THEORY AND PRACTICE OF DYNAMIC VOLTAGE/FREQUENCY SCALING IN THE HIGH PERFORMANCE COMPUTING ENVIRONMENT

by

Barry Louis Rountree

Creative Commons 3.0 Atrribution-Share Alike License

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2010

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Barry Louis Rountree
entitled Theory and Practice of Dynamic Voltage/Frequency Scaling in the High Performance Computing Environment
and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Date: 15 Nov 2009
David K. Lowenthal

Date: 15 Nov 2009
Bronis R. de Supinski

Date: 15 Nov 2009
Shelby Funk

Date: 15 Nov 2009
John Hartman

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Date: 15 Nov 2009
Dissertation Director: David K. Lowenthal

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. This work is licensed under the Creative Commons Attribution-No Derivative Works 3.0 United States License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/3.0/us/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

SIGNED: Barry Louis Rountree

ACKNOWLEDGEMENTS

I would like to thank the members of my committee. I'm grateful to John Hartman and Chris Gniady for helping out a new transfer student. Shelby Funk made the original suggestion to use linear programming and provided many hours of support and encouragement while I mastered that technique. Nearly all of the research presented in this dissertation can be traced back to consequences of that suggestion.

Bronis de Supiski gave me the opportunity to intern at Lawrence Livermore National Library, then extended the intership, sat in on dozens of conference calls, edited nearly every draft that was sent out, and provided a necessary outsider's perspective.

I can't recall the exact details of meeting David Lowenthal for the first time, but I think I can reconstruct the essense. He probably said something to the effect of "Hi, I'm Dave," and I almost certainly replied "No, you're wrong."

I'd first like to thank Dave for putting up with me. I'd also like to thank him for the amount of freedom he gave me when it came to the scope of my research. I'm told that most doctoral students are given a tractable problem to solve and a few possible paths to a solution. Dave trusted me enough to give me a domain to work in and allowed me to pick problems that he didn't know how to solve, and continued to do so even after a few initial false starts. Doing this allowed me to start thinking early on about what made problems interesting and what made solutions tractable. These are not trivial lessons and I can't say that I've mastered them, but the fault there lies in the student, not the teacher.

Dave also provided multiple opportunities to review manuscripts, and his careful attention to my comments has both taught me how to read the technical literature as well as how to be a thoughtful reviewer. He has read and reread every line of every draft I've submitted for publication, sat through every practice talk, and consistently pushed me to set my own expectations higher. The process would certainly have been easier if he had cared less, but then I wouldn't be nearly as well-prepared as I am.

In short, David Lowenthal didn't do much except teach me how to read, write and think.

Thanks, Dave.

Finally, I would like to thank Joyce Rountree for looking after the finances, Ange Kahn for looking after the cats, and Robin Snyder for looking after me.

TABLE OF CONTENTS

LIST OF FIGURES 8

LIST OF TABLES 9

ABSTRACT 10

CHAPTER 1 INTRODUCTION 11
- 1.1 Motivation 11
- 1.2 Contribution 13
- 1.3 Outline of the Dissertation 14

CHAPTER 2 DYNAMIC VOLTAGE/FREQUENCY SCALING OVERVIEW . . 15
- 2.1 Mathematical Model and Experimental Observations 15
- 2.2 DVFS in Practice 17
- 2.3 Other Approaches to Using DVFS in HPC 17
- 2.4 Other Domains 19

CHAPTER 3 BOUNDING THE POTENTIAL ENERGY SAVINGS 20
- 3.1 Execution Model 21
- 3.2 Linear Programming Formulation 25
- 3.3 Implementation 26
  - 3.3.1 Trace collection 27
  - 3.3.2 Solution via linear programming 27
  - 3.3.3 Task binding 28
  - 3.3.4 Validation 28
  - 3.3.5 Limitations 28
- 3.4 Experimental Results 29
  - 3.4.1 Experimental Methodology 29
  - 3.4.2 Applications 30
- 3.5 Related Work 35
- 3.6 Summary and Future Work 37

CHAPTER 4 THE *ADAGIO* RUNTIME SYSTEM 39
- 4.1 Overview 41
  - 4.1.1 Definitions and Basic Assumptions 41
  - 4.1.2 Taxonomy 42

TABLE OF CONTENTS – *Continued*

4.2 Adagio . . . 45
  4.2.1 Adagio Implementation . . . 48
  4.2.2 Optimizations . . . 50
  4.2.3 Large Message Handling . . . 52
4.3 Results . . . 52
  4.3.1 Algorithms . . . 54
  4.3.2 UMT2K . . . 55
  4.3.3 ParaDiS . . . 57
  4.3.4 Summary of UMT2K and ParaDiS results . . . 59
  4.3.5 NAS Parallel Benchmarks . . . 60
4.4 Related Work . . . 63
  4.4.1 Scheduled Communication . . . 64
  4.4.2 Scheduled Iteration . . . 64
  4.4.3 Scheduled Timeslice . . . 64
  4.4.4 Other Related Work . . . 65
4.5 Summary and Future Work . . . 66

CHAPTER 5 ARCHITECTURAL MODEL . . . 67
5.1 Introduction . . . 67
5.2 Overview . . . 69
5.3 Evaluation of Existing Models . . . 71
  5.3.1 Existing Models and Implementations. . . . 71
  5.3.2 Discussion . . . 73
  5.3.3 Limitations of current models . . . 74
5.4 Architectural Model . . . 76
  5.4.1 Assumptions, definitions and simplifications . . . 76
  5.4.2 In-Order Execution Model . . . 79
  5.4.3 Out-of-Order, Single Read Execution Model . . . 79
  5.4.4 Multiple Read, Out-of-order Execution Model . . . 81
  5.4.5 The *Leading Load* Technique . . . 83
  5.4.6 Implementation Issues. . . . 84
5.5 Evaluation of the *Leading Loads* Technique . . . 84
  5.5.1 Experimental Setup . . . 85
  5.5.2 Discussion . . . 86
5.6 Related Work . . . 88
5.7 Conclusions and Future Work . . . 90

TABLE OF CONTENTS – *Continued*

CHAPTER 6 SUMMARY AND FUTURE WORK 92
6.1 Summary 92
6.2 Future work 93
6.2.1 Performance Prediction 93
6.2.2 User and Compiler Guidance 94
6.2.3 DVFS, Overclocking and Multicore 95
6.2.4 Adoption of Policies 95

REFERENCES 97

LIST OF FIGURES

2.1 Frequency vs. Power for two systems. 16

3.1 Sample program and resulting task graph. 21
3.2 Jacobi with various algorithms. 31
3.3 Particle with various algorithms. 31
3.4 *UMT2K* with various algorithms. 32
3.5 *LPS* schedule used by each node for *UMT2K*, MHz on y-axis. 34
3.6 *Comm* schedule on node 0 for *UMT2K*, MHz on y-axis. 35

4.1 Typical SPMD Execution (left) and its task graph (right). 41
4.2 Adagio algorithm with no optimizations. 47
4.3 Normalized time, energy, and power for *UMT2K*. 55
4.4 Normalized time, energy, and power for *ParaDiS*. 58
4.5 Normalized time, energy, and power for load-balanced *ParaDiS*. 58

5.1 Regression over “outstanding read” cycles per cycle (RPC). 75
5.2 Interval Models: Three architectures and CPU frequencies. 80
5.3 Simulated performance of RMI (gray) and Leading Loads (black). 87

LIST OF TABLES

3.1 Key variables used in execution model. 23
3.2 Measured power per frequency for *UMT2K*. 27
3.3 Scheduled seconds per frequency. 33

4.1 Comparison of near-optimal offline scheduling to runtime classes. 43
4.2 Variables used within *Adagio* and their purpose. 46
4.3 Normalized Time and Energy for the NAS Parallel Benchmarks. 61

5.1 Evaluation of Existing Models on the Core2 Architecture 73
5.2 Evaluation of *leading loads* using PTLSim 86

ABSTRACT

This dissertation provides a comprehensive overview of the theory and practice of *Dynamic Voltage/Frequency Scaling* (DVFS) in the *High Performance Computing* (HPC) environment. We summarize the overall problem as follows: how can the same level of computational performance be achieved using less electrical power? Equivalently, how can computational performance be increased using the same amount of electrical power? In this dissertation we present performance and architecture models of DVFS as well as the *Adagio* runtime system. The performance model recasts the question as an optimization problem that we solve using linear programming, thus establishing a bound on potential energy savings. The architectural model provides a low-level explanation of how memory bus and CPU clock frequencies interact to determine execution time. Using insights provided from these models, we have designed and implemented the *Adagio* runtime system. This system realizes near-optimal energy savings on real-world scientific applications without the use of training runs or source code modification, and under the constraint that only negligible delay will be tolerated by the user. This work has opened up several new avenues of research, and we conclude by enumerating these.

CHAPTER 1

INTRODUCTION

1.1 Motivation

In 2002, Japan's *Earth Simulator* supercomputer (Antony et al., 2006) went online and became the fastest supercomputer in the top500.org rankings (Dongarra et al., 2009). This architecture depended on 5120 specialized processors and required 6.4MW of system power. While the raw performance was impressive, this architecture generated only 5.6 FLOPS per watt (FpW) (Feng and Cameron, 2010) and ultimately proved to be a dead end in HPC architecture.

In 2004, the Earth Simulator was replaced at the top of the rankings by Lawrence Livermore's *Blue Gene/L* (Gara et al., 2005) supercomputer. In its current form, 212,992 processor cores require only 2.3MW of system power, produce 205.27 FpW, and are roughly 13 times more powerful than the Earth Simulator in terms of total system performance.

The reason for this dramatic change in architecture and increase in performance can be traced in part to the physics of CPUs. The performance of CPU-bound computation is dominated by the clock frequency, and a linear increase in clock frequency, all other things being equal, requires a quadratic increase in power. While the Earth Simulator was considered to be a massively parallel architecture, the bulk of its processing power arose from a relatively small number of relatively fast processors. The quadratic relationship between power and processor performance ensured diminishing returns for a strategy of increasing individual CPU performance.

The Blue Gene architecture took the opposite approach and *reduced* individual CPU performance, spending the power budget on hundreds of thousands of low-power processors. The current number two machine in the Top500 list (the *Roadrunner* system at Los Alamos) is even more efficient than Blue Gene/L, with a FLOPS per Watt rating of 458.33 and 2.5MW total system power. Yet the top system, the *Jaguar* supercom-

puter (Alam et al., 2007) at Oak Ridge, is only slightly slower than Roadrunner, much less efficient (152.36 FpW), and uses far more total system power (7.0MW). While this is still a substantial improvement over the efficiency of the *Earth Simulator*, if low-voltage CPUs have proven to be so effective, why are machines such as Jaguar still being built?

Jaguar and Blue Gene/L are not so much two distinct classes of systems as they are points on a continuum. Consider two clusters with identical components, except the processors on the first cluster have twice the CPU clock frequency of the second, and the second has twice as many processors. For some instances of memory-bound programs the decrease in CPU clock frequency will have a relatively small impact on performance, and the large-node-count approach will provide higher performance. Working against this gain in efficiency is the increased time spent communicating between processors for non-point-to-point communication and the increased number of communication calls needed due to each processor having a smaller amount of RAM. If load imbalance is introduced, the balance may decisively shift in favor of the machine with fewer, faster nodes, as performance will be limited by the capabilities of a subset of nodes.

DVFS allows a cluster to occupy a segment of this continuum rather than a single point. At a coarse level of control, a larger number of slower nodes may be used for programs that can take advantage of this configuration, while a smaller number of faster nodes can be used when individual CPU speed dominates execution time. This approach may be considered *static* voltage/frequency scaling. Our work focuses on how to achieve the best of both worlds *dynamically*: nodes that are determined to be on the critical path of execution use the fastest available CPU clock frequency, thus optimizing execution time. Nodes that are off the critical path execute using slower CPU clock frequencies, thus optimizing energy savings. The overall savings in energy reduce the overall cost of the system, which in turn can lead to either cheaper systems for a given level of performance or faster systems for a given level of cost.

The savings is not due only to reducing the monetary cost of electricity used at the processor. Follow-on effects are also significant. Watts consumed by the CPU are converted to waste heat, and the removal of this heat from the system requires still more watts to be expended by the cooling system. The increased heat decreases component density

and increases the footprint of the entire system. Increased heat also leads to shortened component lifetime and thus a decrease in mean time between failure. Making processors more energy-efficient leads not only to more cores, but more reliable cores using less cooling while packed into a smaller space

1.2 Contribution

The contributions of this dissertation are novel performance and architectural models of DVFS and their synthesis in an implemented runtime system.

Significant previous research exists that investigates how to predict the effects of DVFS on execution time (Ge et al., 2005; Lee et al., 2007; Snowdon et al., 2005; Freeh et al., 2005). The standard approach relies on finding a correlation between a set of hardware performance monitors (HPMs) and change in performance, then using a linear regression over this set to predict future performance. While this approach can lead to low median error, the error is sensitive to how well benchmarks used for training mimic the behavior of the programs to be predicted. This approach also does not provide any guidance as to why outliers exist or under what conditions the regression would be expected to perform poorly.

To address these issues, we create an architectural model of the interface between the CPU and memory. We use this model to explain why HPMs used in the best existing models still have measurable error. We derive a technique based on our model that will allow performance across DVFS to be predicted directly without use of regression and show a significant improvement in prediction based on results obtained from a cycle-accurate processor simulation.

Prior to this work, a consensus existed that DVFS could be used to save energy in the HPC environment, but no models existed that bounded potential savings. Different techniques could be compared against each other, but there was little sense of how close these solutions came to an ideal solution. By casting the problems first in terms of graph theory and then as an optimization problem, our performance model places a tight bound on the amount of potential energy savings.

We have distilled this theoretical work into *Adagio*, a runtime system that uses DVFS to save power for parallel scientific applications. At runtime, *Adagio* determines which compute cores are off the critical path and when, and slows these cores to the point where they join the critical path. Computation on the critical path is not slowed, and slowdown caused by the system is due only to the overhead of executing *Adagio*. The performance and prediction models used in *Adagio* are direct descendants of earlier versions of the above theoretical work.

1.3 Outline of the Dissertation

Chapter 2 provides an overview of how DVFS works and how it has been applied in HPC and other domains. Chapter 3 covers the performance model used to solve the bounding problem, Chapter 4 explains the *Adagio* runtime system. Chapter 5 details the work on performance prediction Conclusions and future work comprise chapter 6.

CHAPTER 2

DYNAMIC VOLTAGE/FREQUENCY SCALING OVERVIEW

This chapter provides a mathematical description of how DVFS works and places the technique in a larger context of saving power in many varied computational domains.

2.1 Mathematical Model and Experimental Observations

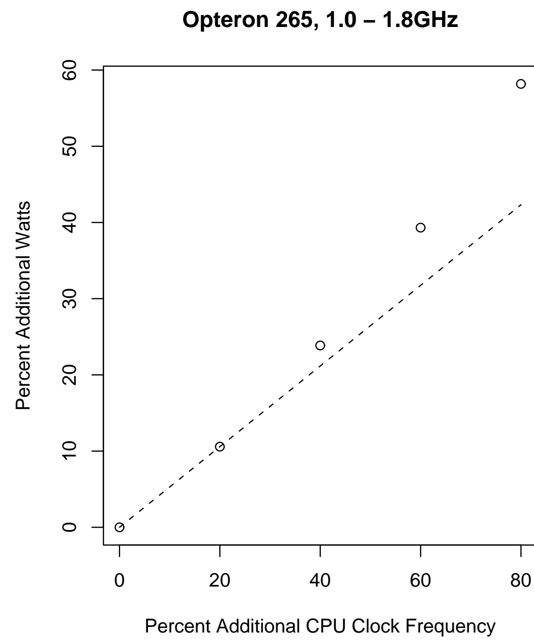
We describe the relationship between CPU clock frequency, power and energy using the equations provided in the Intel optimization documentation (Intel, 2007). We let V_{dd} represent the supply voltage and f represent the CPU clock frequency:

$$Power \propto fV_{dd}^2$$

$$Delay = \frac{1}{f} \propto \frac{1}{V_{dd}}$$

$$Energy \propto V_{dd}^2$$

A linear reduction in delay (by increasing the frequency) will cause a quadratic increase in both power and energy. The utility of DVFS lies in the converse of this statement: a linear *increase* in delay will cause a quadratic *decrease* in the amount of power and energy used. We can express the practical effect much more simply: it takes increasingly more electricity to increase CPU clock frequency. This can be seen in Figure 2.1. This measures total system energy of two nodes, the first based on two dual-core Opteron 265 processors, the other based on four quad-core Intel 5560 processors. All cores on the node were executing spin loops, and power measurements were taken over the range of available frequencies. The dashed line indicates a linear extrapolation from the two lowest frequencies. In the second graph, the meter used was accurate only to ± 2 watts; even so, the trend manifests itself given a large enough range of frequencies.



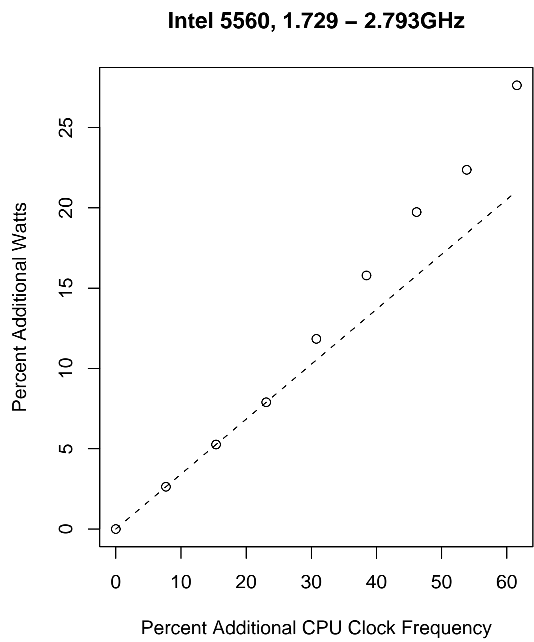


Figure 2.1: Frequency vs. Power for two systems.

2.2 DVFS in Practice

DVFS only affects dynamic power required for useful work. It does not affect the amount of static power necessary to keep the processor in a powered-on state. Other components may have other strategies for energy reduction, in some cases by the component being eliminated altogether (e.g., disk drives are shared by several processors). The focus of our work is reducing total system energy, of which dynamic processor energy remains a significant portion.

For any given unit of computation, an *ideal frequency* exists such that execution at the ideal frequency will complete exactly at the time allotted for computation. We make two simplifying assumptions: the time alloted is at least sufficient for execution to complete at the highest frequency without exceeding the deadline, and that the time alloted is not so long that it is more efficient to run as fast as possible and then turn off the machine until it is needed again. Given these assumptions, the ideal frequency is the most efficient frequency to use. Because only a handful of frequencies are usually available, and it's unlikely that the ideal frequency will be one of these. To work around this, we can approximate the ideal frequency by splitting the execution over two neighboring frequencies that bracket the ideal frequency (Ishihara and Yasuura, 1998).

Finally, as computation becomes more memory-bound, lowering the CPU clock frequency has less of an effect on performance. The bottleneck for performance in this case has become the memory bus, and changing the CPU clock does not have a significant impact on memory latency. We can take advantage of this by allowing more aggressive scheduling of slower frequencies and greater energy savings.

2.3 Other Approaches to Using DVFS in HPC

Most early work in this area focused on the quadratic potential for power savings and made the argument that a significant performance penalty would be acceptable given sufficient savings. This approach led to a debate on what metric should be used to capture both delay and savings, with the consensus settling on some variation of the *energy delay product* (Hsu et al., 2005).

These metrics have not met with much acceptance in the wider HPC community. Supercomputers have a relatively short working lifetime and they are budgeted with the assumption that enough work exists to fill that lifetime. While saving energy is a laudable goal, if significant delay accrues during the course of saving energy, ultimately less work will be done, the machine will be underutilized, and money will have been wasted.

Subsequent research has taken this into account and either assumes only negligible delay will be tolerated (Rountree et al., 2007) or allows the user to specify what delay will be considered acceptable (Ge et al., 2007). We detail four broad approaches below.

Scheduled Communication MPI communication calls are rarely CPU-bound. If these calls take significantly longer than the time required to change the CPU clock frequency, energy can be saved with little to no delay by changing to a slower frequency for the duration of the communication (Rountree et al., 2007)

Scheduled Iteration Scientific kernels are often highly iterative. After identifying these iteration boundaries, the CPU is slowed on cores where there is a significant percentage of *slack*, or idle time. By observing per-core iteration execution time the algorithm can avoid slowing cores that execute computation on the critical path (Freeh et al., 2008a).

Scheduled Timeslice By observing several timeslices in the immediate past, the algorithm predicts the characteristics of the upcoming timeslice and chooses the appropriate CPU clock frequency (Ge et al., 2007).

Offline Scheduling Using one or more training runs, a DVFS schedule is calculated offline and used for subsequent execution (Rountree et al., 2007). This can be particularly useful in embedded and real-time computing. Here, instead of program execution being a unique event, programs are expected to loop continuously and be replicated across every instance of a particular device (e.g., temperature sensors). The cost of scheduling can be amortized over all execution on all devices, justifying even modest energy savings.

2.4 Other Domains

DVFS first gained popularity in battery-powered devices where maximum computing capacity is only needed intermittently, and a lower CPU clock frequency can be tolerated for the remainder. For these kinds of consumer devices, response time and quality-of-service are more important metrics than time to completion of a particular task, and designers expect that the device essentially will be idle most of the time. Real-time computing is a domain that intersects this area and often has similar characteristics: raw speed is not the metric of interest, there is usually a large portion of idle time, and thus there is potential energy savings to be had by slowing down the CPU.

In the HPC domain, these assumptions no longer hold. A supercomputer is rarely idle and the primary metric of success is time to completion. To take advantage of DVFS under these constraints, we exploit *load imbalance* where certain processors have less work than others and thus accrue idle time. DVFS can be used in this case to schedule execution at a lower CPU clock frequency without affecting overall execution time.

Saving energy has also become a concern for datacenters, which are installations that handle large databases and webservers. Workloads are distinct from both supercomputing and consumer domains: utilization is both highly variable and highly cyclical, individual jobs tend to be very short and to take few resources, and overprovisioning is a common strategy for handling peak loads. Performance is usually measured by a combination of execution time and responsiveness. Because jobs tend to be independent of other jobs and can be migrated easily from one server to another, it is possible to craft scheduling algorithms that have a very fine granularity of control. This combination has focused power research on how to power only those portions of the cluster necessary for acceptable quality of service as well as how to predict when additional resources will be needed (Elnozahy et al., 2002; Sharma et al., 2003; Femal and Freeh, 2005).

CHAPTER 3

BOUNDING THE POTENTIAL ENERGY SAVINGS

In this chapter we show how to bound potential DVFS-based energy savings in parallel message-passing programs. We transform the program into a graph where nodes represent work and edges represent dependences. We then transform the graph into a linear programming (LP) problem. The LP solution is the ideal CPU clock frequency schedule for the program, and measuring the energy used by this schedule provides the near-optimal bound on energy.

Earlier approaches resulted in systems that produced varying amounts of energy savings and execution delay. However, all of these systems suffer from a lack of knowledge of (1) the maximum energy savings possible for a given time delay limit; and (2) a schedule of frequencies that achieves that maximum. A system that determines a bound on energy savings and the associated schedule for any MPI application supports assessment of energy-reducing heuristics. Further, the schedule provides insight into *how* to achieve the energy savings. We present an LP system that finds a schedule that tightly bounds the optimal solution for a given application and an allowable time delay. Our LP system exploits *slack*—the difference between a processor's deadline and when it finishes its work—by switching a processor with slack to a lower frequency. Our system is sophisticated enough to handle the non-uniform mapping between CPU frequency and slowdown.

This chapter makes three *contributions*. First, we develop a computationally tractable method to bound energy savings, for any specified time delay, *without* programmer involvement. Second, because we are looking at an entire program run or iteration, we optimize slack reclamation across all processors, taking into account both communication slack and memory pressure. Third, our energy bounds provide a tool with which to evaluate practical run-time algorithms for HPC programs.

We apply our system to three programs: Jacobi iteration, particle simulation, and *UMT2K* (Lawrence Livermore National Laboratory, 2005). All programs have at least

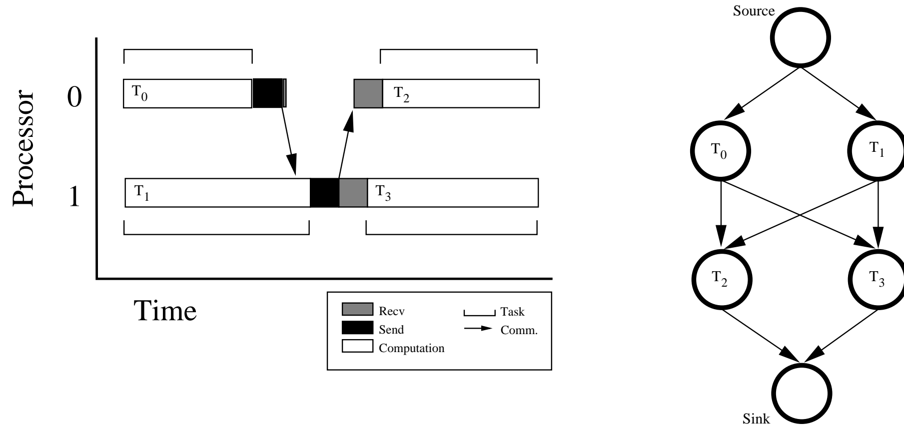


Figure 3.1: Sample program and resulting task graph.

ten thousand MPI communication events. Given zero allowable delay, a likely goal for HPC applications, our LP-based bounding technique shows that while the particle code can save up to 15% energy, *UMT2K* can save only 3.3%.

We emphasize that the significance of this work does not lie in these individual values: a 3.3% savings for an 8K-node cluster is far more interesting than the same savings on a 8-node cluster. Rather, our contribution is the ability to determine the percentages. If a run-time algorithm does not save as much energy as expected for a particular application, we can now distinguish between a suboptimal algorithm and an application with no realizable energy savings.

The rest of this chapter is organized as follows. We discuss our execution model in Section 3.1 and the translation of our execution model to a linear programming formulation in Section 3.2. Next, Section 3.3 presents the implementation of our LP system. Section 3.4 discusses the measured results on a real power-scalable cluster. Finally, Section 3.5 describes related work, and Section 3.6 summarizes and describes future work.

3.1 Execution Model

Our execution model is a distributed-memory multicomputer on which each processor (node) executes a unique, predetermined set of tasks. We define a *task* as a region of code between two communication points. Each task runs to completion. The tasks are

totally ordered on each processor, and MPI communication events can create intertask dependencies across processors. Any remote communication between tasks (e.g., *send* or *receive*) occurs either before or after the task. Figure 3.1 (left) pictorially shows an example program in our framework. The task boundaries, which occur at the *send* and *receive* commands, are shown. The pictorial version of the program on the left side of Figure 3.1 translates directly to the program *task graph* on its right side. This graph represents tasks by vertices and (communication) dependencies by edges. It has a single source vertex and a single sink vertex.

Some communication operations can be more difficult to model, depending on the implementation of the communication library. For example, in most MPI implementations, some *send* operations can block; with large messages, `MPI_Send` as well as `MPI_Isend` (at the matching `Wait`) block until the destination node has arrived at the corresponding `MPI_Recv` or `MPI_Irecv`. This requires an extra dependence (a two-way instead of a one-way model). MPI collective communications are reimplemented using primitives. For example, MPI_Barrier() has been transformed by our runtime library into the requisite number of MPI_Send() and MPI_Recv() calls.

Given a communication graph, the goal is to find both the total time and total energy. Table 3.1 summarizes the key variables in the execution model. We denote the start time of task i as S_i and the execution time of task i as T_i. All tasks that immediately precede task i in the task graph are its predecessors, denoted $Pred_i$. For task j in $Pred_i$, M_i^j is the time for the message to travel from j to i. We assume M_i^j is zero if i and j reside on the same processor. A processor may commence execution of task i when the predecessors of i in the task graph have completed and it has received its input data from all (remote) predecessors. Thus,

$$S_i = \max_{j \in Pred_i} (S_j + T_j + M_i^j).$$

The source vertex has a start time of zero. The program completes when the sink vertex (which is instantaneous) has executed. Thus, total program execution time, ω, is S_{Sink}. Using this information, we can determine ω by finding the longest path through the task graph of the program.

Variable	Meaning
P	Number of processors
$\mathcal{T}$	Set of all tasks
S_i	Task i start time
T_i	Task i total scheduled execution time
C_i^f	Task i execution time if run at frequency f
M_i^j	Latency time for message from task j to task i
$Pred_i$	Predecessors of task i
$\mathcal{F}$	Set of all frequencies
δ_i^f	For a task i, fraction of task completed in frequency f
W_i^f	For a task i, power it consumes in frequency f
W_I	Power consumed when idle
I	Total idle time (over entire task graph)
ω	Program execution time

Table 3.1: Key variables used in execution model.

To minimize energy, we must determine two additional factors. First, we must determine the execution time of each task if it were to run only using a single available frequency. For $f \in \mathcal{F}$, we denote C_i^f as the execution time for task i if it were to run only in frequency f (we discuss how we determine C_i^f in Section 3.3). The fraction of task i completed at frequency f is δ_i^f. We estimate execution time as:[1]

$$T_i = \sum_{f \in \mathcal{F}} (C_i^f \delta_i^f).$$

Second, we find the total energy E. To do this, we first denote the total system power (in watts) that task i consumes while executing in frequency f as W_i^f. Then, as energy is the product of power and time, we can write the energy consumed by a task as:

$$E_i = \sum_{f \in \mathcal{F}} (C_i^f \delta_i^f W_i^f).$$

[1]This assumes that a task is homogeneous in terms of its frequency of memory access.

Then, the total energy due to the computation of all tasks is:

$$E_C = \sum_{i \in \mathcal{T}} (E_i).$$

We denote the power consumed when a processor is idle as W_I. Because the processors are either idling or executing a task, we can find the idle time by subtracting the sum over all individual task execution times from the program execution time. The sum of the task execution times is clearly $\sum_{i \in \mathcal{T}} T_i$. Because we have P processors, each running for a total of ω time (no processor can finish until all have finished), the total elapsed time over all processors is $P \cdot \omega$. Thus, the total idle time is:

$$I = P \cdot \omega - \sum_{i \in \mathcal{T}} T_i.$$

Since our experiments found that the system power while communicating was much closer to idle power than computing power we treat communication as idle time. We explored modeling communication with a fixed amount of (non-idle) time per byte, but found that assuming instantaneous communication gave better results. We assume a more complex communication model that handles the overlap between communication and computation will perform better. The total idle energy is given by

$$E_I = W_I I.$$

Finally, the total system energy is

$$E = E_C + E_I.$$

Our work focuses on minimizing E.

This model ignores the cost of switching from one CPU frequency to another, because including this cost requires integer linear programming. The measured switching time on an Opteron 265 using the `sysfs` interface is rather small, ranging from 32 to 850 microseconds depending in part on the frequencies used.

3.2 Linear Programming Formulation

In this section we describe how we translate our execution model into a linear programming formulation. The linear program takes as input both task and application constraints. For each task we require dependency information and the power requirements per frequency. The single application constraint is overall execution time at the highest frequency. The linear program then creates a schedule for all tasks that minimizes energy consistent with the given constraints.

We minimize the objective function

$$E = W_I I + \sum_{i \in \mathcal{T}} \sum_{f \in \mathcal{F}} W_i^f \delta_i^f C_i^f$$

subject to the following constraints:

- *Start Time:* No task can start before its predecessors complete, so for each task i and every task $j \in Pred_i$, we have:

$$S_i - (S_j + T_j) \geq 0$$

- *Completion Time:* All tasks must complete within the total execution time limit, (ω), so:

$$\omega - (S_i + T_i) \geq 0$$

- *Idle Time:* Since the LP would otherwise treat the energy consumption of idling processors as zero and produce an incorrect schedule, we must account for their energy use and explicitly include total idle time as a constraint:

$$P \cdot \omega - \sum_{i \in \mathcal{T}} T_i = I$$

- *Sufficient time:* The execution time of each task i must be sufficient to complete all of the work of the task using the frequencies selected. We guarantee this using:

$$\sum_{f \in \mathcal{F}} \delta_i^f = 1$$

The decision variables are δ_i^f and S_i in this formulation. All decision variables are constrained to be non-negative.

The most efficient execution will use at most two adjacent frequencies for a given deadline and set of fixed frequencies if the relation between CPU frequency and voltage is at least quadratic (Ishihara and Yasuura, 1998). While this relation holds for CPU energy, it may not necessarily hold for total system energy. The activity of other system components' (e.g., fans) can result in a system power graph that is not strictly quadratic. In this case, we have observed that the LP will still choose at most two frequencies, but that they may not be adjacent.

3.3 Implementation

This section describes the implementation of our LP-based infrastructure that bounds energy savings. It consists of a trace collection mechanism, an LP solver, and a run-time mechanism to leverage slack that the LP solver cannot remove. Broadly speaking, our system determines a near-optimal energy savings using the following procedure:

- First, run the program and generate a communication graph.
- Input the graph to the LP solver, which outputs an *energy schedule*. This schedule contains the CPU frequency (or frequencies) to be used for each task. It also ensures that any remaining communication slack is executed at the lowest frequency.
- For validation, re-run the program using the energy schedule and measure execution time and total system energy.

CPU Frequency (MHz)	1800	1600	1400	1200	1000	1000 (Idle)
Power (watts)	153	142	130	123	116	105

Table 3.2: Measured power per frequency for *UMT2K*.

3.3.1 Trace collection

First, we collect multiple traces by executing the application at each frequency available on the machine. At the fastest available frequency, we use a custom PMPI library that intercepts selected MPI calls to capture the local communication information: the timestamp, source, and destination of each MPI call. This trace is sufficient to determine communication slack We run the program at the other frequencies in order to obtain the execution time of each task at each frequency (which gives us memory slack information) and the average power that an application consumes at each frequency (power usage varies by application). An example of power consumption per frequency for *UMT2K* is shown in Table 3.2.

We note that this does require $|\mathcal{F}|$ runs of the program. An alternative way to measure slowdown in just a single execution is to compute the *memory pressure* of each task in a program—in other words, how much memory traffic it generates—and map this to task slowdown. While that is more efficient, our approach is more accurate for bounding the gains available for on-line heuristic techniques. The inherent inaccuracy of average power computed over several microbenchmarks makes this method unsuitable for this work.

3.3.2 Solution via linear programming

Next, our system combines the data into a single, system-wide task graph, and the graph is transformed into an LP matrix. The matrix, the total execution time limit, and the time for each task at each frequency are passed into the LP solver `glpk`, the GNU Linear Programming Kit (Makhorin, 2005). The solution to the linear program is converted into a schedule, listing how long each task should run in each frequency so that system energy is minimized.

3.3.3 Task binding

Our system input also includes a *task binding threshold*. Any task that has a computation time below this threshold is bound to the fastest frequency. This mechanism avoids excessive frequency changes for tasks with little work. It also greatly improves the running time of the LP solver, which is a function of the non-bound task count. Currently we have set this threshold to 10 ms; we have found that in practice a lower threshold can lead to an execution time increase due to too many changes in CPU frequency. Likewise, assigning these tasks to the lowest frequency can cause the LP solver to fail to converge on a solution.

3.3.4 Validation

For validation, our system re-executes the program using the schedule and intercepts each *send*, *receive*, *wait*, *barrier*, and *allreduce* call, changing the frequency as necessary. Our system must regain control of the program without relying on MPI invocations for any task scheduled to use two frequencies. For these tasks, we set a timer to expire at the time to change frequencies. The timer's signal handler then changes the frequency based on the schedule.

3.3.5 Limitations

Our current prototype has a few limitations; we note that even with these limitations, our system produces schedules that result in a relatively tight bound. First, we do not instrument the computation within the MPI library, treating the entire time spent in the MPI library as communication. Second, in the case of the asynchronous communication operations we do not consider scaling `MPI_Wait`. Scaling `MPI_Wait` operations that match `MPI_Isend` operations could cause an increase in execution time if the MPI implementation blocks until the matching receive is posted—turning asynchronous communication into synchronous communication. As our large-scale benchmark, *UMT2K*, uses these operations to send multi-megabyte messages, this is not simply a theoretical issue. Finally, we only instrument key MPI communication operations.

3.4 Experimental Results

This section reports our performance results. We first provide our experimental methodology. Then, we provide results on three different applications.

3.4.1 Experimental Methodology

For all experiments, we used a cluster of eight machines, each containing two AMD Opteron 265 dual-core processors. We used only one core on each node, so we use the terms interchangeably in this section. We chose OpenMPI (The OpenMPI Team , 2010) as our MPI implementation. The machines are connected by gigabit ethernet and have 2GB of RAM. The Opteron 265 supports CPU frequencies 1000 MHz through 1800 MHz in steps of 200 MHz. All frequency shifting was done through the `sysfs` interface made available by a modified Fedora Core 2 OS running the 2.6.16 kernel. All applications were compiled with `gcc` using the `-O2` optimization flag, except for *UMT2K*, which was compiled with `icc` and `ifort`. No other processes except for the usual daemons ran on the system during our testing.

We measure execution time and energy consumed. Reported results are from the experiment that produced the median energy over five runs. Execution time is elapsed wall clock time. The energy consumed by the entire system is measured by precision multimeters at the wall outlet. We emphasize that we report direct program executions and measurements, not simulations or emulations. In addition, the energy is *total system energy*, not just CPU energy.

For each application, we compute a bound on energy savings using our linear-programming technique described in the previous sections (denoted *LPS* here). We compare our results to several other energy-saving approaches.

The first, called *Comm*, reduces the frequency while the processor is blocked at a communication point. The second, *Slack*, collects the *total* slack time over all nodes and assigns a single (fixed) frequency to each node based on that slack. Slack is based on Jitter (Kappiah et al., 2005). The node with the least slack runs at the fastest frequency, and the other nodes are assigned a frequency intended to save as much energy as possible

without a significant time delay. Third, $Freq_n$ assigns each node a fixed frequency (e.g., $Freq_1$ for the second-fastest; $Freq_0$ is fastest and is used as the baseline for normalization). Finally, *Theoretical Bound* is an upper bound (but *not* in general an achievable one). We determine this bound with the LP solver, using two optimistic assumptions: switching cost is zero and the MPI implementation does not block on any send operation. Whereas *LPS* is conservative on operations such as `MPI_Wait`, *Theoretical Bound* is free to lower the frequency aggressively .

We normalize all execution time and energy results to the unscheduled execution. The unscheduled execution uses the stock OpenMPI library with no communication calls intercepted. For the energy results, a number below one means the scheduled version saves energy.

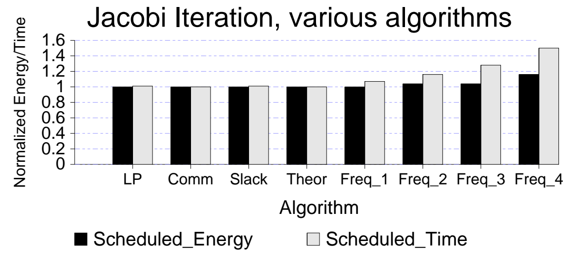
3.4.2 Applications

This section describes our three test applications. The first is *Jacobi*, which is a PDE solver and is a canonical benchmark. The second, denoted *Particle*, is a particle simulation based on MP3D from the Splash suite (Singh et al., 1991). The third is *UMT2K*, a photon transport code that operates on unstructured meshes from the ASC Purple suite (Lawrence Livermore National Laboratory, 2001).

Jacobi is load-balanced and is intended to verify that essentially no energy savings are available, while the latter two programs exhibit some degree of load imbalance and hence are amendable to saving energy.

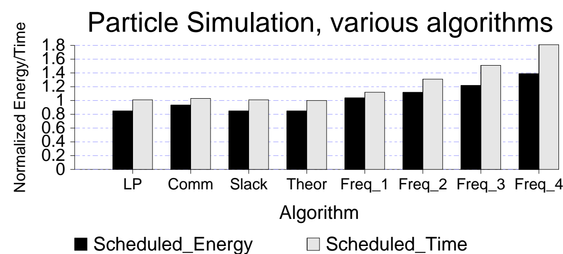
We first consider *Jacobi*; Figure 3.2 presents the results. Clearly, no energy savings are available due to the balanced load, large computation to communication ratio, and modest demand on memory. Our bounding technique, *LPS*, clearly shows this as expected.

Next, Figure 3.3 shows that *Particle* is amenable to saving energy. Specifically, our bounding technique reports a 15% potential energy savings without any time increase. Inspection of the schedule generated revealed that the load imbalance was significant enough to allow early-finishing nodes to execute in the slowest frequency (1000 MHz). For the *Particle* program, the *Slack* technique performs very well; in fact, it saves the maximum amount of energy (15%), also without any significant time increase. This optimal



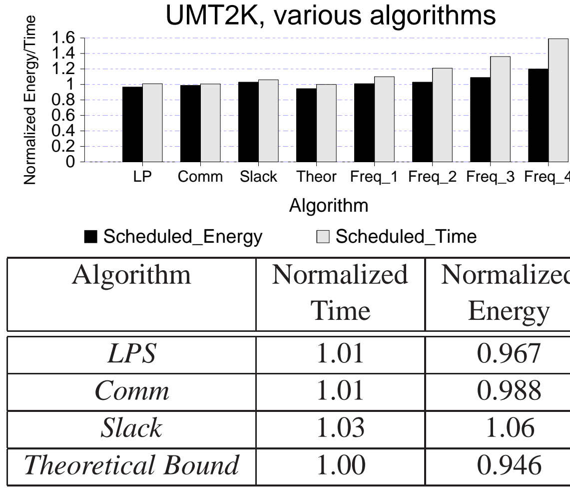
Algorithm	Normalized Time	Normalized Energy
LPS	1.01	1.00
Comm	1.00	1.00
Slack	1.01	1.00
Theoretical Bound	1.00	1.00

Figure 3.2: `Jacobi` with various algorithms.



Algorithm	Normalized Time	Normalized Energy
LPS	1.01	0.85
Comm	1.03	0.93
Slack	1.01	0.85
Theoretical Bound	1.00	0.85

Figure 3.3: `Particle` with various algorithms.



Algorithm	Normalized Time	Normalized Energy
LPS	1.01	0.967
Comm	1.01	0.988
Slack	1.03	1.06
Theoretical Bound	1.00	0.946

Figure 3.4: *UMT2K* with various algorithms.

result is because (by chance) the amount of slack available on each node allows scaling such that execution is nearly identical to the schedule generated by *LPS*. In general, we would expect *Slack* to perform well but not optimally. Finally, *Comm* saves about half as much energy as *Slack* and also has a 3% overhead to do so. Clearly, *Particle* is a program where energy saving is available and relatively straightforward to achieve.

Finally, we studied a large-scale parallel program, *UMT2K*. This program has been categorized as load-imbalanced (Vetter and Yoo, 2002) and uses both MPI and OpenMP (we disabled OpenMP for our tests). To cut down on machine usage, we reduced the number of iterations that the standard *UMT2K* input file uses.

Figure 3.4 presents the results. *LPS* computes a bound of 3.3% energy savings, while *Theoretical Bound* for this application is 5.4%. *LPS* incurs a time delay of just under 1%, which we believe is due to a combination of frequency switching time (not modeled by the LP solver) along with the overhead to execute the schedule. Again, we emphasize that *Theoretical Bound* is an upper bound on energy savings that is not actually achievable, i.e., the optimal energy savings is *smaller* than *Theoretical Bound*. Nevertheless, the results show that *LPS* results in a reasonable bound—it generates a schedule that is close to *Theoretical Bound* in terms of energy savings.

LP Solver Version	1800 MHz	1600 MHz	1400 1MHz	1200 MHz	1000 MHz	Blocked
Standard	381	35.3	110	0	0	33.7
Worst Case	319	200	0	0	0	41.0
Theoretical Bound	324	47.6	119	13.7	32.1	23.6

Table 3.3: Scheduled seconds per frequency.

Unlike the *Particle* program, current energy-saving heuristics have trouble with *UMT2K*. First, *Comm* provides little energy savings, as for this application much of the savings is in the computation, not at blocking points. The schedule provided by *LPS* provides significant additional energy savings—2.1%, more than two and half times that of *Comm*. The *Slack* algorithm, which is so effective for *Particle*, is completely ineffective for *UMT2K*. Because it uses *total* slack per node to arrive at a schedule, which provides no guarantee that the time bound will be honored. *Slack* does execute some nodes in lower frequencies than *LPS*, so it can conceivably save more energy for a given application—but for *UMT2K*, it actually consumes *more* energy because of the combination of the time increase (which will usually occur with *Slack*) *and* the fine task granularity of *UMT2K*. *Slack* works on entire program iterations and so is too coarse-grained to handle programs such as *UMT2K*. Finally, using a fixed lower frequency takes more time, as expected. More importantly, the energy consumption increases no matter the choice of frequency.

Figure 3.5, which shows the *LPS*-derived schedules, makes several things clear. First, nodes 0, 1 and 2 have the most communication slack, while nodes 5, 6, and 7 have the least. Second, while there are some similarities, the schedules are different on each node. Finally, determining such a schedule (on each node) by hand is simply not feasible. As a means for comparison, Figure 3.6 shows the schedule on node 0 for *Comm*. It is simpler—it varies only between 1800 MHz (for computation) and 1000 MHz (when blocked)—but does not result in a program that is as energy-efficient as the ones that use *LPS*.

Next, we investigate the effects of memory slack. To measure the improvement gained by considering memory slack, we implemented an alternate version of our LP solver, denoted *Worst Case*. This version instruments the program in only the fastest frequency and then assumes a worst-case slowdown for all other frequencies. For example, when

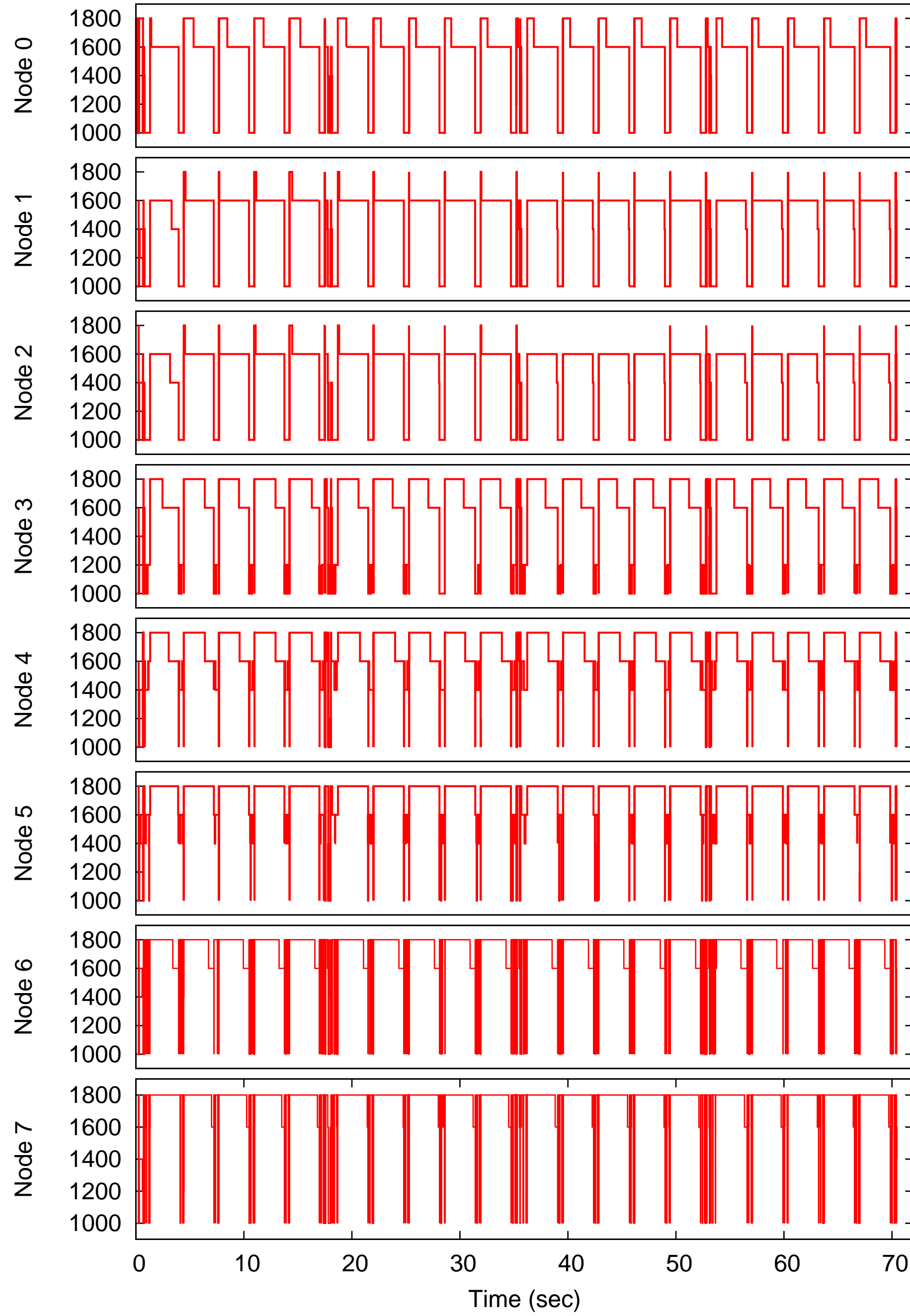


Figure 3.5: *LPS* schedule used by each node for *UMT2K*, MHz on y-axis.

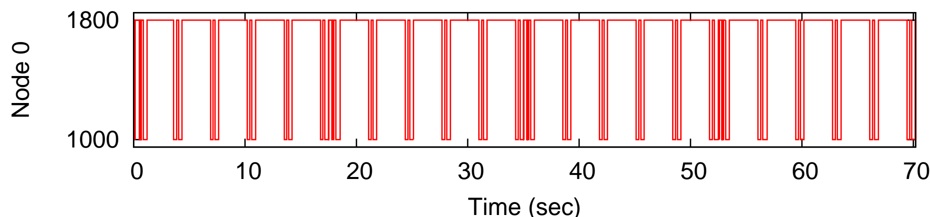


Figure 3.6: *Comm* schedule on node 0 for *UMT2K*, MHz on y-axis.

changing the frequency from 1800 Mhz to 1600 MHz, *Worst Case* assumes that every task slows down by a factor of 1.125 $(1800/1600)$.

The schedules for each version are shown in Table 3.3. For reference, the table also shows the schedule for *Theoretical Bound*. In *UMT2K*, the execution time of some tasks slows down by less than the ratio of the frequencies, which results in the *Worst Case* LP solver generating a schedule that executes more in 1600 MHz and less in 1400 MHz. The schedule when using *Worst Case* results in an increase in predicted energy consumption of about 0.5%. We ran *UMT2K* using this schedule and found the actual energy increase to be slightly greater than 1%.

3.5 Related Work

Several researchers have developed techniques and systems to save energy with a greater or lesser increase in execution time. Cameron et al. (Cameron et al., 2005) as well as Hsu et al. (Hsu and Feng, 2005) developed run-time systems to save energy given a user-specified performance constraint . Kappiah (Kappiah et al., 2005) developed a heuristic to reduce the CPU frequency in an adaptive manner to save energy in load-imbalanced programs. Additionally, Springer et al. (Springer et al., 2006) and Ge et al. (Ge et al., 2007) developed analytic models to predict as well as to understand energy consumption in the context of scaling programs to run on larger numbers of processors.

Our system differs from all of these in that we find a near-optimal solution in terms of energy savings. That is, the schedules that our LP solver generates can act as a baseline for comparison of the techniques described in the papers above—which are “best-effort”

techniques. Furthermore, our system generates the actual schedule that realizes the near-optimal energy savings.

Many have addressed finding optimal energy savings without a time increase in the real-time community. Wu et al. provide a taxonomy of dynamic voltage and frequency scaling: online vs. offline, and formal vs. ad hoc (Wu et al., 2005a). Our work falls into the offline and formal categories. Two of three examples cited by Wu et al. are intended for serial codes (Hsu and Kremer, 2003; Lorch and Smith, 2001). The third example is introduced by Xie, and is closest to our work. Xie's work (Xie et al., 2003, 2005, 2004) uses an ILP model to determine maximum energy savings on a single processor, develops a heuristic that approximates the ILP very well and runs much faster, and proposes an analytic model that allows DVFS to be applied across separate programs. Our goals and techniques are similar in that we both use mathematical models to bound the maximum energy savings. Our work, however, uses linear programming in the distributed computing domain, which introduces significant additional complexity for the LP model.

Another axis of classification is serial vs. parallel vs. distributed. Wu's work covers serial and core multiprocessor (Wu et al., 2005a). The latter assumes that any processor core can be assigned any task that is ready to run. Our domain is slightly different in that all tasks are assigned to specific processors at the start of the program.

Other researchers (Ishihara and Yasuura, 1998; Swaminathan and Chakrabarty, 2000, 2001; Saputra et al., 2002) have used Mixed Integer Linear Programming to solve the DVFS scheduling problem. All have been for single processors. Zhang et al. used an LP approximation of an ILP solution for the parallel real-time domain (Zhang et al., 2002). This work was continued by Mochocki et al. (Mochocki et al., 2002, 2005) with an emphasis on accounting for frequency transition overhead costs. Non-optimal distributed real-time energy scheduling has been investigated by Zhu's et al. work on slack reclamation, and Moncusi's et al. work on hard real time end-to-end deadlines (Zhu et al., 2003; Moncusí et al., 2003).

Dramatic energy savings usually require the user to accept longer execution times. Both the amount of energy saved as well as the priorities of the user determine if delay is considered significant. Thus, no metric of "good" performance has been widely-

accepted (Hsu et al., 2005). We have assumed that the user will tolerate as little delay as possible so our schedules last no longer than the run time at the fastest available frequency.

Ishihara first investigated the implications of DVFS (Ishihara and Yasuura, 1998). DVFS has been used to reduce energy in environments as diverse as web server farms (Elnozahy et al., 2002; Sharma et al., 2003) and mobile devices (Flautner et al., 2002; Noble et al., 1997).

DVFS scheduling algorithms have been explored in great detail. For example, Both Noble et al. (Noble et al., 1997) and Grunwald et al. (Grunwald et al., 2000) examined the effects of multiple algorithms on interactive programs in the mobile computing domain. Weiser et al. used past CPU utilization history to determine a frequency for future time periods (Weiser et al., 1994). Our work is similar only in that we use DVFS—we are looking at specific applications, and we must honor a specific time bound.

Instead of finding the most cost-effective way to speed up a schedule, aka "crashing" (Render and Stair Jr., 2000), we find the most resource efficient way to slow it down. We assume the schedule modifications are deterministic, much like CPM. Also, PERT networks are able to model stochastic activity times, and recent work shows promise in being able to crash these types of networks (Haga and O'keefe, 2001).

Modeling a parallel program by a task graph is not a new idea. It is particularly common in handling programs that exhibit both task and data parallelism—the vertices are tasks, which can be assigned one or more processor, and the edges are dependencies between tasks. The CPR method (Rădulescu et al., 2001) gave an efficient algorithm to find an effective allocation of processors to tasks. Our work focuses on saving energy in tasks, not determining how many processors to assign tasks.

Finally, there are several examples of using program traces to analyze performance. These range from tools such as MetaSim and DIMEMAS, which allow cross-architecture performance prediction (Snavely et al., 2002), to KOJAK (Mohr and Wolf, 2003), which allows a visual representation of performance. The work of Noeth et al. (Noeth et al., 2007), which provides a way to compress MPI traces both within and between nodes, also complements our work.

3.6 Summary and Future Work

We introduced a system that uses linear programming (LP) to bound optimal energy savings tightly for a given MPI application. Our system creates a program trace, generates a communication graph, and uses an LP solver to determine the schedule. The system shows that existing techniques that make use of DVFS can work well for some programs, but for others little energy savings is possible. As our system has scaled to over ten thousand tasks, our work can be used as a baseline for energy-saving algorithms that exist or will be developed in the future.

Our near-term goals include incorporating other MPI operations, such as `MPI_Startall`, into our system. In addition, we will study a range of allowable delay values to investigate how much additional energy can be saved. We also plan to model one-way (asynchronous) versus two-way (synchronous) communication more precisely, which will require a greater understanding of the particular MPI implementation.

In the longer term, we are interested in developing this technique as a design tool. Given the communication trace of a benchmark from a supercomputer that does not use DVFS, we can model a similar system that is equipped with DVFS in order to demonstrate the potential energy savings. The next step would derive energy characteristics given only an architectural specification. Finally, we also plan to model multi-core architectures with an LP-based system.

CHAPTER 4

THE *ADAGIO* RUNTIME SYSTEM

We apply the knowledge gained from teh previous chapter to solve the following problem: how can near-optimal energy savings be realized at runtime, assuming no training runs and no source code modification. In this chapter we present our solution, the *Adagio* runtime system.

Our bounding approach for energy savings relies on scheduling changes at MPI communication calls, identifying the critical path of execution to ensure it is never slowed, and approximating ideal frequencies by splitting execution time over available discrete frequencies. However, offline scheduling requires a *complete* program trace at *each* discrete frequency. Further, the use of a linear programming solver to generate the schedule is far too costly to be done at runtime.

In this chapter, we introduce the *Adagio* runtime system, which achieves significant energy savings with negligible (less than 1%) increase in execution time. We accomplish this by adapting and extending the principles behind offline scheduling as follows:

1. Schedules in *Adagio* are generated from predicted computation time. *Adagio* uses a simple, robust algorithm that requires no application-specific knowledge.

2. Slowdown decisions in *Adagio* occur at runtime. We base initial scheduling on worst-case slowdown with subsequent, more aggressive scheduling based on observed performance.

3. *Adagio* limits critical path detection to information local to the processor. *Adagio* scheduling assumes that users will tolerate only negligible delay of MPI call completion.

4. *Adagio* identifies individual MPI calls through hashing the stack trace. Not only must *Adagio* correctly predict the computation, communication and slowdown associated with the upcoming call, it must also predict the upcoming call itself.

The above *Adagio* features are distinct from our offline bounding approach. In that work (presented in the previous chapter), we solve all of these problems using an execution trace from *all* available discrete frequencies.

Adagio is a unique run-time approach to HPC energy savings in that execution delay is negligible *and* the application source code need not be modified. Previous HPC energy-saving algorithms are fundamentally different. Approaches that use a fine-grain history mechanism to predict future behavior (Ge et al., 2007; Hsu and Feng, 2005) will save less energy than *Adagio* when we can leverage load imbalance to reduce the frequency during computation bursts. Another approach uses per-iteration delay to determine per-processor frequencies (Freeh et al., 2008b), but does not detect the critical path correctly in the general case and therefore risks significant program slowdown. *Adagio* differs from these approaches in that it slows only that computation known to be off of the critical path in order to realize energy savings.

We show the effectiveness of *Adagio* for real-world programs as well as standard benchmarks. This includes *UMT2K* (Lawrence Livermore National Laboratory, 2005) and *ParaDiS* (Bulatov et al., 2004), two complex, real-world programs. While incurring less than 1% delay, *Adagio* reduces total system energy consumption up to 8% for *UMT2K* and 20% for *ParaDiS* on 32 cores. These are significant savings because the power difference between the fastest and slowest frequencies on our experimental platform is only 39%. We include comparisons of *Adagio* to existing energy-saving algorithms. Using larger numbers of nodes and thus increasing the effects of load imbalance allows the increased savings when compared to the smaller runs used for our bounding work.

The chapter is organized as follows. We present definitions, assumptions and a taxonomy of existing runtime algorithms in Section 4.1. Next, Section 4.2 details *Adagio*, our new runtime algorithm. In Section 4.3 we compare the effectiveness of *Adagio* to similar algorithms using real-world applications and standard benchmarks. Finally, we discuss related and future work in Sections 4.4 and 4.5.

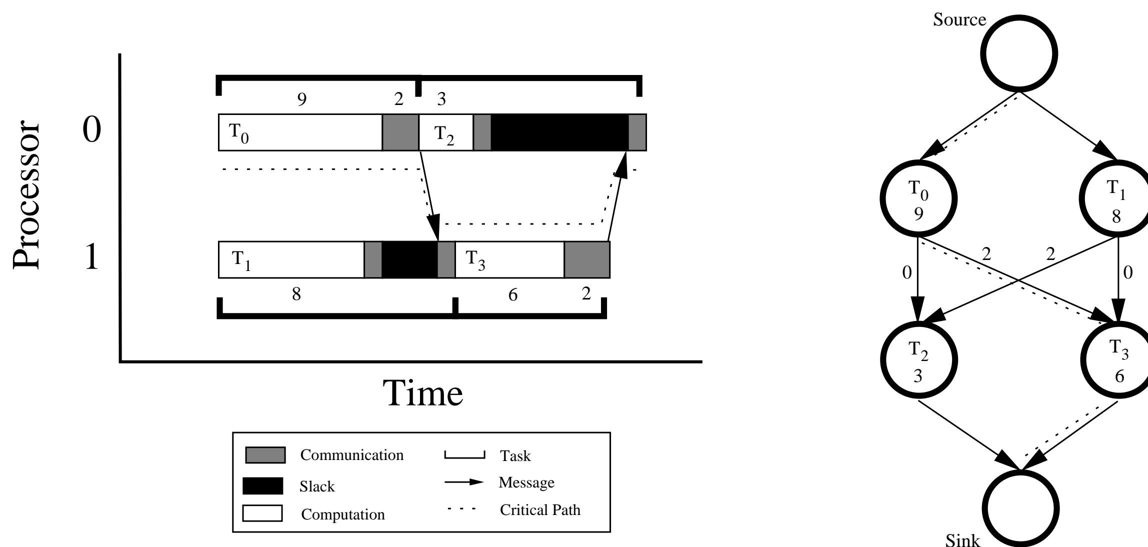


Figure 4.1: Typical SPMD Execution (left) and its task graph (right).

4.1 Overview

We place our work among existing algorithms based on common assumptions and definitions. We then provide a taxonomy of other approaches with their respective strengths and weaknesses.

4.1.1 Definitions and Basic Assumptions

We assume an SPMD (Single Program Multiple Data) programming model on a distributed-memory system using message passing, in our case MPI, for any communication between processes. For simplicity—and without restricting generality—we assume that each process is associated with a single core, although a single machine may have multiple cores. We refer to these cores as processors for the remainder of the dissertation.

Figure 4.1 illustrates our execution model. A *task* is the basic unit of scheduling, comprising total communication and computation that takes place on a single processor between the completion of two successive MPI communication calls. We measure task computation by an instruction count and an observed per-frequency instruction execution rate. We measure communication by recording the time spent within the MPI library.

We use *critical path analysis* to determine which tasks we can slow without incurring overall execution delay. A *critical path* (CP) is the longest path through a directed acyclic graph. For our analysis, each task forms a vertex in the graph and each edge indicates a dependence between tasks (*e.g.*, an edge exists between successive tasks on the same processor, and between a blocking send and its matching receive). Each vertex is weighted with the *normalized execution time* of that task, defined as the time required to complete the computation portion of the task when executing at the fastest frequency. Further, the graph has exactly one source, the `MPI_Init` function call, and one sink, the `MPI_Finalize` call. The critical path can change processors at any receive point (including calls with no explicit data transfer, such as `MPI_Barrier`).

We define time spent blocked in an MPI communication call as *slack*. By definition, while a processor executes on the critical path, it does not block while waiting for data to arrive during MPI communication calls: any process blocked waiting on remote communication can be slowed in order to complete exactly when the remote communication completes without affecting overall execution time. Thus, if a process is blocked, it cannot be on the critical path (non-blocked processes may be either on or off the path).

The *ideal frequency* is the slowest CPU frequency at which a given task can be run without incurring any slack, that is, the frequency necessary to finish “just in time”. The ideal frequency exists in the continuous domain: while the ideal frequency uses the minimum amount of energy (Ishihara and Yasuura, 1998), it is usually not one of the discrete frequencies available on the processor. If the ideal frequency occurs between the fastest and slowest frequencies, we can approximate the ideal frequency by executing part of the task in the faster neighboring frequency and the remaining portion in the slower neighboring frequency. We use the slowest frequency when it is faster than the ideal frequency.

4.1.2 Taxonomy

Ideally, runtime HPC *DVFS* algorithms satisfy three simultaneous goals: save as much energy as possible, increase execution time as little as possible, and support both simple and complex applications. No existing approach meets all of these goals. Table 4.1 summarizes the current state of the art in three classes of runtime algorithms along with

Algorithm Class	Online vs. Offline	Granu-larity	Modified App. Source	Critical Path Aware	Slows Comp.	Slows Comm.	Ideal Freq.
Offline Scheduling (Rountree et al., 2007)	Offline	**MPI call**	**No**	**Yes**	**Yes**	**Yes**	**Yes**
Scheduled Comm. (Lim et al., 2006) (Li et al., 2004) (Rountree et al., 2007)	**Online**	**MPI call**	**No**	**Yes**	No	**Yes**	No
Scheduled Iteration (Freeh et al., 2008b) (Liu et al., 2005)	**Online**	Timestep	Yes	No	**Yes**	No	No
Scheduled Timeslice (Jones, 2007) (Ge et al., 2007)	**Online**	Timeslice	**No**	No	**Yes**	**Yes**	No
Adagio (this work)	**Online**	**MPI call**	**No**	**Yes**	**Yes**	**Yes**	**Yes**

Table 4.1: Comparison of near-optimal offline scheduling to runtime classes.

a near-optimal offline scheduler and compares them to *Adagio*. We discuss existing approaches from each class in more detail in Section 4.4.

Offline Scheduling

We first briefly review the offline scheduler (Rountree et al., 2007) that uses a complete execution trace for *every* available CPU frequency as input. Given this input, linear programming determines a near-optimal schedule based on MPI call granularity, i.e., the critical path could move from one processor to another at any MPI communication call. Thus, this granularity allows critical path identification and prevents slowing of any computation along the critical path. While the MPI communication calls indicate *where* frequencies are to be changed, this algorithm is near-optimal because it lowers the frequency of the computation surrounded by these calls, thus (so far as is possible) eliminating slack.

Often, no single frequency exists that is low enough to remove all slack but not so low that the slowed computation impinges on the critical path. Judiciously splitting the computation across two neighboring frequencies (as detailed in Section 4.2.2) allows close approximation of the ideal frequency, saving additional power with no additional delay.

Using these techniques, the offline scheduler essentially places an upper bound on the effectiveness of any *DVFS* algorithm, either online or offline. While its high cost precludes using it in a production environment or at runtime, the design of *Adagio* reflects lessons learned from this approach.

Scheduled Communication

The simplest class of runtime algorithms, which we term *Scheduled Communication*, uses *DVFS* to reduce energy consumption when program execution blocks on MPI communication (Li et al., 2004; Lim et al., 2006; Rountree et al., 2007). This matches the granularity used in offline scheduling. Because data transfer is not computationally intensive, slowing these transfers generally incurs a negligible increase in overall execution time. Because no computation is slowed, the critical path is not affected, avoiding calculation of the ideal frequency. *Scheduled Communication* algorithms save energy in highly complex, production-quality MPI programs, with no source code modification. However, they leave significant potential energy savings untapped.

Scheduled Iteration

Scheduled Iteration methods (Freeh et al., 2008a; Liu et al., 2005) compute the total slack per processor per timestep, then schedule a single discrete frequency for each processor for the upcoming *timestep* . We define a timestep intuitively as an iteration of a scientific application's outermost loop. This timestep-level granularity works well for simple applications where the critical path remains on a single processor for the duration of each timestep. However, applications with complex communication patterns may have a critical path that crosses several processors during a timestep. In this case, algorithms of this type will choose not to slow any processor that contains any portion of the critical

path, thus forgoing any energy savings that might be had on those processors elsewhere in that timestep. As such, we classify these algorithms as not critical-path-aware and not using ideal frequency, although they can be significantly more effective than *Scheduled Communication* algorithms, at least on simple applications, due to the slowing of computation during the timestep. Also, this class of algorithms may require modification of the application source code to indicate the boundaries of the timestep.

Scheduled Timeslice

Scheduled Timeslice methods (Jones, 2007; Ge et al., 2007) schedule at fixed time intervals. These algorithms predict the execution characteristics of the upcoming interval (i.e., timeslice) based on recent intervals. They generally select the lowest discrete available frequency for each processor such that predicted slowdown does not exceed a user-specified limit. This timeslice granularity cannot track the critical path, nor do these algorithms use the ideal frequency to match the specified delay. Thus, any delay specification smaller than what would be achieved using the second-highest frequency will result in no computation being slowed. This approach does not require modification to the application source and can save significant energy, but only where significant delay can be tolerated.

Conclusions

None of the existing runtime methods achieves all of our goals, because none of them combines the design criteria of MPI-call granularity, slowing computation using ideal frequencies, and respecting the critical path. *Adagio* combines all of these without requiring modification to the application source code.

4.2 Adagio

We begin this section with the *Adagio* implementation. We then discuss optimizations for ideal frequency calculation, slack reclamation, and large message handling.

Variable	**Explanation**	**Variable**	**Explanation**
$\bar{f}$	Slowest frequency available on the machine.	ϕ	Ideal frequency for a task.
$\hat{f}$	Fastest frequency available on the machine.	t_{comp}	Total observed computation time.
t	Total observed task time (includes communication, computation, and slack).	t_{copy}	Total time required for message copying (does not include blocking time).
t_{lib}	Total observed time spent in the MPI library (includes copy and blocking time).	t_{target}	Available time for computation. *Adagio* slows the processor to take exactly this time.
R	Rate at which a processor executes a task computation (instructions per second).	I	Number of instructions executed during computation.
$taskid$	Unique identifier for each task, generated by hashing the stack pointers.	$Rates[taskid][f]$	Table of instructions per second for each task $taskid$ at each discrete frequency f.
		$Sched[taskid]$	Table holding frequency schedule for each $taskid$.

Table 4.2: Variables used within *Adagio* and their purpose.

```
1  PreTask()
2
3     taskid = hash(stack_pointer_chain)
4     if isnew(taskid) then
          /* First instance of a task:  Choose fastest
             frequency.                                      */
5         ; f = f̂
6     else
          /* Look up correct frequency.                      */
7         ; f = Sched[taskid]
8     SetFreq(f)
9     InitPerformanceCounters()
10    RunTask(taskid)
11
12 PostTask()
13
      /* Generate schedule for next execution of this
         task.                                               */
14    ; Record I, t_comp, t_lib.
15    Rates[taskid][f] = I/t_comp
16    t = t_comp + t_lib
17    t_target = t − t_copy
18    if isnew(taskid) then
          /* First instance of a task:  Set slowdown
             rates to worst-case for each available
             frequency.                                      */
19        ; for f ∈ F do
20            Rates[taskid][f] =
21             Rates[taskid][f̂] ∗ f̂/f
22        end
23    /* Find slowest frequency that respects the
         critical path.  Default is fastest frequency.
         */
24    ; Sched[taskid] = f̂
25    for f from slowest (f̄) to fastest (f̂) do
26        if I/Rates[taskid][f] ≤ t_target then
27            Sched[taskid] = f
28            return;
29
30    end
31
```

Figure 4.2: Adagio algorithm with no optimizations.

4.2.1 Adagio Implementation

As *Adagio* is task based, we must predict the properties of the next task that will execute after a given task. This prediction requires that the algorithm first determines which task will occur next. To accomplish this, we create a signature for each task based on a hash of the pointers that make up the stack trace. The hash is generated when the MPI call associated with the task is intercepted by our library. The record of each completed task contains the hash of the task that had been observed to follow it immediately.

Before the computation of a task begins, *Adagio* fetches the frequency schedule for the task and changes the operating frequency to the first one in the schedule. It also initializes hardware performance monitors (HPMs) to monitor the code. After a task, *Adagio* collects data and determines the frequency schedule for the next execution of that task. We stress that *Adagio* executes on each processor and tailors schedules to computation performed on each processor.

The first time a task is observed, we record the task that preceded it and execute it at the highest available frequency. This forms the basis for the prediction of computation, communication, and blocking times. We assume that task behavior will essentially be identical every time it is executed. This very simple predictor captures the behavior of many scientific applications.

Figure 4.2 shows pseudocode for *Adagio* for the simple case of using a single frequency per task. We detail the optimized split-frequency case in Section 4.2.2. As runtime algorithms have no prior information about program execution characteristics, *Adagio* schedules execution at the fastest frequency (represented by $\hat{f}$) when it first encounters a task If the task recurs, *Adagio* schedules it under the assumption of *worst-case slowdown* (execution slowdown proportional to that of the change in frequency). Computation will not slow by more than the ratio of the change in frequencies. For example, running a task at 1.6GHz instead of 1.8GHz will cause no more than a 12.5% delay. The communication and memory-boundedness of a task may lower this delay substantially. Thus, *Adagio* records the observed slowdown when a task is scheduled and executed in a particular frequency. *Adagio* then uses this refined estimate for subsequent scheduling.

Table 4.2 summarizes the variables that we use to describe *Adagio*'s algorithm. Throughout this discussion, we will use $\hat{f}$, and $\bar{f}$ to denote the fastest and slowest operating frequencies (i.e., MHz) and will use them to index into tables $Sched$ and $Rates$ (instead of standard array indices).

Adagio records the number of instructions I and the instructions per second R for the current frequency and task when it completes. Recording R allows *Adagio* to estimate how fast a task would run if it ran at the fastest frequency—a significant contribution beyond previous work (Freeh et al., 2005; Springer et al., 2006), which lacked an algorithm to determine execution time as a function of frequency. Further, recording I allows *Adagio* to determine when execution characteristics (i.e., computation) have changed between task instances. We measure the number of instructions using PAPI (Mucci and the PAPI Team, 2009). We also use `gettimeofday` to measure the total execution time of a task, which we need to compute the instructions per second metric. We emphasize that these HPMs are collected at runtime and only for those frequencies that are actually used. No training runs are necessary.

We record the total task time t, which is the sum of the task computation time, t_{comp}, and the time spent in its associated MPI call, t_{lib}. The target execution time, t_{target}, is set to the difference of t and the copy portion of the communication time, t_{copy}. We predict t_{copy} based on results obtained with microbenchmarks that vary the message size, such as a simple ping pong test. These microbenchmarks are application-independent and only need to be executed once in order to characterize a particular system.

We schedule the task to take time t_{target} during the upcoming timestep by iterating through all frequencies, slowest to fastest, and finding the slowest frequency that does not exceed the target, thus respecting the critical path. By definition, a task that blocks cannot be on the critical path, and so this algorithm will not slow any task that was on the critical path during the previous iteration. We do miss the opportunity to slow tasks that are both off of the critical path and do not block, but the additional algorithmic complexity required to detect such tasks is not warranted due to the limited additional energy that we could save.

As stated above, if we have not yet executed the task in a particular frequency f, we assume *worst-case slowdown*: given quantity $Rates[taskid][\hat{f}]$ (observed during the initial timestep),

$$Rates[taskid][f] = Rates[taskid][\hat{f}] \times f/\hat{f}.$$

Our assumption is conservative: the execution rate will not decrease *more* than the decrease in CPU frequency but might decrease less since memory references (and I/O) are independent of CPU frequency. Thus, slowing the CPU will not in general lead to as much slowdown as the slowdown in frequency. Scheduling conservatively does not increase overall execution time, and as soon as a task is executed at a particular frequency, we replace this pessimistic estimate with an observed value.

4.2.2 Optimizations

We now detail three novel optimizations: approximating ideal frequencies by using two neighboring frequencies, slack reclamation, and large message handling.

Split Frequencies

We determine the *ideal CPU frequency*, ϕ, for a task such that it executes in exactly t_{target} seconds. However, processors used in HPC environments operate only at a few discrete frequencies. To our knowledge, all other existing runtime algorithms choose a single frequency and either lose energy savings by running faster than the ideal frequency or lose time (and possibly energy savings) by running slower than the ideal frequency. In the worst case, $|\mathcal{F}| + 1$ iterations are required to discover the ideal frequency if task behavior is consistent across iterations.

We can approximate the ideal frequency by using its neighboring frequencies (Ishihara and Yasuura, 1998). Our optimized schedule still uses the fastest available frequency when the computation lies on the critical path and the slowest available frequency when the ideal frequency is even slower (slack remains in this case). In any other case, we calculate how long to run the epoch in the two frequencies immediately above and below the ideal frequency.

Let q be the percentage of time to execute at frequency f, and let frequencies $f > \phi > f'$. We must satisfy

$$t_{target} = q \times (I/Rates[e][f]) + (1 - q) \times (I/Rates[e][f']).$$

We solve for q for the given task and use the two frequencies for the corresponding durations ($t_{target} \times q$ seconds for frequency f and $t_{target} \times (1 - q)$ seconds for frequency f'). For this reason, *Adagio* stores q as well as f (f' is always one frequency below f) per task in $Sched$. Thus, each processor can generally execute each task at or very near the target time.

At the beginning of each task, we use `setitimer` to generate an interrupt after $q \times I/Rates[e][f]$ seconds that allows *Adagio* to switch to f'. When *Adagio* catches the `SIGALRM` signal, it records $Rates[e][f]$, and I (up to that point), switches the CPU frequency to f', and continues. We disable the alarm when entering the MPI function that ends the task to avoid interrupting the application when computation completes ahead of schedule. *Adagio* additionally stores $Rates[e][f']$.

Using a split-frequency schedule can lead to using a particular frequency for a small amount of time. This choice would incur a time penalty for the additional switch and decrease system stability (at least on our hardware). To counter this, we have empirically arrived at a *switching threshold* of $100ms$ for our cluster. We require any frequency switch to remain in the new frequency for at least $100ms$. Thus, we schedule no task for a lower frequency if the scheduled time would be less than the threshold, and we do not schedule split frequencies unless the time spent in both frequencies will be greater than the threshold (the higher frequency will be used for the entire task), Finally, we guarantee sheduled time exceeds the threshold time before switching to a lower frequency while in the MPI library.

Slack Reclamation

Tasks may still block during communication. If there was sufficient computation that the task had been scheduled, any blocking in excess of the buffer will use the lowest

frequency, as that will be the frequency chosen for the computation. However, there exists a second case where a task consists of a small amount of computation — too small to be scheduled — followed by a relatively large amount of blocking communication. To prevent using a high frequency for blocking, we determine the amount of time spent blocking during the previous instance of the task and, if this is greater than or equal to twice the switching threshold, set an alarm to expire at the threshold time. If the alarm fires before the task completes, we change to the lowest available frequency and remain there (if our prediction is correct) for at least another duration equal to the threshold.

4.2.3 Large Message Handling

The *FT* benchmark is unusual in several respects (Section 4.3 contains all results). Among these, required communication for a particular `MPI_Alltoall` call is measured in seconds instead of milliseconds. In this case, our threshold value is too short to handle the amount of communication that occurs. We could construct an *FT*-specific solution or we could attempt to model expected communication time for these calls. We have instead created a simple, general solution. A side effect of an all-to-all call is synchronization. We wish to separate out the communication time spent blocking waiting for other processes to arrive at the call from the communication time spent transferring data. So, at the beginning of large `MPI_Alltoall` calls, our library inserts an `MPI_Barrier`. Any slack present at this barrier can be reclaimed by scheduling the computation immediately before it as usual. The task terminated by the following `MPI_Alltoall` has almost no computation, and slack reclamation occurs as outlined in the previous section.

4.3 Results

This section reports our performance results. For all experiments, we used a cluster of sixteen nodes, each containing two AMD Opteron 265 dual-core processors. We used sixteen nodes and one core per processor in all tests (32 cores) except for those NAS tests that required the number of nodes to be a perfect square (in this case, we used a single core per node). We can independently set the frequency of each processor, but this early-

model multicore processor cannot scale core frequencies independently. Consequently, the second core on a processor consumes energy while doing no useful computation, reducing the energy savings we can achieve. *Adagio* has been designed in anticipation of processors with per-core *DVFS* control. Future work will extend *Adagio* to handle assignments of processes to cores consistent with our energy-saving goals.

We chose OpenMPI (The OpenMPI Team , 2010) as our MPI implementation. The nodes are connected by gigabit ethernet and have 2GB RAM each. The Opteron 265 supports CPU frequencies 1000 MHz through 1800 MHz in steps of 200 MHz. We use the `sysfs` interface made available by a modified Fedora Core 2 OS running the 2.6.16 kernel for frequency shifting. We compiled all applications with `gcc` or `g77` using the `-O2` optimization flag. The system ran no other processes during our experiments other than the usual daemons.

Our application set includes two complete applications, *UMT2K* and *ParaDiS*, as well as the programs in the NAS suite (NASA Advanced Supercomputing Division, 2006). For each application, we measure execution time (elapsed wall clock time) and energy consumed. We measure the total system power with precision multimeters at the wall outlet and compute energy using $energy = power \times time$ so energy is *total system energy*, not just CPU energy. While time, and thus energy, can vary across many runs of a benchmark, power does not vary much at all. All results are from direct program executions and measurements, not simulations or emulations. Each benchmark was executed using each indicated algorithm a minimum of five times, with the median time and energy values normalized against the median time and energy for benchmark execution with no *DVFS* scheduling. We also provide a *lower bound* to the energy consumption. This was computed using program traces and indicates the maximum amount of energy that can be saved given perfect knowledge.

When computing in a tight loop, each compute node in the system consumes 180 watts at the fastest available frequency and 117 at the slowest. Blocking at the slowest frequency reduces the 117 watts to 110. Thus, assuming no time increase due to frequency scaling, an overly optimistic upper bound on possible *DVFS* energy savings on these nodes is 39%. Real applications cannot achieve this bound without increasing execution time because at

least one processor must be on the critical path and run at the fastest frequency and, generally, not all non-critical-path processors can be run at the slowest frequency.

4.3.1 Algorithms

In Section 4.2 we described the design and implementation of *Adagio*. We now describe our comparison algorithms: *Fermata-1800*; *Adagio-Comp*; *Timeslice*; and *Jitter*.

From the class *Scheduled Communication*, we use *Fermata-1800* (Rountree et al., 2007), which uses the same technique as the slack reclamation algorithm in *Adagio*. All computation is executed at 1800 MHz, while communication runs at the slowest frequency if blocking time exceeds a 100ms threshold. *Fermata* slows only communication. We use *Adagio-Comp*, the *Adagio* algorithm with no slack reclamation, to show the effect when we slow only computation.

From the *Scheduled Timeslice* algorithms we implement *Timeslice*. These algorithms require the user to specify a delay that they will tolerate in order to save energy. A higher tolerance generally increases energy savings. With a delay tolerance of only 0% or 1%, similar to *Adagio*'s target, these algorithms save the most energy by only slowing communication since they do not use split frequencies. Choosing which communication to slow can only be based on the characteristics of previous timeslices, so at best the first timeslice that includes the communication call runs at the highest frequency, and the first timeslice after the communication call runs at the lowest frequency. Depending on the size of the timeslice, this best case approaches the performance of the *Fermata* algorithm.

To allow a fairer comparison, we instead have implemented an algorithm that gives a bound to the performance of algorithms in this class. We execute all computation at the second-highest frequency (1.6GHz) and use *Fermata* to execute communication at the lowest frequency. This results in a certain amount of delay (never worse than 12.5%) depending on the application. There are two factors influencing the resulting energy savings. Running at a lower frequency lowers power, but running longer increases time.

From the *Scheduled Iteration* algorithms we use *Jitter* (Freeh et al., 2008a). This algorithm slows the processor for an entire iteration. These iterations are not identified automatically: the user must annotate the source code with a call to `MPI_Pcontrol` and

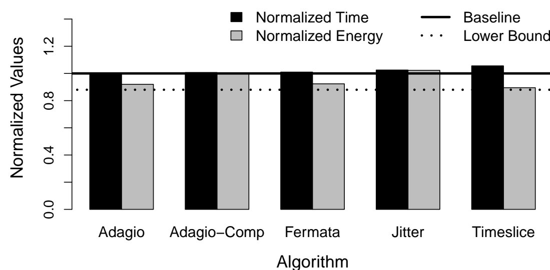


Figure 4.3: Normalized time, energy, and power for *UMT2K*.

recompile. *Jitter* can detect when an iteration on a particular processor ran longer than expected (when, for example, a portion of the critical path crossed that processor and was inadvertently slowed) and restores the processor to the fastest frequency for subsequent iterations. In the worst case, this can occur on every processor that has some amount of slack, and in this case *Jitter* will eventually schedule all the processors to run at the highest frequency. No energy will be saved, but the only delay will come from the slowdown of the few iterations that were tested.

4.3.2 UMT2K

The *UMT2K* benchmark (Lawrence Livermore National Laboratory, 2005) is part of the ASC Purple Benchmark Suite (Lawrence Livermore National Laboratory, 2001) assembled by Lawrence Livermore National Laboratory (LLNL). Extensive studies of this benchmark (Rountree et al., 2007) have shown that it is a very challenging test of energy savings for offline, let alone runtime, scheduling.

UMT2K implements a tree communication pattern that handles large, asynchronous messages (100KB+). The critical path does not stay on the same processor across an entire iteration. The tasks in *UMT2K* tend to have either a great deal of computation ended by a small amount of communication or very short computation followed by a large amount of communication. The implication is that task-level scaling of computation will generally be ineffective, but that significant energy may still be saved by selectively slowing communication.

We see this reflected in our results as shown in Figure 4.3, which indicates the lower bound for energy use as determined by offline scheduling. After finding the median values of runs with no energy scheduling, we recorded the median values of runs for each algorithm and normalized them to the nonscheduled median values. For this application, *Adagio* saved 8% energy with only 0.2% delay.

Fermata, *Adagio-Comp* and *Adagio* ran with less than 1% delay. We can pinpoint the source of the energy savings from examining *Fermata* and *Adagio-Comp*. Because *Fermata* only slows communication, and observing that it achieved 7.6% energy savings doing so, we can conclude that very little energy savings can be picked up from slowing computation. In fact, *Adagio-Comp* (which primarily slows computation) actually uses *more* energy than the nonscheduled runs (0.4% more). We cannot simply add the savings achieved by *Adagio-Comp* and *Fermata* together to estimate total savings since the combination into *Adagio* performed slightly better.

Jitter performs poorly on this application. The *Jitter* algorithm is sophisticated enough to determine when slowing a particular processor leads to overall slowdown, and that processor is returned to executing at the fastest frequency for the following iteration. For this application, *Jitter* is unable to find any processors where slack can be reduced without additional delay, and so ultimately ends up running all processors at the highest available frequency. In making this determination, though, *Jitter* introduces an overall execution delay of 2.5% *increases* energy use by 2.3%.

Our *Timeslice* algorithm saves the most energy for this benchmark: 10.5%. However, this comes at a cost of a 5.6% delay. *Timeslice* is the most effective algorithm if this kind of delay can be tolerated. However, supercomputers are purchased to run programs as fast as possible, and energy savings are likely to be interesting only within that constraint.

UMT2K presents the intriguing possibility that communication could be reordered to yield even greater energy savings. Currently, the application uses a barrier for synchronization after a large computation task. Because the computation is balanced, there is essentially no slack. Then the application performs a sequence of load-imbalanced asynchronous communications that terminate in another barrier or `MPI_Waitall`. Moving to a synchronous communication model could eliminate the need for barriers as well as

placing the inevitable slack into the same task as the load-balanced computation. To our knowledge, no research has been done concerning MPI programming techniques that allow for increased energy savings. We plan to revisit this issue in future work.

In summary, *UMT2K* presents a challenge for energy savings because of its complex communication pattern and the inability to slow computation without adding delay. Because *Adagio* can save energy by both slowing computation *and* communication, *Adagio* outperforms every other algorithm used on this benchmark, although *Fermata* is almost as effective. Unlike *Fermata*, *Adagio* also performs well when computation can be slowed, as we illustrate with the next benchmark: *ParaDiS*.

4.3.3 ParaDiS

ParaDiS (Bulatov et al., 2004) is a dislocation dynamics simulation used at Lawrence Livermore National Laboratory. It is a "chaotic" program that converges using a varying number of iterations for the same initial inputs over different runs. Program performance reflects this behavior—total run times in our experiments that do not use energy savings vary up to 4.9%. This nondeterminism makes the program unsuitable for offline scheduling. The power consumed, however, is consistent within an algorithm: the power requirement for each iteration is the same (to the extent we can measure it), and the varying number of iterations are reflected in the varying energy. Thus, while a lower bound on execution time would require multiple traces, a normalized bound on energy savings can be computed from a single trace.

ParaDiS exhibits load imbalance. It is possible to configure *ParaDiS* to perform dynamic load balancing, which reduces (but does not eliminate) slack available for *DVFS* scheduling and makes tasks more difficult to predict. We have successfully integrated *Adagio* into the dynamic load balancer provided by *ParaDiS* and have saved significant energy with less than 1% execution time delay.

The results are shown in Figure 4.4. Three of the algorithms had less than 1% delay, and all five achieved significant energy savings. This application is structured so that the critical path tends to stay on the same processor throughout the entire program. The task with the largest computation also has a large amount of blocking communication time

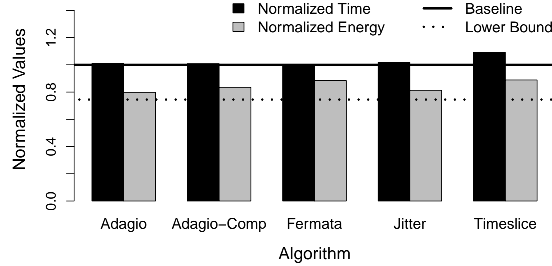


Figure 4.4: Normalized time, energy, and power for *ParaDiS*.

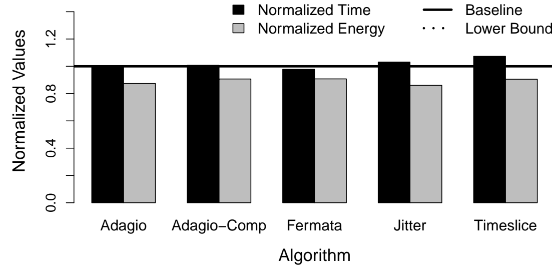


Figure 4.5: Normalized time, energy, and power for load-balanced *ParaDiS*.

(on processors off the critical path). There are also tasks that have short computation combined with long communication. As such, both *Adagio-Comp* and *Fermata* do well in isolation, and the combination into *Adagio* results in a 20.2% energy savings.

Due to this structure, *Jitter* also performed well, achieving 18.7% energy savings with a 1.8% delay. Our *Timeslice* algorithm did poorly. If no communication time existed on the critical path of the program and all of the computation was CPU-bound, *Timeslice* will slow execution by 12.5% (1.8GHz vs 1.6GHz). With *ParaDiS*, the delay is much closer to this worst case (unlike *UMT2K*, which is relatively communication bound). This additional delay does not accrue much energy savings — despite the 9.1% slowdown, *Timeslice* only achieved 11.1% energy savings, the worst of any algorithm.

The challenge with *ParaDiS* does not lie so much in identifying slack, but rather in making sure reclaiming the slack does not slow overall program execution. We must predict the next timestep using only prior information, However, prior information is not

necessarily a reliable guide to performance, especially at the beginning of the program and especially for this kind of benchmark.

The computation time predictor in *Adagio* proves to be robust in this situation. If the critical path time for each succeeding instance increases more than the non-critical path times, *Adagio* schedules the non-critical path processors to complete earlier than necessary. This misprediction leaves some amount of energy savings unexploited, but results in no additional delay. Additionally, when the critical path times decrease more slowly than processors off the critical path, *Adagio* also incurs no additional delay. In the case of *ParaDiS*, the change was similar enough across all processors that additional delay did not become an issue. *Jitter*'s prediction of slack also takes advantage of this. *ParaDiS* is an example for which predicting tasks and predicting iterations give similar results, as opposed to *Timeslice*'s prediction of timeslices.

These *ParaDiS* results show what can be achieved with worst-case load imbalance, but *Adagio* also performs when when on this benchmark when enable the internal load balancer provided with *ParaDiS*. Figure 4.5 shows the results of these runs. We were not able to calculate a lower bound due to the chaotic nature of the benchmark, but even with reduced load imbalance *ParaDiS* saved 13% energy with less than 1% increase in execution time. We achieved these savings not only despite *ParaDiS* not saving energy during load balancing iterations, but also despite having to relearn a new schedule after each of these iterations.

4.3.4 Summary of UMT2K and ParaDiS results

Broadly speaking, there are two methods for saving energy using *DVFS* in MPI programs: slowing communication and slowing computation to reduce slack. The former tends to work well in most programs with significant communication time due to large message sizes, of which *UMT2K* is an excellent example. However, *UMT2K* has relatively little slack and thus presents little opportunity to save energy by slowing computation. *ParaDiS* has far greater load imbalance and thus greater slack. This not only provides an opportunity to save energy by slowing task communication, but also for energy savings by slowing task computation.

ParaDiS shows that *Adagio* outperforms other runtime algorithms when load imbalance allows computation to be slowed, and our *UMT2K* results show that *Adagio* outperforms other runtime algorithms when load imbalance allows only communication to be slowed. For each application, one of the existing techniques attains performance close to that of *Adagio* (*Fermata* for *UMT2K* and *Jitter* for *ParaDiS*), but performs relatively poorly on the other application. *Adagio* is the only algorithm that performs well in both situations. We now turn to the NAS Parallel benchmarks, a well-known suite of load-balanced kernels.

4.3.5 NAS Parallel Benchmarks

The NAS Parallel Benchmark suite (NPB) (NASA Advanced Supercomputing Division, 2006) is a well-known collection of benchmarks for parallel computing maintained and distributed by the NASA Advanced Supercomputing division These benchmarks generally have a well-balanced computational load, implying no communication slack and thus no obvious opportunity for energy savings. However, *FT* and *CG* proved to be the exceptions. While both are load-balanced benchmarks, the ratio of communication time to computation time in *FT* was high enough that energy could be saved by only slowing communication, and *CG* is sufficiently memory-bound that slowing computation on the critical path resulted in a less than 1% delay. We explore the implications of both of these benchmarks later in this section.

We executed each benchmark over 32 processors except in the case of *BT* and *SP*, which require a perfect square for the number of processors (in this case, 16). We chose the class size so that the benchmark would run long enough to guarantee an accurate reading on our power meters. This was class C in all cases except *MG*, where we moved to the larger class D.

We present the results in Table 4.3. As expected, with the exception of *FT*, no significant energy was saved with *Adagio*; the largest delay was 0.4% (in *SP*). This result is important because *Adagio* is able to recognize when saving energy would incur non-negligible delay. Moreover, the tasks in some of the NAS programs are small enough that if *Adagio* tried to schedule them, scheduling overhead (e.g., frequency switching) would

Bench-mark	Normalized Time				
	Adagio	*Adagio-Comp*	*Fermata*	*Timeslice*	*Jitter*
bt.C	0.999	1.000	0.991	1.046	1.038
cg.C	0.999	0.997	1.000	1.009	1.063
ep.C	1.009	1.008	1.016	1.122	n/a
ft.C	1.017	1.015	1.004	1.015	1.027
lu.C	0.994	0.992	0.988	1.096	0.901
mg.D	1.000	0.997	1.003	1.048	1.043
sp.C	1.004	1.001	1.000	1.034	1.119
Bench-mark	Normalized Energy				
	Adagio	*Adagio-Comp*	*Fermata*	*Timeslice*	*Jitter*
bt.C	1.000	1.006	0.994	0.979	1.033
cg.C	0.995	0.992	0.995	0.911	0.812
ep.C	1.008	1.005	1.014	1.017	n/a
ft.C	0.821	0.990	0.823	0.781	0.950
lu.C	0.997	1.000	0.998	0.981	0.913
mg.D	0.983	0.997	0.985	0.940	0.993
sp.C	0.998	1.000	0.995	0.977	1.099

Table 4.3: Normalized Time and Energy for the NAS Parallel Benchmarks.

dominate, again leading to too much delay—and possibly even *increased* energy due to this extra delay. These results show that *Adagio* can be used *safely* on applications. When energy savings are possible, as with *UMT2K* and *ParaDiS*, *Adagio* will realize these savings with negligible delay. Where no energy savings are available, *Adagio* does no harm.

Of particular interest here is the range of performance presented by *Timeslice*. For a CPU-bound benchmark such as *EP*, slowdown in execution time essentially matches the slowdown of the processor. Slowdown occurs in *LU* as well, although to a lesser extent. Despite executing in a lower frequency, there are no significant energy savings due to the increase in execution time.

Jitter performed poorly overall, with the best energy savings (19% on *CG*) associated with the worst delay (6%)[1]. One anomaly is that *Jitter* resulted in 10% speedup on *LU*. In the past, we have observed repeatable small speedups in some benchmark configurations when the CPU is slowed. This appears to be caused by the side effect of staggering communication to reduce contention at individual processors. As this kind of speedup

[1]The previous reported *Jitter* results for the NAS applications are for 8 processors (Freeh et al., 2008b); this explains some of the discrepancy with our results.

has an effect on energy savings, it is an avenue for future study. Finally, the structure of the *EP* benchmark prevents *Jitter* from scheduling it.

At the other end of the scale are the benchmarks that are either memory-bound (*CG*) or communication-bound (*FT*). While both *Fermata* and *Adagio* do well on *FT* (18% savings with 0.4% and 1.7% delay, respectively) *Timeslice* does even better: 22% savings with only 1.5% delay. *Timeslice* saves energy on *CG* (9% savings with 0.9% delay) while no other algorithm does.

This illustrates one area of possible improvement for *Adagio*. The *CG* benchmark is entirely memory bound, but unless a task is associated with slack greater than the threshold, *Adagio* will not attempt to schedule that task, even when dropping the CPU frequency by 12.5% will result in less than 1% slowdown. We could address this in *Adagio* by first scheduling every task that falls below the slack threshold to use a combination of the fastest and second-fastest frequencies. This schedule will waste little time if the task is CPU-bound. However, measuring the execution rate at the second frequency will reveal if the task is memory-bound. In this case, *Adagio* can schedule the task appropriately even in the absence of slack. We can extend this approach iteratively to allow *Adagio* to save significant energy while incurring at most a small bounded delay in the absence of slack. Given an additional hardware performance monitor (outlined in the next chapter) we are able to improve on this and predict the effect of slowdown given an single run at an arbitrary frequency.

We now consider *FT*, which is unusual in two respects. First, communication time is an order of magnitude larger than computation time, due to repeated calls to an `MPI_Alltoall` that took up to ten seconds each in communication time alone. Second, the initial iterations of several tasks varied widely enough to cause significant misprediction and thus greater slowdown than had been expected. While later iterations could be precisely predicted, there were not enough total iterations to amortize the early error.

The former characteristic allowed *Adagio* to save 18% energy. Communication is not CPU bound, so executing it in the lowest possible frequency saved energy with negligible delay (the *Fermata* algorithm accomplished the same savings with only 0.4% delay). The latter characteristic caused an unusually high delay of 1.7%. The *Adagio-Comp* algorithm

saved essentially no energy while causing a 1.5% delay, and the *Timeslice* algorithm accomplished additional savings by scheduling all of the iterations, which *Adagio* cannot do because of its goal of negligible delay.

Several simple additions to *Adagio* could bring the delay results down to our tolerance. As the issue is misprediction, the solution can either make the current predictor less sensitive to variation or use application-specific knowledge to create a better predictor. Since one of our goals is to avoid application source modification or other application programmer intervention, we only examine the former. The simplest modification would hard code a minimum of n executions of a task before beginning to schedule it. This solution succeeds, at least in this case, since the error is confined to the "warm up" iterations of *FT*. A more general solution would require that n iterations are within p percent of the average computation time before scheduling can begin *or* resume. An even more complex solution, implemented in an earlier version of *Adagio*, calculates the accumulated percentage delay at runtime and only allows scheduling to occur when that delay falls below the tolerance. In all three cases, only computation scheduling is affected. *Adagio* will continue to save some amount of energy by slowing the CPU during communication.

We have chosen not to implement any of these solutions because we are not persuaded that a real problem exists. The C class version of *FT* ran for a handful of iterations; a more realistic benchmark would have amortized the error over a greater number of iterations. We have observed a similar pattern of behavior in *ParaDiS*; benchmarking runs of a dozen iterations produced suboptimal results, due to startup variability. But in practice more realistic *ParaDiS* runs show our simple predictor is more than adequate to meet our defined limits.

4.4 Related Work

Previous work on static scheduling (Rountree et al., 2007) has most heavily influenced the design and implementation of *Adagio*. Specifically, the concepts of MPI-level scheduling granularity came from that work, as did the use of split frequencies. The latter ultimately originated in the real-time work of Ishihara and Yasuura (Ishihara and Yasuura, 1998).

Several other dynamic voltage scaling runtime algorithms exist. In this section we detail algorithms from the classes *Scheduled Communication*, *Scheduled Iteration*, and *Scheduled Timeslice*, and briefly describe other related work.

4.4.1 Scheduled Communication

We choose three algorithms to illustrate the class *Scheduled Communication*. We described the first, *Fermata* (Rountree et al., 2007), in Section 4.3. Li et al. (Li et al., 2004), implemented the second, *thrifty barriers*, a similar idea in spirit but aimed at chip multiprocessors. Lim et al. (Lim et al., 2006) developed the third, a technique to infer communication regions and lower the frequency during those regions. Unlike *Fermata*, this approach lowers the frequency on some computation. However, it is not aware of the critical path and so does not provide time guarantees; instead, it attempts to minimize the energy-delay product.

4.4.2 Scheduled Iteration

Section 4.3 described *Jitter* (Freeh et al., 2008b), which is the primary *Scheduled Iteration* algorithm of which we are aware for message passing programs. Liu et al. (Liu et al., 2005) slowed down computation before barriers, a similar idea, for chip multiprocessors. As mentioned earlier, because these approaches make scheduling decisions across the entire timestep, they cannot handle situations where the critical path migrates across processors within a timestep—even if the migration occurs at global synchronization points. Unfortunately, such migration is not unusual; in particular, it occurs in complex applications such as *UMT2K*. As *Adagio* predicts such migration, it provides better results for these kinds of applications.

4.4.3 Scheduled Timeslice

We select two algorithms to illustrate the class *Scheduled Timeslice*. The first, *CPU-Miser* (Ge et al., 2007), divides a timestep into many small timeslices, the size of which depends on the current frequency. *CPU-Miser* gathers hardware performance monitors

(HPMs) for each timeslice and uses past history to select a single frequency for the next timeslice. Another approach uses `cpufreq` (Jones, 2007), a simple command-line interface that makes use of the "userspace" CPU frequency governor provided by the Linux kernel. Frequency switches occur based on user-specified limits on CPU idle levels and/or CPU temperature.

As with *Scheduled Iteration*, these approaches do not respect the critical path, whereas *Adagio* does. While *Adagio* can schedule computation effectively in environments where little or no delay can be tolerated, *CPU-Miser* can require a significant delay (e.g., 5%) to schedule computation effectively. This makes *Adagio* a better fit for many classes of HPC applications, where the primary metric is execution time.

4.4.4 Other Related Work

Several researchers have developed techniques and systems to save energy with a modest increase in execution time. Cameron et al. (Cameron et al., 2005) and Hsu et al. (Hsu and Feng, 2005) developed some of the earliest runtime systems to save energy in a performance constrained manner for HPC applications. Additionally, Springer et al. (Springer et al., 2006) and Ge et al. (Ge et al., 2007) developed analytic models to predict or to understand energy consumption in the context of scalability. Similarly to Springer et al., Li and Martinez (Li and Martìnez, 2006) considered both reducing parallelism and frequency scaling, although in the context of chip multiprocessors. Their results showed power savings in almost every situation.

Recent work has explored reducing the amount of concurrency in programs, with one of the benefits of such reduction being lower energy. Ding et al. adapt behavior when cores on a chip multiprocessor are unavailable (Ding et al., 2008). Curtis-Maury et al. fork fewer threads for parallel regions when beneficial (Curtis-Maury et al., 2006). Both papers use linear regression to predict the effect on performance and minimize energy-delay. In contrast, *Adagio* aims at saving energy with negligible delay.

Many researchers have addressed finding optimal energy savings without a time increase in the real-time community. Several have used Mixed Integer Linear Programming to solve the *DVFS* scheduling problem (Ishihara and Yasuura, 1998; Saputra et al.,

2002; Swaminathan and Chakrabarty, 2001, 2000) but are limited to a single processor. Zhang et al. used an LP approximation of an ILP solution for the parallel real-time domain (Zhang et al., 2002). Mochocki et al. (Mochocki et al., 2002, 2005) continued this work with an emphasis on accounting for frequency transition overhead costs. Zhu's work on slack reclamation (Zhu et al., 2003) and Moncusi's work on hard real time end-to-end deadlines (Moncusí et al., 2003) have dealt with non-optimal distributed real-time energy scheduling. Other work (Rountree et al., 2007) used Linear Programming to derive an approximate upper bound on potential energy savings. Unlike *Adagio*, these solutions are all offline.

4.5 Summary and Future Work

In this paper, we have presented *Adagio*, a runtime *DVFS* algorithm aimed at saving energy in HPC applications with negligible delay. *Adagio* improves on existing runtime algorithms by using the proper semantic level of granularity, split frequencies, and normalized execution time. We applied *Adagio* to two real-world HPC applications—*UMT2K* and *ParaDiS*—and obtained significant energy savings with negligible execution delay.

We are exploring important open issues including the development of techniques that guarantee no added delay when slowing MPI communication. Porting this work to OpenMP as well as providing a hybrid MPI/OpenMP solution will allow many more applications to save energy. Incorporating processor sleep states into *Adagio* may allow savings of even greater amounts of energy if we manage the longer delays required to transition out of these states.

Many architectural issues also remain. Multicore chips should enable per-core *DVFS* control. Multicore optimization is a very active area of research, and the possibilities of leveraging this work while simultaneously saving energy are intriguing. We are actively working on using this approach in real-time systems, where bounding delay is vitally important. Finally, we provide in the next chapter an architectural model of how *DVFS* affects arbitrary applications and propose a novel hardware performance monitor that allows us to calculate the effect of changing the CPU clock frequency.

CHAPTER 5

ARCHITECTURAL MODEL

5.1 Introduction

While power consumption is also an important problem in embedded and consumer devices, the metrics used to measure effectiveness in these domains—latency and responsiveness—differ markedly from the primary HPC metric: execution time. Previous work has shown how selectively slowing processors in parallel, scientific applications can achieve significant energy savings with moderate impact on overall execution time (Rountree et al., 2009; Hsu et al., 2005; Kappiah et al., 2005; Ge et al., 2007). These techniques require mechanisms to predict how a lower CPU clock frequency will affect execution. Most work determines the appropriate CPU clock frequency under the assumption of *worst-case slowdown*: the code in question is assumed to be *CPU-bound* and will respond proportionally to the change in frequency. This conservative approach ignores the program's diminished sensitivity to changes in CPU frequency if any portion is memory-bound. Such conservative approaches unnecessarily sacrifice potential energy savings in order to prevent unexpected execution delay. A better approach would predict the effect of lowering frequency on execution time more accurately, thus saving greater energy, without increasing execution time.

Several researchers have proposed using regression over various sets of hardware performance monitors (HPMs) to predict the degradation in execution time given a single program trace. The general approach is to find correlations to CPU time, memory bus time, or both, and scale CPU time proportionally to the change in CPU clock frequency while holding the bus time constant. Neither characteristic can be measured directly, so work in this area has focused on identifying other, more easily measured aspects of program behavior that correlate well with memory boundedness. Instruction count rates and last-level cache miss rates are the two most commonly used metrics. The former hypothe-

sizes that increased memory boundedness will lead to a lower measure of instructions per cycle, and the latter hypothesizes that the degrees of memory boundedness can be defined by how often memory is accessed. While both ideas are true to an extent, complicating factors prevent either or both from being used as an effective performance predictor.

We begin this work by examining two additional hardware performance monitors (HPMs) provided by newer architectures: bus access counts and counts of cycles with at least one outstanding bus request. Both provide significant improvement compared to regressions based on L2 cache miss rates and instructions count rates, reducing standard error from 0.071 (using misses per cycle) to 0.024 (using bus request cycles per cycle). However, this correlation approach suffers from the same difficulties as earlier correlation approaches: improvement is dependent on the quality of training data, worst-case error can be relatively high even when median error is minimal, and the correlation does not provide any guidance for further reduction of error.

To address this, we have developed an architectural model of the interface between the processor and memory. We first examine the simplest useful architecture: an in-order processor where at most one memory request may be outstanding at a time. We analyze this model and show how previous models used HPMs to predict slowdown accurately . We then make this model more realistic by adding out-of-order execution and a memory bus that can handle multiple in-flight requests. This model is sufficiently complex to capture the behavior of a wide range of modern processors.

We then divide computation into intervals based on the time of issue and completion of memory access instructions. At this level, we assume all computation scales proportionally with the change in CPU clock frequency, and memory latency is modeled as constant time. At this point, we can translate the model into a graph on which we can perform critical path analysis. While multiple critical paths can exist, they are by definition equivalent and we only need to study one.

We separate loads into *load groups* in such a way that the *effective bus time* is the sum of the memory latencies of the last load in each group. This sum is constant across changes in CPU clock frequency (given that the bus clock frequency remains unchanged), and the remaining time (CPU time) is scaled proportional to the change in CPU clock

frequency. With this model in hand, we can now calculate performance at an arbitrary CPU clock frequency without using regression.

To evaluate this model, we chose to simulate how a simple additional HPM can be used to identify load groups and thus predict performance across changes in CPU clock frequency. We implemented this HPM in the PTLSim cycle-accurate simulator and executed several synthetic benchmarks as well as selected benchmarks from the SpecCPU 2006 and NAS Parallel Benchmark (NPB) suites. We compare our results to regressions over bus request cycles per cycle and show an improvement on standard error on the SpecCPU suite from 1.232% to 0.446% and from 0.888% to 0.122% over the NPB suite. Further, we identify a source of the remaining error and describe how to address it.

This chapter makes the following contributions:

- We propose an architectural model of CPU/memory interaction that identifies critical features required to predict performance across changes in CPU clock frequency.
- Based on this model, we propose a technique that allows direct calculation of performance at arbitrary CPU clock frequencies.
- We evaluate this technique using a cycle-accurate simulator and show reduced prediction error over the best regression-based solutions.

The remainder of the chapter is organized as follows. Section 5.2 gives a more detailed overview of the problem we are addressing. Section 5.3 discusses current best practice with regression-based models and improves on these using two recently-introduced HPMs. Section 5.4 provides the architectural model, interval analysis, and critical path analysis. Section 5.5 describes the performance of our leading loads approach under simulation. Section 5.6 covers related work, and Section 5.7 states our conclusions and outlines future work.

5.2 Overview

Power and frequency are quadratically related: increases in CPU clock frequency, f, require a quadratic increase in power, but by the same token reducing f provides an opportunity for saving quadratically more power. The domain of high-performance computing (HPC) can leverage this relationship in two different ways. For the same power budget, a dense and highly integrated architecture such as Blue Gene may use orders of magnitude more low-frequency processors than a similar cluster with high-frequency processors. For highly parallel workloads, the tradeoff of increased processor count for decreased processor performance has led to a substantial increase in overall performance.

Dynamic Voltage/Frequency Scaling (DVFS) is orthogonal to the low power processor approach. Instead of always using the same frequency, *DVFS* allows runtime f selections that best meet power consumption and performance requirements. This capability can be particularly effective with parallel programs. One can select the highest available frequency for overloaded processors while using slower frequencies that allow the remainder of the processors to complete "just in time". Recent work has shown this technique can realize the combined goal for *DVFS* in HPC: save significant energy while only incurring a negligible performance penalty (Rountree et al., 2009; Hsu et al., 2005; Kappiah et al., 2005; Ge et al., 2007).

We can more effectively realize this energy savings if we can reliably predict performance at a particular f—otherwise processors may complete their work early and waste energy while idling, or inadvertently introduce execution delay by taking too long to complete. For *CPU-bound* computation, which requires no main memory accesses, we can easily calculate the change in performance. Given an execution time t_0 at f f_0 and a target f f_n, the execution time t_n at the target f is simply:

$$t_n = \frac{f_0}{f_n} t_0$$

As the number of memory accesses increases, the program becomes increasingly *memory bound*, with a greater fraction of overall performance dependent on memory latency. Theoretically, an entirely memory-bound program, one that spends all execution time ac-

cessing main memory, experiences no slowdown when f is reduced since the change does not impact main memory access times. Although perfectly memory-bound programs do not exist, we must account for the degree of memory boundedness (Hsu and Feng, 2005) in order to predict performance. Time spent waiting for memory at f_0 does not incur performance loss at f_n and out-of-order processors may overlap those stall cycles with computation at the lower frequency.

The following section describes our improvements to the traditional regression-based HPM approach.

5.3 Evaluation of Existing Models

In this section, we review the current use of hardware performance monitors (HPMs) as proxies for predicting performance at arbitrary CPU clock frequencies. We show that while existing intuitive models based on instruction rates and miss rates have some predictive power, these models can be improved by using HPMs that more closely correlate with performance. By quantifying the success of these HPMs on real hardware, we can better estimate the effectiveness of our novel model presented in the next section.

5.3.1 Existing Models and Implementations.

Models A common model runs through nearly all of the work on performance prediction under changes in CPU clock frequency. Program execution is divided into CPU time and bus time. As the CPU clock frequency changes, CPU time is scaled proportional to the change in frequency and bus time remains the same. Each specific model provides its own variations on this theme. For example, Snowdon et al. break execution time down into CPU, bus, memory and IO components (Snowdon et al., 2007); Lee et al. distinguish between memory instructions and non-memory instructions (Lee et al., 2007); and Ge et al. distinguish between on-chip and off-chip time (Ge and Cameron, 2007).

Implementations Each of the above models captures the intuition of how performance is related to the proportion of CPU-boundedness vs. memory-boundedness. These are

effective models on single-issue architectures: memory access and computation do not overlap, determining the amount of time accessing memory is straightforward, and any remaining time scales proportionally to the change in CPU clock frequency. On out-of-order issue architectures, however, the sum of computation time and bus time is greater than program execution time.

To compensate for this, researchers have attempted to find indirect correlations between program characteristics and performance, with linear regression being the tool of choice. In general, we would expect that programs with a higher instructions-per-cycle (IPC) would be more memory bound than a program with a lower IPC, and we would expect that level 2 cache misses per cycle (MPC) would be higher in a memory-bound program than a CPU-bound program. A typical regression would look like:

$$\frac{T_{f'}}{T_f} = \alpha_0 + \alpha_1 IPC@f + \alpha_2 MPC@f$$

where f is the reference CPU clock frequency, f' is the target CPU clock frequency, T is execution time at a particular CPU clock frequency, and each α is a constant that minimizes error across the set of measured applications. In this case, to predict performance of a new application at frequency f', the cycles, misses, instructions and execution time are measured at f, and the regression equation computes the ratio of the measured time to the predicted time.

Experimental Results We draw benchmarks from the serial versions of the NAS Parallel Benchmark suite classes A through E modified to limit the total number of iterations. Table 5.1 summarizes our results on a quad-core Xeon 5440 processor. We compiled all benchmarks using *gcc* with the optimization level set to $-O2$. Unless otherwise noted, we obtained results by using a linear regression over hardware performance monitor (HPM) values per cycle at 2.833GHz (e.g., L2 data cache misses per cycle). We predict the ratio of cycles at 2.000GHz to cycles at 2.833GHz. A hypothetical completely CPU-bound program that takes time t at 2.833GHz would be expected to take time $\frac{2.833/2.000}{t} \approx 1.42t$ at 2.000GHz, while a hypothetical completely memory-bound program would take the

Table 5.1: Evaluation of Existing Models on the Core2 Architecture

Model	R-squared	Mean Absolute Error	Median Absolute Error	Standard Deviation of Error
I/C[1,3,4]	0.147	5.582	4.867	6.986
I/M[5]	0.445	5.161	4.912	5.986
M/C[1,2,4]	0.520	4.856	5.153	5.754
X/C[1,3]	0.909	1.897	1.060	2.546
R/C[6]	0.945	1.432	1.088	1.943
(I,M)/C[4]	0.505	4.845	5.092	5.772
(X,C)/I[3]	0.923	1.533	0.682	2.278
(I,M,X)/C[1]	0.941	1.278	0.567	1.987

I = Instructions, C = Cycles, M = L2 Cache Misses,
X = Bus Accesses, R = Cycle with 1+ Outstanding Reads.
[1] (Curtis-Maury et al., 2006) [2] (Snowdon et al., 2007)
[3] (Lee et al., 2007) [4] (Ge et al., 2007)
[5] (Freeh et al., 2005) [6] (Intel, 2007)

same amount of time at either CPU clock frequency. This bounds the ratio of change in execution time to the range $(1.0, 1.42)$.

The R^2 value is the standard measure of “fit” for regressions. For individual benchmarks (e.g., Figure 5.3.2) error is calculated for each benchmark as follows:

$$\text{error} = 100 \times \frac{\text{predicted} - \text{actual}}{\text{actual}}$$

For benchmarks as a group, we report the standard deviation of the error values as well as the mean and median of the absolute error values (e.g., Table 5.1). We list counters as well as models with the HPMs treated as single-HPM model in order to assess their contribution. As this is a summary, only principle HPMs of the models are listed. For details of the full models, consult the cited reference.

5.3.2 Discussion

Snowdon et al. created an effective model based on MPC for the PXA255 processor (Snowdon et al., 2007). This single-issue architecture creates a direct relationship be-

tween the number of cache misses and the amount of time spent on the bus. In out-of-order issue architectures, this link in not nearly as strong, and consequently this HPM is not nearly as useful. Likewise, papers published by Lee, Freeh and Ge examine performance of their models primarily on combinations of cache miss rates instruction rates (Ge et al., 2007; Freeh et al., 2005). Despite their wide use, these two metrics metric provide little additional accuracy on out-of-order issue architectures. For example, we report a mean absolute error of 1.897% for a regression using only bus transactions per cycle, and adding both I/C and M/C terms to the regression only reduces the error to 1.278%.

More recent processors provide HPMs that allow more direct measurement of bus time. Both Curtis-Maury et al. and Lee et al. take advantage of these HPMs. Curits-Maury et al. makes online predictions of IPC in order to select near-optimal thread granularity (Curtis-Maury et al., 2006) while Lee et al. predict execution time (Lee et al., 2007). The Curtis-Maury model was targeted to a different area of performance prediction. Here, we strip the model down to the three HPMs that are relevant for predicting performance across changes in CPU clock frequency: bus transactions, cache misses and IPC rates. This modification provides the best regression model we have found in the literature.

Lastly, the processor documentation provided by Intel suggests that the Outstanding Bus Requests HPM be used with a CMASK=1 in order to count the number of cycles with one or more outstanding reads (Intel, 2007). We abbreviate this as *1+ Reads/Cycle* or simply *RPC*. This metric is highly correlated with bus transactions and provides a slightly improved predictor of performance (mean absolute error 1.432%). To our knowledge, this HPM has not been used in any existing work in this area, even though its performance is equivalent to models requiring additional HPMs.

5.3.3 Limitations of current models

Regression-based approaches have a common limitation: the quality of the prediction depends on the quality of the training data. If a new program is sufficiently different, the error could be much greater than the regression would lead us to expect. If the phenomenon underlying the regression is understood in sufficient detail, we should be able

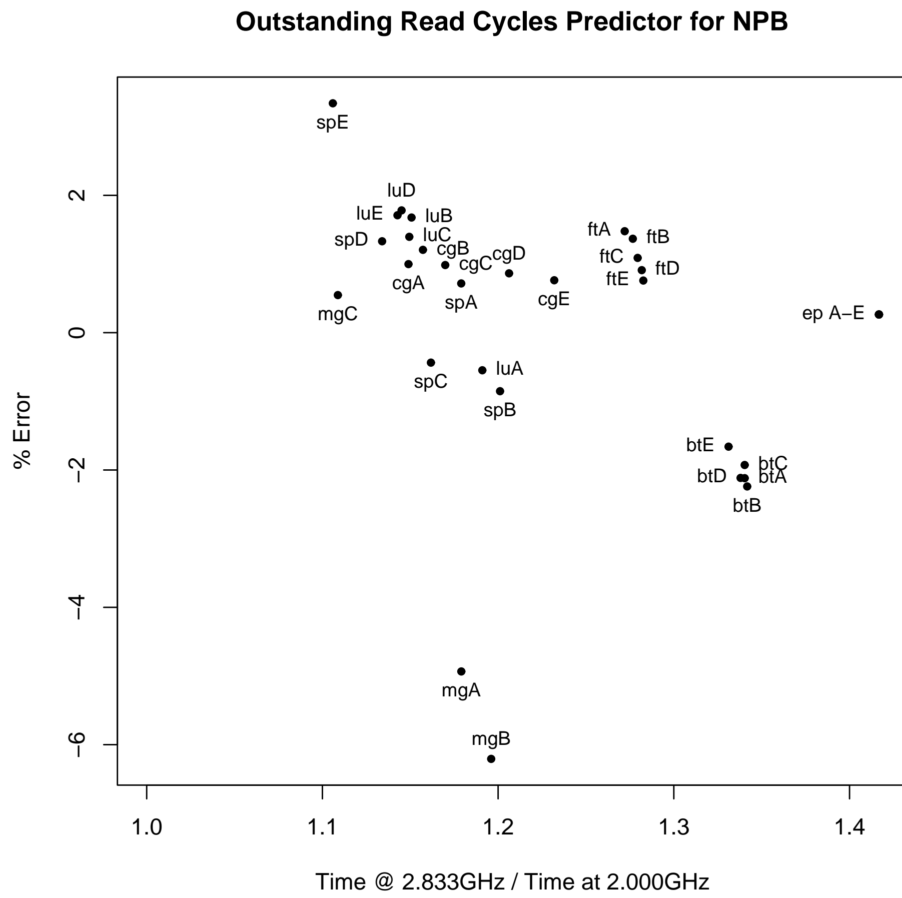


Figure 5.1: Regression over “outstanding read” cycles per cycle (RPC).

to predict which programs will be the outliers. Current models do not provide this level of understanding.

Figure 5.3.2 illustrates this. For most cases, the RPC predictor works well as we would expect from the low reported error and high R^2 values in table 5.1. However, both MG classes A and B are significant outliers, yet class C of this program is not. The BT benchmarks form another significant outlier. Current models do not explain these outliers and cannot predict if a new program will become yet another outlier. In the next section, we address these weakness by presenting a novel model that explicitly takes into account the out-of-order nature of modern server-class processors and use this model to create an improved technique for predicting performance.

5.4 Architectural Model

In the previous section, we showed that performance prediction using regression-based models offers only, and at best, good median performance. Significant error exists at the outliers and, far more important, a regression does not give any suggestion as to why this error occurs or how this error can be reduced. In this section we present an architectural model of the interface between the memory and CPU and apply it to create a consistently better prediction technique.

We analyze three progressively more realistic architectures:

1. An in-order execution architecture (Figure 5.2, a-c).
2. An out-of-order execution architecture with a bus capacity limited to a single outstanding read (Figure 5.2, d-f).
3. An out-of-order execution architecture with a bus capacity limited to an arbitrary number of reads (Figure 5.2, g-i).

We provide *interval analysis* for each model using *critical path identification* where applicable. We define these terms in the following subsection. Based on our analysis of each architecture, we provide guidance as to what should be measured in order to predict the effects of a change in the CPU clock frequency.

5.4.1 Assumptions, definitions and simplifications

We define a *load* as a non-speculative read that results in a last-level cache miss. Execution of each load begins with its *issue* on the cycle when the read begins and terminates some constant time *latency* later with its *completion* on the cycle when the data arrives from main memory. We assume the results of the load are required by one or more instructions further downstream, and that these instructions cannot begin execution until the load completes.

We divide execution into several *intervals*. An interval begins when any of the following conditions hold:

1. Program execution begins.
2. Beginning execution of instructions immediately following the issue of a load. (Implicitly, these instructions do not depend on the completion of the load.)
3. Beginning execution of instructions that would have blocked on the completion of a previous load given arbitrarily long latency.

The current interval ends when any of the following conditions hold:

1. Program execution ends.
2. A load issues.
3. Any condition indicating a new interval begins.

Intervals map to continuous execution: no interval maps to time spent with the entire CPU blocked on memory.

We represent individual interval execution duration as d_i and a constant memory latency time as L. Interval execution duration is strictly a function of the CPU clock frequency f. Memory latency remains independent of f. The model allows us to determine total execution time t_i from program start to the completion of interval i for a given f. Examples in this section use relative frequencies f, $f/2$ and $2f$.

These examples contain $2n$ intervals with the following dependences, where $a \rightarrow b$ implies b must complete before a can begin.:

- $I_0 \rightarrow \emptyset$. The first interval, I_0, does not depend on any other interval.
- $I_i \rightarrow I_{i-1} \forall_i \in I_{1 \ldots n-1}$. Any other interval cannot begin until the previous interval completes.
- $I_{0'} \rightarrow \{I_0, I_{n-1}\}$. The first interval that depends on a load completion ($I_{0'}$) cannot begin until *both* the previous interval (I_{n-1}) and the interval that issues the corresponding load (I_0) completes.
- $I_{i'} \rightarrow \{I_i, I_{i'-1}\} \forall_{i'} \in \{I_{1' \ldots n'-1}\}$. The same holds for every other interval that depends on a load completion.

We map intervals onto a graph theoretic representation where the weight of the vertexes represents the execution duration of the intervals, edges represent dependences between intervals, and the weights of the edges represent latency (which will be either zero or memory latency L). We define the graph *source* vertex as the first interval to execute and the graph *sink* vertex as the final interval to execute. We assume instruction scheduling and dependences are independent of CPU clock frequency.

The length of the longest path (or paths) from source to sink determines execution time; these are the *critical paths*. If a node or edge is on the critical path and the weight (duration) increases, the length of the critical path (overall execution time) with increase by the same amount. If the weight decreases that particular path length will decrease by the same amount. Only if there are no other critical paths through the graph at that point in time, this decreases the critical path. Increasing or decreasing weight of nodes or edges off of the critical path will not affect the length of the critical path unless the increase is sufficiently large to place the node or edge on a new, longer critical path.

Most modern processors use write buffers to hide store miss penalties, making the effective latency zero. Our model can also be extended to handle write-bound applications where write latency is a limiting factor on performance, but as none of the benchmarks we use demonstrate this behavior, we do not model writes in this dissertation. Our experimental results using cycle-accurate simulation presented in Section 5.5 demonstrate that we achieve significant accuracy despite the above simplifications.

5.4.2 In-Order Execution Model

We begin with a simple, in-order execution architecture. The first three examples of Figure 5.2 (a-c) shows the initial interval I_0 terminating in a load, followed by the latency associated with the load, a second interval I_1 also terminating in a load, the I_1 latency, and finally the intervals $I_{0'}$ and $I_{1'}$. We show this over three CPU clock frequencies $2f$, f, and $f/2$.

All intervals and latency lie on the critical path: if any interval or latency period took longer, the delay would affect total execution time. In terms of the model,

$$t_{1'} = d_0 + L + d_1 + L + d_{0'} + d_{1'}$$

Changing f results in a proportional change in d_i. Predicting the effect of a change of f in this architecture reduces to the problem of separating computation time from blocking time. Given that time spent blocked is both equal to latency L and constant, accurate prediction can be as simple as counting last-level cache misses, multiplying by L, and keeping this amount of time constant and adding to it remaining time scaled to the change in CPU clock frequency. If deriving L is difficult, then a simple regression over misses per cycle will provide satisfactory results (Snowdon et al., 2007).

5.4.3 Out-of-Order, Single Read Execution Model

We now begin to relax the assumptions for our initial model by considering an out-of-order architecture that, as in the previous architecture, can only accommodate a single outstanding read at a time. This architecture allows execution of an interval during the latency incurred by a load 5.2(d-f). Interval I_0 ends with a load immediately followed by interval I_1 during the latency associated with I_0. When the I_0 load completes, $I_{0'}$ begins during the latency of I_1, and when the I_1 load completes $I_{1'}$ begins.

This small change complicates the model.

$$T_{1'} = d_0 + \max(L, d_1) + \max(L, d_{0'}) + d_{1'}$$

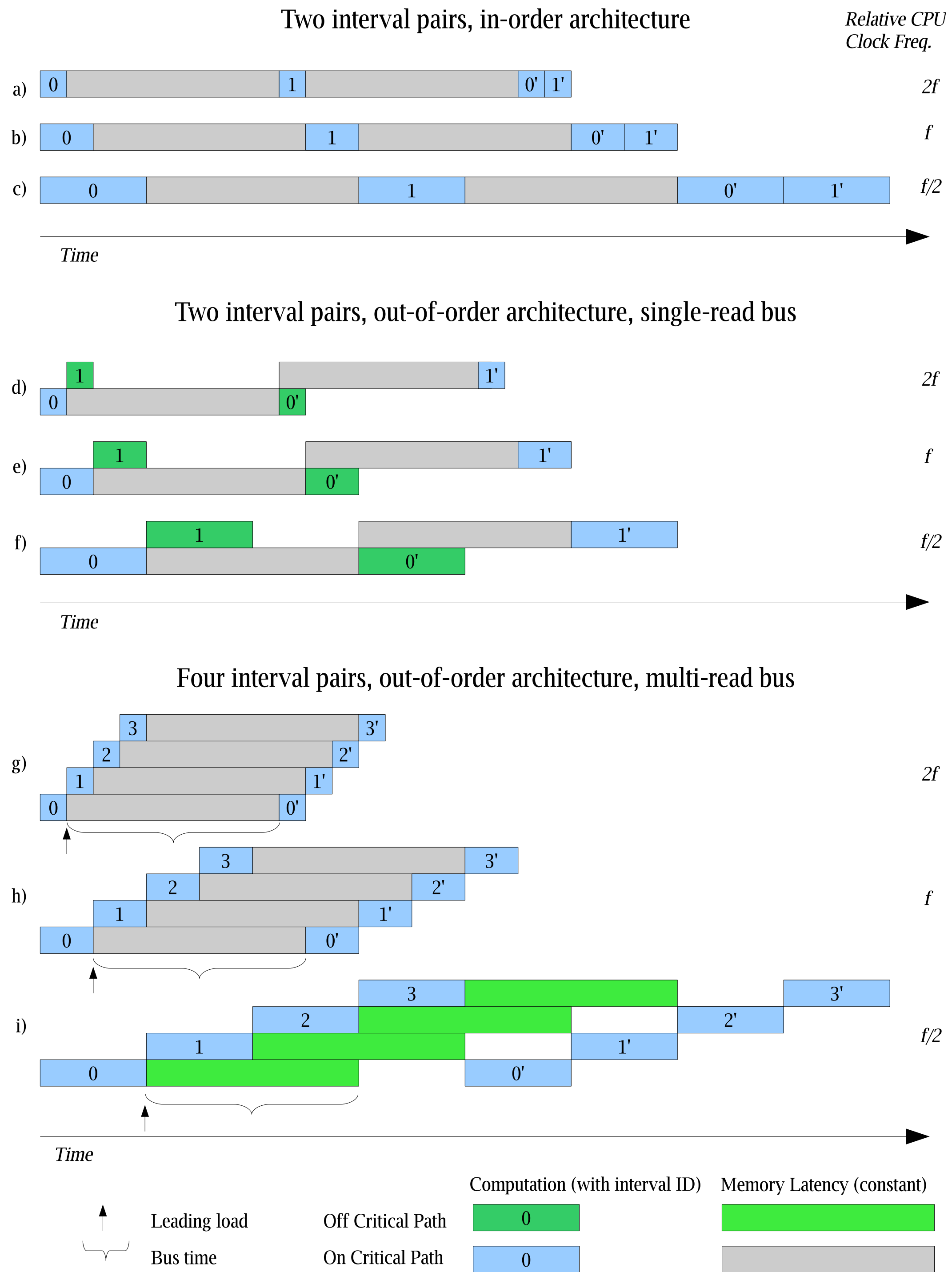


Figure 5.2: Interval Models: Three architectures and CPU frequencies.

If the duration of intervals d_1 and $d_{0'}$ exceeds the duration of the memory latency, this portion of the program is effectively *CPU bound*: slowing f will be reflected in a proportional change to $T_{1'}$. However, if $d_1, d_{0'} < L$, a change in f will only affect d_0 and $d_{1'}$, making the program (partially) memory bound: speeding up f will improve the execution time of only a portion of the program. The graph-theoretic representation shows that the critical path through the program also depends on whether sufficient computation exists (or the CPU is sufficiently slow) to fill the time taken by memory latency.

Making the distinction between "bus time" and "CPU time" is less useful on this architecture. If we make the assumption that there will generally not be enough out-of-order computation to fill the time taken by memory latency, we can reuse the same model developed for the previous architecture above. The number of loads multiplied by the memory latency time will give a total time not affected by changes to the CPU clock frequency (assuming further that the changes are small enough to prevent a *phase change*, defined below). If these assumptions do not hold, the model will tend to overestimate the amount of bus time available and underestimate the effects of changing the CPU clock frequency.

A *phase change* exists in this example when decreasing f causes intervals $d_1, d_{0'}$ to go from being less than L to exceeding L, or when increasing f causes $d_1, d_{0'}$ to go from exceeding L to being less than L. Given a large enough change in the relation of the CPU frequency to bus frequency, any execution where loads are present can be made either CPU bound (when the CPU clock frequency is arbitrarily slow relative to the bus clock frequency)) or partially memory bound (where the CPU clock frequency is arbitrarily fast relative to the bus clock frequency).

5.4.4 Multiple Read, Out-of-order Execution Model

Figure 5.2 shows the most realistic architecture: an out-of-order execution processor with sufficient bus capacity to allow multiple outstanding reads. This architecture breaks the link between load count and "bus time." Multiple intervals can occur during the memory latency of an initial load. Each of these can be terminated by another load. This models server-class processors. Previous models approximated simpler embedded processors.

The lessons learned from the previous model hold here as well. Computation that occurs during an earlier load's latency time is effectively "bus time". In this architecture, however, computation can occur under multiple load latencies. Changing CPU clock frequency changes the cumulative latency time as well as the time spent computing during latency. Using the model we have developed we are able to provide simple recursive definitions of total elapsed time.

$$\begin{aligned} t_0 &= d_0 \\ t_1 &= d_0 + d_1 \\ t_{n-1} &= \sum_{i=0}^{n-1} d_i \\ t_{0'} &= \max(t_0 + L, t_{n-1}) + d_{0'} \\ t_{1'} &= \max(t_1 + L, t_{0'}) + d_{1'} \\ t_{n'-1} &= \max(t_{n-1} + L, t_{n'-2}) + d_{n'-1} \end{aligned}$$

If interval durations are arbitrary, this mathematical apparatus will be of limited use. In practice, however, interval durations tend to be both homogeneous and short (due to limited instruction-level parallelism exploitable by the out-of-order core). To incorporate this into our model, we assume intervals are of a constant size c such that $\forall i \in I_{0\ldots n-1,0'\ldots n'-1}, d_i = c$. Given this constraint, we now have a direct solution to the prediction problem. If $\sum_{i=0}^{n-1} d_i < L$, then $t_{n'-1} = d_0 + L + \sum_{i=0'}^{n'-1}$. Alternatively, if $\sum_{i=0}^{n-1} d_i \geq L$, then $t_{n'-1} = \sum_{i=0}^{n-1} d_i + \sum_{i=0'}^{n'-1}$.

This captures two intuitions that are obvious from our explication of our model. If there exists sufficient computation to completely fill the load latency of the initial load, then the CPU is never completely blocked waiting for memory and (so long as $\sum_{i=0}^{n-1} d_i \geq L$ holds) execution time will be a function of CPU clock speed. However, if there is not enough computation to fill the latency incurred by the initial load, then the critical path of the graph proceeds through I_0, its latency L, and finally $I_{0'\ldots n'-1}$. Changing the CPU

clock frequency will affect the duration of any of these intervals (so long as $\sum_{i=0}^{n-1} d_i < L$ holds), but *not* intervals $I_{1\ldots n-1}$.

5.4.5 The *Leading Load* Technique

The above forms a rigorous description of a simple (albeit non-obvious) idea. For a set of intervals as we have defined them, the *bus time* is exactly L for the $\sum_{i=0}^{n-1} d_i < L$ case and exactly zero for the $\sum_{i=0}^{n-1} d_i \geq L$ case. The latter case is sufficiently uncommon that we are able to disregard it with negligible effect on the model's accuracy.

The former we measure as follows. We have a state variable `LeadingLoadPresent` initialized to zero and a `LeadingLoadCounter` also initialize to zero. The first load issued (resulting in a last-level cache miss) changes `LeadingLoadPresent` to 1 and increments `LeadingLoadCounter`. Neither variable is affected by an further loads until the initial load completes. At this point `LeadingLoadPresent` is reset to zero and the process continues at the next load.

The *bus time* of the program now becomes $L \times \texttt{LeadingLoadCounter}$. We hold this duration constant and scale the remaining program execution time proportionally to the change in the CPU clock frequency. The sum of this *CPU time* and *bus time* becomes the predicted execution time at the chosen frequency. Given a p percent difference in CPU performance, we formalize this as:

$$\begin{aligned}
\text{Predicted Execution Time} &= \text{Observed Bus Time} + p \times \text{Observed CPU Time} \\
\text{Observed Bus Time} &= L \times \texttt{LeadingLoadCounter} \\
\text{Observed CPU Time} &= \text{Observed Execution Time} - \text{Observed Bus Time}
\end{aligned}$$

We implemented this model in a cycle-accurate simulator and found it to be superior to the best existing models over synthetic, architectural and scientific benchmarks.

5.4.6 Implementation Issues.

It is beyond the scope of this dissertation to give a detailed proposal of how to implement this hardware performance monitor (HPM) *in silico*. However, we expect an eventual implementation should be relatively simple. We begin by adding an extra bit to every entry in the load/store queue. Insertions into this queue check to see if any entry in the queue has this bit set. If none do, the bit on the new entry is set high. While any entry has a high bit, an `LeadingLoadCounter` HPM increments once per cycle. When the load completes, the bit on the entry is reset along with the remainder of the entry and the HPM no longer increments.

As described above, we would prefer to only count leading load where the sum of the interval durations are less than the latency. It may be possible to extend this implementation so that the `LeadingLoadCounter` is only incremented if there exists both a leading load and a core-wide, memory-related stall cycle between issue and completion. Disambiguating memory stall cycles from other sources of stall can be problematic at best and it may be that the extra accuracy gained does not justify the additional cost. We plan to quantify this additional accuracy in future work.

We wish to emphasize that the implementation does not require any knowledge of intervals, durations or critical paths. These idea are present only in the model. Determination of dependences and instruction ordering ahead of execution along with cache modeling requires a prohibitive amount of complexity if done offline and a processor's worth of complexity if done online. The advantage of our model lies in abstracting away all of this complexity into a single counter while retaining remaining accurate enough to outperform existing approaches.

5.5 Evaluation of the *Leading Loads* Technique

After a thorough exploration of available hardware performance monitors (HPMs) we were unable to find a combination of metrics that could approximate our concept of *Leading Loads*. In this section we show how we validated this approach to performance prediction using the PTLSim cycle-accurate simulator.

5.5.1 Experimental Setup

The PTLSim Simulator Validation of this technique requires a cycle-accurate processor and cache simulator. We chose the PTLSim simulator and made the necessary changes to the source code to identify and to track leading loads. This simulator uses a compile-time setting to specify the memory latency: the number of CPU cycles taken by any last-level cache miss. Assuming that memory bus frequency is unaffected by changes to the CPU clock frequency, we can simulate changes in the CPU clock frequency simply by changing the number of cycles of memory latency (faster CPUs fill the same time with more cycles). As the unit of time in the simulator is the cycle and as this cycle does not have a defined duration, we predict execution time at n GHz given measurements taken at $2n$ Ghz. In other words, we predict performance for a processor running half as fast.

Benchmarks We used three benchmarks sets for the simulations: single iterations of 20 instances of the NAS Parallel Benchmark suite classes S, W, and A-C; complete runs of 28 instances of the SpecCPU 2006 benchmarks using the *test* class; and 170 unique instances of a synthetic benchmark that we created to increase test coverage. While these results are not directly comparable to the results obtained on the Intel Xeon, we show similar behavior across HPMs measuring misses, instructions and bus cycles and compare these to the leading loads technique.

Regression Models For each of the prior models, we calculate results via linear regression for each benchmark sets and the combined NAS Parallel Benchmarks and SpecCPU sets. The model-wide results use standard error, and the error of individual results is calculated as described in section 5.3. *Because we effectively use the same data for both testing and training, we expect the error of the regression models to be higher in practice.* These results should be considered as a lower bound of potential error rather than as an accurate estimate of how these models would perform in the real world. Table 5.2 summarizes three single-variable regressions (read cycles, L2 cache misses and instructions) as well as a regression combining all three variables (RMI).

Table 5.2: Evaluation of *leading loads* using PTLSim

Model	Benchmarks	R^2	Mean Absolute Error	Median Absolute Error	Standard Deviation of Error
(R) Read Cycles	all	0.9446	1.935	1.412	2.597
(M) L2 Misses	all	0.3324	6.648	4.814	10.010
(I) Instructions	all	0.6211	5.551	4.651	7.652
RMI	all	0.9727	1.470	1.058	1.901
RMI	npb+spec	0.9865	1.894	1.929	2.180
RMI	npb	0.9934	0.888	0.760	1.277
RMI	spec	0.9762	1.232	0.915	1.666
RMI	synth	0.9738	0.992	0.744	1.348
Leading Loads	all	n/a	0.293	0.205	0.434
Leading Loads	npb+spec	n/a	0.311	0.118	0.638
Leading Loads	npb	n/a	0.122	0.095	0.108
Leading Loads	spec	n/a	0.446	0.166	0.829
Leading Loads	synth	n/a	0.288	0.225	0.353

Leading Load Models Leading load models do not use regression. Instead, we record the number of total cycles and leading loads at frequency $2n$ and use this to predict execution time at frequency n. Table 5.2 summarizes the results. *Note that the error of individual leading load experiments is completely independent of the results of any other experiment.* Because we require no training data, the error is not dependent on surrounding results.

5.5.2 Discussion

The first third of Table 5.2 provides a reference point for judging the remainder of the results. As in the results obtained from the Xeon processor, we observe models based only on cache misses or instruction counts perform poorly when compared to a model based solely on bus cycles. Combining all three HPMs gives a slight improvement over the bus-cycle-only model, so we use this model in the remainder of the experiments. The middle third of Table 5.2 demonstrates the synthetic benchmarks are roughly as difficult to predict as the NPB benchmarks.

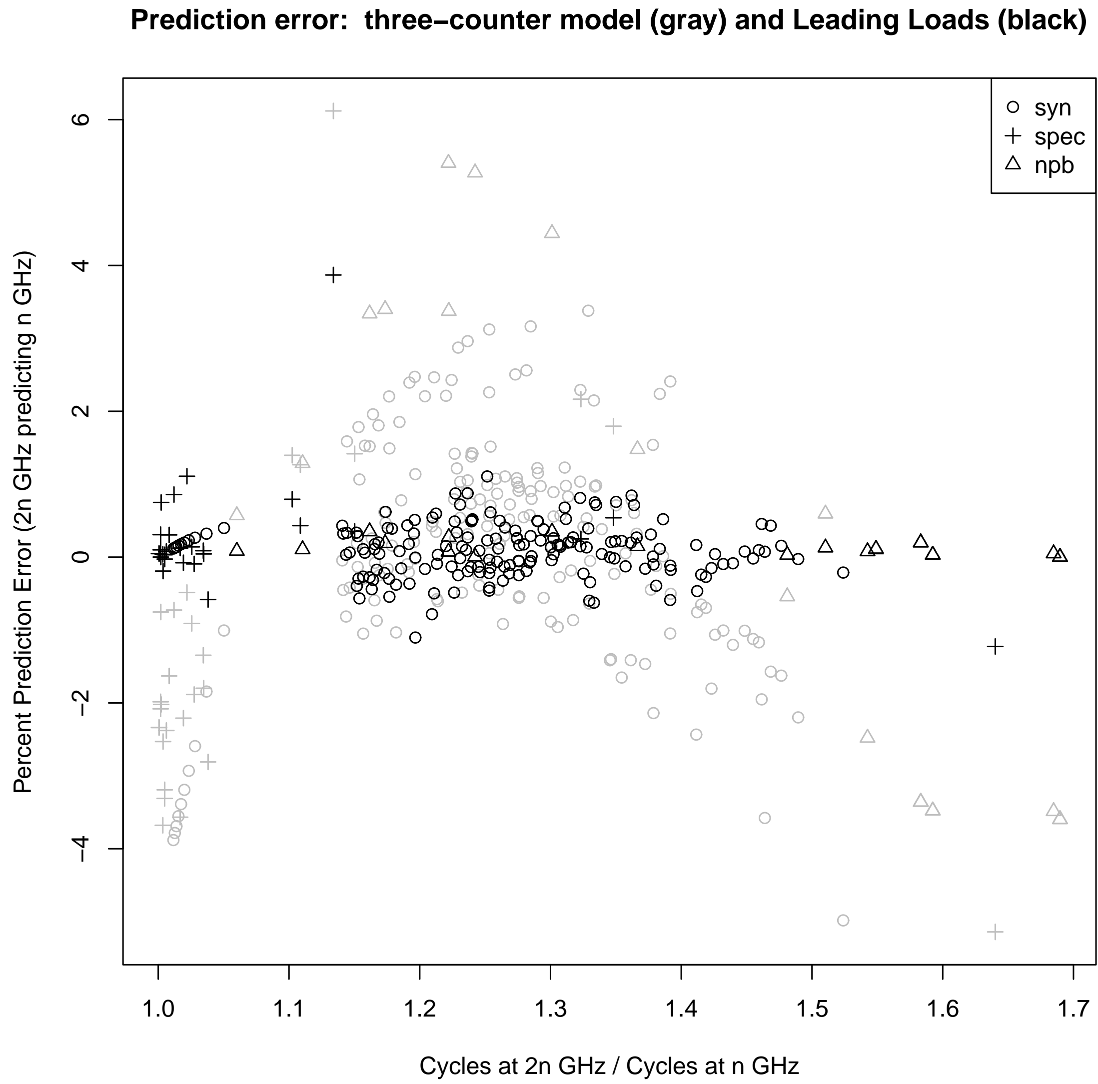


Figure 5.3: Simulated performance of RMI (gray) and Leading Loads (black).

The final third of Table 5.2 shows the effectiveness of leading loads in performance prediction. Error decreased across the SpecCPU suite from 1.232% to 0.445%, and error decreased across the NPB suite from 0.888 to 0.122. We have also broken out the combination of NPB and SpecCPU to show how our technique performs across all third-party benchmarks. Mean error changes from 0.311% to 0.118%. Across all 218 experiments, error decreased from 1.470% to 0.293%. We show individual results from this set in Figure 5.3.

5.6 Related Work

This research is at the intersection of several disparate fields of study. The problem of energy as a design constraint in high-performance computing has led to the use of low-power processors in high-density supercomputers. The two most notable examples of this are Blue Gene/L (Gara et al., 2005) and Roadrunner (Barker et al., 2008). DVFS complements these efforts. This technique has been applied to determine the most efficient per-program allocation of processors and frequencies (Springer et al., 2006) as well as to application-oblivious runtime algorithms (Ge et al., 2007; Freeh et al., 2008b; Rountree et al., 2009). These applications have been limited by the absence of a reliable predictive model of program performance under changes in CPU clock frequency. Work in bounding potential energy savings required complete program execution at every frequency (Rountree et al., 2007), and runtime algorithms commonly assume all slowdown is scaled slowdown until demonstrated otherwise (Rountree et al., 2009).

Several predictive models have been proposed, usually in combination with predicting system energy. Snowdon et al. (Snowdon et al., 2007) begin with a component model that sums coefficients over frequencies as follows:

$$T = \frac{C_{cpu}}{f_{cpu}} + \frac{C_{bus}}{f_{bus}} + \frac{C_{mem}}{f_{mem}} + \cdots$$

Both bus and mem coefficients are derived using linear regressions over hardware performance monitors (HPMs). Their work focused on the PXA255 processor. Because this processor has only fourteen different HPMs, an exhaustive search was possible. Us-

ing several applications from the MiBench suite, the authors determined that *data cache misses* was the most effective single hardware performance monitor (HPM) to predict performance at another frequency, and the best dual-HPM results were achieved by adding *TLB misses*. The next most effective HPMs were *data cache buffer stalls* and *data dependency stall cycles*.

The model used by Lee et al. is simple and effective. The cycles per instruction (CPI) at target frequency f' is predicted using CPI and Memory Instructions at reference frequency f. The HPMs used to approximate these on the Pentium-M are *INSTRUCTIONS_RETIRED* and *BUS_TRANS_MEM*. This work targets both energy savings and execution time at a lower CPU clock frequency. While effective for non-HPC benchmarks, the results in Section 5.3 show that using bus transactions lead to at least a 4.5% error, and combining this with other metrics such as instruction counts and miss counts does not lead to a significant improvement.

The work of Ge et al. (Ge et al., 2007) begins with a complex model that interrelates CPU-bound time (w_{on}), cycles per instruction (CPI_{on}), memory time and latency (w_{mem} and t_{mem}, respectively), IO and idle time (t_{IO} and t_0 respectively) as well as an overlap factor for CPU and memory time (α).

Curtis-Maury's work (Curtis-Maury et al., 2006) develops the most sophisticated model of the several existing ones Their model is the only one that handles the multicore case and focuses on identifying and predicting *useful Instructions Per Cycle* ($uIPC$). The HPMs are selected not by an exhaustive search through all possible combinations, but by using a multivariate analysis to determine the contribution made by each HPM individually. For the single-hyperthreaded case, the HPMs selected from those available on the Pentium 4 are: *cycles active, L2 cache misses, branches retired, TC deliver mode, memory accesses cancelled, double-precision SIMD uops, machine clears, stall cycles* and *instructions retired*. Experiments were conducted on the NAS Parallel benchmark suite, but as in the case with the other models, predicted time at lower frequency was not explicitly evaluated. Li and Martinez (Li and Martìnez, 2006), who use IPC and iterate to converge to the solution, and Contreras and Martonosi (Contreras and Martonosi, 2005), who use regression over HPMs to predict CPU and memory power.

We have taken a different approach. Our interval analysis is inspired by the work of Eyerman et al. (Eyerman et al., 2006, 2007) in designing HPMs for accurately calculating IPC stacks. While both their work and our work study the effects of cache misses on performance, their work focuses on the effects across the processor pipeline for a single frequency. Our analysis focuses only on a single section of the pipeline, abstracts away IPC, and answers the question of which existing HPMs can best predict performance across multiple CPU clock frequencies.

We also use a much different definition of *interval*. Eyerman et al. use the term to define the execution of some significant number of instructions. We adapt the term to be the time between load events, whether issue or completion. This definition allows us to reduce the question of slowdown to the sum of intervals with no slowdown combined with the sum of intervals with scaled slowdown. The end result is similar to Duesterwald's work (Duesterwald et al., 2003) on cross-metric prediction. Predicting change in execution time is equivalent to predicting change in IPC, and we have found a set of metrics that correlate well to this.

5.7 Conclusions and Future Work

To this point, HPM-based models for predicting the effects of changes in CPU clock frequency could be classified as either (1) high-level models with significant error or (2) exhaustive-search models that could not provide an explanation as to why the chosen counters predicted as well or as poorly as they did. In this dissertation, we have provided a detailed architecture model of the relationship between CPU clock frequency, memory usage, and performance. We have validated this model on hardware using realistic scientific benchmarks that stress both the memory and CPU subsystems. We have constructed an architectural model that has provided additional insight for a novel counter, leading loads, and validated this counter under simulation. We have observed improvement across all benchmarks, and this improvement is independent of any issues of training set quality. Finally, our model allows us to analyze these results and provides a framework for research on further improvements.

The first such improvement is in modeling the change in the number of leading loads as a function of the difference between the observed and target CPU clock frequencies. While this change is minor, it does contribute to the remaining error. Given a small amount of additional complexity in the model, we may be able to reduce this impact.

The next improvement will handle the multicore case. All of our benchmarks executed on a single core. We expect that leading loads can be extended to multiple cores, but the memory model of PTLSim is not sufficiently detailed to show the effects of bus contention. Researching this may require moving to a different simulator or contributing improvements to PTLSim.

Whole-program analysis is a useful first step, but ultimately this technique is intended for runtime systems where decisions on changes in CPU clock frequency are made on a millisecond-by-millisecond basis. Reducing the granularity will inevitably increase worst-case error, and the benefits of improved prediction would be made far more useful if the runtime system could assign a confidence level to each prediction.

CHAPTER 6

SUMMARY AND FUTURE WORK

In this chapter we summarize the contribution made by the research presented in this dissertation. We then detail how we could extend this work.

6.1 Summary

Most parallel computing research can be reduced to the question “How do I get the answer to my problem faster?”, which is a special case of the more general question “How do I get the answer to my problem more cheaply?” (given the convertibility of time to units of currency). Current best practice assumes that each component has a fixed cost and a fixed level of performance. In cases where the initial equipment outlay dominates operating cost, this is an acceptable assumption.

The research presented in this dissertation targets the increasingly common case where operating costs are greater than hardware costs. In this case, provisioning a supercomputer for rare peak performance wastes both equipment and operating resources when the system operates at less than peak efficiency. If the performance of components can be adjusted on the fly, and reducing performance reduces operating costs, then it may be cost-effective to tune performance (and cost) to meet the current load on the system.

Specifically, this dissertation has focused on tuning CPU performance at the processor level. Lowering performance (measured in CPU clock frequency) lowers cost (measured in energy consumed by the processor). Doing this indiscriminately results in longer time-to-solution and does not necessarily save any energy. But by identifying the *critical path* of execution, this research has shown that supercomputers can be tuned at runtime to match the performance required by the application with only negligible impact on time-to-solution.

To accomplish this, this dissertation has made the following novel contributions:

1. An execution model that allows identification of the critical path and near-optimal offline scheduling of changes in CPU clock frequency,
2. An architectural model that allows precise and accurate prediction of the effects on performance of changes in CPU clock frequency, and
3. Translating the execution model into a runtime system that preserved most of the energy savings while treating the program as a black box.

6.2 Future work

The strength of this work lies not so much in the amount of energy saved as in the techniques that were developed to save that energy. These techniques can be applied to more general problems as we illustrate below.

6.2.1 Performance Prediction

While much recent research has focus on performance prediction as a function of processor count or CPU clock frequency, most has used a level of granularity of the whole program and has at best only dealt with phases. By predicting individual, per-processor task behavior as a function of processor count, we will be able to preserve the information necessary to identify the critical path of execution, and thus execution time. Integrating this with CPU clock speed and energy awareness will allow users to determine the ideal number of processor to request for a given job as well as allowing administrators to charge the users only for the performance required.

To do this, several program characteristics must be predicted:

1. *Communication Call Path.* Changing the number of nodes can affect the path of program execution, and the effects will not necessarily be the same across all processors. We must not only predict how call paths change, but also predict which added processors will get which predicted call paths.

2. *Communication Call Parameters.* The amount of data to be transfered as well as basic communication patterns can change as a function of the number of processors.

3. *Communication Execution Time.* Similar MPI calls with similar parameters may take longer to execute as the number of processors increases.

4. *Computation Time.* One of the strengths of the work presented in this dissertation has been the absence of any required source code annotation. We wish to carry this strength over to performance prediction and treat computation between MPI calls as a black box.

5. *Per-frequency Energy.* As the number of processors changes, the load placed on the processor, memory and interconnect changes as well.

Given these predictors, we can begin to think about not only how to schedule processor counts and CPU clock frequencies for individual jobs, but also how to design supercomputers to take advantage of this level of optimization.

Capturing the above requires trace analysis, and predicting individual portions of the traces would be a novel contribution. Zhai et al. have created a technique to predict performance at n processors given a communication trace from n processors from a different machine (Zhai et al., 2010). Barnes et al. predict performance at n processors given performance characteristics at $m << n$ processors using regression over computation and communication time as well as input parameters (Barnes et al., 2008, 2010). Midorikawa's work on PEMPIs has some similarities to this approach in that it uses a graph theoretic model to summarize the results of static program analysis. The used this analysis with measured execution characteristics to predict MPI performance at larger node counts (Midorikawa et al., 2004). Prediction of MPI message types and size has been explored by Freitag et al in (Freitag et al., 2003). Sulistio et al provides a taxonomy of simulation approaches to performance prediction (Sulistio et al., 2002).

6.2.2 User and Compiler Guidance

Up to this point, our techniques have assumed that program computation is a black box. While hardware performance monitors can characterize this computation, no other information is required in order to save energy. However, the effectiveness of the runtime system in particular relies on accurate prediction of upcoming communication and computation, with most of the observed delay coming from computation mispredictions. It may be possible for the user, the compiler, or both to provide hints to the runtime system ahead of execution that may be difficult or impossible to predict. These hints could lead to both a further reduction in delay and energy consumption.

Hsu's work in this area focused on source-to-source transformation using SUIF2 to choose regions of code where slowdown would have minimal impact on performance (Hsu and Kremer, 2003). Saputra used loop-level optimization and integer programming to allow the compiler to choose both static and dynamic power levels (Saputra et al., 2002). Valluri et al. focused on compile-time prioritization of instruction scheduling in order to reduce energy consumption (Valluri et al., 2005), and Wu explored the effects of using DVFS in a dynamic compilation environment (Wu et al., 2005b). Finally, Xie presents an compiler-level analytic model of intra-program DVFS effectiveness (Xie et al., 2004).

6.2.3 DVFS, Overclocking and Multicore

Determining the correct level of thread granularity is a nontrivial problem (Curtis-Maury et al., 2006), and it can be further complicated given per-core control over DVFS. If one core can be overclocked but only at the expense of other cores, how does this affect bounding energy savings, runtime realization of those savings, and how processes are assigned to CPUs? This question appears to invalidate the linear programming approach, as CPU clock frequencies are no longer independent. One possible solution is scheduling groups of threads rather than individual threads and allowing the operating system to sort things out at the processor level.

6.2.4 Adoption of Policies

Perhaps the most difficult problem lies with convincing supercomputer users that performance-at-any-cost is not only inefficient, but reduces the amount of work that they can do. Given a supercomputer that has been budgeted to run as fast as possible during its expected lifetime, it would not be surprising to find users reluctant to slow down their programs for the sake of savings on the institution's electric bill. We could charge users per kilowatt hour as well per node hour, but even in this situation the default behavior will probably be running as fast as possible.

To change this attitude, supercomputers might be provisioned with more cores than can be run at the highest CPU clock frequency, but such that all cores can execute at the lowest possible CPU clock frequency. In either case, all provisioned power is utilized, but the power is scheduled to optimize for execution time. If a set of cores is scheduled to execute at a higher CPU clock frequency, this might require turning other cores off completely.

The user now has a much more interesting choice: pay for more nodes or pay for faster nodes, but not both. Springer investigated the boundaries of this tradeoff using the NAS Parallel Benchmark suite on a small number of nodes (Springer et al., 2006). Given this scenario, the user has far more incentive to use targeted DVFS to speed up the nodes on the critical path while keeping the loss of useful nodes to a minimum. This approach should result in a machine that is used far more efficiently, and that is the ultimate goal of this research.

REFERENCES

Alam, S. R., R. F. Barrett, M. R. Fahey, J. A. Kuehn, J. M. Larkin, R. Sankaran, and P. H. Worley (2007). Cray XT4: An Early Evaluation for PetaScale Scientific Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Antony, J., P. P. Janes, and A. P. Rendell (2006). Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In *13th International Conference on High Performance Computing (HiPC)*. Bangalore, India.

Barker, K. J., K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho (2008). Entering the Petaflop Era: The Architecture and Performance of Roadrunner. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Austin, Texas.

Barnes, B., J. Garren, D. K. Lowenthal, J. Reeves, B. de Supinski, artin Schulz, and B. Rountree (2010). Using Focused Regression for Accurate Time-Constrained Scaling of Scientific Applications. In *24th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS)*. Atlanta, Georgia.

Barnes, B., B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, , and M. Schulz (2008). A Regression-Based Approach to Scalability Prediction. In *International Conference on Supercomputing (ICS)*. Island of Kos, Greece.

Bulatov, V., W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis (2004). Scalable Line Dynamics in ParaDiS. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Cameron, K. W., X. Feng, and R. Ge (2005). Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Seattle, Washington.

Contreras, G. and M. Martonosi (2005). Power Prediction for Intel XScale Processors Using Performance Monitoring Unit Events. In *International Symposium on Low Power Electronics and Design (ISLPED)*. San Diego, California.

Curtis-Maury, M., J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos (2006). Online Power-Performance Adaptation of Multithreaded Programs using Hardware Event-Based Prediction. In *International Conference on Supercomputing (ICS)*. Queensland, Australia.

Ding, Y., M. Kandemir, P. Raghavan, and M. Irwin (2008). A Helper Thread Based EDP Reduction Scheme for Adapting Application Execution in CMPs. In *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Miami, Florida.

Dongarra, J., H. Meuer, H. Simon, and E. Strohmaier (2009). Top 500 Supercomputer Sites. `http://www.top500.org/`.

Duesterwald, E., C. Cascaval, and S. Dwarkadas (2003). Characterizing and Predicting Program Behavior and its Variability. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*. New Orleans, Louisiana.

Elnozahy, E., M. Kistler, and R. Rajamony (2002). Energy-Efficient Server Clusters. In *Workshop on Power-Aware Computing Systems (PACS)*. Cambridge, Massachusetts.

Eyerman, S., L. Eeckhout, T. Karkhanis, and J. E. Smith (2006). A Performance Counter Architecture for Computing Accurate CPI Components. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. San Jose, California.

Eyerman, S., L. Eeckhout, T. Karkhanis, and J. E. Smith (2007). A Top-Down Approach to Architecting CPI Component Performance Counters. *IEEE Micro*, **27**(1). Special Issue on Top Picks from 2006 Microarchitecture Conferences.

Femal, M. E. and V. W. Freeh (2005). Boosting Data Center Performance Through Non-Uniform Power Allocation. In *Conference on Autonomic Computing (ICAC)*. Seattle, Washington.

Feng, W.-C. and K. W. Cameron (2010). The Green 500. http://www.green500.org/.

Flautner, K., S. Reinhardt, and T. Mudge (2002). Automatic Performance Setting for Dynamic Voltage Scaling. *Wireless Networks*.

Freeh, V. W., N. Kappiah, D. K. Lowenthal, and T. K. Bletsch (2008a). Just-In-Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs. *Journal of Parallel and Distributed Computing*, **68**(9), pp. 1175–1185.

Freeh, V. W., N. Kappiah, D. K. Lowenthal, and T. K. Bletsch (2008b). Just-In-Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs. *Journal of Parallel and Distributed Computing*, **68**(9), pp. 1175–1185.

Freeh, V. W., D. K. Lowenthal, F. Pan, N. Kappiah, and R. Springer. (2005). Exploring the Energy-Time Tradeoff in MPI Programs on a Power-Scalable Cluster. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Denver, Colorado.

Freitag, F., J. Caubet, M. Farrera, T. Cortes, and J. Labarta (2003). Exploring the Predictability of MPI Messages. In *17th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Nice, France.

Gara, A., M. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. Haring, P. Heidelberger, D. Hoenicke, G. Kopcsay, T. Liebsch, M. Ohmacht, B. Steinmacher-Burow, T. Takken, and P. Vranas (2005). Overview of the Blue Gene/L System Architecture. *IBM Journal of Research and Development*, **49**(2-3), pp. 195–212.

Ge, R. and K. W. Cameron (2007). Power-Aware Speedup. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Long Beach, California.

Ge, R., X. Feng, and K. Cameron (2005). Improvement of Power-Performance Efficiency for High-End Computing. In *Workshop on High-Performance, Power-Aware Computing (HPPAC)*. Denver, Colorado.

Ge, R., X. Feng, W. Feng, and K. W. Cameron (2007). CPU Miser: A performance-Directed, Run-Time System for Power-aware Clusters. In *Proceedings of the 2007 International Conference on Parallel Processing (ICPP)*. Xi'An, China.

Grunwald, D., C. B. M. III, P. Levis, M. Neufeld, and K. I. Farkas (2000). Policies for Dynamic Clock Scheduling. In *Operating Systems Design and Implementation (OSDI)*. San Diego, California.

Haga, W. A. and T. O'keefe (2001). Crashing PERT Networks: A Simulation Approach. In *Fourth International Conference of the Academy of Business and Administrative Science Conference*. Quebec Citcy, Canada.

Hsu, C. and U. Kremer (2003). The Design, Implementation and Evaluation of a Compiler Algorithm for CPU Energy Reduction. In *Programming Language Design and Implementation (PLDI)*. San Diego, California.

Hsu, C.-H. and W.-C. Feng (2005). A Power-Aware Run-Time System for High-Performance Computing. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Seattle, Washington.

Hsu, C.-H., W.-C. Feng, and J. S. Archuleta (2005). Towards Efficient Supercomputing: A Quest for the Right Metric. In *Workshop on High-Performance Power-Aware Computing (HPPAC)*. Denver, Colorado.

Intel (2007). *Intel 64 and IA-32 Architectures Optimization Reference Guide*. 248966-016. Intel Corporation.

Ishihara, T. and H. Yasuura (1998). Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In *International Symposium on Low power Electronics and Design (ISLPED)*. Monterey, California.

Jones, D. (2007). Linux Kernel CPUfreq Subsystem. http://www.kernel.org/ ↪ pub/linux/utils/kernel/cpufreq/cpufreq.html.

Kappiah, N., V. W. Freeh, D. K. Lowenthal, and F. Pan (2005). Exploiting Slack Time in Power-Aware, High-Performance Programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Seattle, Washington.

Lawrence Livermore National Laboratory (2001). The ASCI Purple Benchmarks. http://www.llnl.gov/asci/platforms/purple/rfp/benchmarks.

Lawrence Livermore National Laboratory (2005). The UMT Benchmark Code. http://www.llnl.gov/asci/platforms/purple/rfp/benchmarks/ ↪ limited/umt/.

Lee, S.-J., H.-K. Lee, and P.-C. Yew (2007). Runtime Performance Projection Model for Dynamic Power Management. In *12th Asia-Pacific Conference on Advances in Computer Systems Architecture (ACSAC)*. Seoul, Korea.

Li, J. and J. F. Martìnez (2006). Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors. In *12th International Symposium on High-Performance Computer Architecture (HPCA)*. Austin, Texas.

Li, J., J. F. Martínez, and M. C. Huang (2004). The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors. In *10th International Symposium on High Performance Computer Architecture (HPCA)*. Madrid, Spain.

Lim, M. Y., V. W. Freeh, and D. K. Lowenthal (2006). Adaptive, Transparent Frequency and Voltage Scaling of Communication Phases in MPI Programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Tampa, Florida.

Liu, C., A. Sivasubramaniam, M. Kandemir, and M. J. Irwin (2005). Exploiting Barriers to Optimize Power Consumption of CMPs. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Denver, Colorado.

Lorch, J. R. and A. J. Smith (2001). Improving Dynamic Voltage Scaling Algorithms with PACE. In *International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. Cambridge, Massachusetts.

Makhorin, A. (2005). The GNU Linear Programming Kit (GLPK). http://www.gnu.org/software/glpk/glpk.html.

Midorikawa, E. T., H. M. de Oliveira, and J. M. Laine (2004). PEMPIs: A New Methodology for Modeling and Predition of MPI Programs. In *Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*. Foz do Iguaçu, Brazil.

Mochocki, B., X. S. Hu, and G. Quan (2002). A Realistic Variable Voltage Scheduling Model for Real-Time Applications. In *International Conference on Computer-aided Design (ICCAD)*. San Jose, California.

Mochocki, B., X. S. Hu, and G. Quan (2005). Practical On-line DVS Scheduling for Fixed-Priority Real-Time Systems. In *Real Time and Embedded Technology and Applications Symposium (RTAS)*. San Francisco, California.

Mohr, B. and F. Wolf (2003). KOJAK: A Tool Set for Automatic Performance Analysis of Parallel Programs. In *9th International Euro-Par Conference (EUROPAR)*. Klagenfurt, Austria.

Moncusí, M. A., A. Arenas, and J. Labarta (2003). Energy Aware EDF Scheduling in Distributed Hard Real Time Systems. In *Real-Time Systems Symposium (RTSS)*. Cancun, Mexico.

Mucci, P. and the PAPI Team (2009). *Performance Application Programming Interface*. `http://icl.cs.utk.edu/papi/`.

NASA Advanced Supercomputing Division (2006). *NAS Parallel Benchmark Suite*. `http://www.nas.nasa.gov/Resources/Software/npb.html`. Version 3.3.

Noble, B. D., M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker (1997). Agile Application-Aware Adaptation for Mobility. In *Symposium on Operating Systems and Principles (SOSP)*. Saint Malo, France.

Noeth, M., F. Mueller, M. Schulz, and B. R. de Supinski (2007). Scalable Compression and Replay of Communication Traces in Massively Parallel Environments. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Long Beach, California.

Render, B. and R. M. Stair Jr. (2000). *Quantitative Analysis for Management*. Prentice–Hall, seventh edition.

Rountree, B., D. K. Lowenthal, B. de Supinski, M. Schulz, and V. W. Freeh (2009). Adagio: Making DVS Practial for Complex HPC Applications. In *International Conference on Supercompuing (ICS)*. Yorktown Heights, New York.

Rountree, B., D. K. Lowenthal, S. H. Funk, V. W. Freeh, B. R. de Supinski, and M. Schulz (2007). Bounding Energy Consumption in Large-Scale MPI Programs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Reno, Nevada.

Rădulescu, A., C. Nicolescu, A. J. C. van Gemund, and P. P. Jonker (2001). CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems. In *15th IEEE International Parallel and Distributed Processing Symposium. (IPDPS)*. San Francisco, California.

Saputra, H., M. Kandemir, N. Vijaykrishnan, M. Irwin, J. Hu, C.-H. Hsu, and U. Kremer (2002). Energy-Conscious Compilation Based on Voltage Scaling. In *Languages, Compilers and Tools for Embedded Systems (LCTES)*. Berlin, Germany.

Sharma, V., A. Thomas, T. Abdelzaher, K. Skadron, and Z. Lu (2003). Power-aware QoS Management in Web Servers. In *Real-Time Systems Symposium (RTSS)*. Cancun, Mexico.

Singh, J. P., W. Weber, and A. Gupta (1991). SPLASH: Stanford Parallel Applications for Shared Memory. Technical report, Stanford University.

Snavely, A., L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha (2002). A Framework for Performance Modeling and Prediction. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Baltimore, Maryland.

Snowdon, D. C., S. M. Petters, and G. Heiser (2005). Power Measurement as the Basis for Power Management. In *Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERTA)*. Palma de Mallorca, Spain.

Snowdon, D. C., G. van der Linden, S. M. Petters, and G. Heiser (2007). Accurate Run-Time Prediction of Performance Degradation Under Frequency Scaling. In *Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERTA)*.

Springer, R., D. K. Lowenthal, B. Rountree, and V. W. Freeh (2006). Minimizing Execution Time in MPI Programs on an Energy-Constrained, Power-Scalable Cluster. In *Principles and Practice of Parallel Programming (PPoPP)*. New York City, New York.

Sulistio, A., C. Yeo, and R. Buyya (2002). Simulation of Parallel and Distributed Systems: A Taxonomy and Survey of Tools. *International Journal of Software: Practice and Experience*, pp. 61–80.

Swaminathan, V. and K. Chakrabarty (2000). Real-Time Task Scheduling for Energy-Aware Embedded Systems. In *Real-Time Systems Symposium (RTSS)*. Orlando, Florida.

Swaminathan, V. and K. Chakrabarty (2001). Investigating the Effect of Voltage-Switching on Low-Energy Task Scheduling in Hard Real-Time Systems. In *Asia South Pacific Design Automation Conference (ASP-DAC)*. Yokohama, Japan.

The OpenMPI Team (2010). OpenMPI. `http://www.open-mpi.org/`. Version 1.4.1.

Valluri, M. G., L. K. John, and K. S. McKinley (2005). Low-Power, Low-Complexity Instruction Issue Using Compiler Assistance. In *International Conference on Supercomputing (ICS)*. Cambridge, Massachusettes.

Vetter, J. S. and A. Yoo (2002). An Empirical Performance Evaluation of Scalable Scientific Applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Baltimore, Maryland.

Weiser, M., B. Welch, A. Demers, and S. Shenker (1994). Scheduling for Reduced CPU Energy. In *Operating Systems Design and Implementation (OSDI)*. Monterey, California.

Wu, Q., P. Juang, M. Martonosi, L.-S. Peh, and D. W. Clark (2005a). Formal Control Techniques for Power-Performance Management. *IEEE Micro*, **25**(5), pp. 52–63.

Wu, Q., V. J. Reddi, Y. Wu, J. Lee, D. Connors, D. Brooks, M. Martonosi, , and D. W. Clark (2005b). A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In *Symposium on Microarchitecture (MICRO)*. Barcelona, Spain.

Xie, F., M. Martonosi, and S. Malik (2003). Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits. In *Programming Language Design and Implementation (PLDI)*. San Diego, California.

Xie, F., M. Martonosi, and S. Malik (2004). Intra-program Dynamic Voltage Scaling: Bounding Opportunities with Analytical Modeling. *ACM Transactions on Architecture and Code Optimizations (TACO)*, **1**(3), pp. 323–367.

Xie, F., M. Martonosi, and S. Malik (2005). Bounds on Power Savings Using Runtime Dynamic Voltage/Frequency Scaling: An Exact Algorithm and A Linear-time Heuristic Approximation. In *International Symposium on Low Power Electronics and Design (ISLPED)*. San Diego, California.

Zhai, J., W. Chen, and W. Zheng (2010). PHANTOM: Predicting Performance of Parallel Applications on Large-Scale Parallel Machines Using a Single Node. In *Principles and Practice of Parallel Programming (PPoPP)*. Bangalore, India.

Zhang, Y., X. S. Hu, and D. Z. Chen (2002). Task scheduling voltage selection for energy minimization. In *Proceedings of the 39th annual Design Automation Conference (DAC)*.

Zhu, D., R. Melhem, and B. R. Childers (2003). Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems. *Transactions on Parallel and Distributed Systems*, **14**(7), pp. 686–700.