For each task, there is an inflow / outflow of power and bandwidth. We denote the inflow of power to a task t as $\mathcal{I}^{\rho}_t$, and the outflow as $\mathcal{O}^{\rho}_t$. Similarly, the inflow / outflow of bandwidth is denoted as $\mathcal{I}^{\beta}_t$ and $\mathcal{O}^{\beta}_t$ respectively.

To support network flow, we add two placeholder tasks: a start task t_s and a finish task t_f. Both tasks complete in zero seconds. t_s is a predecessor of tasks and has no predecessors in G, and t_f is a successor to all tasks and has no successors in G.

The following are variables in the model:

- s_t is the start time of task t,
- d_t is the duration of task t,
- μ_t^w is the percentage of task t's work executed at power bound w,
- v_t^b is the percentage of task t's data transmitted at bandwidth b.

Finally, the model uses the following constants:

- $\mathbb{P}$ is the power bound enforced over the entire system.
- $\mathbb{B}$ is the bandwidth bound enforced over the entire system.

4.3.2 Constraints

Conservation of flow

The first set of constraints enforce the flow conservation rule.

$$\forall\, t \in V \cup \{t_f\} : \sum_{t' \in P_t^+} \mathcal{F}^{\rho}_{(t',t)} = \mathcal{I}^{\rho}_t \tag{4.1}$$

$$\forall\, t \in V \cup \{t_s\} : \sum_{t' \in S_t^+} \mathcal{F}^{\rho}_{(t,t')} = \mathcal{O}^{\rho}_t \tag{4.2}$$

$$\forall\, t \in V \cup \{t_s, t_f\} : \mathcal{I}^{\rho}_t = \mathcal{O}^{\rho}_t \tag{4.3}$$

That is, the sum of all flow of power entering a task is equal to the flow exiting the task to its successors. Constraints 4.1, 4.2 and 4.3 are defined in the same way for bandwidth (replacing ρ with β).

To enforce the entire graph to obey the power and bandwidth bounds, we constrain the placeholder tasks as follows:

$$\mathcal{I}^{\rho}_{t_s} = \mathcal{O}^{\rho}_{t_f} = \mathbb{P} \tag{4.4}$$

$$\mathcal{I}^{\beta}_{t_s} = \mathcal{O}^{\beta}_{t_f} = \mathbb{B} \tag{4.5}$$

Task duration

The inflowing power determines the duration of a computation task, whereas the inflowing bandwidth determines the duration of a communication task. We use the variables μ_t^w and v_t^b to indicate portions of the task spent at different power levels / bandwidths. The purpose of this is to approximate the non-linear relation between power/bandwidth and task duration. Variables μ_t^w and v_t^b are constrained to add up to 1.0 such that the entire task is included in the solution:

$$\forall\, t^c \in V : \sum_{w \in W} \mu_{t^c}^w = 1.0 \tag{4.6}$$

$$\forall\, t^n \in V : \sum_{b \in B} v_{t^n}^b = 1.0 \tag{4.7}$$

The duration of a computation task is calculated as follows:

$$\forall\, t^c \in V : d_{t^c} = \sum_{w \in W} C_{t^c}(w) \times \mu_{t^c}^w \tag{4.8}$$

Similarly, the duration of a network communication task is calculated as follows:

$$\forall\, t^n \in V : d_{t^n} = \sum_{b \in B} N_{t^n}(b) \times v_{t^n}^b \tag{4.9}$$

As mentioned earlier, the duration of placeholder tasks is zero: $d_{t_s} = d_{t_f} = 0$.

The effective power/bandwidth used is the weighted average of all power levels/bandwidths utilized in the task's execution.

$$\forall\, t^c \in V : \sum_{w \in W} w \times \mu_{t^c}^w = \mathcal{I}_{t^c}^{\rho} \tag{4.10}$$

$$\forall\, t^n \in V : \sum_{b \in B} b \times v_{t^n}^b = \mathcal{I}_{t^n}^{\beta} \tag{4.11}$$

Task start time

To ensure that the resulting schedule obeys task precedence, we use constraints to assign start times to the tasks.

$$\forall\, t \in V \cup \{t_f\} : s_t \geq \max_{t' \in P_t} \{s_{t'} + d_{t'}\} \tag{4.12}$$

These constraints ensure that the start time of a task is at or after the time the last predecessor finishes execution. As an initial condition, $s_{t_s} = 0$.

Objective

Finally, the objective of the LP model is to minimize the start time of the placeholder final task, i.e. minimizing the makespan of the schedule.

$$\min s_{t_f} \tag{4.13}$$

4.4 Modelling MPI programs

In this section, we demonstrate how to use the task graph model in Section 4.3.1 to represent simple MPI programs.

4.4.1 `Send/Recv`

Our first example is a two rank blocking send/receive (Figure 4.1). As mentioned earlier, t_s and t_f are placeholder tasks to control power/bandwidth flow. t_1 and t_2 are computation tasks that ranks 0 and 1 perform at startup respectively. Then, rank 0 initiates a blocking send, represented by network communication task t_3. t_4 represents the receive operation performed by rank 1, which is dependent on the send operation. Finally, the program terminates after receive is complete.

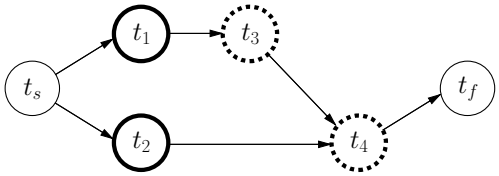


Figure 4.1: Simple two rank blocking send / receive. Tasks with a thick solid border are computation tasks, tasks with a thick dotted border are communication tasks, and tasks with a thin border are placeholder tasks.

4.4.2 `ISend/IRecv`

Next, we demonstrate how to model non-blocking communication (Figure 4.2). In this example, both ranks perform computation at startup, represented by tasks t_1 and t_2. Then,

rank 0 initiates an asynchronous send (t_3) and continues computation represented by task t_4. Similarly, rank 1 initiates an asynchronous receive (t_5) and continues computation represented by task t_6. Both ranks then wait for communication completion in tasks t_7 and t_8 respectively.

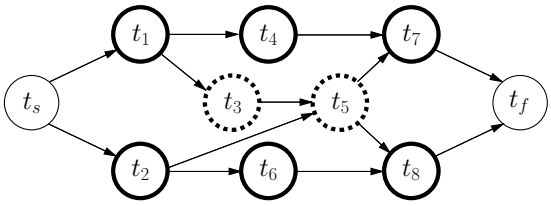


Figure 4.2: Simple two rank non-blocking send / receive.

4.4.3 `IAlltoAll`

Figure 4.3 illustrates the task graph of a two-rank non-blocking all-to-all. In this example, both ranks send (t_3, t_4) and receive (t_7, t_8) data asynchronously, while performing computations in parallel (t_5, t_6). Finally, both ranks wait until the all-to-all operation is complete (t_9, t_{10}) before terminating.

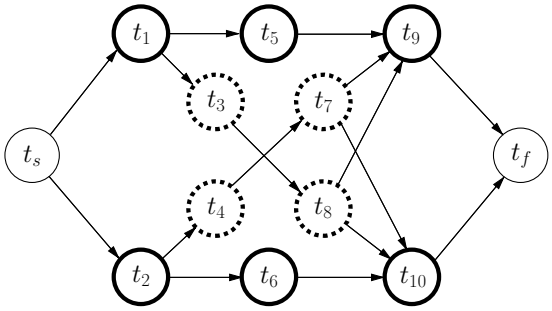


Figure 4.3: Simple two rank non-blocking all-to-all.

4.5 LP Model Example

In this section, we demonstrate how the LP model can be used to identify a schedule of the example MPI program in Section 4.4.3.

Table 4.1: The execution time of computation tasks in Figure 4.3 at different power bounds.

Task \ Power (watts)	50	60	70	80
t_1	60s	22s	15s	13s
t_2	71s	35s	22s	15s
t_5	50s	29s	18s	14s
t_6	40s	25s	15s	10s
t_9	45s	22s	16s	14s
t_{10}	45s	22s	16s	14s

Table 4.2: The execution time of communication tasks in Figure 4.3 at different bandwidths.

Task \ Bandwidth (mbps)	40	60	80	100
t_3, t_4, t_7, t_8	25s	12.5s	5s	2.5s

4.5.1 Assumptions

Let us assume that the two ranks in Figure 4.3 run on two nodes connected on the same switch. The power bound for this two-node cluster is 150w. The bandwidth bound is 100mbps. Table 4.1 shows the execution time of the computation tasks in Figure 4.3 at different power bounds. Table 4.2 shows the execution time of the communication tasks in Figure 4.3 at different bandwidths.

4.5.2 Network flow solution

We identify a schedule for the `IAlltoAll` example using the network flow representation in Section 4.3. We use the CPLEX solver to identify an optimal makespan for the model we provide. Based on the parameters of the example, the optimal makespan is 45.62s.

4.5.3 Proposed schedule

Figure 4.4 illustrates the optimal schedule found by the LP solver. In the figure, each block represents a task. Timing is indicated from left to right and scaled using the axis at the bottom of the figure. For instance, task t_1 starts at time 0s, and ends at approximately 14s. The makespan of the schedule is 45.62s. Computation tasks indicate how much power they receive at the left side of the respective block. Similarly, communication tasks indicate how much bandwidth they receive. The top row contains node 1 computation tasks (t_1, t_5, and t_9). The second row contains node 1 communication tasks (t_3 and t_7). The third and forth rows contain node 2 computation and communication tasks respectively.

As can be seen in the solution, power is 100% utilized across the two nodes (tasks t_1, t_5, t_9 versus t_2, t_6, t_{10}. Send tasks fully utilize bandwidth (t_3 and t_4), while receive tasks only consume enough bandwidth to complete transmission before t_9 and t_{10} which depend on the received data. The most notable example of flow is that from t_4 to t_7 which increases the bandwidth available for node 1, aiding in completing communication before the data is required.

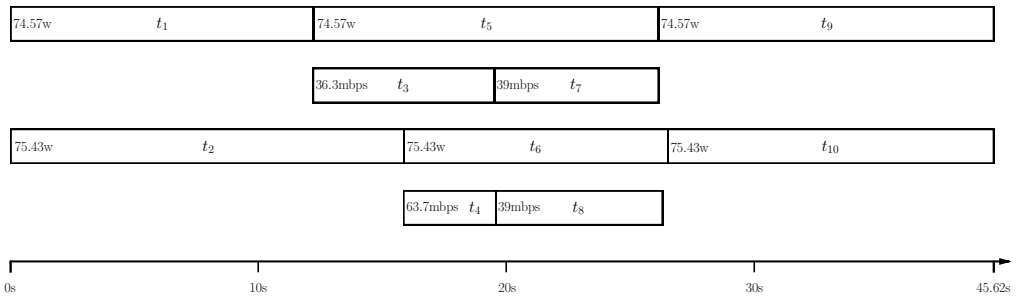


Figure 4.4: A schedule for the `IAlltoAll` example.

4.6 Advanced modelling

The network flow model can be used to model more complex scenarios. We utilize the multi-commodity aspect of the network flow to model more realistic networking setups as well as supporting resources other than power and networking. The following subsections detail some examples of such models.

4.6.1 Modelling local bandwidth bottlenecks

The model in Section 4.3 applies a single bandwidth bound $\mathbb{B}$ on the entire system. This is most probably a loose approximation of a realistic network configuration. In reality, bandwidth constrains a subgroup of nodes in the cluster connected directly to the same physical switch (for instance Infiniband on the same rack).

To model these switches individually, we introduce bandwidth resources specific to each switch. Thus, communication tasks do not share a single bandwidth bound over the entire cluster. In fact, they share individual bandwidths available to each sub group.

Let $\mathcal{S}$ be a set of switches $\{s_1, s_2, \cdots, s_k, \cdots\}$. Let $\sigma(t^n)$ denote the switch on which t^n transmits data. Let $\mathcal{B}_k$ be the bandwidth available on switch s_k. In Section 4.3, the placeholder start task t_s received $\mathbb{P}$ as the input power bound, and $\mathbb{B}$ as the input bandwidth. To model switches individually, t_s must receive $\mathcal{B}_k$ for all $s_k \in \mathcal{S}$. The conservation of flow constraints 4.1, 4.2, 4.3 are repeated for each $\mathcal{B}_k$. Constraint 4.11 is modified as follows:

$$\forall\, t^n \in V : \sum_{b \in B} b \times v^b_{t^n} = \mathcal{I}^{\sigma(t^n)}_{t^n} \tag{4.14}$$

such that the weighted average of $v^b_{t^n}$ is equal to the inflow of bandwidth in t^n's switch $\sigma(t^n)$. That forces the task to use only bandwidth provided by its designated switch.

4.6.2 Modelling interconnection network topologies

Depending on the interconnection network topology, data can be transferred across different layers of switches until it reaches its destination. For instance, a fat tree interconnection network will transfer data up the tree into switches with larger bandwidth and down again until the data is delivered to the receiving node. Such topologies can be modelled as explained above: using separate tasks for data transmission on individual switches. These tasks only utilize the bandwidth of their respective switches, and all bandwidths of all switches are inputted into the network through the placeholder start task t_s. Thus, we model each switch as a *commodity* that flows through the network, only being utilized by tasks that use this commodity.

4.6.3 Modelling other renewable resources

Using the same method described above, more renewable resources can be modelled using the multi-commodity network flow. For instance, disk I/O whether used for reading /

writing job input / output, or for checkpointing, can be a commodity flowing through the network. The same applies to tasks that utilize GPUs, where task duration will be affected by power, number of kernels used, contention on the PCI Express bus, and other configurable parameters.

4.6.4 Modelling non-renewable resources

Our model also supports non-renewable resources such as energy or money. A user could provide an energy consumption bound on the entire job, or a monetary bound on the cost of processing the job in paid cloud computing services. Such resources, such as the energy bound or the monetary budget can be modelled by modifying the constraints in Section 4.3. To demonstrate how the model changes to support renewable resources, we will use energy as an example. The model is modified as follows:

- Similar to non-renewable resources, energy flows through the network. This implies adding flow variables ($\mathcal{F}^{\epsilon}_{(x,y)}$) between vertices, as well as inflow and outflow variables ($\mathcal{I}^{\epsilon}_{t}$ and $\mathcal{O}^{\epsilon}_{t}$).
- The pLaceholder start task receives the energy bound as its energy inflow:

$$\mathcal{I}^{\epsilon}_{t_s} = \mathbb{E}$$

- The placeholder task constrains the outflowing energy to be greater than zero, preventing the network from using energy beyond the available budget:

$$\mathcal{O}^{\epsilon}_{t_f} \geq 0$$

- Conservation of flow is modified to be positive only:

$$\forall\, t \in V \cup \{t_s, t_f\} : \mathcal{I}^{\epsilon}_{t} \geq \mathcal{O}^{\epsilon}_{t}$$

- The energy used by a task is defined as follows:

$$\forall\, t \in V : \mathcal{E}_t = \mathcal{I}^{\epsilon}_{t} - \mathcal{O}^{\epsilon}_{t}$$

Energy can then be used to determine the duration of the task. A non-linear relationship can be modelled using piece-wise linear formulations by creating variables to denote the percentage of a task that is run at a specific energy budget. The energy consumed by the task is the weighted average, similar to Constraint 4.11.

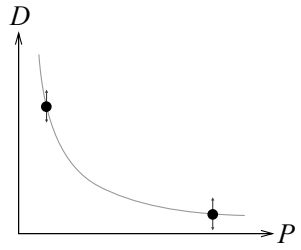


Figure 4.5: Randomizing task execution time using the inverse relationship between power and duration.

4.7 LP Model Benchmark

In this section we introduce our experiments on the scalability of the LP model and the improvement it provides over a trivial schedule.

4.7.1 Experiment Setup

Model

The model is based on the `IAlltoAll` example in Figure 4.3. We modify the example such that is configurable by number of ranks, number of synchronization points (number of iterations an `IAlltoAll` is executed), and the size of the data being transmitted at each iteration.

We introduce variability in the size of parallel tasks, which models real-life variability across nodes even in homogeneous clusters. We model this by assuming $P \propto\!\!\!\sim \frac{1}{D^2}$ where P is the power bound of the task, and D is the duration. Figure 4.5 shows how we generate random task execution times. The two anchor points on the graph are drawn from normal distributions and the curve is adjusted to pass through the new random points. Then, we can use piece-wise linear approximation to model the whole curve. The two points represent the task duration at the lowest and highest allowable power bounds respectively. The normal distributions have a tight variance of $\mu \pm 10\%$.

Environment

We use the python Pulp library for modelling coupled with the CPLEX solver.

Parameters

We experiment with 2, 4, 8, 16, and 32 ranks, 1, 3, and 5 synchronization points or iterations, and finally 10, 100, and 1000 MB of data transmitted at each iteration. We generate 5 random problems for each configuration.

Metrics

We study the solution time, the makespan of the resulting schedule, as well as the makespan of a schedule that evenly distributes resources.

4.7.2 Results

Figure 4.6 shows the solution time of different problem sizes. As expected, the growth in the number of variables increases the solution time exponentially. The largest problem we experimented with had 20800 variables, and was solved in approximately 26 minutes. Note that the solution times are overlayed at ranks 4 and above due to having very close values.

Figure 4.7 shows the speedup gained by using the LP model. We compare our LP against a version of the LP that assigns equal power bounds and bandwidth bounds to all ranks. As shown in the figure, speedup is mostly between 5% and 10%. Note that the variation between parallel tasks is limited to within 10% of the mean. Higher variation will further disadvantage a simple equal share schedule. Similarly, tighter power/bandwidth bounds will increase the benefit of power/bandwidth distribution.

Figure 4.8 shows the speedup grouped by the size of transmitted data. The figure shows that a smaller data size results in better improvement. We examined the resulting schedules, and discovered that data transmission of 10MB is too small to change the entire schedule. Refer to Figure 4.3. Since the communication tasks are very short, they are never on the critical path, making t_9 and t_{10} rely on the time t_5 and t_6 complete. Hence, the improvement in the figure is solely due to power manipulations of the computation tasks. At 100MB, the LP tries to balance between power and bandwidth, resorting to staggering computation tasks to be able to efficiently manipulate the bandwidth being distributed to communication tasks. At 1000MB, communication tasks dominate the schedule and there is little staggering that could be done using power that would improve the makespan.

The utility of staggering is demonstrated in Table 4.3. The table demonstrates the start and end times of the computation and communication tasks in a single iteration of

Table 4.3: Solution of a single iteration of an 8-rank AlltoAll program.

Rank	t^c start	t^c W	t^c end	t^n start	t^n mbps	t^n end
0	0.00	62.78	57.46	57.46	24.47	93.88
1	0.00	60.00	58.61	58.61	25.91	93.88
2	0.00	58.89	53.88	53.88	20.00	93.88
3	0.00	60.00	59.44	59.44	26.94	93.88
4	0.00	58.33	53.88	53.88	20.00	93.88
5	0.00	60.00	54.85	54.85	21.21	93.88
6	0.00	60.00	58.90	58.90	26.28	93.88
7	0.00	60.00	66.04	66.04	35.30	93.88

the IAlltoAll program. The computation tasks in the table are equivalent to tasks t_1 and t_2 in Figure 4.3, while the communication tasks are equivalent to tasks t_3 and t_4.

The tasks have almost the same execution times ($\pm 10\%$) and could be made to complete at the same time. However, we observe that tasks are staggered to complete at different times, after which the successor communication tasks begin execution. Communication tasks that start later receive more bandwidth, which benefits the entire schedule by reducing the time at which the next synchronization point occurs. As shown in the table, all communication tasks finish at the same time, right before the `MPI_Wait` call.

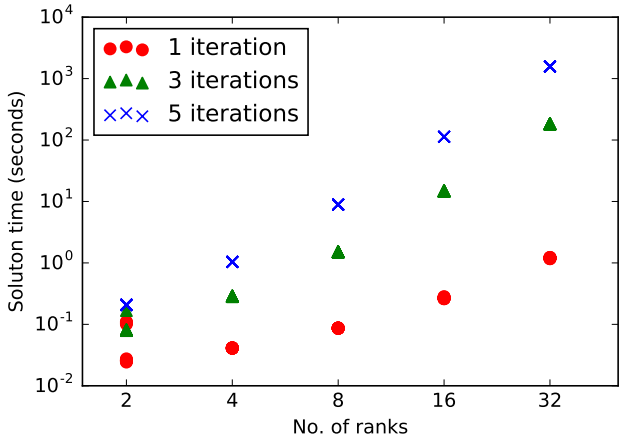


Figure 4.6: The solution time of problems of different ranks and number of iterations.

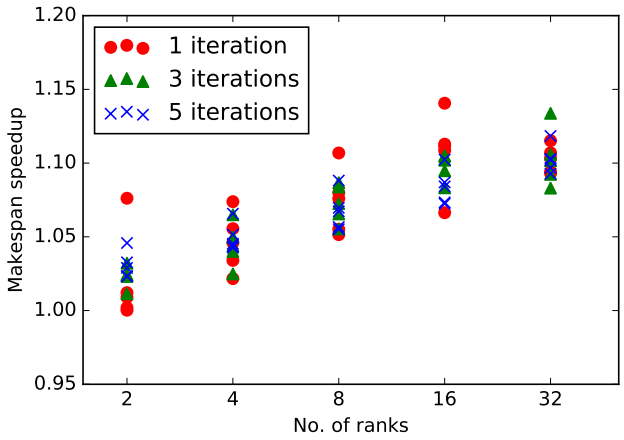


Figure 4.7: Speedup of the LP optimized schedule versus a trivial equal share power and bandwidth distribution.

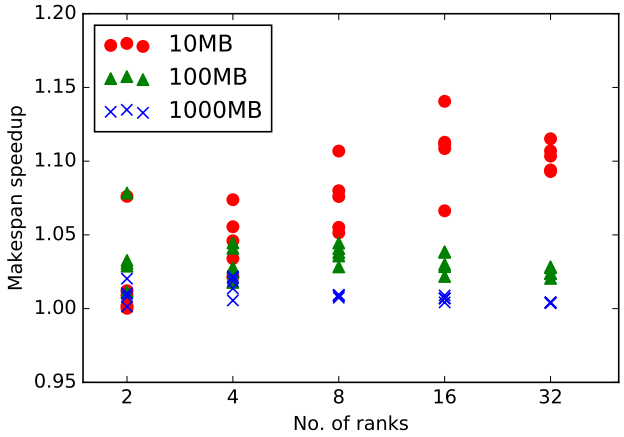


Figure 4.8: Speedup of the LP optimized schedule grouped by transmitted data size.

4.8 Synthetic MPI experiments

In this section we explore the applicability of the staggering technique proposed by the LP in real-life scenarios.

4.8.1 Experiment Design

As seen in Table 4.3, power is used to stagger the completion time of computation tasks such that the communication bandwidth is not overloaded all at once. To replicate this, we implement two synthetic MPI benchmarks that utilize the staggering suggested by the LP. The benchmarks are (1) an asynchronous ring communication program, and (2) an `IAlltoAll`. The `IAlltoAll` is based on the example we use in Section 4.7. We have also modelled a ring communication program and solved it using our LP, yielding similar results to that presented in Section 4.7.

To simulate the staggered start of communication, we use timers that trigger at different times for different ranks. These triggers initiate asynchronous communication and we record the time it takes for data to transferred. We compare these results to the communication time of a perfectly synchronized program that uses the same power bound across all nodes, and results in computation tasks finishing at the same time.

We run 1000 iterations of the ring / `IAlltoAll` programs to get reliable communication time measurements. We experiment with different transmission data sizes similar to our approach in Section 4.7. We also experiment with different number of nodes: 2, 4, 8, 16, and 32. We run each configuration 5 times.

4.8.2 Results

Figure 4.9 illustrates the observed speedup in communication time when using the staggered schedule of the LP model. This is measured as the total time it takes all ranks to send/receive data, starting from the moment `ISend` is called, through `IRecv`, and until `MPI_WaitAll`. As shown in the figure, an improvement of approximately 2.5% is observed at small sizes of transmitted data, which reaches up to 15% when each node transmits 100MB. At 10MB per node, the improvement is approximately 5%. This improvement is in line with the results of the LP. Moreover, it indicates that communication time can be reduced despite the lack of control mechanisms such as RAPL that are capable of throttling bandwidth.

Figure 4.10 illustrates the communication time speedup when studying the `IAlltoAll` example. In this example, the improvement is not consistent, and staggering can actually hurt the performance in more than half of the runs. We attribute this to the way collective operations handle late arrivals. This has been demonstrated extensively in the work by Faraj [38]. In that work, the authors demonstrate that collective operations, including `IAlltoAll` have a weak tolerance to late arrivals. This weak tolerance is manifests as

unreliable and unpredictable performance. Our working explanation is that since we are manufacturing late arrivals by staggering, we are creating a disturbance with respect to the collective operation. The MPI implementation is failing to control this disturbance at larger data sizes, causing a high variability in performance and a penalty to staggered arrival of data. Further investigation into this behavior is needed, which is our ongoing work.

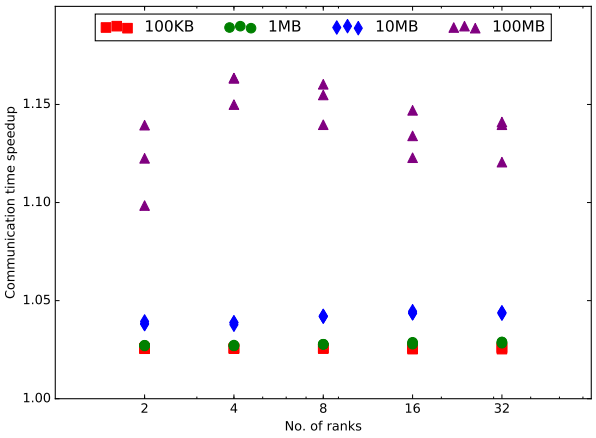


Figure 4.9: Speedup of communication time in a ring communication MPI program.

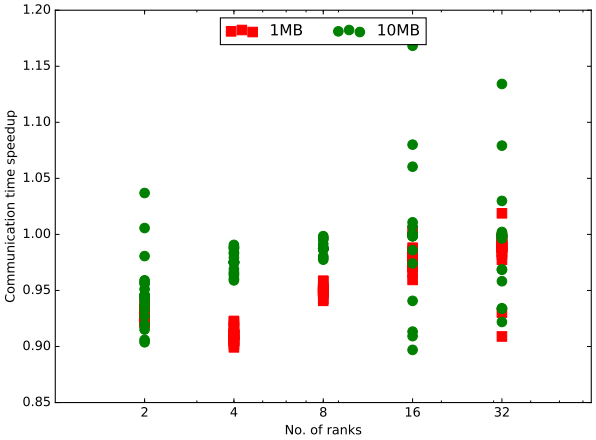


Figure 4.10: Speedup of communication time in an `IAlltoAll` communication MPI program.

4.9 Summary

In this chapter, we presented an efficient method to bound the performance of MPI applications that utilize various renewable resources. We outlined a modelling approach that utilizes the network flow paradigm to produce a relatively efficient linear program. We demonstrated how our approach can model different MPI programs, different interconnection networks, as well as different types of resources. We presented benchmarks to evaluate the performance of the linear program in producing an optimal schedule, and we validated that the output of the linear program can be used to improve the communication time via staggering. We presented examples where staggering can be useful or harmful to performance.

Chapter 5

Power, Energy and Precision

Modern embedded systems are becoming more reliant on real-valued arithmetic as they employ mathematically complex vision algorithms and sensor signal processing. Machine learning in large scale data centers heavily uses floating-point arithmetic in complex and iterative processes. Double-precision floating point is the most commonly used precision in many algorithm implementations due to the convenience of no-compromise accuracy. A single-precision floating point can provide a performance boost due to less memory transfers, less cache occupancy, and relatively faster mathematical operations on some architectures. However, adopting it can result in loss of accuracy. Identifying which parts of the program can run in single-precision floating point with low impact on error is a manual and tedious process. In this chapter, we propose an automatic approach to identify parts of the program that have a low impact on error using shadow-value analysis. Our approach provides the user with a performance / error tradeoff, using which the user can decide how much accuracy can be sacrificed in return for performance improvement. We illustrate the impact of the approach using a well known implementation of Apriltag detection used in robotics vision. We demonstrate that an average 1.3x speedup can be achieved with no impact on tag detection, and a 1.7x speedup with only 4% false negatives.

5.1 Introduction

Embedded systems are becoming more and more sophisticated as they utilize software that was traditionally only used on desktop machines. Computer vision, speech recognition, and machine learning modules are now becoming more common in embedded systems such as self-driving cars, IoT devices, and robots. One common feature of such applications

is their extensive use of floating-point arithmetic to perform computationally intensive mathematics. When technology migrates to the embedded domain, it becomes even more crucial to optimize its performance and energy efficiency. Software that traditionally ran on powerful desktop machines may now be required to run on small, battery-powered embedded devices. In this context, heavy floating-point arithmetic can become a serious obstacle in developing resource-efficient embedded software.

Naturally, domains of traditional reliance on floating-point arithmetic stand to benefit significantly from floating-point performance optimizations. The HPC community has researched various aspects of floating-point efficiency, including floating-point alternatives. Machine learning applications, now commonly deployed in large scale data centers, rely heavily on complex mathematics to construct models of massive amounts of data. The performance and energy efficiency of data centers is of the utmost importance, especially if the gains in performance and energy are made at a measured and acceptable loss of accuracy.

Double precision is commonly used in mathematically complex algorithms since it provides the highest level of accuracy available in standard hardware. However, using single precision provides a performance boost due to less memory transfer, less cache occupancy, and fewer clock cycles for some mathematical operations. Better performance implies less energy consumption, especially with modern architectures optimizing idle power significantly. A single-precision implementation that is idle for a longer period of time is more energy efficient.

The advantages of single precision can potentially come at the cost of accuracy, especially in algorithms that require complex mathematics, such as algorithms involving computer vision, speech recognition, and AI. In this chapter, we show the quantifiable impact of single-precision on the accuracy of a computer vision library [115]. In some cases, it is feasible to convert the entire code to single-precision and retain an acceptable level of accuracy. In fact, modern AI libraries provide the user with a switch to toggle between single and double precision [102]. However, as we show in this chapter, the issue is often more subtle and complicated. To maximize performance while maintaining a level of accuracy that the developer can accept, the developer has to perform the tedious process of analyzing each part of their code to identify which variables or functions can be downgraded to single-precision without significant loss in accuracy. This involves an understanding of the algorithm, its input, and floating-point arithmetic.

To tackle the aforementioned subtleties, developers need an automatic approach to guide the process of precision downgrade without drilling into the details of the code. More specifically, developers need an automatic approach to identify which parts of the

program are candidates for a downgrade from double to single precision. Candidates are parts of the program (variables, functions, instructions) which when downgraded result in an acceptable loss of accuracy. Such an automatic approach should aid the developer in managing the performance/error tradeoff, by providing information that can tell the user how much performance they can gain versus how much accuracy they can lose.

In this chapter, we present a novel automatic approach for managing the performance/error tradeoff by identifying parts of the program that can tolerate lower precision with little increase in overall error. Our approach uses dynamic instrumentation to compute how a single-precision version of the program impacts error at the function granularity. We present methods to quantify and prioritize functions that can be converted to single precision. Then, we demonstrate our approach on a robotics vision case study, where robots detect 2D barcodes to identify locations and orientation. The proposed approach can present the developer with a spectrum of choices to manage the performance / error tradeoff. We present performance and error results of several points along that spectrum where we made manual changes to the code guided by the analysis results. In our case study, we demonstrate that we can achieve a speedup of 1.3x without any impact on the accuracy of detection. We also demonstrate that a speedup of up to 1.7x can be achieved with only a 4% drop in accuracy. We also study the energy and power consumption of the reduced precision implementations. We show that we can achieve a 16% energy reduction without sacrificing accuracy.

The chapter presents the following contributions:

- An analysis tool that can trace the evolution of error per memory location / instruction, providing insight into how error changes as execution progresses. This insight can support sophisticated precision-switching mechanisms.
- An analysis tool that can isolate the error per function, canceling out the effect of error propagation and determining the isolated impact of a function's code on error.
- A method that provides developers with a quantifiable tradeoff between error and performance per function. This tradeoff aids the developer in making decisions regarding precision downgrade at the function level. Our tool provides recommendations as to which functions to downgrade iteratively. These recommendations are relatively simple to apply manually, since they are at the function level.

Organization The rest of the chapter is organized as follows: Section 5.2 describes the proposed approach. Section 5.3 describes the process of managing the performance / error

tradeoff. Section 5.4 introduces the case study and the results of the analysis. Section 5.5 details our experiments with multiple precision levels and validates the quantification of performance and error provided at the analysis phase. Section 5.6 discusses our proposed analysis and the experimental results. Finally Section 5.7 summarizes the chapter.

5.2 Proposed Approach

5.2.1 Problem Statement

In this chapter, we address the following problems:

- At function granularity, automatically quantify the error resulting from converting that function to use single precision floating-point.
- At function granularity, automatically estimate the performance benefit gained by converting that function to use single precision floating-point.
- Provide recommendations to the developer as to which mix of single and double precision functions to use to reach error and performance targets.

5.2.2 Evolution of Precision Error

For every memory location containing a floating-point variable in the original precision, shadow value analysis refers to maintaining a *shadow* value in the alternative precision. All mathematical computations on the original variables are repeated in the alternative precision on the shadow variables. This analysis produces error estimates that support decision-making regarding full or partial conversion of floating-point variables and code to the alternate precision.

In this chapter, we build on top of an existing floating-point shadow value analysis tool [84] by extending it to support the proposed analysis. The tool uses Intel's Pin [95] to modify the target program via just-in-time instrumentation. To perform the analysis, our tool monitors memory access to double precision floating-point locations. It then maintains a map of memory addresses and shadow values, which can be in native single precision or any arbitrary precision. The tool detects SSE instructions and replicates them using

the target precision. After the program under inspection terminates, the tool reports the relative error per memory location which is calculated as

$$\left| \frac{v_s - v_o}{v_o} \right|$$

where v_s is the shadow value and v_o is the original value.

In the remainder of this section, we describe the modifications we made to the tool in [84] to support the proposed analysis. Our objective is to study how error evolves as the program progresses with respect to both memory locations and program instructions. This analysis can help identify parts of the program that can be converted from double precision to single precision with an acceptable impact on error.

5.2.3 Tracing Memory Error

The first modification to the tool is to track the error at each memory write. This analysis can help us identify error behavior for floating-point variables. The following are possible scenarios:

- The error is consistently low, making the variable a good candidate for conversion to single precision.
- The error is consistently high, indicating that the variable should remain at the original double precision.
- The error increases as execution progresses. We have observed this behavior in scientific computations [77], where rounding error accumulates as the application proceeds. This indicates potential for migrating the variable to single precision mid-execution. We discuss this further in Section 5.7.
- The error decreases as execution progresses. We have observed this behavior in a laser beam stabilizing control system. In this system, the initial state exhibits large physical error. This reflects in a larger relative error as the control system tries to stabilize the laser on the target. Once the laser is stable, the error drops, even during the rhythmic disturbance caused by a motor. This scenario makes the case for running at single precision only after the controller has stabilized. This is also further discussed in the future work section.

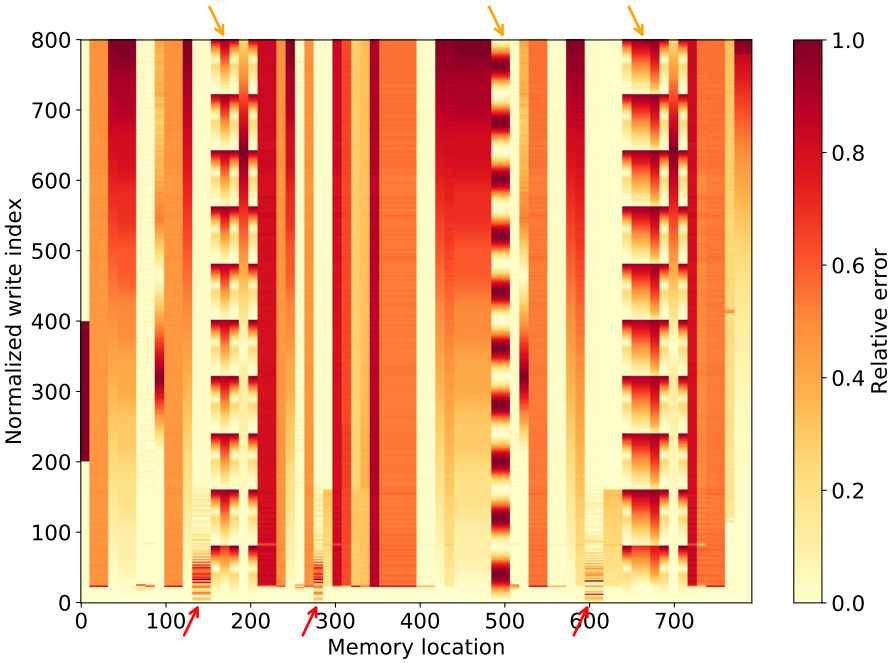


Figure 5.1: Error trace per memory location. A darker pixel indicates higher error.

To avoid large CPU and memory overhead due to instrumentation, our modification constructs an error trace by sampling the memory write operations. This is achieved by adding a knob to the Pin tool that configures the number of memory writes to skip before storing an error value. We found this useful when experimenting with time-sensitive applications such as control systems, where instrumentation overhead can spin the system out of control.

We use a visualization of the evolution of error to aid the user in identifying variables that are candidates for precision downgrade. In this visualization, each pixel represents the relative error between the original value and the shadow value of a specific memory location at a specific write operation during the run of the program. Figure 5.1 demonstrates an example of this visualization for the laser beam stabilizing control system mentioned earlier. The x-axis represents floating point memory locations observed during instrumentation. The y-axis represents the normalized write operations made to each memory location, and serves as a rough proxy to a time axis. In Figure 5.1, each memory location is normalized to be written to 800 times. Each pixel in the image represents the relative error, and a darker pixel indicates higher error.

Figure 5.1 demonstrates multiple behaviors that should guide the precision downgrade process. For some locations, the error increases as time progresses (as we move north along the y-axis). For other locations, the error is initially high and drops shortly after the start of execution. This can be seen around memory locations 140, 280, and 600 (indicated by the red arrows at the bottom). There is also a significant portion of memory where error changes periodically (indicated by orange arrows at the top). This periodic change is indicative of repetitive behavior in the controller due to the motorized disturbance applied to the stabilizer. Thus, this visualization suggests that there are locations that can be downgraded to single precision permanently, and others that can be downgraded dynamically depending on the error trace.

5.2.4 Tracing Instruction Error

The next modification made to the tool is to support tracing the error caused by each floating point instruction. To do this, we record the result of every floating point instruction, tracking the output whether written to memory or an XMM register. We compare this result with the single-precision version of that instruction. Similar to memory error, we sample instruction error to avoid large overhead.

We visualize instruction error in a similar fashion to memory error. Figure 5.2 illustrates the error per instruction at each execution of an instruction in the laser beam stabilizer mentioned earlier. The x-axis represents instructions executed during instrumentation grouped by function. For instance, `control_output` is a function whose instructions begin under the function label and continue to the right until the `control_update` label. The y-axis represents the normalized executions of the instructions, where every instruction is normalized to be executed 120 times. The color of the pixel represents the log of the relative error for the respective instruction on the x-axis and the respective execution on the y-axis.

As shown in Figure 5.2, `control_output` is the largest function. The code for the laser beam stabilizer is generated using Quanser's Simulink package [133] which generates a single large function containing almost all the logic. In general, the error is mostly high (see red braces at the top of the plot), yet there are parts of the code where the error is high at the beginning but drops shortly after the start of the execution (see orange braces at the bottom of the plot). This indicates parts of the function that can be dynamically downgraded to single-precision. `control_update` is also a candidate for permanent downgrade to single-precision since it has mostly low error (black brace at the bottom of the plot).

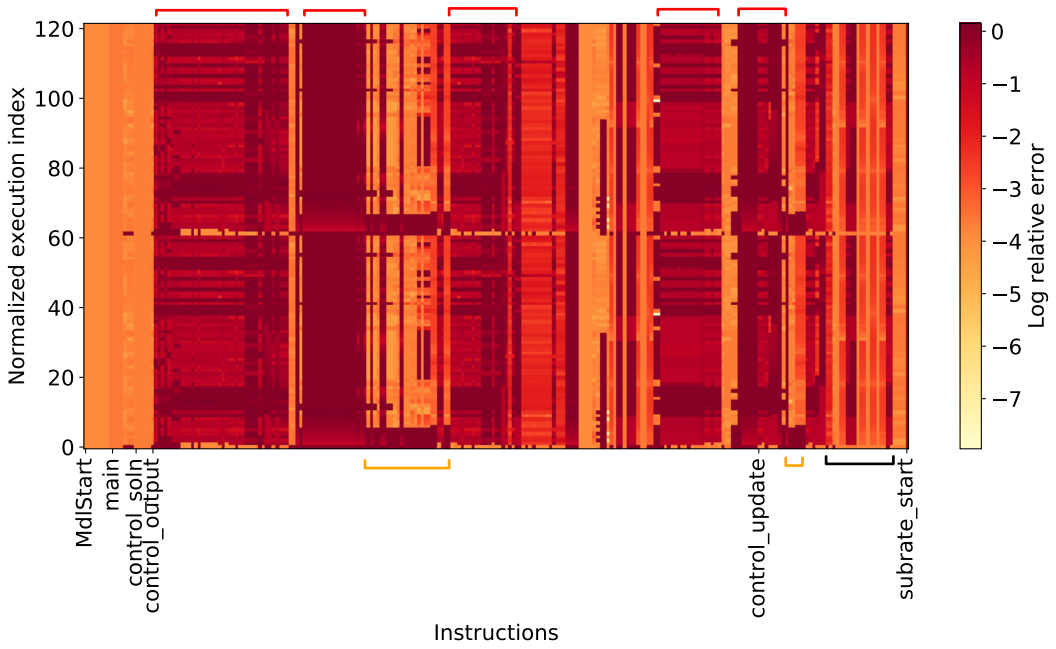


Figure 5.2: Error trace per instruction. A darker pixel indicates higher error. Relative error is in $\log_{10}$ scale.

5.2.5 Isolating Function Error

Simply tracing the error per instruction is not sufficient to determine the real impact of the instruction on error. This is because error propagates through instructions via their operands. We mitigate this by isolating error at the function level. Upon entering a function, we push all shadow values onto a stack, and reset the working map of memory addresses to shadow values. Any instruction within the function uses the original values in double precision, and the error we track is solely due to the rounding error of performing the instruction in single precision. The stack matches the call stack, and so calling one function from another resets the shadow values upon entering the called function, and pops the shadow values upon returning from the called function.

The process of isolating the error per function helps to make relatively coarse decisions about downgrading parts of the program to single-precision. A downgrade of a function to single-precision entails changing all stack variables to floats, all doubles passed by value to floats, and all mathematical operations to their single precision counterparts (ex. MULSD to MULSS). There are several advantages to performing the downgrade at the function

level:

- The downgrade is simple enough to apply manually, which we do in the case study in Section 5.4. It does not require changing data types that impact other functions, which makes it easier to apply.
- The downgrade is coarse enough that it would not introduce too much overhead. Obviously this varies depending on the size of the function, but applying the downgrade at the function level will in general introduce less casting and copying of double precision variables than a downgrade at the instruction level.
- Analysis at the function level reduces the search space of interactions with respect to error. While we can measure the isolated impact of a function on error, we cannot predict the interaction when two functions are simultaneously downgraded. The error might be canceled out or magnified. Studying these interactions can result in a huge search space if done at the instruction level. We discuss further techniques to reduce the search space in the next section.
- Isolating the error at the function level helps create a set of *independent* random variables representing the error per function. Without isolation, the error of a function is dependent on other functions' errors. Such independent random variables simplify experimental design to understand interactions and compound effect on error.

5.3 Managing the performance / error tradeoff

The visualizations in the previous section illustrate how error changes as execution progresses with respect to the input data set. These illustrations will look different for different test data sets. If the user provides a data set that is representative enough of the application's real world use, results should be predictable. This is the same for any system that uses training data and is also true of performance analysis, which requires a representative workload to achieve useful optimization.

The visualizations also do not capture the tradeoff between the performance gain when downgrading a specific function, and the error resulting from the downgrade. To aid the user in managing this tradeoff, we use plots similar to the one in Figure 5.3.

Figure 5.3 illustrates the tradeoff between performance gain and average error for every function. The x-axis represents the number of executed instructions per function. This

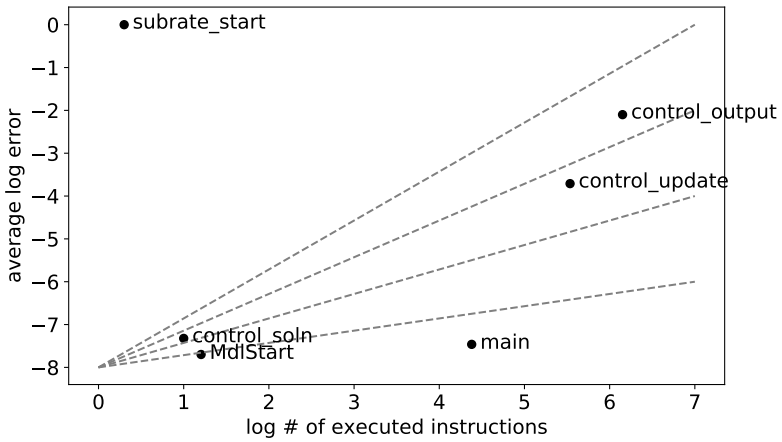


Figure 5.3: Tradeoff between performance gain and error for every function. Average error is in $\log_{10}$ scale.

is a proxy for the performance gain achieved by downgrading the precision of each function, with a larger value indicating larger gains. The rationale for the use of this proxy is that single precision instructions are generally faster than double precision on most architectures due to fewer memory transfers, less cache occupancy, and fewer cycles for some mathematical operations. This advantage also extends to energy, since a single precision implementation is idle for a longer time, resulting in reduced energy consumption. The y-axis represents the average error over all instructions in the function. Naturally, a lower average error is more desirable. The objective of the plot is to aid the user in identifying which functions to downgrade in precision. In general, functions residing in the lower right-hand quadrant are the best candidates for downgrading.

We use two methods to select functions for precision downgrade:

1. **Arbitrary separation.** An arbitrary, positively-sloped line can be drawn originating from the plot's origin, under which all functions are downgraded, and above which all functions remain at double precision. An example of such lines is shown in Figure 5.3. The lines in the figure gradually increase in slope resulting in the inclusion of functions that have higher error but also high performance gains.

2. **Scoring.** A score could be associated with each function in the form of

$$s = \frac{\varkappa}{\varepsilon} \tag{5.1}$$

where s is the score, $\varkappa$ is the number of executed instructions and ε is the average error. Functions with the highest score should be downgraded first.

An automatic search could be employed to find the optimum subset of functions to downgrade such that the compound error is minimized and the performance gain is maximized. This is somewhat similar to a knapsack where functions have value (performance gain) and weight (error), except that the combination of functions in the knapsack affects the total weight. This is similar to the work in [82, 144]. We discuss this further in Section 5.7.

5.4 Apriltag detection case study

5.4.1 The Apriltag Library

Apriltags [115] are two-dimensional bar codes that are similar to QR codes and are used in a variety of applications such as robotics and augmented reality. The advantage of Apriltags over QR codes is that they encode less information, allowing more robust detection at longer distances. Apriltags can also identify a 3D position of the tag with respect to a calibrated camera. Figure 5.4 shows an example of an Apriltag.

Figure 5.4: An Apriltag [4].

Apriltag detection is implemented in a self-contained C library [4] that can process static images as well as live video. We compile it with `gcc` 5.4.0 with `-O4` optimization. Our first step was to verify whether the library uses double precision floating-point arithmetic. We used Intel Pin's `Insmix` tool to extract statistics of instruction usage. Table 5.1 lists the percentage of instructions executed per category. A small percentage of single precision instructions were executed but double precision dominated, accounting for almost a third of all executed instructions. This indicates a large potential for performance improvement if all or some of these double precision instructions are downgraded to single precision.

Table 5.1: Apriltags library instruction breakdown

Category	Percentage
General purpose	50%
MMX	1%
Single-precision SSE	5%
Double-precision SSE2	27%
Others	18%

5.4.2 Test Dataset

We use the PennCOSYVIO dataset [122] to test the reduced-precision versions of the Apriltags library. The dataset consists of a set of videos shot on a university campus using a GoPro camera mounted on a robot. The path of the robot is populated with physical Apriltags printed at various locations. The videos are approximately 3 minutes long at 1080p resolution.

5.4.3 Initial Comparison of Double vs. Single Precision

The next step was to study whether a full single precision implementation can result in acceptable error. It might be the case that converting all variables and instructions to single precision yields low error, in which case there is no need for sophisticated analysis of specific memory locations and functions. To perform this check, we converted all double precision variables in the source code to single precision and tested a set of sample images, some containing Apriltags and some not.

We observed two behaviors when testing a full single-precision version of the library. In some cases, the modified version did not terminate due to strict convergence criteria on a particular loop. Capping the iterations of this loop resulted in false negatives.

These results demonstrate that while single precision computation does often run faster than its double precision counterpart, algorithms with value-based control flow or strict termination criteria might exhibit worse performance when using single precision arithmetic. This is an important factor that should be considered when applying precision-related optimizations.

Next, we compared double and single precision implementations against a video from the dataset. The video is split into 2888 frames and we measure the number of tags

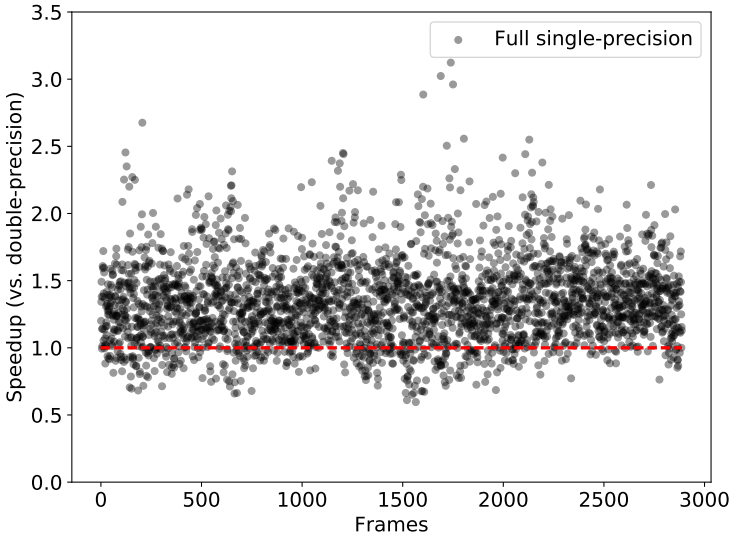


Figure 5.5: Speedup of the full single-precision implementation.

detected by both implementations in each frame. False negatives are abundant and the single precision version misses 75% of all tags. False positives also occur yet are very infrequent ($< 0.1\%$).

Figure 5.5 demonstrates the per frame speedup gained by using single precision. The red line indicates 1x speedup, above which every point is an improvement, and below which every point is a slow down in performance. As can be seen in the plot, while there are significant improvements reaching up to 3x, the single precision implementation is not reliable. It runs longer than the original double precision implementation 11.4% of the time and generally has a higher variance. The occasional increase in run time is due to the algorithm taking more time to converge at single precision. This indicates that a trivial single-precision implementation is not appropriate because it yields high error with no reliable performance improvement.

5.4.4 Using the Proposed Approach to Analyze Error

The results in Section 5.4.3 indicate there is potential to find a sweet spot between error and performance. While a full single-precision implementation can provide attractive speedup, the accuracy drops significantly. Next, we used the analysis approach described in Section 5.2 to understand the behavior of the Apriltags library and find a compromise

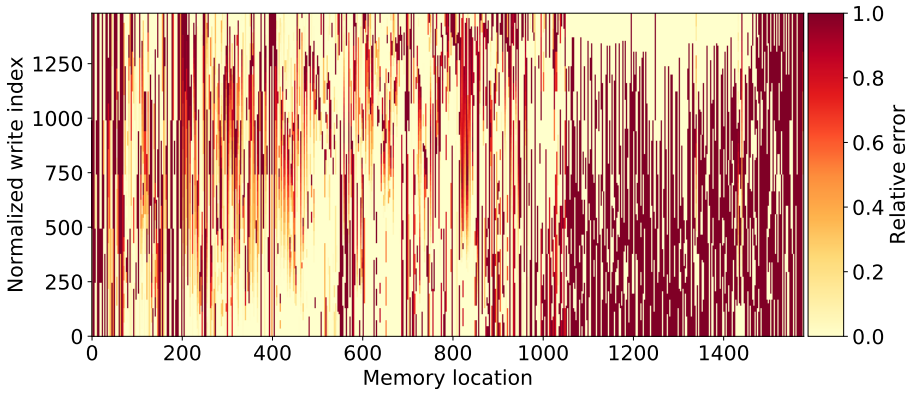


Figure 5.6: Memory error trace of the Apriltags library.

between error and performance.

Apriltags memory error trace

We first studied error with respect to double-precision memory locations. As explained in Section 5.2.3, we use our Pin tool to capture the error behavior during execution. Figure 5.6 illustrates the error trace per memory location. As shown in the figure, the error is generally high in all memory locations at some point during execution, making per-variable optimizations (such as the optimizations suggested in [144]) difficult.

Apriltags instruction error trace

We next studied the error caused by each executed instruction (see Section 5.2.4). Figure 5.7 (top) demonstrates the non isolated error per instruction grouped by function. As can be seen in the figure, there are functions that are reasonable candidates for precision downgrade, such as `fit_line`, `fit_quad`, and `quad_decode_task`.

Apriltags isolated instruction error trace

In Section 5.2.5, we introduced a method to isolate the error per function such that the measured error is not affected by propagated error passed to the function as parameters or accessed by the function using pointers. Upon repeating the analysis using such

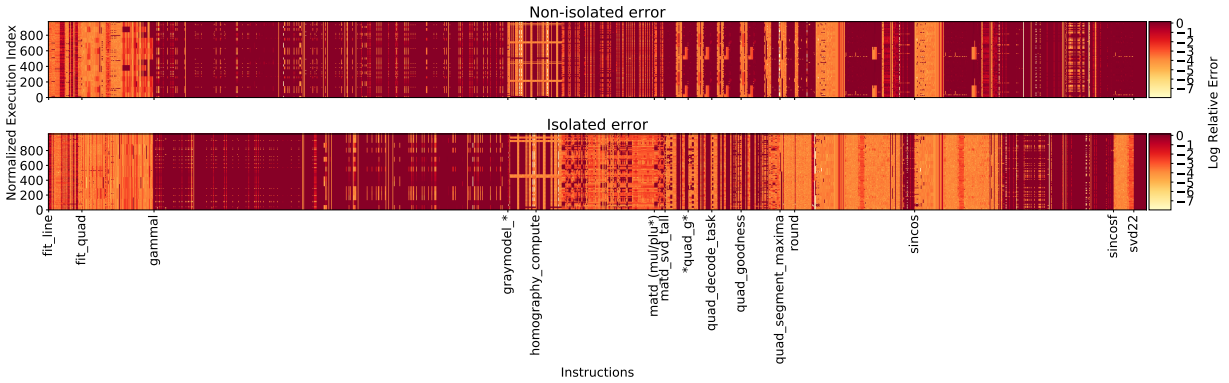


Figure 5.7: Instruction error trace of the Apriltags library isolated per function and non-isolated.

isolation, the error profile of some functions differed drastically. Figure 5.7 (bottom) illustrates the error per instruction isolated by function. As shown in the figure, functions like `quad_goodness` are now prominent candidates for downgrade due to their low error. This indicates that the error observed within `quad_goodness`'s instructions was actually a byproduct of propagated error by different functions. We can identify such functions using this simple plot, whereas tracing through the code manually is an extremely tedious process. Other functions like `homography_compute` and `quad_segment_maxima` exhibit similar behavior and are also candidates for precision downgrade.

Apriltags performance vs error tradeoff

Finally, we used our tool to visualize the tradeoff between performance gain and error per function. Figure 5.8 illustrates the tradeoff. There is a cluster of functions at the bottom right, which are strong candidates for precision downgrade due to low error and a relatively high number of executed instructions. However, while we know the isolated impact of individual functions on error, we do not know what interactions will arise when combinations of functions are downgraded simultaneously. In the next section we describe experimentation to explore this effect.

5.4.5 Identifying Precision Levels

Using the information provided in Figure 5.7 and 5.8, we can start downgrading parts of the program to single precision. The downgrade process is currently manual, although it

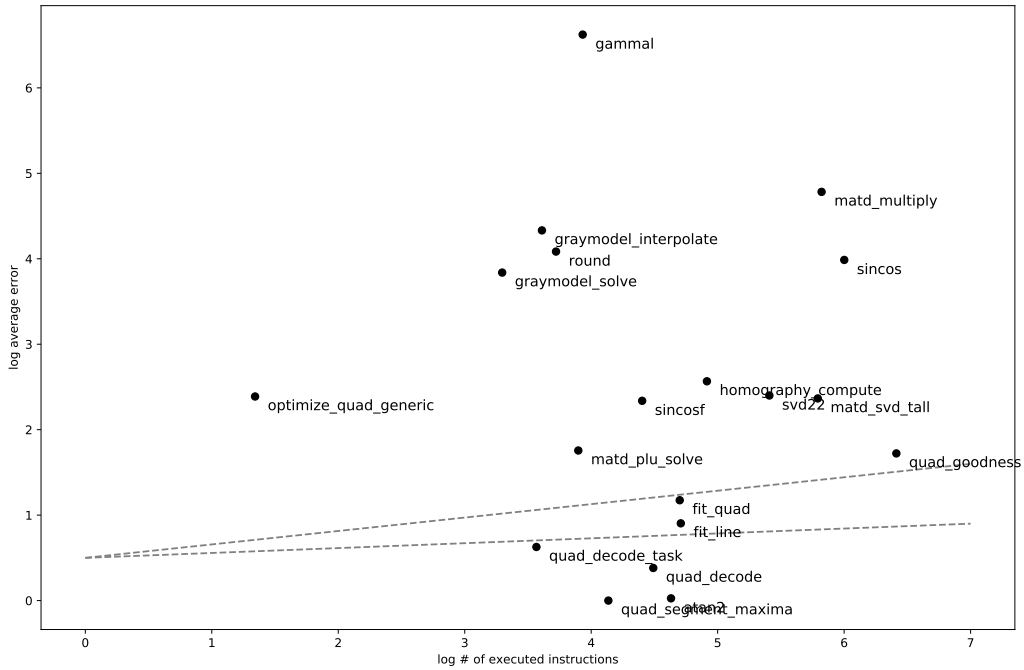


Figure 5.8: Tradeoff between performance gain and error for every function in the Apriltags library.

could be automated as discussed in Section 5.7. Currently, the downgrade process includes:

- changing double-precision passed-by-value parameters to single precision, and
- changing local variables to single precision.

In this case study, each function downgrade took a few minutes and was mostly a find and replace process. The harder problem is identifying which functions to downgrade. Since there is a prohibitively large number of function combinations, we are only concerned with testing a limited number of combinations. We use the term *precision level* to refer to an implementation where a particular subset of all functions is downgraded to single precision. The following sections detail the proposed methods of defining different precision levels.

Levels defined by score

The first method attaches a score to a function, and converts functions with the highest score first, greedily adding more functions. We use the score formula in Equation 5.1 with

$c = 1$. The top 5 functions are `atan2` (a `libmath` function that cannot be downgraded), `quad_goodness`, `matd_svd_tall`, `sincos` (similar to `atan2`), and `matd_multiply`.

We define two levels to experiment with levels by score:

- **Level-1**: `quad_goodness`
- **Level-2**: `quad_goodness` and `matd_svd_tall`

Levels defined visually

The second method is to use an arbitrary line on the tradeoff graph to decide which functions are downgraded (below the line) and which are not. An example of such lines is shown in Figure 5.8. This method helps make changes more coarse and can be used in combination with scores, as we will show in the next section. We use the lines in Figure 5.8 to define our levels visually:

- **Level-3**: `quad_decode_task`, `quad_segment_maxima`, and `quad_decode`.
- **Level-4**: `fit_line`, `fit_quad`, `quad_segment_maxima`, `quad_decode`, and `quad_decode_task`.

5.5 Experiments and Results

In this section, we demonstrate how to define precision levels and ultimately find a compromise between performance and error.

5.5.1 Metrics

We use a set of metrics to measure the quality of an implementation from the perspective of performance as well as error. The following are the metrics we measure in each run:

- Speedup vs. the double-precision version by measuring frame processing time.
- Energy vs. the double-precision version.
- Power vs. the double-precision version.

- Number of false positives per frame (i.e., the library detects tags that do not exist or have the wrong ID).
- Number of false negatives per frame (i.e., the library fails to detect existing tags in the frame).

5.5.2 Experimental setting

We perform our tests on two architectures: an Intel Core i5 machine (i5-4460S CPU @ 2.90GHz, 4 cores, 6MB Cache, 8GB RAM) and a Raspberry Pi 3 (1.2 GHZ quad-core ARM Cortex A53, Broadcom VideoCore IV GPU, and 1 GB LPDDR2-900 SDRAM). Testing on Raspberry Pi illustrates the portability of the analysis, which was done on an Intel machine using Intel's Pin instrumentation tool, to a different architecture. The Raspberry Pi is widely used in robotics with some interesting applications [111]. As a test set, we used a video from [122] to verify the significance of our conclusions against thousands of frames.

5.5.3 Speedup results

Figure 5.9 shows the speedup per precision level. As can be seen, all precision levels almost always dominate the double-precision implementation. Table 5.2 shows the percentage of frames where single precision runs faster than double precision (*% with speedup*). The table also shows the average speedup across all frames.

There are some interesting points to observe:

- **Level-1** is the most reliable in terms of speedup, since the speedups are tightly distributed, and the number of times a frame runs slower in **Level-1** is less than 1%.
- **Level-2** improves speedup significantly. This is due to the downgrade of `matd_svd_tall` to single precision. Figure 5.8 indicates that this function is in the top five functions with most executed instructions. However, speedup is loosely distributed indicating that the function affects a part of the program that relies on precision to terminate. This is also supported by the function's high error (see Figure 5.8).
- **Level-3** and **Level-4** provide higher speedup than **Level-1** with a relatively tight distribution. However this comes at the cost of consistency.

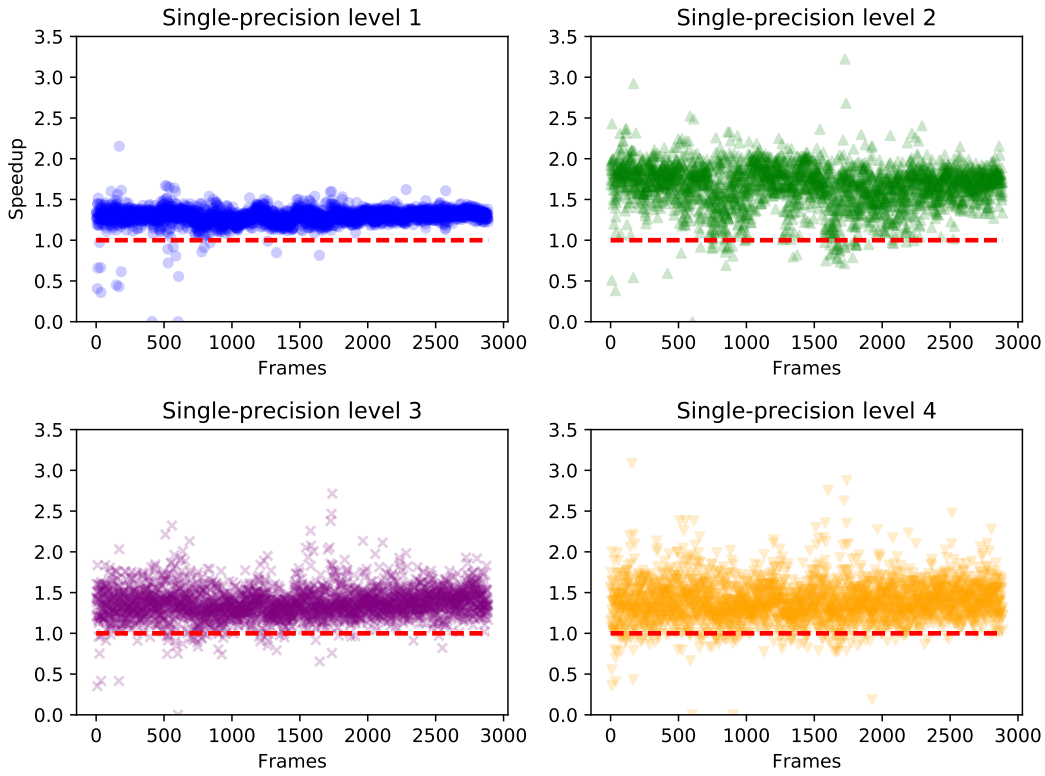


Figure 5.9: Speedup per precision level.

Table 5.2: Apriltags levels performance and accuracy.

Level	Avg. Speedup	% with Speedup	Accuracy
Level 1	1.30x±0.09	99.30%	100%
Level 2	1.70x±0.25	98.60%	96.64%
Level 3	1.35x±0.18	98.06%	61.48%
Level 4	1.36x±0.249	95.77%	60.51%

5.5.4 Accuracy results

Figure 5.10 shows the number of false positives and negatives per level. As can be seen in the figure, **Level-1** has no false positives or negatives since tags in all frames match those in the double-precision implementation. However, the accuracy drops significantly when downgrading the **Level-3** functions as well as the **Level-4** functions. This is indicative of

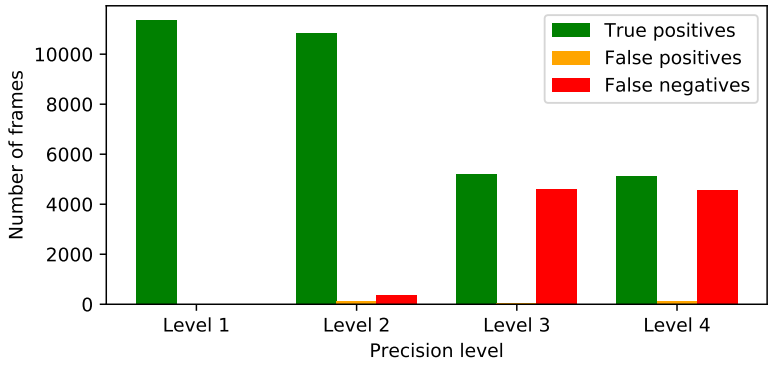


Figure 5.10: False positives / negatives for each level.

a compounding effect that increases error when these functions are downgraded. Table 5.2 lists the accuracy (# of frames with no false positives or negatives / number of frames). Thus, **Level-1** appears to achieve perfect accuracy (against our test dataset) while providing a 1.3x speedup. However, the decision of which level to select is to be made by the user, who can tweak their implementation to match a desired accuracy. In some cases, 96% might be acceptable, and **Level-2** could be selected for its more significant 1.7x speedup.

5.5.5 Energy results

Figure 5.11 shows the ratio between the energy consumption of each precision level versus the energy consumption of the original double precision implementation. This is measured per frame such that the comparison is fair. We use Intel's RAPL [155] to determine energy consumption. As expected, performance gain also results in energy reduction. **Level-1** reduces energy consumption on average by 16%, **Level-2** by 33%, **Level-3** by 20% and **Level-4** by 21%. Similar to our remarks on speedup, **Level-1** seems to be more consistent in its improvements. This is attributed to the **Level-1** modification not affecting the iterative process of convergence, which can sometimes cause longer run times.

Figure 5.12 shows the ratio between the power consumption of each precision level versus that of the original double-precision implementation. We use Intel's RAPL to determine the average power consumption. Interestingly, all precision levels consume *more* power than the double-precision implementation. On average, all levels increase power consumption by around 10%. We investigated the reason behind the increase in power consumption, first exploring the CPU frequency and idle status during execution. We used `powertop` [154] to log the percentage of time the CPU spends in each frequency as well as

Table 5.3: Apriltags `perf` statistics.

Implementation	CPU Cycles	Instructions	IPC
Double precision	18.1×10^9	25.0×10^9	1.39
Level-1	13.9×10^9	26.7×10^9	1.92
Level-2	10.0×10^9	20.5×10^9	2.04
Level-3	13.2×10^9	25.3×10^9	1.91
Level-4	13.4×10^9	25.7×10^9	1.91

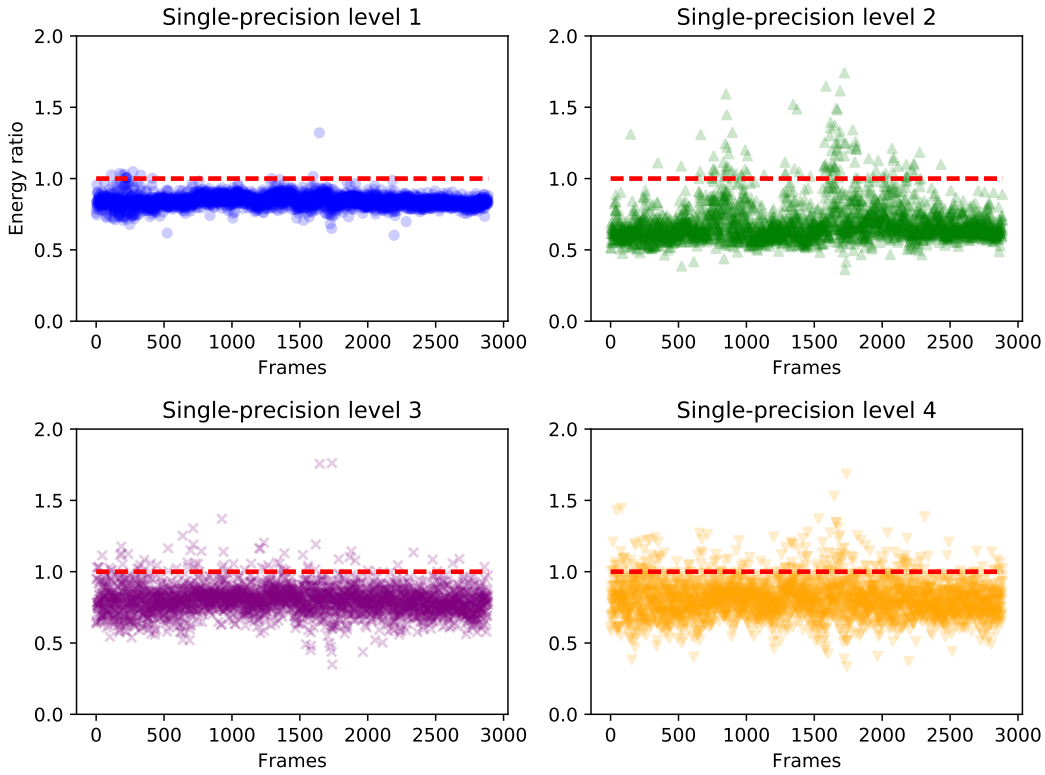


Figure 5.11: Energy ratio per precision level.

idle. We used this log to calculate the weighted average CPU frequency. After comparing the average CPU frequency across different levels and different frames, we failed to prove that there is any significant difference between levels in terms of CPU frequency. This was confirmed using a t-test.

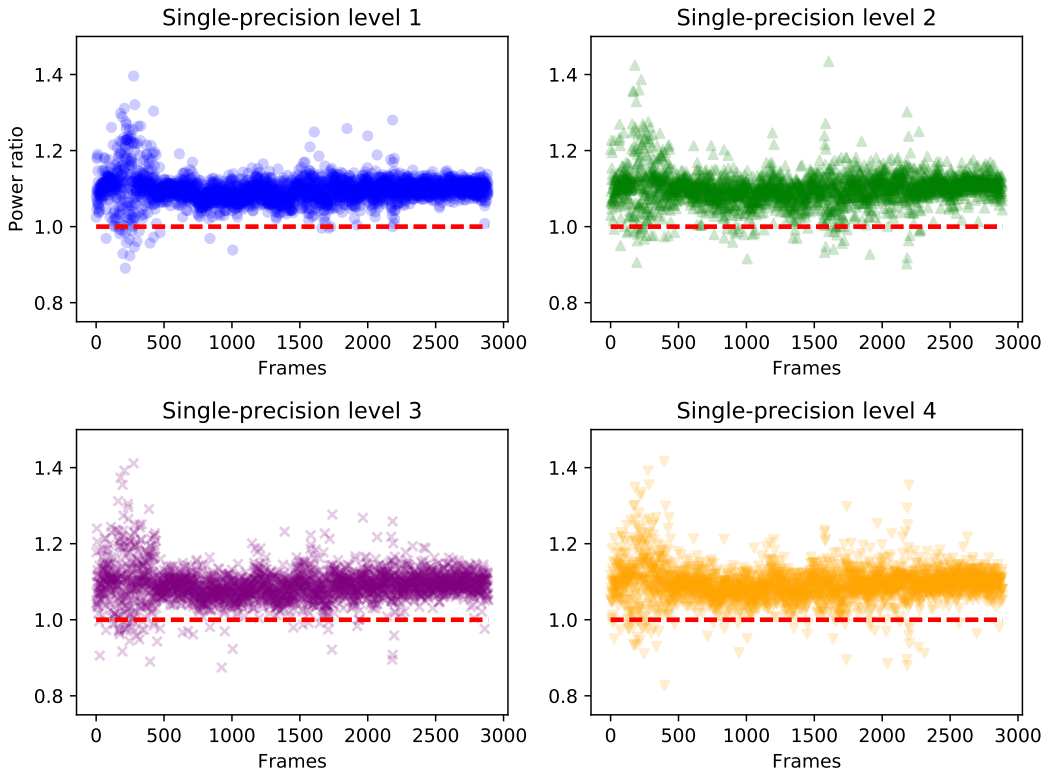


Figure 5.12: Power ratio per precision level.

Next, we verified whether there is a significant difference in the number of CPU cycles used by the single precision levels versus double precision. We used `perf` to count the number of CPU cycles and instructions. Table 5.3 shows the average number of cycles and instructions per precision level, as well as the instructions per cycle (IPC). As shown in the table, reduced-precision levels lower the number of cycles by approximately $25\% - 55\%$ while increasing the number of instructions per cycle by $37\% - 47\%$. The increase in instructions per cycle shows more optimized pipelining due to the use of more efficient 32-bit vectorization. This explains the 10% increase in power consumption despite all implementations running at roughly the same CPU frequency.

5.5.6 Raspberry Pi results

We repeated our experiments on a Raspberry Pi 3 board to validate the impact of reduced precision on different architectures. The Raspberry Pi 3 is equipped with an ARM Cortex

Table 5.4: Apriltags on Raspberry Pi.

Implementation	Speedup	Energy Ratio	Power Ratio
Level-1	1.21x±0.07	0.74	0.92
Level-2	1.61x±0.23	0.59	0.91
Level-3	1.27x±0.17	0.79	1.03
Level-4	1.23x±0.25	0.78	0.97

A53 (ARMv8) quad core processor. The analysis was based on Intel's instrumentation tool, running on an Intel processor, so it is interesting to see the performance and error of precision levels on an ARM processor. To measure power, we use a USB power meter. Due to timing constraints, we tested only 100 frames on the Raspberry Pi as it is significantly slower than an Intel Core i5.

Table 5.4 summarizes the performance of the precision levels on the Raspberry Pi. As shown in the table, speedup is similar to that on the Intel processor. Energy savings are higher on the Raspberry Pi than on the Intel processor, which is encouraging for a versatile device in embedded systems. The power consumption is generally the same across different levels versus double precision. While the Intel processor employs sophisticated pipelining technologies resulting in higher power consumption of single precision levels, the Raspberry Pi power consumption is slightly less for single precision versus double precision in most cases.

5.6 Discussion

In this section we discuss several remarks on our proposed analysis.

Effectiveness of isolated error analysis. Since our analysis isolates error per function, inter-functional error propagation is not captured. As mentioned earlier, identifying the impact of any combination of functions on error is a combinatorial problem. Our analysis approach approximates this using isolated error, which we have shown is capable of yielding positive results. Moreover, using test data to verify different precision levels helps identify levels where actual error does not match predicted error.

Portability of analysis results. Since our tool uses Intel Pin instrumentation, it can only be used on architectures where Pin is supported. If the program cannot run on Pin-supported machines, we will not be able to collect error data. On the other hand, the program itself can be deployed on architectures different from the one used to collect error data. If the floating-point behavior is drastically different between two architectures, our function downgrade recommendations may be erroneous. However, most architectures adhere closely to the IEEE 754 specification for floating-point arithmetic, ensuring that results are portable up to compiler-related differences. We have also demonstrated that our recommendations were useful on at least one very different architecture (the ARM-based Raspberry Pi).

Utility of methods for defining precision levels. We proposed two methods to recommend which functions to pick first when experimenting with downgrading precision: a score method and a line based separation method. The difference between the two methods is as follows: line based separation will include cheap conversions, i.e. small functions that have low impact on error and a low number of instructions. Score based selection will target the functions with the highest executions to error ratio. This can result in different outcomes depending on the application. In our case study, it appears that cheap conversions in the line based approach (levels 3 and 4) cluster into a significant drop in accuracy, due to interactions. However, the score based method results in higher accuracy and speedup. This outcome could be different for different applications, and thus it is important to use selection heuristics such as those we propose to reduce the search space and provide acceptable performance improvements.

Recursive functions. Since we reset error upon entering the function, recursion should still be supported. An interesting scenario would occur if recursion depends on the floating point value converging. We encountered a similar phenomenon but with loops. For instance, let's assume the function recurses 10 times at double precision and 20 times at single precision. Since we are using Pin to shadow the original double precision execution, we only record error for 10 recursive calls of the function at single precision. It is safe to assume that the error at the end of these 10 calls will be higher for single precision. This higher error will proxy the cost of downgrading the precision of the recursive function, and produce an appropriate recommendation.

Effect of sampling. Sampling can result in missing key computations that affect the error reported for the function. The user is responsible for managing the tradeoff between

accuracy and instrumentation time. However, since we are using average error to make recommendations, a few extreme error values will not have a strong impact on the average error. Our approach in the case study of this chapter was to start at a low sampling frequency and incrementally increase the frequency until there was no change in the shape of error curve.

5.7 Summary

In this chapter, we presented a novel approach for identifying parts of a program that can tolerate lower precision with little increase in overall error. The approach focuses on the function granularity, quantifying the error and performance gain resulting from downgrading a function to single precision. We proposed multiple approaches to select functions for reduced precision based on the quantification made by our tool.

To demonstrate the applicability of the proposed approach, we applied our methodology on a robotics vision case study. We analyzed the code of a C library responsible for detecting 2D barcodes in input video streams. We demonstrated that our approach can identify functions that have the least impact on error while providing a large performance boost. To validate our tool's analysis results, we downgraded several suggested functions to single precision and measured accuracy and performance. Results indicate that we can achieve a speedup of 1.3x at no cost to accuracy with respect to our test dataset. Results also show that a speedup of $1.7x$ can be achieved at a loss of only 4% of accuracy.

We further expanded on the utility of the proposed approach for embedded systems by studying the energy and power consumption of the reduced precision implementations. We show that we can achieve a 16% energy reduction on Intel and a 26% energy reduction on ARM with an implementation that has perfect accuracy yet reduced precision. By sacrificing 4% of accuracy, energy reduction can reach 33% on Intel, and 41% on ARM.

Our analysis of the applicability of our approach to several systems resulted in the conclusion that the proposed approach is most effective in floating-point intensive applications. Applications that utilize floating-points in computations on low-precision sensory data are not suitable candidates for our approach, since it is often the case that the sensory precision eclipses the difference between single and double floating-point precisions. Our approach is most suitable for applications involving complex mathematics, which is the case for almost all vision, speech, and AI applications.

Chapter 6

Generalized Model

In Chapter 4, we introduced a flexible model that supports managing tradeoffs of renewable and non-renewable resources for task-based systems. We demonstrated that the model can be used to minimize the makespan of the program by controlling resources such as power and bandwidth. Such a model can be used to bound the performance of batch systems with inter-node dependencies.

In this chapter, we combine concepts from Chapters 2, 3 , 4 and 5 to construct a generalized model for streaming applications. The producer consumer problem discussed in Chapter 2 is a suitable abstraction for many stream processing systems, and the model in Chapter 3 which was improved and generalized in Chapter 4 is modified to support streaming applications instead of task graphs. Moreover, we generalize the energy precision tradeoff in Chapter 5 to model consumers capable of processing data at various quality levels.

This Chapter is organized as follows: Section 6.1 discusses the problem statement, Section 6.2 introduces the proposed generalized model, Section 6.3 demonstrates how the model can be used to capture the tradeoffs of a toy example, Section 6.4 discusses how to construct a linear program that optimizes resource consumption, and finally Section 6.5 summarizes the chapter.

6.1 Problem Statement

Parallel stream processing applications involve complicated tradeoffs. The rate of incoming items forces a tradeoff between infrastructure cost and processing quality. High quality

output – which could be more complex processing algorithms or a higher sampling rate – will commonly have an impact on power, energy, and bandwidth. In multistage systems where nodes are receiving data from one layer, processing it and feeding it into another layer of nodes, the tradeoff becomes difficult to manage. This is magnified in systems where load dynamically shifts from one part of the network to another. In such systems, it is difficult to determine an optimal configuration that would balance tradeoffs cheaply and dynamically.

Most systems employ manual controls to manage tradeoffs. For instance, commercial Data Loss Prevention software allows administrators to control the quality of malicious activity detection according to the expected load [104]. This approach is often targeting a specific subset of resources, and lacks the flexibility required by large dynamic systems. Another approach is to aggressively over-provision the processing infrastructure in terms of machine capabilities (CPU, memory, etc...), network bandwidth, and allocated power budget such that no limits are hit. This is a very costly approach that is often not possible and is not future proof.

To that end, this chapter tackles the following problem: assume a stream processing network where each node in the network encounters a tradeoff between quality of processing and resource usage. Nodes are subject to resource constraints that can be individual or collective. Lower quality results in higher error, which propagates through the network and can potentially impact the quality of subsequent nodes. Our objective is to provide an efficient methodology to model such a system such that resource bounds are respected and a chosen goal by the designer is optimized. This goal could be minimizing error, maximizing quality, minimizing consumption of energy or any other resource modelled in the system.

In this chapter, we propose an abstract model of stream processing applications. In our model, the processing nodes are modelled as a network of producers and consumers. Each node in the network is a consumer of data flowing through its incoming edges, and a producer of data flowing through its outgoing edges. Processing of data being consumed / produced by a node can be performed at a variable quality level. The quality level of processing dictates the amount of resources used by the node. Resources include power, energy, RAM, disk, or network bandwidth. In addition to these resources, we model *accuracy* as a non-renewable resource that flows through the network and is partially depleted when a node reduces its quality level. We demonstrate how to use the model to determine the optimal quality level assignment that minimizes the usage of a specific resource.

6.2 Model

In this section, we present our model used to capture parallel stream processing applications.

6.2.1 The Producer / Consumer Network

A node in the producer consumer network can be a producer, a consumer, or both a producer and consumer. We define a node $v = \langle P_v, S_v \rangle$ as a tuple where P_v is the set of predecessor nodes from which v receives data, and S_v is the set of successor nodes which receive data from v.

Let $G = (\mathbb{V}, \mathbb{E})$ be a directed acyclic graph. $\mathbb{V} = \{v_1, v_2, \cdots, v_N\}$ is a set of vertices representing nodes, and E is a set of edges represented as ordered pairs of vertices such that

$$\mathbb{E} = \{(x, y) \mid y \in S_x\}$$

where S_x is the set of successors of task x. An edge from x to y represents a stream of items flowing from x to y, in which case x is a producer (potentially also a consumer) and y is a consumer (potentially also a producer).

6.2.2 Resources

In this subsection, we demonstrate how to model both renewable and non-renewable resources.

Renewable Resources

Renewable resources are not depleted when an item is processed. Examples of renewable resources are CPU, power, memory, network bandwidth, and quality. These resources are instantly reclaimed once an item is processed. We define the set of renewable resources in the system as follows:

$$\mathbb{R} = \{R_1, R_2, \cdots\} \tag{6.1}$$

Non-renewable Resources

Non-renewable resources are depleted once an item is processed. Examples of non-renewable resources are energy, time, and accuracy. For instance, once error is encountered during the processing of an item along its path through the network, it cannot be reclaimed. We define the set of non-renewable (depletable) resources as follows:

$$\mathbb{D} = \{D_1, D_2, \cdots\} \tag{6.2}$$

6.2.3 Resource Flows

A resource (renewable or non-renewable) can flow through the network using network flow principles. We define three variables for every node in the graph:

- ζ_v^γ denotes the inflow of resource γ into node v.
- μ_v^γ denotes the consumption of resource γ in node v.
- ξ_v^γ denotes the outflow of resource γ from node v.

We define the amount of flow going through an edge in the network as ν_e^γ where $e \in \mathbb{E}$.

Resource Inflow

The amount of a resource flowing into a node depends on the amount of flow carried over all its incoming edges. Traditionally, the inflow is the sum of all flows on incoming edges. We generalize this notion using function Z:

$$\zeta_v^\gamma = Z_v^\gamma \left(\{\nu_e^\gamma \mid e \in \{(v', v) \mid v' \in P_v\}\}\right) \tag{6.3}$$

For instance, power is a resource that can be summed over incoming edges. Accuracy is more complicated since inaccurate data coming from different nodes has a different impact on the receiving node's accuracy.

Resource Outflow

The amount of a resource flowing out of a node is traditionally equal to the amount of the resource flowing in, which is the conservation of flow principle. This models renewable resources efficiently, yet does not capture non-renewable resources. We generalize the outflow using function Ξ:

$$\xi_v^\gamma = \Xi_v^\gamma \left(\zeta_v^\gamma, \mu_v^\gamma\right) \tag{6.4}$$

For instance, power is a renewable resource and thus $\Xi_v^{\mathrm{PWR}} \left(\zeta_v^{\mathrm{PWR}}, \mu_v^{\mathrm{PWR}}\right) = \zeta_v^{\mathrm{PWR}}$, whereas energy is depletable and thus $\Xi_v^{\mathrm{E}} \left(\zeta_v^{\mathrm{E}}, \mu_v^{\mathrm{E}}\right) = \zeta_v^{\mathrm{E}} - \mu_v^{\mathrm{E}}$.

6.2.4 Resource Bounds

Our model supports bounding resources on either nodes or edges. A bound on nodes is defined as

$$b_v = \langle \gamma, \mathcal{N}, \beta_L, \beta_U \rangle$$

where $\gamma \in \mathbb{R} \cup \mathbb{D}$ is the bounded resource, $\mathcal{N}$ is a set of nodes that are bounded, β_L and β_U are the lower and upper limit of the bound respectively. A bound on edges is defined as

$$b_{v,v'} = \langle \gamma, \mathcal{E}, \beta_L, \beta_U \rangle$$

which is similar to b_v except that $\mathcal{E}$ is a set of nodes that are bounded. A bound on nodes enforces the following:

$$\beta_L \leq \sum_{v \in \mathcal{N}} \mu_v^\gamma \leq \beta_U \tag{6.5}$$

A bound on edges enforces the following:

$$\beta_L \leq \sum_{e \in \mathcal{E}} \nu_e^\gamma \leq \beta_U \tag{6.6}$$

For instance, if a node v has 8 cores, the maximum CPU usage is 800% as in the common notation of multicore systems. In this case, a bound $b = \langle R_{\mathrm{CPU}}, \{v\}, 0, 800 \rangle$ is applied. The set of bounds in the system is

$$\mathbb{B} = \{b_{v_1}, b_{v_2}, \cdots\} \cup \{b_{e_1}, b_{e_2}, \cdots\}$$

Another example is a cluster of nodes that is power bounded. This implies that the sum of power consumed by all nodes in the cluster should not exceed a specific value. A bound

$b = \langle R_{\text{PWR}}, \{v_1, v_2, v_3\}, 0, 500\rangle$ denotes that the total power consumption of nodes v_1, v_2 and v_3 should not exceed 500 watts.

We use bounds on edges to control the distribution of resources across outgoing edges of a node. We identify two main methods of assigning flow to outgoing edges: *broadcast* and *distribution* resources.

Broadcast In this case, nodes broadcast their outflow to outgoing edges. Accuracy is broadcasted, since all outgoing edges of a node carry data with the same accuracy level that the node produces. We can enforce resource γ to be broadcast using the following bounds:

$$\forall (v, v') \in \mathbb{E} : b_{(v,v')} = \langle \gamma, \{(v, v')\}, \xi_v^\gamma, \xi_v^\gamma \rangle \tag{6.7}$$

Distribution In this case, the outflow is distributed across all outgoing edges. For instance, in a multiple consumer setting any one of a set of receiving nodes can process items. In this case, the outgoing data flow of the producer is distributed among all consumer nodes. The objective here is to determine the fraction of data flowing to each consumer such that resource bounds are respected and the usage of a specific resource is optimized. We can enforce resource γ to be distributed using the following bounds:

$$\forall v : b_v = \langle \gamma, \{(v, v') \mid v' \in S_v\}, \xi_v^\gamma, \xi_v^\gamma \rangle \tag{6.8}$$

6.2.5 Configurations

There are various configuration parameters that impact the resource usage of a node and the accuracy of its output.

- Sampling rate: in some systems, sampling can be supported. In this case, the amount of resources consumed by a node is probably proportional to the sampling rate. On the other hand, lower sampling rate is commonly associated with reduced accuracy or confidence. Hence, sampling rate is a configuration parameter that controls the tradeoff between resource usage and accuracy.
- Outgoing data rate: the outgoing data rate of a node impacts the resource usage of subsequent nodes. If subsequent nodes decide to sample this data, then accuracy is negatively impacted.

- Precision: some algorithms support controllable precision. The work in [98] demonstrates the how configurable precision impacts accuracy and resource usage.
- Algorithm alternatives: in some systems, there are different algorithms that can be used to process the data, with varying degrees of resource usage and accuracy. For instance, DLP (Data Loss Prevention) systems employ different classifiers for malicious activity that are designed to have different processing costs [104].

To simplify our model, we abstract all the above parameters into a single *quality* parameter. First, we define the incoming and outgoing rates of data. Let O_v be the outgoing data rate of node v, and I_v be the rate of incoming data to node v. The rate of incoming data is defined as follows:

$$I_v = \sum_{v' \in P_v} O_{v'}$$

where P_v is the set of v's predecessor nodes. Hence, the rate of incoming data is the sum of all the rates of outgoing data of predecessor nodes.

We use a single parameter to identify the quality of a node's processing, which encompasses sampling, precision and algorithmic alternatives. Let $Q_v = \{q_{v,1}, q_{v,2}, \cdots, q_{v,k}\}$ be a set of quality levels that node v could use to process items. Also, let κ_v denote the quality level of node v such that $\kappa_v \in Q_v$. Let $\vartheta_v : \Lambda \times Q_v \to \Lambda$ be a function that maps an incoming data rate and a quality level to an outgoing data rate. Note that Λ refers to the domain of data rates.

Thus, we can define the outgoing rate of a node as follows:

$$O_v = \vartheta(I_v, \kappa_v) \tag{6.9}$$

6.2.6 Relationships

In this subsection, we define the relationships between configurations and resources. Let $\varphi_v^\gamma : \Lambda \times Q_v \to \Omega_\gamma$ be a function that maps node v's rate of incoming data and its quality level to a value in the domain Ω_γ of resource γ. Resource γ can be either renewable or non-renewable. Hence, each node defines a set of functions as follows:

$$\Phi_v = \{\varphi_v^\gamma \mid \gamma \in \mathbb{R} \cup \mathbb{D} - \{A\}\}$$

where A is the accuracy resource. We exclude accuracy since it is defined differently.

While accuracy depends on the quality level of a node like other resources, it also depends on the accuracy of incoming data. We incorporate this notion to model systems where error is compound, i.e. receiving erroneous data may impact the accuracy of produced data differently even at the same quality level. This behaviour is common in precision based quality levels, where rounding error is compounded as more mathematical operations are performed on a data path [105]. Thus, the accuracy of node v is defined as follows:

$$\alpha_v = \psi_v(\kappa_v, \{\alpha_{v'} \mid v' \in P_v\}) \tag{6.10}$$

where v' is a predecessor node of v, and the set $\{\alpha_{v'} \mid v' \in P_v\}$ is a set of accuracies of all predecessor nodes of v.

6.2.7 Revised definitions

Based on the definitions introduced in the previous subsections, we now redefine a node as follows:

$$v = \langle P_v, S_v, Q_v, \vartheta_v, \Phi_v, \psi_v \rangle \tag{6.11}$$

Thus, the node now includes a set of quality levels (Q_v), a function that determines the rate of outgoing data (ϑ_v), a set of functions that determine resource usage (Φ_v), and a function that determines the accuracy of the node's output (ψ_v).

Finally, we redefine the graph as follows:

$$G = \langle \mathbb{V}, \mathbb{E}, \mathbb{R}, \mathbb{D}, \mathbb{B} \rangle \tag{6.12}$$

Thus, the graph now defines a set of non-renewable resources ($\mathbb{R}$), a set of renewable resources ($\mathbb{D}$), and a set of bounds on resources ($\mathbb{B}$).

6.3 Toy Example

In this section, we introduce a toy example we use to demonstrate how our proposed model can be used to optimize various resources. Figure 6.1 illustrates a simple example of a producer / consumer network. The network models a hierarchical monitoring system where $v_{[1,4]}$ are producers of events that are consumed and manipulated by nodes $v_{[5,8]}$. Nodes $v_{[5,8]}$ then transmit the manipulated events into v_9 for aggregation. Node v_s is a placeholder node for enforcing flow-based bounds, which we elaborate on later.

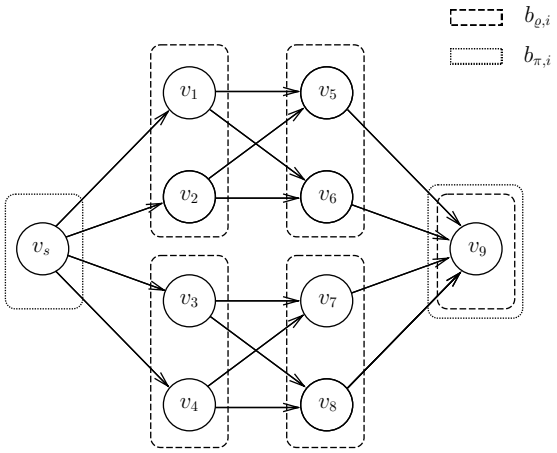


Figure 6.1: A simple producer / consumer network.

6.3.1 Resource Relationships

First, let us introduce the characteristics of the nodes in the network. Table 6.1 lists the outgoing rate (O), power (ϱ), and response time (π) of nodes $v_{[1,4]}$ at different qualities. the quality level for nodes $v_{[1,4]}$ is designated by the sampling rate. Thus, q_1 which is the highest quality level has the highest rate of outgoing items, versus the lowest quality level q_3.

Table 6.1: Nodes $v_{[1,4]}$ resource usage.

	O (items per sec)			ϱ (watts)			π (msec)		
	q_1	q_2	q_3	q_1	q_2	q_3	q_1	q_2	q_3
v_1	100	75	50	80	65	40	10	13.3	20
v_2	90	60	30	75	55	35	11.1	16.6	33.3
v_3	100	70	40	80	65	40	10	14.3	25
v_4	120	90	60	85	70	40	8.3	11.1	16.6

Second, we introduce the power usage and response time of $v_{[5,9]}$. Resource usage of nodes $v_{[5,9]}$ is affected by the rate of incoming data and the node's quality level. The rate of incoming data is the sum of the rates of outgoing data of predecessors ($v_{[1,4]}$). Figure 6.2 illustrates the relationship of incoming rate and quality versus power and response time. As quality decreases, power consumption decreases and response time increases. Note that outgoing rate of nodes $v_{[5,8]}$ is the same as their incoming rate.

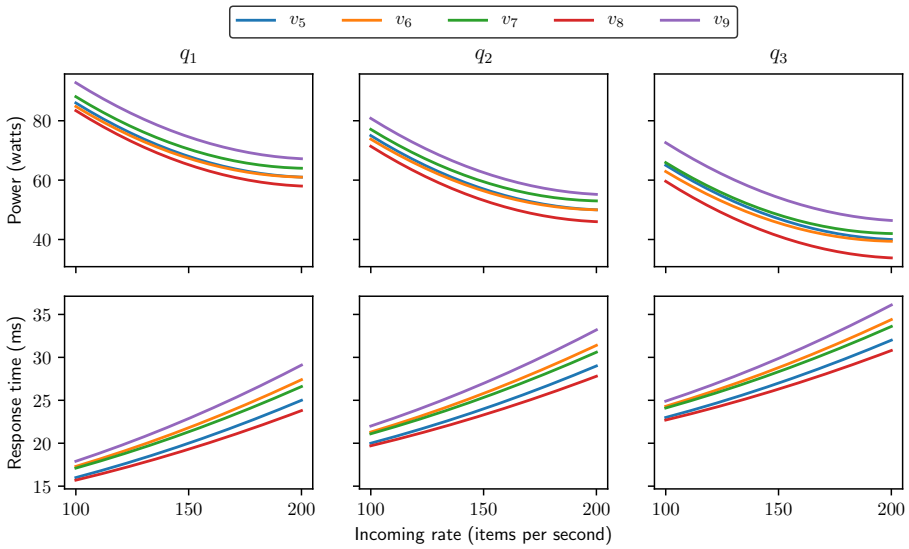


Figure 6.2: Relationships of $v_{[5,9]}$ quality and incoming rate versus power and response time.

6.3.2 Accuracy

Next, we define accuracy for nodes $v_{[5,9]}$. Accuracy is affected by the quality level of the node as well as the accuracy of its predecessors. In this example we model this relationship as a linear equation in the following form:

$$\alpha_v = A \times \mathcal{P}_v + B \times \hat{\mathcal{Q}}_v + c \tag{6.13}$$

where c is a constant, $\mathcal{P}_v \in \Omega_\alpha^{|P_v|}$ is a vector of accuracies of node v's predecessors, and A is a vector of the coefficients of these accuracies. $\hat{\mathcal{Q}}_v \in [0,1]^{|Q_v|}$ is a unit vector of binary values indicating the quality assigned to node v while B is a vector of the coefficients of each quality level. That is $\hat{\mathcal{Q}}_v = (\rho_{v,1}, \rho_{v,2}, \cdots, \rho_{v,|Q_v|})$ such that:

$$\rho_{v,i} = \begin{cases} 1 & \kappa_v = q_{v,i} \\ 0 & \kappa_v \neq q_{v,i} \end{cases} \tag{6.14}$$

Table 6.2 assigns values to vectors A and B and constant c in Equation 6.13 to concretely define the accuracy of nodes $v_{[5,8]}$. We define the accuracy of node v_9 separately as follows:

$$\begin{aligned} \alpha_{v_9} = &- 0.12\alpha_{v_5} - 0.06\alpha_{v_6} - 0.07\alpha_{v_7} - 0.04\alpha_{v_8} \\ &+ \rho_{v_8,1} + 0.89\rho_{v_8,2} + 0.71\rho_{v_8,3} + 4.0 \end{aligned} \tag{6.15}$$

Table 6.2: Coefficients of the accuracies of nodes $v_{[5,8]}$ in Equation 6.13.

	A_0	A_1	B_0	B_1	B_2	c
v_5	-0.15	-0.01	1.0	0.82	0.67	2.0
v_6	-0.14	0.00	1.0	0.85	0.73	2.0
v_7	-0.08	-0.09	1.0	0.79	0.62	2.0
v_8	-0.12	-0.09	1.0	0.81	0.59	2.0

6.3.3 Resource Flows

In our example, two resources flow through the network: accuracy and response time. Below we define the functions that dictate how these resources flow in and out of nodes.

Accuracy

we have defined the inflow and outflow of accuracy implicitly in Subsection 6.3.2. We now detail how the definition maps to the flow of accuracy as a resource. The inflow of accuracy depends on the specific weights we assign to specific incoming edges. Table 6.2 demonstrates these weights in vector A. For instance, the inflow of v_5 is defined as follows:

$$\zeta^{\alpha}_{v_5} = Z^{\alpha}_{v_5}(\{\nu^{\alpha}_{(v_1,v_5)}, \nu^{\alpha}_{(v_2,v_5)}\}) = -0.15\nu^{\alpha}_{(v_1,v_5)} - 0.01\nu^{\alpha}_{(v_2,v_5)}$$

The outflow of accuracy depends on the accuracy of the node as follows:

$$\xi^{\alpha}_{v_5} = \Xi(\zeta^{\alpha}_{v_5}, \mu^{\alpha}_{v_5}) = \zeta^{\alpha}_{v_5} + \rho_{v_5,1} + 0.82\rho_{v_5,2} + 0.67\rho_{v_5,3} + 2.0$$

which matches v_5's coefficients (B and c) in Table 6.2.

Response time

response time increases as an item accumulates processing time while flowing through the network. Thus, the inflow response time of a node is the maximum response time across all it's predecessors. Let π be the response time resource. We define Z^{π} (see Equation 6.3) as follows:

$$Z^{\pi}_{v} = \max\Big\{\{\nu^{\pi}_{e} \mid e \in \{(v', v) \mid v' \in P_v\}\}\Big\}$$

The outflow of response time is defined as follows:

$$\Xi^{\pi}_{v}(\zeta^{\pi}_{v}, \mu^{\pi}_{v}) = \zeta^{\pi}_{v} + \mu^{\pi}_{v}$$

6.3.4 Resource Bounds

We define seven resource bounds on the example system in Figure 6.1. First, we enforce five power bounds, denoted in the figure by the dashed rectangular borders. These bounds are defined as follows:

$$\begin{aligned} b_{\varrho,1} &= \langle \varrho, \{v_1, v_2\}, 140 \rangle \\ b_{\varrho,2} &= \langle \varrho, \{v_3, v_4\}, 140 \rangle \\ b_{\varrho,3} &= \langle \varrho, \{v_5, v_6\}, 150 \rangle \\ b_{\varrho,4} &= \langle \varrho, \{v_7, v_8\}, 150 \rangle \\ b_{\varrho,5} &= \langle \varrho, \{v_9\}, 90 \rangle \end{aligned} \tag{6.16}$$

Next, we define bounds on the response time.

$$b_{\pi,1} = \langle \pi, \{v_s\}, 0, 0 \rangle$$

This bound serves as an initialization of the response time resource. All response time flows initially from v_s onto the entire network. For a resource like energy, the bound at v_s would enforce that the energy budget flows from v_s to the rest of the network.

$$b_{\pi,2} = \langle \pi, \{v_9\}, 0, 50 \rangle$$

Thus, the maximum response time allowed at the aggregation node v_9 is 50ms.

Finally, we can also initialize accuracy as follows:

$$b_{\alpha} = \langle \alpha, \{v_s\}, 1.0, 1.0 \rangle$$

Thus, accuracy is initially 100% at node v_s.

6.3.5 Objective

The objective for the toy example is to maximize accuracy at the aggregation node v_9:

$$\max \alpha_{v_9} \tag{6.17}$$

6.4 Linear Program

The transformation of the model described in Section 6.2 to an ILP is straightforward, given all flow and resource functions are linear (Z, Ξ, ϑ, φ, and ψ). The integer variables in the model are the quality variables ρ which indicate which quality level a node uses. In this section, we introduce an LP relaxation of the problem to support solving larger models in reasonable execution time. The following subsections outline the relaxation.

6.4.1 Quality Fractions

The first step in constructing a linear program relaxation is to replace binary variables $\rho_{v,i}$ with continuous versions. Recall that $\rho_{v,i}$ indicates whether node v is configured to run at quality level $q_{v,i}$. To transform the discrete choice between quality levels to a continuous one, we use $\lambda_{v,i} \in [0, 1]$ to indicate the fraction of items that are processed at quality level $q_{v,i}$. This implies the following:

$$\forall v : \sum_{i}^{|Q_v|} \lambda_{v,i} = 1 \tag{6.18}$$

Since we are dealing with stream processing applications, it is not unreasonable to assume that a consumer can process incoming items at different quality levels based on the designated fractions $\lambda_{v,i}$. This assumption becomes more appropriate if the total resource consumed is linear in the consumption of the individual fractions. Power is a resource that can be approximated in such a way without significant loss of accuracy [72]. This assumption allows us to define the resource relationship functions φ_v^γ in a piece wise linear fashion:

$$\varphi_v^\gamma(r_v, \lambda_{v,1}, \cdots, \lambda_{v,|Q_v|}) = \varphi_{v,1}^\gamma(r_v, \lambda_{v,1}) + \cdots + \varphi_{v,|Q_v|}^\gamma(r_v, \lambda_{v,|Q_v|}) \tag{6.19}$$

Where $\varphi_{v,i}^\gamma$ is a function that calculates the consumption of resource γ by node v at quality level i. Assuming $\varphi_{v,i}^\gamma(r_v, \lambda_{v,i})$ is linear in the incoming data rate r_v and the fraction $\lambda_{v,i}$ of items processed at quality level $q_{v,i}$, we now have a linear expression for the consumption of resource γ by node v.

6.4.2 Non-linear relationships

The assumption that $\varphi_{v,i}^\gamma(r_v, \lambda_{v,i})$ is linear in its parameters might be too simplified for some relationships. A more reasonable assumption is that the fraction $\lambda_{v,i}$ is multiplied

by the rate of incoming data r_v to scale it proportionally to the fraction of items being processed at quality level $q_{v,i}$. In such a case, we have a term that is the multiplication of two continuous variables, which takes us outside the realm of linear programming. In this case, we resort back to using the binary variables $\rho_{v,i}$ and formulating the problem as an ILP. Hence, $\varphi^{\gamma}_{v,i}$ is defined as follows:

$$\varphi^{\gamma}_{v}(r_v, \rho_{v,1}, \cdots, \rho_{v,|Q_v|}) = \varphi^{\gamma}_{v,1}(r_v) \times \rho_{v,1} + \cdots + \varphi^{\gamma}_{v,|Q_v|}(r_v) \times \rho_{v,|Q_v|} \tag{6.20}$$

This definition results in the multiplication of a continuous variable (r_v) by a binary variable ($\rho_{v,i}$), assuming $\varphi^{\gamma}_{v,i}$ is linear in the incoming data rate r_v. Fortunately, the multiplication term can be eliminated using techniques well known in optimization literature [18]. To eliminate the multiplication term, we replace it with the variable $y_{v,i} = r_v \times \rho_{v,i}$ such that:

$$\begin{aligned} y_{v,i} &\leq u_{r_v} \rho_{v,i} \\ y_{v,i} &\leq r_v \\ y_{v,i} &\geq r_v - u_{r_v}(1 - \rho_{v,i}) \\ y_{v,i} &\geq 0 \end{aligned} \tag{6.21}$$

where u_{r_v} is the upper bound of the incoming data rate r_v.

To obtain an LP from this formulation, we simply relax the problem by replacing the binary variable $\rho_{v,i}$ with $\lambda_{v,i}$ in every constraint in Inequalities 6.21. Thus, the final form of φ^{γ}_{v} is as follows:

$$\varphi^{\gamma}_{v} = \sum_{i}^{|Q_v|} C_{v,i} \times y_{v,i} + C_v \tag{6.22}$$

where $C_{v,i}$ and C_v are constants, and $y_{v,i} = r_v \times \lambda_{v,i}$.

6.4.3 Accuracy

We showed in Subsection 6.2.6 that accuracy is calculated differently, since it depends on the quality level as well as the accuracies of the nodes connected to incoming edges (see Equation 6.10). However, our approach to relaxing accuracy is the same. Let us first define the accuracy of node v in terms of the binary variables $\rho_{v,i}$:

$$\alpha_v = \sum_{i}^{|Q_v|} \rho_{v_i} \times \psi_{v,i}(\{\alpha_{v'} \mid v' \in P_v\}) \tag{6.23}$$

where $\psi_{v,i}$ is a function that calculates the accuracy of node v at quality $q_{v,i}$ and based on the accuracy of all predecessor nodes. We assume that $\psi_{v,i}$ is linear in the accuracies of predecessors $\alpha_{v'}$. Since the expression now contains the multiplication of binary variables ($\rho_{v,i}$) by continuous variables ($\alpha_{v'}$), we replace it with

$$z_{v,i,v'} = \rho_{v,i} \times \alpha_{v'}$$

which is subject to the same constraints as in Inequalities 6.21. Next, we replace $\rho_{v,i}$ with $\lambda_{v,i}$ to relax the problem. The final form of α_v is as follows:

$$\alpha_v = \sum_{v' \in P_v} \sum_{i}^{|Q_v|} C_{v,i,v'} \times z_{v,i,v'} + C'_v \tag{6.24}$$

where $C_{v,i,v'}$ and C'_v are constants.

6.4.4 Objective Function

The objective function can target any resource modelled in the system, either to be maximized or minimized. For instance, one objective is to maximize the accuracy at the final aggregator node. In systems where there are multiple leaf nodes (nodes with no outgoing edges), the objective could be to maximize a weighted average of leaf accuracy, or maximizing the minimum of all leaf accuracies.

Multiple resources can be optimized if the objective function is linear in all the resource values. Normalizing and using a weighted average is a well-known approach to combine multiple objectives into one objective function.

6.5 Summary

In this chapter, we presented a generalized model for representing stream processing applications with support for complex data flows. We argue that the model can capture complex systems such as Big Data stream processing workflows, where multiple stages of manipulations are applied to data to produce the end result. The model represents such systems as a network of producers and consumers, where nodes can be consumers of data arriving on their incoming edges, and producers of data flowing through their outgoing edges.

The purpose of the model is to manage multiple resource tradeoffs. To that end, we designed the model to support renewable and non-renewable resources as abstractions for many system resources present in practical systems. We generalized the model to capture complex relationships between resources in a simplified manner, that allows for bounding of performance. We demonstrated that the model can be used to bound multiple resources in different parts of the network, while optimizing the consumption of a user-specified resource.

To demonstrate the use of the model in determining optimal configurations, we presented a toy example which simplifies many relationships with linear abstractions. This simplification allows using linear programming relaxations to efficiently find optimal resource configurations for large and complex systems. Moreover, we discussed the challenges of constructing a linear program to determine an optimal combination of configurations that adhere to resource bounds and optimize user-specified resource consumption. We presented methods to relax the problem to obtain a bound on the objective function.

Chapter 7

Literature Review

This chapter presents a literature review of of work related to the problems discussed in the thesis. We begin in Section 7.1 by discussing work related to the low-level optimization of energy consumption. This covers work that deals with operating system schedulers and energy management device drivers that optimize energy consumption at the level of p-states and c-states. In Section 7.2, we discuss work related to the management of power and energy at the task-level across multiple nodes. Finally, Section 7.3 discusses work related to the management of precision in floating-point intensive applications.

7.1 Low-Level Energy-Efficient Algorithms

This section discusses the literature related to low-level energy-efficient algorithms. To organize this section, we divide related work into a taxonomy of three categories:

1. **DVS scheduling.** An extensive overview of work on DVS scheduling and how it relates to power-efficient algorithms for concurrency problems.
2. **DPM techniques.** An extensive overview of published dynamic power management techniques. However this line of work does not strictly target the scheduling community.
3. **Energy models for concurrency problems.** An extensive review of work that target the specific niche proposed by our research problem.

The chapter is organized as follows: Subsection 7.1.1 discusses the literature on DVS scheduling, Subsection 7.1.2 introduces previous work on DPM techniques, and finally Subsection 7.1.3 discusses algorithms tied to concurrency problems.

7.1.1 DVS Scheduling

The work on DVS scheduling is diverse between real-time systems, and a workstation-like model where latency is not an issue, rather a quality of service metric. The following subsections discuss the different approaches taken in DVS scheduling.

Interval-based approaches

The work in [52,120,123,171,186] discusses interval-based approaches to voltage scaling. In these techniques, time is divided into intervals during which CPU utilization is studied and a decision is made on whether to change the CPU frequency. Different classes of schedulers are considered in this work, mainly differentiated with the approach used to make a CPU frequency change decision. The two main categories are future-based algorithms that determine the minimum clockrate to operate at given varying constraints, or past-based algorithms that utilize recent past to predict the future.

Slack exploitation

The initial techniques to utilize DVS-enabled processors in real-time systems revolved around static slack reclamation. This is based on the observation that generally processor utilization is often far lower than 100%, which creates an opportunity to reduce energy consumption by statically reducing CPU frequency, while still honoring deadlines. The work in [131] was among the first to introduce DVS-based fixed priority scheduling for real-time systems. The work in [186] introduced an offline EDF based approach to reducing CPU energy. The work in [7, 23, 194–196] introduce online scheduling algorithms for real-time systems.

Stochastic DVS scheduling

A line of work considered devising strategies to reduce the *expected* energy consumption of a certain workload on an ideal processor. In this work [54, 55, 181, 193], a probability distribution of the workload is provided. The work in [54,55] considered scheduling multiple

tasks based on their worst case execution time. The work in [181] incorporated dynamic task behavior with slack reclamation to minimize expected energy. The work in [193] presents offline and online algorithms for hard real-time systems using stochastic workload information.

Static analysis techniques

A complementary approach is utilizing profiling information obtaining from studying the control flow graph (CFG) of the program and employing path locality [148]. The work in [128] argues the scalability issues of obtaining all branch probabilities of large programs, and instead utilizes knowledge of frequently executed paths.

DVS with power-down strategies

With the observation of the cost of leakage power, it became apparent that consideration of power-down strategies (DPM) in combination with slowdown (DVS) should be considered. The work in [70] considered the combined problem for a real-time system. Following the authors' work on surveying the algorithmic problems in energy management [69], they proposed a 3-approximation offline algorithm, and was later followed by an online algorithm [68]. The work in [132] considered the problem for fixed-priority systems. The work in [88] studied procrastination based scheduling in periodic real-time systems.

Feedback control techniques

Upon realizing the inadequacy of pure DVS techniques in adapting to systems with variable load, the work in [87, 148, 149] attempted a feedback based approach for soft real-time systems. The work in [78] extended these approaches to cover hard real-time systems.

Comparison to our work

Our problem differs from work on DVS scheduling in the following aspects:

- Our work does not tackle the problem from a scheduling perspective; specifically not real-time scheduling, on which most of the focus exists. While DVS scheduling in real-time systems makes sense, for general systems pure DVS scheduling has been argued to lack adaptivity to variable load and varying system dynamics [149]. In

fact, work in feedback control techniques for real-time systems attempts to counter this deficiency.

- We focus on idle power as well as voltage scaling. The majority of work on DVS scheduling deals with the problem of selecting the frequency to assign to a task, with the exception of the line of work by Irani [70], which also considers power-down strategies. However, similar to the first point, the work is rooted in scheduling theory and thus studies competitive ratios and performs adversary analyses. This is a different objective than our problem proposes: namely studying the efficiency of concurrency algorithms and data structures.

With the differences mentioned, we should also demonstrate how our approach to the problem builds on top of basic concepts used in DVS scheduling:

- **Interval-based scheduling.** We utilize the basic concept of interval based scheduling in constructing our adaptive approach. By observing the system periodically decisions can be made adaptively according to varying workload.
- **Slack exploitation.** Slack exploitation is a fundamental concept that is utilized in many approaches in energy management. While it is not used in its strict sense with respect to scheduling, we study idle periods during which a process is waiting for input in an under-utilized environment.
- **Feedback control.** As mentioned above, feedback control is a fundamental component in constructing adaptive approaches. We utilize different feedback mechanisms in designing our adaptive approaches.

7.1.2 DPM techniques

DPM techniques have been extensively studied by both the research community [66, 68, 136, 150, 157] and industry [3, 32, 67]. ACPI [32] demonstrates the realization and combined effort of the industry to tackle the problem of energy efficiency in a mobile and battery powered era.

Designing DPM strategies

According to a survey on designing DPM strategies [94], research has been classified into two categories: *predictive schemes* and *stochastic schemes*. Recently, approaches based on

machine learning algorithms have emerged, which creates a third category: *learning-based schemes*.

Predictive schemes Predictive schemes generally attempt to predict a device's usage based on history, and consequently decides to change the device power state accordingly. When discussing DPM techniques, we focus on power states in which parts (or all) of the CPU is idle. The majority of work on predictive schemes follows an adaptive approach, in which observed system behavior with respect to idle periods affects the prediction of future idle periods. Various work in the literature has tackled this problem [29, 66]. The work in [66] uses an exponential-average methodology in predicting the next idle period and utilizes sleep states accordingly. The work in [29] utilizes a sliding window in the characterization of non-stationary (initially unknown) service requests.

Stochastic schemes Stochastic schemes use observations to construct a probability distribution of usage patterns. This is then used to formulate an optimization problem, for which the solution is a DPM scheme. Various work has tackled such problem [112, 113, 130, 151]. The work in [130, 151] formulates the optimization problem on Markov decision processes. The work in [112, 113] uses formal methods to verify the correctness of a DPM strategy.

Machine-learning schemes Work on machine-learning-based techniques in designing DPM strategies argues that simple prediction techniques such as those presented in [66, 157] do not perform well when requests are not highly correlated, and do not consider performance constraints, resulting in a lack of a trade-off control between performance and energy. The work in [37] utilizes machine learning to select a strategy online. The work in [160, 161, 164, 170, 189] uses reinforcement learning to construct a DPM strategy. The work in [160] utilizes Q-learning to learn the best next state given the current system state, using a reward and penalty approach. However, this approach has a large overhead. The work in [170] improved the convergence rate by using TD learning.

Further approaches leveraging machine learning target multi-core systems, such as those presented in [51, 59, 71, 73, 187].

OS level DPM techniques

This section introduces various prominent work on OS-level dynamic power management [14] that have been integrated in actual systems.

Linux on-demand governor The on-demand governor [118] has become the defacto power management controller in linux systems due to its simplicity and effectiveness. It operates by observing CPU utilization and manipulates CPU frequency such that the CPU maintains approximately 80% utilization. It supports SMP as well as multicore and multithreading architectures.

ECOSystem ECOSystem [191, 192] treats energy as a first-class system resource. The work introduces a *currentcy* model with which energy accounting is possible per application, and a target energy discharge can be achieved by fairly allocating *currentcy* to running applications and services.

Nemesis OS The work in [110] is similar to ECOSystem in the sense that the objective is managing energy as a resource. Nemesis OS also employs per process energy accounting, yet also incorporates a feedback model with applications, where the OS communicates energy usage to the application which should act accordingly. This is similar to what widely exists in some mobile platforms with respect to memory management: the OS communicates a memory warning to the sandboxed application, which should consequently release some objects to reduce its memory footprint.

GRACE project The Illinois GRACE project [146, 165] argues that all layers of the system should coordinate to adapt to usage needs while targeting reduced energy consumption. Thus, the project proposes a framework that comprises global adaptation, per application adaptation, and per layer adaptation.

PowerNap The work in [101] targets server based systems, which are shown to have generally low utilization. The authors propose a method to transition the system from fully active to fully idle faster, saving time and energy in the transition process.

Comparison to our work

The approach taken in research on DPM techniques resembles our approach in many ways. The following points illustrate the similarities:

- Contrary to DVS scheduling, the work on DPM techniques focuses on exploiting idle time, either exclusively or in unison with frequency scaling. This allows for a more

integrated approach to managing energy, versus working solely with either DPM or DVS. Granted, a holistic approach that includes DVFS, DPM, and scheduling is still considered by part of the community as a goal that would allow for seamless integration between the three components, versus the independence and lack of communication that currently exists [85].

- Our problem shares the demand for online adaptive approaches with work on DPM techniques. More specifically, the work mentioned above on DPM veered from the path of theoretical scheduling and competitive analyses, which is the approach we followed.

- Work on DPM targets general purpose systems that can experience variable workloads, which has directed most of the work towards experiments with real workloads [101], or integration in the linux kernel [118, 146, 192]. This is illustrated in Subsection 7.1.2.

- Work on DPM involves some kind of prediction mechanism to adapt to varying workload (Subsection 7.1.2). Our approach to the proposed problem also utilizes prediction methods.

Our approach compared to work on DPM techniques is better illustrated as a complementary relationship. In fact, our work should integrate nicely with work on DPM to form a diverse basis of OS-level energy management. This is further clarified in the following points:

- Our work assumes a DPM strategy is in place. This is a fundamental differentiator between our work and the literature on DPM. Our objective is not to replace current DPM techniques with smarter / more robust alternatives. We assume the existence of a DPM strategy and a DVFS governor. Our objective is to improve the energy efficiency of concurrency primitives given the existence of DPM and DVFS.

- Hence, our focus on concurrency primitives complements the work on DPM by following mostly the same approach but targeting a different part of the OS: namely the part most involved in idling and activating the CPU, i.e concurrency. While DPM techniques are closer to the hardware level, mostly concerned with management of CPU idle states or frequencies, targeting concurrency allows us to be closer to the application level, covering multithreaded behavior and resource producer / consumer relationships.

- Most of the work on DPM uses as input some low level utilization monitor such as CPU utilization, performance counters, interrupt monitors, etc... However, since our work is concerned with concurrency primitives, our input is data item streams or data structures involved in concurrency operations. This allows for more insight into the behavior of the application, without getting involved into the user code. This is a main objective in energy management research in general, to be seamless and transparent to the developer. We believe by targeting concurrency primitives, we reside in a niche that is closest to the application yet can be fully automated without developer assistance. This is in comparison to some currently existing techniques that require a developer to call certain API that provides information on the energy demand of a certain task.

7.1.3 Energy Models for Concurrency Problems

This section discusses work most closely related to our work in Chapter 2. The work cited in this section mostly targets energy modelling of concurrency data structures.

Work on energy models

The work in [31] uses performance counters to estimate the power consumption of a system. The paper proposes *power weights*, which map hardware performance counters to power consumption. The work in [168] proposes a method to estimate per-core power consumption using CPU frequency and IPC as the only PMC used. The work in [163] proposed a per-instruction power model, in which they calculate a cost per instruction and extrapolate power consumption of compositions.

Work on energy models for queues

The work in [65] experimentally evaluates the performance and energy of both lock-based and lock-free FIFO queues using a set of contentious workloads. By comparing execution time, peak power, and total energy consumption, the paper concludes that lock-free queues are more energy efficient than their lock-based alternatives.

The work in [44] compares locks to software transactional memory in terms of energy efficiency. The premise of the paper is based on the fact that synchronization techniques are designed to exploit multicore architectures and are optimized for performance. With the increasing demand for energy efficient systems, it is necessary to study the energy

efficiency of popular synchronization techniques. While comparing locks with STM has been done before [108], this work is the first to execute the comparison on real hardware. The paper also studies the impact of DVFS and DPM strategies on the energy consumption of synchronization techniques.

The work in [6] introduces an analytical model for the energy consumption of lock-free queues. The model focuses on steady state behavior as its basis. The paper studies both performance and power dissipation of lock-free queues, by constructing models for each. Such models are based on a subset of data points and thus are more feasible to construct than exhaustive stochastic approaches that are used to build energy models.

Comparison to our work

Our work can be compared to the work on energy models for concurrency problems with the following points:

- Energy modelling provides the distinct advantage of simplicity in prediction of energy demand of a system and its feasibility by avoiding exhaustive empirical analysis. However, such models suffer from two drawbacks: (1) assumptions that limit the applicability of the model to a diverse range of systems, and (2) low accuracy due to extrapolation. Active research in energy modelling should result in high fidelity models with reduced empirical dependence as shown in [6]. In that sense, currently we rely on online adaptation to construct *real-time models* used to make energy-aware decisions, yet leveraging advanced energy models partially or fully can further improve our results. Thus, the work on energy modelling should provide a launching point for our energy management techniques.

- Our objective is managing energy consumption online. An inherent condition is that these approaches should have a low energy footprint, otherwise its existence defeats its purpose. This requires lightweight models with minimal computational and energy demands. This objective is not stressed in work on energy modelling, since their purpose is to construct a high fidelity model and not a robust online mechanism.

- Another differentiator to consider is that we assume the existence of a DPM strategy and a DVFS governor. An energy model that takes these factors into account will be considerably complex, again defeating its purpose.

- In comparison to the work in [6], the lack of a strict model allows us to adapt to variable workloads and provide a more robust experience. While the work in [6] is based on the assumption of a steady state system, we use actual datasets and real life experiments to provide more confidence in the applicability and adaptability of the proposed approach.

7.2 Cross-node Power and Energy Management

This section discusses the literature related to the problem of managing power and energy in compute clusters. To organize this chapter, we divide related work into a taxonomy of three categories:

1. **Dynamic workload distribution.** An overview of work on dynamic workload distribution as an early alternative to power distribution.
2. **Energy saving.** An overview of published work on reducing energy consumption of compute clusters. We show how this problem differs from the proposed problem.
3. **Power distribution.** An extensive review of work that target the specific niche proposed by our research problem.

The section is organized as follows: Subsection 7.2.1 discusses the literature on dynamic workload distribution, Subsection 7.2.2 introduces previous work on energy efficient computing clusters. Subsection 7.2.3 discusses algorithms and runtimes for power distribution. Finally, Subsection 7.2.4 presents a summary of the work on power management in large scale data centers and illustrates how our research problems fits in this spectrum.

7.2.1 Dynamic workload distribution

Load balancing has been studied extensively decades ago. With the explosion of cloud computing, load balancing in internet scale services is crucial, and the literature on the topic is dense. The survey in [114] provides an overview of the current state of load balancing in cloud computing. However, the rising relevance of heterogeneous clusters mutated the problem towards dynamic workload distribution based on node capabilities [137]. In this section, we discuss different approaches to improving performance in heterogeneous clusters.

Performance in heterogeneous clusters

The work in [13,173,175,176,185] is among the first to study the performance of a parallel program in a heterogeneous environment. The work in [13,173,185] attempts to construct models to evaluate and predict the performance of a parallel program running in a heterogeneous cluster. The work in [175,176] is among the first to suggest adaptive techniques to manage a resource - namely memory - in a heterogeneous setting.

Load balancing in heterogeneous clusters

The work in [16,19,129] tackles load balancing in heterogeneous settings. The work in [19] specifically addresses the problem of assigning tasks to nodes according to node capabilities. The work in [129] focuses on I/O intensive applications, and approaches the problem by proposing a method to migrate high intensity I/O bound workloads to low I/O utilization node.

The series of papers by Yang et al [27,183,184] tackle the interesting problem of scheduling parallel loops on heterogeneous clusters. They propose a series of methods and enhancements to build self-scheduling parallel loops.

MPI

Naturally, the evolution of the work presented in the previous sections led to the focus on specific sets of applications that are widely used in HPC. The first candidate is MPI. The work in [17] targets building an adaptive MPI layer that relies on migration to achieve load balancing. The book in [74] compiles most of the work on object migration, and includes work on related frameworks such as MPI and Charm++. Papers such as [53,86] target building a heterogeneous message-passing platform. The work in [86] provides the programmer with interfaces that allow for describing an underlying parallel algorithm that supports heterogeneity. The work in [53] presents an architecture for heterogeneous support for Open MPI that is transparent to the user. The paper classifies heterogeneity into four categories: processor, network, run-time environment, and binary heterogeneity. Our main focus is processor heterogeneity.

MapReduce

The work on optimizing the performance of MapReduce on heterogeneous has gathered especially significant attention. Due to its widespread use in cloud computing, performance

optimization of MapReduce on heterogeneous clusters has been extensively studied. The work in [2, 24, 127, 162, 180, 190] all deals with optimizing MapReduce in heterogeneous environments. In [190], the authors identify that the task scheduler of Hadoop [172] can cause significant performance degradation in a heterogeneous cluster. The authors propose a new scheduling algorithm (LATE) that improves performance in a heterogeneous cluster. The work in [162] also proposes a Hadoop scheduler but one that caters to heterogeneous workloads. With the same objective, the work in [180] focuses adapting Hadoop to heterogeneous clusters, yet via data placement. The work in [24] criticizes the LATE scheduler due to its static operation, and suggests an adaptive scheduler that constantly updates its decisions according to the continuously varying environment. The work in [127] extends the adaptive scheduler with knowledge of the underlying hardware capabilities.

Comparison to the proposed problem

The proposed problem has one fundamental difference from the problems tackled in this section: power distribution versus workload distribution. The basic argument is that workload distribution suffers from the following drawbacks:

- *Static workload distribution is difficult.* The redesign of parallel algorithms accumulating decades of work on design and implementation is difficult and costly to achieve. It also relies heavily on models of the heterogeneous cluster and its varied capabilities. This is a fragile dependence, since small changes in the cluster can vary the output significantly. Further complications such as multiple tasks running concurrently increases the complexity of the model and the difficulty of the static distribution decision. Another example is software aging of a node, which again has to be mitigated in the static distribution decision.

- *Adaptive workload distribution is costly.* While adaptive workload distribution is an alternative to static distribution that avoids most of its problems, it is costly in terms of communication, overhead, and subsequently energy. Transferring of data between nodes to mitigate heterogeneity is bound to introduce overhead that does not exist in a static distribution setting.

Another major difference is that the work mentioned above does not consider power or energy efficiency. The next section presents an overview of work targeting energy efficiency in HPC clusters.

7.2.2 Energy saving

The work on energy efficiency in clusters spans multiple research directions. The survey in [92] covers the various techniques in power management for HPC. Most of the basic techniques have been discussed earlier in this article, however the survey discusses metrics applicable to the HPC domain, and profiling techniques for parallel applications. The following subsections detail the different research directions beginning with metrics and profiling, and then categorizing research efforts into work related to runtime systems, and work related to studying optimal energy savings.

Metrics

As mentioned above, the survey in [92] discusses metrics used for evaluation of the energy efficiency of an HPC cluster. The most basic metric is energy-delay to the n^{th} (ED^n) [119], which originated in the VLSI community. ED2P (ED^2) was first introduced by [20] attempts to cancel out the exponential effect of frequency scaling (energy is directly proportional to the square of frequency in most contexts). In [47], the author proposes a generalized metric called *Weighed ED2P*, which accounts for the variation in systems. The work in [63] is among the first to argue that classical metrics such as the performance-power ratio and energy-delay product are biased for massively parallel systems.

Energy profiling

The work in [46] proposes an analytical model (PowerPack) to predict execution time in power-constrained clusters. The model also predicts the energy-delay tradeoff according to various system configurations. The work in [152] uses PowerPack to study the power and energy profiles of the HPC Challenge benchmark. This paper is able to use PowerPack to correlate application functions to power profiles, memory access patterns to power consumption, and energy consumption to workload. PowerPack is further extended in [50] to support multicore multiprocessor nodes and resolves the issues arising from applying DVFS to such nodes. Similarly, the work in [153] proposes an analytical model that is based on iso-efficiency [80]. The paper denotes their proposed model as *iso-energy-efficiency*, and applies it to various applications in the NAS parallel benchmark [9].

Runtime systems

The work in [48, 62] proposes a distributed DVS scheduling framework to reduce energy consumption in HPC clusters. The work studies the energy-delay tradeoff when using their respective runtime systems. The work in [49] extends that in [48] to support autonomous DVS scheduling on clusters, as well as support for multicore multiprocessor systems. The work also couples the run time system with a performance prediction model. The work in [49] relies on a feedback mechanism, where history based performance counter readings are used to determine the next frequency to use.

Optimal energy savings

Many researchers targeted the problem of finding optimal energy savings without impacting performance. The work in [43] studies the trade-off between energy and delay for a wide set of applications. The paper also studies metrics to use to predict memory or communication bottlenecks. The paper makes two significant conclusions: (1) in the case of programs that have a memory or communication bottleneck, power management can reduce energy consumption significantly while costing small delay. (2) for some programs, increasing the number of nodes and decreasing their operating frequency can in fact reduce energy consumption and concurrently improve performance.

The work in [64, 93] attempted to tackle the problem on a single processor. The work mainly proposes an alternative DVFS strategy that maintains the same performance at reduced energy consumption. The work in [174] discusses the effectiveness of using control theory in power management. The series of papers [177–179] constructs an ILP model to determine the minimum energy consumption that a program can consume on a single processor. Then, the authors propose a heuristic to approximate the ILP, and couples this with an analytical model for energy consumption prediction.

Comparison to the proposed problem

The proposed problem differs in multiple aspects from the work on energy saving. These aspects are as follows:

- Our objective is not to reduce energy consumption. The issue with this objective is that it sets the work on track to the highly argued territory of energy versus delay. With exascale computing demands becoming increasingly imminent, attempting to

improve performance given a provisioned power bound is a more realistic objective, specially that delay is and will generally always be intolerable in the HPC domain.

- Very little of the work on energy efficiency in HPC clusters targets heterogeneous clusters. We deal with two types of heterogeneity: processor and workload. While there is an abundance of work on heterogeneity with the objective of performance (see Section 7.2.1), very little work consider the problem in a power-constrained setting or from an energy efficiency perspective.

7.2.3 Task-Level Power Distribution

In this section, we discuss work most closely related to our work in Chapters 3, 4, and 6. We will discuss five main papers, each in its own subsection, with a comparison to the our work.

Minimizing execution time on energy-constrained clusters

In [156], the authors propose an analytical model coupled with actual program execution to determine a schedule that minimizes execution time given a bound on energy consumption. A schedule in the context of the paper is a tuple identifying the number of nodes to be used, and the sequence of frequencies to assign to each computational phase. The authors propose a method that combines performance modelling, prediction and actual program execution, to determine the minimum execution time. The authors claim that the proposed method is an improvement over an exhaustive search in the set of all possible schedules.

Our research differs from this work in the following points:

- Our setting is fundamentally different than that of [156]. We attempt to reduce execution time (which is a common goal) given a power bound (not an energy bound), in a heterogeneous cluster (versus homogeneous) online (versus offline).
- We do not require trial runs of the program to construct a performance model. In fact, we do not attempt to construct performance models.
- The task model in this work is simple and hides complex dependencies and relationships that exist in most parallel applications.

Jitter

In [75], the authors attempt to exploit slack time to reduce energy consumption. The proposed framework decreases the frequency of nodes that are assigned less computation, thus resulting in an overall reduction in the cluster energy consumption. The idea behind the paper is as follows: if a node is assigned less computation than its peers, it will probably arrive at a global synchronization point earlier than other nodes. If we were to decrease its frequency while it executes its job, it would arrive *just in time* to the synchronization point concurrently with its peers, which will result in all nodes unblocking and continuing execution.

Our work and proposed approach differs from this work in the following key points:

- Our problem setting is different. The objective of the work in [75] is to reduce energy consumption while minimally impacting performance. It has been argued repeatedly that delay is not acceptable in the HPC community. We attempt to improve performance given a power budget, which is a more acceptable goal and which supports exascale computing.
- This point is further exemplified in the results mentioned in the paper. While energy is reduced by 8%, execution time is increased by 2.5%. This launches the debate on the value of both energy reduction versus execution time increase. Such a comparison depends on many external factors and hence limits the applicability of such approaches in general.
- Our proposed approach builds on top of a task dependency model, where blocking nodes are detected in run time.
- Jitter requires a special directive to indicate the completion of a computation phase. Our problem requires a developer-transparent approach.
- Our proposed approach simultaneously decreases the frequency of a blocked node and increases the frequency of the blocking nodes to improve overall performance. We propose a ranking mechanism to sort blocking nodes and determine their quota of extra power.

Bounding energy consumption

In [141], the author addresses the problem of determining a DVS schedule across the cluster such that energy consumption is minimized. While the problem of assigning frequencies to

jobs to minimize energy consumption is NP-complete, the authors proposes an LP model to bound the energy consumption of an application given its communication trace and the cluster characteristics. This is very useful to evaluate run time algorithms, since the quality of the algorithm can be quantified against the energy consumption bound.

Our research problem and proposed approach differs from this work in the following key points:

- The work in [141] proposes a method to bound the energy consumption of an MPI program using an LP model that knows the execution time of jobs on machines and the effect of changing the frequency on their speedup. Our work proposes an online heuristic that distributes power dynamically without knowledge of the jobs. This is a major distinction.
- Bounding energy is a different problem versus minimizing execution time given a power budget. The constraints are different, and as of yet there is no LP formulation for the problem (specifically for a heterogeneous cluster), versus the availability of an LP formulation for the energy bounding problem.
- While our execution model is similar to the work in [141], our execution model extends it by identifying which jobs run concurrently via the "stretching" mechanism. This is preprocessed ahead of formulating the ILP model.
- Another major distinction is that our execution model supports heterogeneous clusters, versus the homogeneous cluster discussed in [141] based on its execution model.

MIMO

In [169], the authors present a control theoretic approach to power management in a cluster setting. The authors argue that single-input-single-output (SISO) control models used in the literature are only capable of controlling a single node, and fail to take into consideration a cluster in which multiple nodes share the workload. Thus, the paper focuses on multiple-input-multiple output (MIMO) control models. The solution presented follows standard controller design methodology: first, system identification is used to construct a model of the cluster, then a MIMO controller is designed based on the cluster model.

Our research problem differs from the work in [169] in the following points:

- The solution in [169] requires using system identification to build a controller that is tailored for a specific cluster. The solution cannot be automatically deployed and hence requires significant preprocessing.

- The solution does not consider data dependency between nodes, which most probably will require a non-trivial power distribution across the cluster.
- Concordantly, the paper does not construct a theoretical optimal solution that could be used a measuring stick with which we evaluate online solutions.
- The work does not support heterogeneity in the cluster hardware.

Adagio

In [142], the authors present a run time system similar to Jitter, yet improves on it in many aspects. First of all, the authors claim the delay is negligible (less than 1%). The run time system is transparent to the developer and thus does not require extra directives to be added to the code. Adagio works as follows:

- Adagio first makes predications about the next task using historical observed stack traces which are recorded when Adagio intercepts MPI calls.
- The first time a task runs, it is executed at maximum frequency.
- Adagio uses slack information to gradually slow down the task while maintaining its historical information.

Our research problem and proposed approach differs from this work in the following key points:

- Adagio targets reducing energy consumption. This is why it can be deployed separately on each node in the cluster. All the operations are local and thus optimizations cannot come from a holistic point of view, which an offline optimal scheduler will be able to achieve. This is contrary to our problem, which targets reducing execution time while enforcing a cluster power bound. Consequently, our proposed approach involves communication among nodes to coordinate power distribution.
- Critical path detection is also localized and hence there is lack of support for heterogeneous clusters where a critical path consistently spans multiple nodes. Our approach is designed to support heterogeneous clusters by adapting power distribution dynamically to mitigate differences in capabilities among nodes.

- Our approach builds on top of a task dependency model which Adagio does not support due to its locality. In our approach, a task dependency graph is constructed in run time to determine blocking nodes and rank their criticality, which is used to determine their power quota.
- Adagio only employs slowdown, while our problem requires optimal power distribution, which includes slow down and speed up. As mentioned earlier, our proposed approach simultaneously decreases the frequency of a blocked node and increases the frequency of the blocking nodes to improve overall performance.

7.2.4 Summary

This subsection presents a summary of research on power management in HPC which has been compiled in [92]. We extend this compilation by illustrating how our research problem fits in the spectrum of research on the topic. Table 7.1, which is presented in [92] compiles most of the work we mentioned in the previous section, as well as others. We have extended the table by adding one row at the end to describe our research problem using the same comparison points present in the table. As shown in the table, our research problem proposes a question that is not answered by the spectrum of work in the area: how can performance be optimized online in a heterogeneous cluster given a power bound.

Table 7.1: Summary of power management techniques for HPC presented in [92].

Technique / Tool	**Constraints**	**Mechanism**	**Profiling**	**Applicable Systems**
Load Concentration [125]	Acceptable performance	Node on/off		Homogeneous clusters
Muse [22]	SLA	Node on/off	Analytical modeling	Homogeneous clusters
Chan et al. [26]	SLA	Node on/off, DVFS	Analytical modeling	Homogeneous clusters
Horvath et al. [60, 61]	SLA	Node on/off, DVFS, Multiple sleep states	Analytical modeling	Multi-tier clusters
Heath et al. [58]		Node on/off	Analytical modeling	Heterogeneous clusters
PDC [124]	Disk bandwidth	Multi-speed disk	Simulation	Disk arrays
MAID [30]	Request time	Spin-down/up	Simulation	Disk arrays
DIV [126]	Redundancy, Throughput	Disk active or standby	Analytical modeling	Disk arrays with redundancy
Ranganathan et al. [139]	SLA, Power budget	CPU DVFS	Simulation, Online measurement	Homogeneous clusters
Femal et al. [40]	Power budget, Minimum service level	CPU DVFS, Device on/off	Analytical modeling	Homogeneous clusters with non-uniform load
MIMO [169]	Power budget	CPU DVFS	Analytical modeling, Online measurement	Homogeneous clusters
Mercury and Freon [57]	Temperature threshold, Utilization threshold	Node on/off	Analytical modeling	Homogeneous data centers
Moore et al. [107]	Cooling cost, Temperature threshold	Node on/off, CRAC supply temperature	Analytical modeling	Homogeneous data centers
Hsu et al. [62]	Relative performance, slowdown	CPU DVFS	Analytical modeling	HPC
Ge et al. [49]	Relative performance, slowdown	CPU DVFS	Analytical modeling	HPC
Freeh et al. [42]	(Energy saving) / (time delay)	CPU DVFS	Online measurement	HPC
Lim et al. [90]	E*D	CPU DVFS		HPC
Jitter [75]	Net slack	CPU DVFS		HPC
Adagio [142]	Slack	CPU DVFS	Analytical modeling, Online measurement	HPC
SLURM [188]				HPC
EETDS [197]	Performance, Energy	Node on/off	Analytical modeling	Heterogeneous HPC
pMapper [166, 167]	Power budget, Performance, Migration cost	Node on/off	Analytical modeling	Virtualized heterogeneous HPC
Stoess et al. [158]	Failure rate, Temperature constraints, Power budget, QoS	CPU throttling, Disk active/idle	PMC-based, Analytical modeling	Hypervisor-based virtual machines
VirtualPower [109]	SLA, Power budget, Power/Performance	DVFS, Node on/sleep/off		Virtualized heterogeneous HPC
Research problem	**Power budget**	**CPU DVFS**	**Analytical modeling, Simulation, Online measurements**	**Heterogeneous Clusters**

7.3 Precision Management

Approximate computing is an emerging approach to improve the performance and energy efficiency of embedded systems by sacrificing accuracy [56]. The work on approximate computing has targeted both hardware and software. In hardware, researchers have designed arithmetic circuitry that supports approximate computing to achieve higher energy efficiency [91]. In software, researchers have written tools to simulate program behavior on approximate computing hardware [103] and characterized the resilience of applications to approximate computing using computation models [28].

Various work has focused on optimizing precision for performance gains. A primary motivation for this work is reducing energy consumption. One approach uses qualifiers explicitly declared by developers to indicate parts of the program that can be approximately computed [147]. These qualifiers are used to guide an automated system into improving performance by managing the accuracy of the qualified parts of the program. This approach has also been explored from a runtime perspective, with dynamic monitoring and adaptation to maintain a QoS [8]. However, these techniques all require developer input. Our approach automatically provides the developer with recommendations supported by our error analysis method without first modifying the source code.

Utilizing lower-precision computation in machine learning applications is an emerging trend that is gaining traction [33]. Popular open source machine learning libraries such as Microsoft's Cognitive Toolkit [102] and Tensorflow [1] use both single and double precision. However, the selection of precision is left to the developer. In our work, we present an automatic approach that provides more granularity than a binary whole-program precision switch.

Finally, various work has used dynamic instrumentation to optimize performance of floating point arithmetic. This includes a technique for online cancellation detection [83], an automated mixed precision search framework [82], and a rounding-error-based general precision analysis tool [81]. Others have implemented similar techniques using a compiler framework [144]. In this work, an automatic search is employed to determine the combination of variables to switch to single precision. The tool injects instructions to cast converted variables when reading from / writing to memory. Shadow value analysis has also been used to investigate the effects of reduced precision levels [84, 143]. However, these techniques either do not focus specifically on making recommendations, or do not do so from the perspective of instructions and functions. We have demonstrated that our approach provides insight into a spectrum for managing the performance/error tradeoff using simple manipulations to the source code. We have also shown in the case study that focusing only on memory is not always as effective as considering instructions.

Chapter 8

Conclusion

This thesis has tackled the problem of managing the resource tradeoffs of power/energy constrained applications. We approached the problem in a bottom up approach, starting with studying the nuances of power/energy tradeoffs at the CPU level for streaming applications, then moving towards a higher abstraction of entire tasks, where we studied how performance can be controlled with power. Next, we designed an efficient model to capture the tradeoffs of the task-level abstraction. We next examined more resources that can affect and are affect by power and energy: namely bandwidth and precision. Finally, we combined all studied tradeoffs into a generalized model that supports constraining and/or optimizing the consumption of multiple resources. This chapter is organized as follows: Section 8.1 presents summaries and contributions of each chapter in the thesis and Section 8.2 presents the future work.

8.1 Summary and Contributions

In Chapter 2, we tackled the energy consumption of applications that can be modelled using the producer consumer problem. This covers most stream processing applications from embedded sensor fusion applications to web servers to Big Data stream processing. We made the following contributions:

- A detailed study into the energy consumption of different implementations using different concurrency primitives.

- A novel algorithm that solves the producer consumer problem while providing significant energy savings. In fact, our algorithm demonstrated a 51% reduction in energy consumption versus the widely used mutex/semaphore implementation of the producer consumer problem.
- An algorithm that strategically manages the allocated memory among parallel consumers in order to alleviate pressure from transient loads. We demonstrate that the algorithm reduces energy consumption by 10%.

In Chapter 3, we moved beyond the details of every p-state and c-state transition and studied the power performance tradeoff of entire computation tasks in parallel applications. We tackled the problem of how performance can be maximized when a cluster of nodes is operating under a power bound and executing a parallel program with inter-node dependencies. To that end, we made the following contributions:

- We designed an online power distribution heuristic that intelligently moves power from blocked nodes to the most nodes that are most likely on the critical path.
- We demonstrated how the heuristic operates in simulation and real-life experiments. Our online heuristic produces a speedup of up to a factor of 1.8 in CPU bound programs.
- We designed an integer program to identify the optimum distribution of power such that the total execution time of the program is minimized.

In Chapter 4, we built on top of our work in Chapter 3 in two ways: (1) we tackled the scalability issues of the proposed ILP, and (2) we introduced a new resource to the model: network bandwidth. To that end, we made the following contributions:

- We introduced a linear program that uses network flow principles to optimize the power distribution in a task graph with the objective of minimizing the total execution time. We demonstrated that this new model is significantly more efficient than ILP solutions, and can be used to identify an optimal for a 20800 variable problem in 26 minutes.
- We proposed using power-based staggering of parallel tasks to alleviate network bandwidth bottlenecks.
- We demonstrated how bandwidth can be included in the linear program as a resource in addition to power, energy, and time.

- We used the new model to show that the linear program staggers tasks using power when it is beneficial for the total execution time.
- We studied the practical application of the proposed staggering and demonstrate that it reduces total communication time by up to 15%. This study demonstrates that power can be used as an indirect control mechanism of other resources for which there are no explicit controls.

In Chapter 5, we tackled precision not as a binary choice of either single or double-precision floating point, but as a resource with a spectrum of control points, starting from a program that is 100% in single-precision, to a program that is 100% in double precision. This spectrum resulted in a tradeoff between performance and accuracy, with the 100% single-precision version being the fastest and least accurate, and the 100% double precision version being the slowest and most accurate. In this chapter, we introduced the novel idea that parts of the program can be converted to single precision to manage the performance accuracy tradeoff. Moreover, we made the following contributions:

- We presented a novel approach for scoring parts of the program with respect to their impact on performance versus accuracy. Our approach is based on dynamic binary instrumentation of the program under inspection.
- We provided a publicly available tool that performs this scoring of functions in a program.
- We demonstrated the applicability of the tool on a well known computer vision library used for detecting QR-code-like tags for use in visual inertial odometry by producing recommendations for functions that are most beneficial to downgrade in precision.
- We showed that we can easily produce multiple versions of the library with varying degrees of precision use, and we demonstrated how the increase of use of single precision gradually decreased accuracy and energy consumption, but interestingly increased power consumption.
- Finally, we demonstrated that our tool correctly recommended a function for downgrade which resulted in 1.3x speedup, a 16% reduction in energy consumption, and zero impact on accuracy.

In Chapter 6, we tackled the problem of constraining and optimizing resource consumption in stream processing applications that exhibit multiple resource tradeoffs. We

modelled such applications as an abstract network of producers and consumers, which can be used to model a multitude of stream processing applications. To that end, we made the following contributions:

- We proposed a novel abstract model based on network flow that captures multiple resource tradeoffs in a network of inter-dependent producers/consumers. We formalized the interactions of power, energy, execution time, accuracy, precision, bandwidth, and sampling amongst one another.
- We introduced quality as a resource, which encapsulates precision and algorithmic alternatives to capture the tradeoff between the quality of processing and its cost.
- We presented an LP relaxation formulated using the model, which allows for bounding the objective resource consumption of large networks of stream processing applications efficiently.

8.2 Future Work

The thesis has tackled the problem of managing resource tradeoffs at different granularities. It will be interesting to further expand this study horizontally.

Concurrency primitives. The producer consumer problem models many systems, however it is interesting to study how different concurrency problems and primitives can be redesigned with energy efficiency as a first class requirement. Our research approach could be extended to different low-level primitives such as `epoll` or go's channels. Such primitives are highly popular in modern applications at internet scale.

Task models. There are more complicated task models that could benefit from our modelling approach in order to provide more tailored resource management. In complex MapReduce systems, checkpointing for fault tolerance and redundancy for straggler mitigation is an integral aspect of design. The models that can be captured from such systems will introduce new challenges and potential for more tailored resource management.

Automatic precision management. Our work on precision management creates room for many extensions that could improve the experience of managing precision. A fully automatic approach will simplify the optimization process drastically and will allow for a more optimal exploration of the vast search space of precision levels. Moreover, the design and implementation of dynamic precision switching has great potential based on our studies on error evolution and its visualization.

Improved optimization of tradeoff models. We have demonstrated the complexity of the resource tradeoff model and how to reduce this complexity with approximations. There are approaches to improve the efficiency of optimization in the operations research literature as well as machine learning. Exploring these techniques allows us to reduce the relaxations and produce a tighter bound on optimality.

Explicit resource control. Our work was based around the management of power and energy versus other resources. It will be interesting to develop explicit control knobs for other physical resources and explore how our resource tradeoff models can improve resource distribution. For instance, explicit control knobs for the network bandwidth in different levels of the interconnection network would allow for more granular optimization, versus relying on power staggering to control bottlenecks.

Battery driven optimization. Battery consumption is affected by many factors: the peak, the duration of high load, the ambient temperature, the number of charge cycles, etc... Modelling these aspects allows for more detailed control over the user experience. Designing a methodology to incorporate these aspects into a model allows software to make decisions about resource tradeoffs that are more representative of the physics of the device, versus approximations of an energy source.

Power source driven optimization. Similar to battery modelling, the modelling of the power plant affects the approach with which power is distributed in a data center. There are interesting aspects such as the cost model of power at different times and different seasons, the availability of solar power and the lifetime of batteries that can provide said power, the impact of weather on renewable energy and predicting a surge/drop in availability. A model that captures such aspects could allow us to improve efficiency and reduce energy costs.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[2] Faraz Ahmad, Srimat T Chakradhar, Anand Raghunathan, and TN Vijaykumar. Tarazu: optimizing mapreduce on heterogeneous clusters. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 61–74. ACM, 2012.

[3] AMD. AMD PowerNow!, 2015. http://support.amd.com/TechDocs/24404a.pdf.

[4] University of Michigan APRIL Robotics Laboratory. AprilTags. https://april.eecs.umich.edu/software/apriltag.html, 2017. [Online; accessed 03-April-2017].

[5] Martin Arlitt and Tai Jin. 1998 World Cup web site access logs, 1998.

[6] Aras Atalar, Anders Gidenstam, Paul Renaud-Goud, and Philippas Tsigas. Modeling energy consumption of lock-free queue implementations. Technical report, Chalmers University of Technology, 2015.

[7] Hakan Aydin, Vinay Devadas, and Dakai Zhu. System-level energy management for periodic real-time tasks. In *Real-Time Systems Symposium, 2006. RTSS'06. 27th IEEE International*, pages 313–322. IEEE, 2006.

[8] Woongki Baek and Trishul M Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Sigplan Notices*, volume 45, pages 198–209. ACM, 2010.

[9] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[10] David H Bailey, Leonardo Dagum, Eric Barszcz, and Horst D Simon. NAS parallel benchmark results. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 386–393. IEEE Computer Society Press, 1992.

[11] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computers*, 40(12):33–37, 2007.

[12] Luiz André Barroso. The price of performance. *Queue*, 3(7):48–53, 2005.

[13] Victor E Bazterra, Martin Cuma, Marta B Ferraro, and Julio C Facelli. A general framework to understand parallel performance in heterogeneous clusters: analysis of a new adaptive parallel genetic algorithm. *Journal of Parallel and Distributed Computing*, 65(1):48–57, 2005.

[14] Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, Albert Zomaya, et al. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in computers*, 82(2):47–111, 2011.

[15] Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 15, 2008.

[16] Alessandro Bevilacqua. A dynamic load balancing method on a heterogeneous cluster of workstations. *INFORMATICA-LJUBLJANA-*, 23:49–56, 1999.

[17] Milind Bhandarkar, Laxmikant V Kalé, Eric de Sturler, and Jay Hoeflinger. Adaptive load balancing for MPI programs. In *Computational Science-ICCS 2001*, pages 108–117. Springer, 2001.

[18] Johannes Bisschop. *AIMMS optimization modelling*. Lulu.com, 2006.

[19] Christopher A Bohn and Gary B Lamont. Load balancing for heterogeneous clusters of PCs. *Future Generation Computer Systems*, 18(3):389–400, 2002.

[20] Pradip Bose, Margaret Martonosi, and David Brooks. Modeling and analyzing CPU power and performance: Metrics, methods, and abstractions. *Tutorial, ACM SIGMETRICS*, 2001.

[21] Jeffrey Burt. Intel, ARM take competition into HPC arena. http://www.eweek.com/servers/intel-arm-take-competition-into-hpc-arena.html, 2014. [Online; accessed 18-October-2014].

[22] Jeffrey S Chase, Darrell C Anderson, Prachi N Thakar, Amin M Vahdat, and Ronald P Doyle. Managing energy and server resources in hosting centers. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 103–116. ACM, 2001.

[23] Jian-Jia Chen and Chin-Fu Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms. In *null*, pages 28–38. IEEE, 2007.

[24] Quan Chen, Daqiang Zhang, Minyi Guo, Qianni Deng, and Song Guo. Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pages 2736–2743. IEEE, 2010.

[25] Yan Chen, Shunqing Zhang, Shugong Xu, and Geoffrey Ye Li. Fundamental trade-offs on green wireless networks. *IEEE Communications Magazine*, 49(6), 2011.

[26] Yiyu Chen, Amitayu Das, Wubi Qin, Anand Sivasubramaniam, Qian Wang, and Natarajan Gautam. Managing server energy and operational costs in hosting centers. In *ACM SIGMETRICS Performance Evaluation Review*, volume 33, pages 303–314. ACM, 2005.

[27] Kuan-Wei Cheng, Chao-Tung Yang, Chuan-Lin Lai, and Shun-Chyi Chang. A parallel loop self-scheduling on grid computing environments. In *Parallel Architectures, Algorithms and Networks, 2004. Proceedings. 7th International Symposium on*, pages 409–414. IEEE, 2004.

[28] Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, page 113. ACM, 2013.

[29] Eui-Young Chung, Luca Benini, Alessandro Bogliolo, Yung-Hsiang Lu, and Giovanni De Micheli. Dynamic power management for nonstationary service requests. *Computers, IEEE Transactions on*, 51(11):1345–1361, 2002.

[30] Dennis Colarelli and Dirk Grunwald. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–11. IEEE Computer Society Press, 2002.

[31] Gilberto Contreras and Margaret Martonosi. Power prediction for intel xscale® processors using performance monitoring unit events. In *Low Power Electronics and Design, 2005. ISLPED'05. Proceedings of the 2005 International Symposium on*, pages 221–226. IEEE, 2005.

[32] Compaq Computer Corporation and Revision B. Advanced configuration and power interface specification, 2000.

[33] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[34] Simon J Cox, James T Cox, Richard P Boardman, Steven J Johnston, Mark Scott, and Neil S OBrien. Iridis-pi: a low-cost, compact demonstration cluster. *Cluster Computing*, 17(2):349–358, 2014.

[35] E. W. Davis. Resource allocation in project network models – a survey. *Journal of Industrial Engineering*, 17, April 1966.

[36] E. W. Davis. Project scheduling under resource constraints – historical review and categorization of procedures. *AIIE Transactions*, 5, 1973.

[37] Gaurav Dhiman and Tajana Simunic Rosing. Dynamic power management using machine learning. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 747–754. ACM, 2006.

[38] Ahmad Faraj, Pitch Patarasuk, and Xin Yuan. A study of process arrival patterns for MPI collective operations. *International Journal of Parallel Programming*, 36(6):543–570, 2008.

[39] Michael Feldman. Less is More: Exploiting Single Precision Math in HPC. https://www.hpcwire.com/2006/06/16/less_is_more_exploiting_single_precision_math_in_hpc-1/, 2006. [Online; accessed 31-October-2017].

[40] Mark E Femal and Vincent W Freeh. Boosting data center performance through non-uniform power allocation. In *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 250–261. IEEE, 2005.

[41] Shane Fogerty, Siddhartha Bishnu, Yuliana Zamora, Laura Monroe, Steve Poole, Michael Lam, Joe Schoonover, and Robert Robey. Thoughtful precision in mini-apps. In *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*, pages 858–865. IEEE, 2017.

[42] Vincent W Freeh and David K Lowenthal. Using multiple energy gears in MPI programs on a power-scalable cluster. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 164–173. ACM, 2005.

[43] Vincent W Freeh, David K Lowenthal, Feng Pan, Nandini Kappiah, Rob Springer, Barry L Rountree, and Mark E Femal. Analyzing the energy-time trade-off in high-performance computing applications. *Parallel and Distributed Systems, IEEE Transactions on*, 18(6):835–848, 2007.

[44] Ashok Gautham, Kunal Korgaonkar, Patanjali Slpsk, Shankar Balachandran, and Kamakoti Veezhinathan. The implications of shared data synchronization techniques on multi-core energy efficiency. In *Proceedings of the 2012 USENIX conference on Power-Aware Computing and Systems*, pages 6–6. USENIX Association, 2012.

[45] R. Ge, X. Feng, and K. W. Cameron. Improvement of power-performance efficiency for high-end computing. In *Proceedings. 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 8, 2005.

[46] Rong Ge and Kirk W Cameron. Power-aware speedup. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[47] Rong Ge, Xizhou Feng, and Kirk W Cameron. Improvement of power-performance efficiency for high-end computing. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8–pp. IEEE, 2005.

[48] Rong Ge, Xizhou Feng, and Kirk W Cameron. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 34. IEEE Computer Society, 2005.

[49] Rong Ge, Xizhou Feng, Wu-chun Feng, and Kirk W Cameron. Cpu miser: A performance-directed, run-time system for power-aware clusters. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, pages 18–18. IEEE, 2007.

[50] Rong Ge, Xizhou Feng, Shuaiwen Song, Hung-Ching Chang, Dong Li, and Kirk W Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *Parallel and Distributed Systems, IEEE Transactions on*, 21(5):658–671, 2010.

[51] Mohammad Ghasemazar, Ehsan Pakbaznia, and Massoud Pedram. Minimizing the power consumption of a chip multiprocessor under an average throughput constraint. In *Quality Electronic Design (ISQED), 2010 11th International Symposium on*, pages 362–371. IEEE, 2010.

[52] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power CPU. In *Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25. ACM, 1995.

[53] Richard L Graham, Galen M Shipman, Brian W Barrett, Ralph H Castain, George Bosilca, and Andrew Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–9. IEEE, 2006.

[54] Flavius Gruian. Hard real-time scheduling for low-energy using stochastic data and DVS processors. In *Proceedings of the 2001 international symposium on Low power electronics and design*, pages 46–51. ACM, 2001.

[55] Flavius Gruian and Krzysztof Kuchcinski. Uncertainty-based scheduling: energy-efficient ordering for tasks with variable execution time [processor scheduling]. In *Low Power Electronics and Design, 2003. ISLPED'03. Proceedings of the 2003 International Symposium on*, pages 465–468. IEEE, 2003.

[56] Jie Han and Michael Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *Test Symposium (ETS), 2013 18th IEEE European*, pages 1–6. IEEE, 2013.

[57] Taliver Heath, Ana Paula Centeno, Pradeep George, Luiz Ramos, Yogesh Jaluria, and Ricardo Bianchini. Mercury and freon: temperature emulation and management for server systems. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 106–116. ACM, 2006.

[58] Taliver Heath, Bruno Diniz, Enrique V Carrera, Wagner Meira Jr, and Ricardo Bianchini. Energy conservation in heterogeneous server clusters. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 186–195. ACM, 2005.

[59] Steven Herbert and Diana Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, pages 38–43. IEEE, 2007.

[60] Tibor Horvath, Tarek Abdelzaher, Kevin Skadron, and Xue Liu. Dynamic voltage scaling in multitier web servers with end-to-end delay control. *Computers, IEEE Transactions on*, 56(4):444–458, 2007.

[61] Tibor Horvath and Kevin Skadron. Multi-mode energy management for multi-tier server clusters. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 270–279. ACM, 2008.

[62] Chung-hsing Hsu and Wu-chun Feng. A power-aware run-time system for high-performance computing. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 1. IEEE Computer Society, 2005.

[63] Chung-Hsing Hsu, Wu-chun Feng, and Jeremy S Archuleta. Towards efficient supercomputing: A quest for the right metric. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 8–pp. IEEE, 2005.

[64] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *ACM SIGPLAN Notices*, volume 38, pages 38–48. ACM, 2003.

[65] Nicholas Hunt, Paramjit Singh Sandhu, and Luis Ceze. Characterizing the performance and energy efficiency of lock-free data structures. In *Interaction between Compilers and Computer Architectures (INTERACT), 2011 15th Workshop on*, pages 63–70. IEEE, 2011.

[66] Chi-Hong Hwang and Allen C-H Wu. A predictive system shutdown method for energy saving of event-driven computation. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 5(2):226–241, 2000.

[67] Intel. Intel TurboBoost, 2015. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[68] Sandy Irani, R Gupta, and S Shukla. Competitive analysis of dynamic power management strategies for systems with multiple power savings states. In *Proceedings of the conference on Design, automation and test in Europe*, page 117. IEEE Computer Society, 2002.

[69] Sandy Irani and Kirk R Pruhs. Algorithmic problems in power management. *ACM Sigact News*, 36(2):63–76, 2005.

[70] Sandy Irani, Sandeep Shukla, and Rajesh Gupta. Algorithms for power savings. *ACM Transactions on Algorithms (TALG)*, 3(4):41, 2007.

[71] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th annual IEEE/ACM international symposium on microarchitecture*, pages 347–358. IEEE Computer Society, 2006.

[72] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*, pages 197–202. IEEE, 1998.

[73] Hwisung Jung and Massoud Pedram. Supervised learning based power management for multicore processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(9):1395–1408, 2010.

[74] Laxmikant V Kale and Gengbin Zheng. Charm++ and ampi: Adaptive runtime strategies via migratable objects. *Advanced Computational Infrastructures for Parallel and Distributed Applications*, pages 265–282, 2009.

[75] Nandini Kappiah, Vincent W Freeh, and David K Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 33. IEEE Computer Society, 2005.

[76] Anna R Karlin, Mark S Manasse, Larry Rudolph, and Daniel D Sleator. Competitive snoopy caching. *Algorithmica*, 3(1-4):79–119, 1988.

[77] Ian Karlin, Abhinav Bhatele, Bradford L Chamberlain, Jonathan Cohen, Zachary Devito, Maya Gokhale, Riyaz Haque, Rich Hornung, Jeff Keasler, Dan Laney, et al. Lulesh programming model and performance ports overview. *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep*, 2012.

[78] Woonseok Kim, Jihong Kim, and S Min. A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In *Proceedings of the conference on Design, automation and test in Europe*, page 788. IEEE Computer Society, 2002.

[79] Jonathan Koomey. Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times*, 2011.

[80] Vijay P. Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. *Journal of parallel and distributed computing*, 22(3):379–391, 1994.

[81] Michael O. Lam and Jeffrey K. Hollingsworth. Fine-Grained Floating-Point Precision Analysis. *International Journal of High Performance Computing Applications*, page 1094342016652462, jun 2016.

[82] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. Automatically Adapting Programs for Mixed-Precision Floating-Point Computation. In *Proceedings of the 27th International ACM Conference on Supercomputing (ICS '13)*, page 369, New York, New York, USA, jun 2013. ACM Press.

[83] Michael O. Lam, Jeffrey K. Hollingsworth, and G.W. Stewart. Dynamic floating-point cancellation detection. *Parallel Computing*, 39(3):146–155, mar 2013.

[84] Michael O Lam and Barry L Rountree. Floating-point shadow value analysis. In *Proceedings of the 5th Workshop on Extreme-Scale Programming Tools*, pages 18–25. IEEE Press, 2016.

[85] Michael Larabel. Linux still working on power-aware scheduling, 2015. http://www.phoronix.com/scan.php?page=news_item&px=MTQ4Mzg.

[86] Alexey Lastovetsky and Ravi Reddy. HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. *Journal of Parallel and Distributed Computing*, 66(2):197–220, 2006.

[87] Seongsoo Lee and Takayasu Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proceedings of the 37th Annual Design Automation Conference*, pages 806–809. ACM, 2000.

[88] Yann-Hang Lee, Krishna P Reddy, and C Mani Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 105–112. IEEE, 2003.

[89] Geoffrey Ye Li, Zhikun Xu, Cong Xiong, Chenyang Yang, Shunqing Zhang, Yan Chen, and Shugong Xu. Energy-efficient wireless communications: tutorial, survey, and open issues. *IEEE Wireless Communications*, 18(6), 2011.

[90] Min Yeol Lim, Vincent W Freeh, and David K Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs. In *SC 2006 conference, proceedings of the ACM/IEEE*, pages 14–14. IEEE, 2006.

[91] Cong Liu, Jie Han, and Fabrizio Lombardi. A low-power, high-performance approximate multiplier with configurable partial error recovery. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 95. European Design and Automation Association, 2014.

[92] Yongpeng Liu and Hong Zhu. A survey of the research on power management techniques for high-performance systems. *Software: Practice and Experience*, 40(11):943–964, 2010.

[93] Jacob R Lorch and Alan Jay Smith. Improving dynamic voltage scaling algorithms with pace. In *ACM SIGMETRICS Performance Evaluation Review*, volume 29, pages 50–61. ACM, 2001.

[94] Yung-Hsiang Lu and Giovanni De Micheli. Comparing system level power management policies. *Design & Test of Computers, IEEE*, 18(2):10–19, 2001.

[95] 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. In *Acm sigplan notices*, volume 40, pages 190–200. ACM, 2005.

[96] Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. Power-efficient multiple producer-consumer. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 669–678. IEEE, 2014.

[97] Ramy Medhat, Borzoo Bonakdarpour, Deepak Kumar, and Sebastian Fischmeister. Runtime monitoring of cyber-physical systems under timing and memory constraints. *ACM Transactions on Embedded Computing Systems (TECS)*, 14(4):79, 2015.

[98] Ramy Medhat, Shelby Funk, and Barry Rountree. Scalable performance bounding under multiple constrained renewable resources. *ACM Transactions on Embedded Computing Systems (TECS)*, 16, 2017.

[99] Ramy Medhat, Deepak Kumar, Borzoo Bonakdarpour, and Sebastian Fischmeister. Sacrificing a little space can significantly improve monitoring of time-sensitive cyber-physical systems. In *ICCPS'14: ACM/IEEE 5th International Conference on Cyber-Physical Systems (with CPS Week 2014)*, pages 115–126. IEEE Computer Society, 2014.

[100] Ramy Medhat, Michael Lam, Barry Rountree, Borzoo Bonakdarpour, and Sebastian Fischmeister. Managing the performance/error tradeoff of floating-point intensive applications. In *5th International Workshop on Energy Efficient Supercomputing (E2SC)*. ACM, 2017.

[101] David Meisner, Brian T Gold, and Thomas F Wenisch. PowerNap: eliminating server idle power. In *ACM Sigplan Notices*, volume 44, pages 205–216. ACM, 2009.

[102] Microsoft. Microsoft Cognitive Toolkit. https://www.microsoft.com/en-us/research/product/cognitive-toolkit/, 2017. [Online; accessed 03-April-2017].

[103] Asit K Mishra, Rajkishore Barik, and Somnath Paul. iact: A software-hardware framework for understanding the scope of approximate computing. In *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[104] Rich Mogull and LLC Securosis. Understanding and selecting a data loss prevention solution. *Technicalreport, SANS Institute*, 2007.

[105] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):12, 2008.

[106] J. D. Moore, J. S. Chase, P. Ranganathan, and R. K. Sharma. Making scheduling "cool": Temperature-aware workload placement in data centers. In *USENIX Annual Technical Conference, General Track*, pages 61–75, 2005.

[107] Justin D Moore, Jeffrey S Chase, Parthasarathy Ranganathan, and Ratnesh K Sharma. Making scheduling "cool": Temperature-aware workload placement in data centers. In *USENIX annual technical conference, General Track*, pages 61–75, 2005.

[108] Tali Moreshet, R Bahar, and Maurice Herlihy. Energy reduction in multiprocessor systems using transactional memory. In *Low Power Electronics and Design, 2005. ISLPED'05. Proceedings of the 2005 International Symposium on*, pages 331–334. IEEE, 2005.

[109] Ripal Nathuji and Karsten Schwan. VirtualPower: coordinated power management in virtualized enterprise systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 265–278. ACM, 2007.

[110] Rolf Neugebauer and Derek McAuley. Energy is just another resource: Energy accounting and energy pricing in the nemesis OS. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 67–72. IEEE, 2001.

[111] Ricardo Neves and Anibal C Matos. Raspberry Pi based stereo vision for small size ASVs. In *Oceans-San Diego, 2013*, pages 1–6. IEEE, 2013.

[112] Gethin Norman, David Parker, Marta Kwiatkowska, Sandeep Shukla, and Rajesh Gupta. Using probabilistic model checking for dynamic power management. *Formal aspects of computing*, 17(2):160–176, 2005.

[113] Gethin Norman, David Parker, Marta Kwiatkowska, Sandeep K Shukla, and Rajesh K Gupta. Formal analysis and validation of continuous-time markov chain based system level power management strategies. In *High-Level Design Validation and Test Workshop, 2002. Seventh IEEE International*, pages 45–50. IEEE, 2002.

[114] Klaithem Al Nuaimi, Nader Mohamed, Mariam Al Nuaimi, and Jameela Al-Jaroodi. A survey of load balancing in cloud computing: challenges and algorithms. In *Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on*, pages 137–142. IEEE, 2012.

[115] Edwin Olson. Apriltag: A robust and flexible visual fiducial system. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3400–3407. IEEE, 2011.

[116] Chris Olston, Boon Thau Loo, and Jennifer Widom. Adaptive precision setting for cached approximate values. *ACM SIGMOD Record*, 30(2):355–366, 2001.

[117] Chris Olston and Jennifer Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. Technical report, Stanford, 2000.

[118] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor. In *Proceedings of the Linux Symposium*, volume 2, pages 215–230. sn, 2006.

[119] Paul I Pénzes and Alain J Martin. Energy-delay efficiency of VLSI computations. In *Proceedings of the 12th ACM Great Lakes symposium on VLSI*, pages 104–111. ACM, 2002.

[120] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 international symposium on Low power electronics and design*, pages 76–81. ACM, 1998.

[121] Aaron M Pfalzgraf and Joseph A Driscoll. A low-cost computer cluster for high-performance computing education. In *Electro/Information Technology (EIT), 2014 IEEE International Conference on*, pages 362–366. IEEE, 2014.

[122] Bernd Pfrommer. PennCOSYVIO Data Set. https://daniilidis-group.github.io/penncosyvio/, 2017. [Online; accessed 03-April-2017].

[123] Padmanabhan Pillai and Kang G Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *ACM SIGOPS Operating Systems Review*, volume 35, pages 89–102. ACM, 2001.

[124] Eduardo Pinheiro and Ricardo Bianchini. Energy conservation techniques for disk array-based servers. pages 369–379, 2014.

[125] Eduardo Pinheiro, Ricardo Bianchini, Enrique V Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Workshop on compilers and operating systems for low power*, volume 180, pages 182–195. Barcelona, Spain, 2001.

[126] Eduardo Pinheiro, Ricardo Bianchini, and Cezary Dubnicki. Exploiting redundancy to conserve energy in storage systems. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 15–26. ACM, 2006.

[127] Jorda Polo, David Carrera, Yolanda Becerra, Vicenç Beltran, Jordi Torres, and Eduard Ayguadé. Performance management of accelerated mapreduce workloads in heterogeneous clusters. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 653–662. IEEE, 2010.

[128] Johan Pouwelse, Koen Langendoen, and Henk J Sips. Application-directed voltage scaling. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 11(5):812–826, 2003.

[129] Xiao Qin, Hong Jiang, Yifeng Zhu, and David R Swanson. Dynamic load balancing for I/O-intensive tasks on heterogeneous clusters. In *High Performance Computing-HiPC 2003*, pages 300–309. Springer, 2003.

[130] Qinru Qiu and Massoud Pedram. Dynamic power management based on continuous-time Markov decision processes. In *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 555–561. ACM, 1999.

[131] Gang Quan and Xiaobo Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Design Automation Conference, 2001. Proceedings*, pages 828–833. IEEE, 2001.

[132] Gang Quan, Linwei Niu, Xiaobo Sharon Hu, and Bren Mochocki. Fixed priority scheduling for reducing overall energy on variable voltage processors. In *Real-Time Systems Symposium, 2004. Proceedings. 25th IEEE International*, pages 309–318. IEEE, 2004.

[133] Quanser. QUARC Real-Time Control Software. http://www.quanser.com/Products/quarc, 2017. [Online; accessed 03-April-2017].

[134] Nikola Rajovic, Alejandro Rico, Nikola Puzovic, Chris Adeniyi-Jones, and Alex Ramirez. Tibidabo: Making the case for an arm-based HPC system. *Future Generation Computer Systems*, 36:322–334, 2014.

[135] Daler Rakhmatov, Sarma Vrudhula, and Deborah A Wallach. A model for battery lifetime analysis for organizing applications on a pocket computer. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 11(6):1019–1030, 2003.

[136] Dinesh Ramanathan, Sandy Irani, and Rajesh Gupta. Latency effects of system level power management algorithms. In *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 350–356. IEEE Press, 2000.

[137] Lavanya Ramapantulu, Bogdan Marius Tudor, Dumitrel Loghin, Trang Vu, and Yong Meng Teo. Modeling the energy efficiency of heterogeneous clusters. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 321–330. IEEE, 2014.

[138] P. Ranganathan, P. Leech, D. E. Irwin, and J. S. Chase. Ensemble-level power management for dense blade servers. In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*, pages 66–77, 2006.

[139] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey Chase. Ensemble-level power management for dense blade servers. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 66–77. IEEE Computer Society, 2006.

[140] Esteban Rodriguez, Gustavo Alkmim, Daniel M Batista, and Nelson LS da Fonseca. Trade-off between bandwidth and energy consumption minimization in virtual network mapping. In *Communications (LATINCOM), 2012 IEEE Latin-America Conference on*, pages 1–6. IEEE, 2012.

[141] Barry Rountree, David K Lowenthal, Shelby Funk, Vincent W Freeh, Bronis R De Supinski, and Martin Schulz. Bounding energy consumption in large-scale MPI

programs. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–9. IEEE, 2007.

[142] Barry Rountree, David K Lownenthal, Bronis R de Supinski, Martin Schulz, Vincent W Freeh, and Tyler Bletsch. Adagio: making DVS practical for complex HPC applications. In *Proceedings of the 23rd international conference on Supercomputing*, pages 460–469. ACM, 2009.

[143] Cindy Rubio-González, Cuong Nguyen, Benjamin Mehne, Koushik Sen, James Demmel, William Kahan, Costin Iancu, Wim Lavrijsen, David H Bailey, and David Hough. Floating-point precision tuning using blame analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1074–1085. ACM, 2016.

[144] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. Precimonious: Tuning assistant for floating-point precision. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 27. ACM, 2013.

[145] M. Sachenbacher, M. Leucker, A. Artmeier, and J. Haselmayr. Efficient energy-optimal routing for electric vehicles. In *Proceedings of the 25th Conference on Artificial Intelligence, (AAAI)*, 2011.

[146] Daniel Grobe Sachs, Wanghong Yuan, Christopher J. Hughes, Albert Harris, Sarita V. Adve, Douglas L. Jones, Robin H. Kravets, and Klara Nahrstedt. GRACE: A hierarchical adaptation framework for saving energy, 2004.

[147] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *ACM SIGPLAN Notices*, volume 46, pages 164–174. ACM, 2011.

[148] Jaewon Seo, Taewhan Kim, and Ki-Seok Chung. Profile-based optimal intra-task voltage scheduling for hard real-time applications. In *Proceedings of the 41st annual Design Automation Conference*, pages 87–92. ACM, 2004.

[149] Dongkun Shin, Jihong Kim, and Seongsoo Lee. Intra-task voltage scheduling for low-energy, hard real-time applications. *IEEE Design & Test of Computers*, (2):20–30, 2001.

[150] Sandeep K Shukla and Rajesh K Gupta. A model checking approach to evaluating system level dynamic power management policies for embedded systems. In *High-Level Design Validation and Test Workshop, 2001. Proceedings. Sixth IEEE International*, pages 53–57. IEEE, 2001.

[151] Tajana Simunic, Giovanni De Micheli, and Luca Benini. Event-driven power management of portable systems. In *Proceedings of the 12th international symposium on System synthesis*, page 18. IEEE Computer Society, 1999.

[152] Shuaiwen Song, Rong Ge, Xizhou Feng, and Kirk W Cameron. Energy profiling and analysis of the HPC challenge benchmarks. *International Journal of High Performance Computing Applications*, 2009.

[153] Shuaiwen Song, Chun-Yi Su, Rong Ge, Abhinav Vishnu, and Kirk W Cameron. Iso-energy-efficiency: An approach to power-constrained parallel computation. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 128–139. IEEE, 2011.

[154] Intel Open Source. Powertop. https://01.org/powertop, 2017. [Online; accessed 03-April-2017].

[155] Intel Open Source. RAPL. https://01.org/rapl-power-meter, 2017. [Online; accessed 03-April-2017].

[156] Robert Springer, David K Lowenthal, Barry Rountree, and Vincent W Freeh. Minimizing execution time in MPI programs on an energy-constrained, power-scalable cluster. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 230–238. ACM, 2006.

[157] Mani B Srivastava, Anantha P Chandrakasan, and Robert W Brodersen. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 4(1):42–55, 1996.

[158] Jan Stoess, Christian Lang, and Frank Bellosa. Energy management for hypervisor-based virtual machines. In *USENIX annual technical conference*, pages 1–14, 2007.

[159] János Sztrik. Basic queueing theory. *University of Debrecen, Faculty of Informatics*, 193, 2012.

[160] Ying Tan, Wei Liu, and Qinru Qiu. Adaptive power management using reinforcement learning. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 461–467. ACM, 2009.

[161] Gerald Tesauro, Rajarshi Das, Hoi Chan, Jeffrey Kephart, David Levine, Freeman Rawson, and Charles Lefurgy. Managing power consumption and performance of computing systems using reinforcement learning. In *Advances in Neural Information Processing Systems*, pages 1497–1504, 2007.

[162] Chao Tian, Haojie Zhou, Yongqiang He, and Li Zha. A dynamic mapreduce scheduler for heterogeneous workloads. In *Grid and Cooperative Computing, 2009. GCC'09. Eighth International Conference on*, pages 218–224. IEEE, 2009.

[163] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: a first step towards software power minimization. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 2(4):437–445, 1994.

[164] M Triki, Y Wang, AC Ammari, and M Pedram. Hierarchical power management of a system with autonomously power-managed components using reinforcement learning. *Integration, the VLSI Journal*, 48:10–20, 2015.

[165] Vibhore Vardhan, Wanghong Yuan, Albert F Harris, Sarita V Adve, Robin Kravets, Klara Nahrstedt, Daniel Sachs, and Douglas Jones. Grace-2: integrating fine-grained application adaptation with global adaptation for saving energy. *international Journal of embedded Systems*, 4(2):152–169, 2009.

[166] Akshat Verma, Puneet Ahuja, and Anindya Neogi. pMapper: power and migration cost aware application placement in virtualized systems. In *Middleware 2008*, pages 243–264. Springer, 2008.

[167] Akshat Verma, Puneet Ahuja, and Anindya Neogi. Power-aware dynamic placement of HPC applications. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 175–184. ACM, 2008.

[168] Shinan Wang, Hui Chen, and Weisong Shi. SPAN: A software power analyzer for multicore computer systems. *Sustainable Computing: Informatics and Systems*, 1(1):23–34, 2011.

[169] Xiaorui Wang and Ming Chen. Cluster-level feedback power control for performance optimization. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 101–110. IEEE, 2008.

[170] Yanzhi Wang, Qing Xie, Ahmed Ammari, and Massoud Pedram. Deriving a near-optimal power management policy using model-free reinforcement learning and bayesian classification. In *Proceedings of the 48th Design Automation Conference*, pages 41–46. ACM, 2011.

[171] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Mobile Computing*, pages 449–471. Springer, 1996.

[172] Tom White. *Hadoop: The definitive guide*. "O'Reilly Media, Inc.", 2012.

[173] Tiffani L Williams and Rebecca J Parsons. The heterogeneous bulk synchronous parallel model. In *Parallel and Distributed Processing*, pages 102–108. Springer, 2000.

[174] Qiang Wu, Philo Juang, Margaret Martonosi, Li-Shiuan Peh, and Douglas W Clark. Formal control techniques for power-performance management. *IEEE Micro*, (5):52–62, 2005.

[175] Li Xiao, Songqing Chen, and Xiaodong Zhang. Dynamic cluster resource allocations for jobs with known and unknown memory demands. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):223–240, 2002.

[176] Li Xiao, Songqing Chen, and Xiaodong Zhang. Adaptive memory allocations in clusters to handle unexpectedly large data-intensive jobs. *Parallel and Distributed Systems, IEEE Transactions on*, 15(7):577–592, 2004.

[177] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. In *ACM SIGPLAN Notices*, volume 38, pages 49–62. ACM, 2003.

[178] Fen Xie, Margaret Martonosi, and Sharad Malik. Intraprogram dynamic voltage scaling: Bounding opportunities with analytic modeling. *ACM Transactions on Architecture and Code Optimization (TACO)*, 1(3):323–367, 2004.

[179] Fen Xie, Margaret Martonosi, and Sharad Malik. Bounds on power savings using runtime dynamic voltage scaling: an exact algorithm and a linear-time heuristic approximation. In *Proceedings of the 2005 international symposium on Low power electronics and design*, pages 287–292. ACM, 2005.

[180] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, James Majors, Adam Manzanares, and Xiao Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–9. IEEE, 2010.

[181] Ruibin Xu, Daniel Mossé, and Rami Melhem. Minimizing expected energy in real-time embedded systems. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 251–254. ACM, 2005.

[182] B. Yang, J. Geunes, and W. J. Obrien. Resource-constrained project scheduling: Past work and new directions. Technical report, University of Florida, 2001.

[183] Chao-Tung Yang, Kuan-Wei Cheng, and Kuan-Ching Li. An enhanced parallel loop self-scheduling scheme for cluster environments. *The Journal of Supercomputing*, 34(3):315–335, 2005.

[184] Chao-Tung Yang, Kuan-Wei Cheng, and Wen-Chung Shih. On development of an efficient parallel loop self-scheduling for grid computing environments. *Parallel Computing*, 33(7):467–487, 2007.

[185] Yong Yang, Xiaodong Zhang, and Yongsheng Song. An effective and practical performance prediction model for parallel computing on nondedicated heterogeneous NOW. *Journal of Parallel and Distributed Computing*, 38(1):63–80, 1996.

[186] Frances Yao, Alan Demers, and Scott Shenker. A scheduling model for reduced CPU energy. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 374–382. IEEE, 1995.

[187] Rong Ye and Qiang Xu. Learning-based power management for multicore processors via idle period manipulation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 33(7):1043–1055, 2014.

[188] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[189] Siyu Yue, Di Zhu, Yanzhi Wang, and Massoud Pedram. Reinforcement learning based dynamic power management with a hybrid power supply. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 81–86. IEEE, 2012.

[190] Matei Zaharia, Andy Konwinski, Anthony D Joseph, Randy H Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *OSDI*, volume 8, page 7, 2008.

[191] Heng Zeng, Carla S Ellis, and Alvin R Lebeck. Experiences in managing energy with ecosystem. *Pervasive Computing, IEEE*, 4(1):62–68, 2005.

[192] Heng Zeng, Carla S Ellis, Alvin R Lebeck, and Amin Vahdat. ECOSystem: Managing energy as a first class operating system resource. *ACM SIGPLAN Notices*, 37(10):123–132, 2002.

[193] Yan Zhang, Zhijian Lu, John Lach, Kevin Skadron, and Mircea R Stan. Optimal procrastinating voltage scheduling for hard real-time systems. In *Proceedings of the 42nd annual Design Automation Conference*, pages 905–908. ACM, 2005.

[194] Xiliang Zhong and Cheng-Zhong Xu. System-wide energy minimization for real-time tasks: Lower bound and approximation. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):28, 2008.

[195] Jianli Zhuo and Chaitali Chakrabarti. System-level energy-efficient dynamic task scheduling. In *Design Automation Conference, 2005. Proceedings. 42nd*, pages 628–631. IEEE, 2005.

[196] Jianli Zhuo and Chaitali Chakrabarti. Energy-efficient dynamic task scheduling algorithms for DVS systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(2):17, 2008.

[197] Ziliang Zong, Xiao Qin, Xiaojun Ruan, Kiranmai Bellam, Mais Nijim, and Mohamed Alghamdi. Energy-efficient scheduling for parallel applications running on heterogeneous clusters. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, pages 19–19. IEEE, 2007.