Theories and Techniques for Efficient High-End Computing

Rong Ge

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Kirk W. Cameron, Chair
Godmar Back
Michael Hsiao
Dennis Kafura
Calvin Ribbens

August 27, 2007
Blacksburg, Virginia

Keywords: High End Computing, Performance Modeling and Analysis, Power-Performance Efficiency, Power and Performance Management

Theories and Techniques for Efficient High-End Computing

RONG GE

ABSTRACT

As high-end computing systems grow tremendously in size and capacity, reducing power and improving efficiency becomes a compelling issue. Today it is common for a supercomputer to consume several megawatts of electric power but deliver only 10–15% of peak system performance for applications. The enormous power consumption costs millions of dollars annually and the resulting heat reduces system reliability.

To address these issues, this thesis provides theories to model performance and power in high-end systems and techniques to optimize power and performance efficiency.

The proposed communication performance models ($log_n P$ and $log_3 P$) analytically describe the communication cost in distributed systems. By explicitly quantifying the cost of middleware communication and data distribution that are ignored by previous models, these models provide more accurate performance prediction and aid more efficient algorithm design.

The power-performance model (power-aware speedup) predicts parallel application performance in power scalable systems. This model quantifies the impact of processor frequency and power on instruction throughput for parallel codes, and forms the theoretical foundation for designing and scheduling a system for a given application to improve efficiency.

We have developed techniques to profile and control power in high-end computing systems. Power profiling techniques directly measure the power consumption of multiple system compo-

nents simultaneously for programs and their functions. The achieved profiles are fine-grained, compared to system or building level profiles from existing methods. Such fine granularity is required to identify the potential opportunities for power reduction, and evaluate the effectiveness of power reduction techniques. Power controlling techniques exploit off-the-shelf power-aware technologies, and integrate performance modeling and workload prediction into power-aware scheduling algorithms. The resulting schedulers can reduce power and energy consumption while meeting user specified performance constraints.

Our theories and techniques have been applied to high-end computing systems. Results show that our theoretical models can improve algorithm performance by up to 59%; our power profiling techniques provide previously unavailable insight to parallel scientific application power consumption; and our power-aware techniques can save up to 36% energy with little performance degradation.

Acknowledgments

First and foremost, I would like to thank my graduate advisor Professor Kirk W. Cameron. His insightful advice, constant support, and never-ending encouragement have been extremely valuable to my graduate study. I am particularly grateful to him for guiding me into the challenging and exciting area of system research, providing me inspiration and flexibility in research directions and ideas.

Special thanks go to my thesis committee members, Professor Michael Hsiao, Professor Dennis Kafura, Professor Godmar Back, and Professor Calvin Ribbens. Each of them devoted significant time and effort to my thesis, and provided me valuable suggestions and insightful comments on research direction and approaches. These suggestions and comments led to substantial improvement in the final product.

The Center for High-End Computing Systems (CHECS) provides an ideal environment for graduate students to grow. In this center I had the opportunity to frequently discuss and communicate, both research and otherwise, with many professors, especially with Professor Wuchun Feng, Professor Dimitris Nikolopoulos, Professor Eli Tilevich, and Professor Ali Butt. I received unselfish help from each individual and learned a great deal from their experiences.

My colleagues in the SCAPE research group have created a positive work environment for me. Matthew Tolentino and Joseph Turner have always provided good advice and been very helpful to

my speaking, writing, and oral presentation when possible. Dong Li and Leon Song brought new blood to the group and their hard work provided me with additional time for my thesis.

At last, I owe a great debt to my family for their deep love and constant support. My parents have been extremely supportive and helpful during my graduate study, especially when they were visiting me. My brothers would be always willing to help us when needed even with their tight schedules. The completion of this dissertation would be impossible without the understanding and encouragement from my husband, Xizhou, all the time and when I was most stressed. I often felt sorry for Kevin and Katherine when I had to work on the thesis during weekends and late nights. Though they won't remember the tough times, they should know they made my life much more enjoyable and satisfying.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past several decades, high-end computing has experienced tremendous growth in peak performance. However, increasing peak performance without improvements in sustained performance and power efficiency is impractical. Today it is not unusual for a large-scale high-end computer system running a typical application to achieve less than 10% peak performance while consuming several megawatts of electric power. This inefficiency costs supercomputer centers millions of dollars annually and the heat produced by the power consumed can reduce system reliability. Power and performance are both critical constraints in high-end computing systems.

Using a strategy that combines experimental profiling and theoretical analysis, this thesis provides theories and techniques to address emerging power and performance constraints in high-end computing. The proposed theories enable evaluation of power and its impact on performance and the techniques lead to power and performance efficiency improvement.

1.1 Current Challenges in High-end Computing

We define a high-end computing (HEC) system as a scientific parallel computing platform such as a supercomputer or a cluster of server-class computing systems. Such systems consist of multiple processors that can be used to execute a program in parallel to reduce program execution time significantly.

High-end computing is indispensable for scientific discovery and technological revolution. The unprecedented computational capability of supercomputers enables scientists to solve complex problems previously deemed intractable. With the aid of supercomputers, scientists are able to make breakthroughs in a wide spectrum of fields such as nanoscience, fusion, climate modeling and astrophysics [78, 118].

Performance has been the primary concern in HEC for decades. To shorten program execution time and increase the resolution of computational simulations, the high-end computing community is continuously pursuing ever faster systems. Thanks to Moore's law [115], the doubling of chip frequency about every 2 years, microprocessor performance has increased significantly. HEC rides the wave in chip design by adopting cutting edge processors and other components. At the same time, HEC exploits coarse-grain parallelism using many processing elements. The number of processors in high-end computing systems has increased from thousands in the 1990's, to hundreds of thousands today [3]. In 2006, the peak performance of the No. 1 supercomputer in the Top500 list [138] reached 280 TFlop/s, 2800X faster than the No. 1 supercomputer in 1993.

The sole focus on performance results in serious problems. First, increasing system size for peak performance results in inefficient system utilization. There is an increasing gap between

"sustained" performance and designed peak performance. Empirical data indicates that the sustained performance achieved by average scientific applications is about 10-15% of the peak performance. Even after years of collaborative work by teams of experts, Gordon Bell prize winning applications [8, 123, 129] can only sustain 35% to 65% of peak performance. The LINPACK benchmark [45], arguably the most scalable and optimized benchmark code, achieves 67% of the designed peak performance on average for the top five machines [46].

Second, high-end computing systems consume huge amounts of electrical power. Most use tens of thousands of cutting edge components that increase their power consumption 2.7 times every two years [17]. The Earth Simulator requires 12 megawatts of power. Future petaflop systems may require 100 megawatts of power [13], one third of the output of a small power plant (300 megawatts). High power consumption leads to intolerable operating cost and failure rate. For example, such a petaflop system may cost $10,000 per hour at $100 per megawatt, excluding the additional cost of dedicated cooling. Considering commodity components fail at an annual rate of 2-3% [122], this system with 12,000 nodes will sustain hardware failure once every twenty-four hours. On current systems, the actual interrupts and failures are even worse. The mean time between interrupts (MTBI) or mean time between failures (MTBF) [142] is 6.5 hours for the LANL ASCI Q supercomputer, and 5.0 hours for the LLNL ASCI white supercomputer [47].

Consequently, power and performance are critical constraints in the design of today's HEC systems. Power consumption must be limited to reduce cost and improve reliability. However, power and performance are inextricably interwoven; reducing power often results in degradation in per-

formance. Achieving high performance while reducing power consumption, and hence improving efficiency, are challenging problems that the HEC community now faces.

1.2 Research Objectives and Approaches

The objective of this thesis is to improve HEC power and performance efficiency by addressing several issues. Research in power-performance efficiency in HEC was nonexistent in the literature prior to 2004 [28]. In this work, we propose four steps encompassing fundamental theories and techniques to 1) accurately predict performance and improve efficiency in systems with advanced hierarchical memories, 2) understand and evaluate power, 3) quantify the impacts of power on performance, and 4) control power and performance for maximum efficiency.

First, we analytically study application performance and seek opportunities for design improvement. Our focus is on memory and communication costs since they dominate the overall execution time for many scientific distributed applications and are often times the cause of performance inefficiency. The challenge is to keep our analytical models simple enough for practical use while incorporating enough detail to guarantee accuracy.

Next, we profile and analyze the power and energy of scientific applications at fine granularity on distributed systems. Power and energy consumption in HEC varies with systems and their components, applications and their functions or phases, and even input data size. Nevertheless, due to the challenges involved in scaling fine-grain power measurement, prior to our work, power studies of server-class systems have been primarily at the cluster level or single node level, and thus lack the details required to identify opportunities for power reduction in HEC computing systems.

Third, we analytically study the impacts of power on application performance and evaluate the tradeoffs between performance and power. We propose models that quantify and predict the impact of processor frequency and power on instruction throughput for parallel codes. The challenge in this work is to create models that are accurate and informative yet practically useful on real systems.

At last, we create power reduction and energy savings techniques that do not sacrifice scientific application performance. We exploit off-the-shelf power-aware technologies, e.g. dynamic voltage and frequency scaling (DVFS) processors, present in server class systems today. Such processors have a set of power/performance modes available, where each mode is identified with a frequency and voltage pair. The mode with higher frequency uses higher voltage and consumes additional power. The challenge is to schedule power modes without reducing performance. Our modeling efforts provide critical insight that we leverage to address this challenge.

1.3 Research Contributions

Overall, this thesis provides theories and techniques that enable efficiency evaluation and improvement of HEC systems.

- We have developed a distributed communication model and deployed it to predict parallel application performance and identify optimal algorithm designs [25, 29]. This model includes the costs of memory middleware in communications. As a result, this model is more accurate in predicting parallel application performance (within 5%), and identifying optimal algorithm designs (up to 59% performance improvement) than existing models.

- We have designed and developed a power/energy profiling and estimation framework, and deployed it on distributed systems [56, 26]. This framework is the first to profile and estimate system power consumption at fine granularity in components as well as functions. Such granularity is required for identifying the potential opportunities for power reduction, and evaluating the effectiveness of power reduction techniques.

- We have developed a performance model that analytically describes the effects of power on application performance and a corresponding methodology for model parameterizations [65]. The accuracy of performance prediction on a cluster using the proposed model and methodology is within 10%. By quickly analyzing the effects of power and parallelism, this model forms the theoretical foundation for designing and controlling power modes to improve distributed system efficiency.

- We have designed and implemented HEC power-aware scheduling techniques on parallel and distributed computing systems for power reduction and energy savings [66, 67, 69]. The basic technique, phase-based power aware scheduling, introduces phase categorization and proposes power/performance modes according to CPU processing needs for each phase. With the aid of performance profiling for phase detection and processing need evaluation, phase-based power aware scheduling provides up to 36% energy savings. Furthermore, we have automated deployment of our techniques in a run-time power management system. In contrast to existing approaches, this run-time power management system exploits all the potential opportunities for energy savings with minimal performance impact; thus promoting the practice of deploying power-aware technology for power/energy reduction in HEC.

1.4 Organization of This Dissertation

The organization for the rest of the dissertation is as follows. Chapter 2 presents the background and reviews related work. First, we review the terms and metrics that are required to describe and evaluate performance, power, and efficiency. Then we discuss the related work in detail.

Chapter 3 describes our analytical approach to improving performance efficiency. We present models of point-to-point communication in distributed systems and their application in algorithm design for performance improvement. Two models are presented in this chapter. The first one is a general model that describes the communication cost at fine granularity. To make this comprehensive model more practical for general use, a simplified model is also presented. The models are validated on real high-end computing systems. We exploit the accuracy of the models to predict application performance and design algorithms for performance improvement.

To reduce power consumption, we must first understand how power is consumed on real systems running parallel scientific codes. Chapter 4 presents an innovative framework that profiles and characterizes power and performance for parallel programs at various levels and correlates usage to application phases. The power profiles obtained provide us with a complete picture of application and system power consumption. We use these profiles to evaluate the energy efficiency of parallel applications. This chapter also presents an accurate power estimation method using performance profiles as a complement for systems where direct measurement is not available.

Chapter 5 focuses on the correlation between performance and power. An analytical performance model for power scalable clusters is presented in this chapter. This model quantifies the performance effects of power, parallelism, and their combination. We show how to derive model

parameters in real systems and apply the model to predict efficiency for various system configurations. The power and performance management in the next two chapters use this model as the theoretical foundation.

Our approach for power reduction is phase-based DVFS power-aware scheduling, described in Chapter 6. We first present phase categorization, and then describe how to identify phases and schedule CPU performance/power modes for applications using performance profiles.

Phase-based power aware scheduling can provide power bounds for a given performance constraint, but our initial attempts required offline scheduling. Run-time schedulers may be more practical given their automation and transparency. Chapter 7 describes a run-time power-aware scheduler that leverages our phase-based approach and its implementation on dual-core dual processor clusters. This run-time scheduler approximates the application execution phases with execution intervals and predictively schedules CPU performance/power modes with integrated performance modeling and prediction. We show this run-time system can automatically and transparently reduce power while meeting performance constraints for scientific applications on real systems.

We conclude this thesis with Chapter 8, where we summarize conclusions and findings, and discuss some directions for future research.

Chapter 2

Background and Literature Survey

We begin with a review of terms and metrics that are required to evaluate power, performance, and efficiency. In section 2.2 we focus our detailed discussion on related work.

2.1 Measures of Performance, Power, and Energy

2.1.1 Performance

Typically, system performance is evaluated using the execution time T (or delay D for the same meaning) for one or a set of representative applications [2]. The execution time for an application is affected by the CPU speed, memory hierarchy and application execution pattern.

The sequential execution time $T(1)$ for a program on a single processor consists of two parts: the time that the processor is busy executing instructions T_{comp}, and the time that the processor waits for data from the local memory system $T_{memoryaccess}$ [**?**], i.e.,

$$T(1) = T_{comp}(1) + T_{memoryaccess}(1)\,. \tag{2.1}$$

Memory access is expensive: the latency for a single memory access can be the same as the time for the CPU to execute hundreds of instructions. The term $T_{memoryaccess}$ can consume up to 50% of execution time even for an application 99% of whose data accesses hit in the closest caches.

The parallel execution time on n processors $T(n)$ is affected by additional components such as parallel overhead. *Synchronization time*, $T_{sync}(n)$, is due to load imbalance and serialization. *Communication time*, $T_{comm}(n)$, occurs when the processor is stalled waiting for a data transfer from or to a remote processing node. The time that the processor is busy executing extra work, $T_{extrawork}(n)$, is due to overhead from parallel data decomposition and task assignment. The parallel execution time can be expressed as

$$T(n) = T_{comp}(n) + T_{memoryaccess}(n) + T_{sync}(n) + T_{comm}(n) + T_{extrawork}(n)\,. \tag{2.2}$$

Parallel overhead $(T_{sync}(n) + T_{comm}(n) + T_{extrawork}(n))$ is quite expensive. For example, the communication time for a single piece of data can be as large as the computation time for thousands of instructions. Moreover, parallel overhead tends to increase with the number of processing nodes.

The ratio of sequential execution time to parallel execution time on n processors is the parallel speedup, i.e.

$$speedup(n) = \frac{T(1)}{T(n)}\,. \tag{2.3}$$

Ideally, the speedup on n processors is equal to n for a fixed-size problem, such that the speedup grows linearly with the number of processors. However, the achieved speedup for real applications is typically sub-linear due to parallel overhead.

2.1.2 Power

The power consumption of CMOS logic circuits [110] such as processor and cache logic is approximated by

$$P = ACV^2 f + P_{short} + P_{leak}\,. \tag{2.4}$$

The power consumption of CMOS logic consists of three components: dynamic power $P_d = ACV^2 f$ which is caused by signal line switching; short circuit power P_{short} which is caused by through-type current within the cell; and leak power P_{leak} which is caused by leakage current. Here f is the operating frequency, A is the activity of the gates in the system, C is the total capacitance seen by the gate outputs, and V is the supply voltage.

For many years, dynamic power (P_d) has dominated total power (P), accounting for 70% or more, P_{short} typically accounts for 10-30% of (P) and P_{leak} accounts for about 1% [92] of (P). Therefore, CMOS circuit power consumption is approximately proportional to the operating frequency and the square of supply voltage when ignoring the effects of short circuit power and leak power.

With continuing decreases in feature size and increases in processor frequency, leakage power (P_{leak}) is growing in importance. Thus, reducing leakage power is an important area of research being addressed primarily by other communities such as electrical engineering [75, 35, 117]. In contrast, our work focuses on using software to reduce dynamic power (P_d) since it still dominates total power (P) and has throughout the work described herein.

2.1.3 Energy

Whereas power (P) describes consumption at a discrete point in time, energy (E) specifies the number of joules used for a time interval (t_1,t_2) as the product of the average power and the delay ($D = t_2 - t_1$):

$$E = \int_{t_1}^{t_2} P dt = P_{avg} \times (t_2 - t_1) = P_{avg} \times D\,. \tag{2.5}$$

Equation 2.5 specifies the relation between power, delay, and energy. To save energy, we need to reduce the delay, the average power, or both. Performance improvements such as code transformation, memory remapping, and communication optimization can decrease the delay [119, 29]. Clever system scheduling among various power-performance modes can effectively reduce average power without affecting delay.

In the context of parallel processing, by increasing the number of processors, we can speedup the application but also increase the total power consumption. Depending on the parallel scalability of the application, the energy consumed by an application may be constant, grow slowly or grow very quickly with the number of processors.

2.1.4 Power-Performance Efficiency

As discussed earlier, power and performance often conflict with one another. Some relation between power and performance is needed to define optimal power-performance efficiency in this context. To this end, certain product forms of delay D (i.e. execution time T) and power P are used to quantify power-performance efficiency. Smaller products represent better efficiency. Commonly used metrics include PDP (the $P \cdot D$ product, i.e. energy E), PD2P (the $P \cdot D^2$ product), and

PD3P (the $P \cdot D^3$ product). These metrics can also be represented in the forms of energy and delay products such as EDP and ED2P.

These metrics put different emphasis on power and performance, and are appropriate for evaluating power-performance efficiency for different systems. PDP or energy is appropriate for low power portable systems where battery life is the major concern. PD2P [21] metrics emphasize both performance and power; this metric is appropriate for systems which need to save energy with some allowable performance loss. PD3P [15] emphasizes performance; this metric is appropriate for high-end systems where performance is the major concern but energy conservation is desirable.

2.2 Related Work

Related work can be organized in four groups: communication performance models, power profiling and estimation techniques, scalability models, and power reduction and energy savings techniques. We introduce each group followed by its limitations and motivation for our work. Each group corresponds to one of the major challenges and contributions presented in Chapter 1.

2.2.1 Communication Performance Models

Research on communication performance models in high-end computing has been active for several decades. These models use hardware and software parameters to describe communication costs analytically. Representative analytical models which are applicable to a wide range of machines include the PRAM model [61], BSP model [141], LogP model [40], and memory logP model

[30]. These models are useful for predicting performance, guiding algorithm design, and analyzing system procurement and installation.

PRAM model and variants

The PRAM model [61] is widely used for parallel algorithm design. A parallel random access machine (PRAM) consists of a set of processors that communicate through a global random access memory. PRAM is abstract and simple; it helps designers to focus on the structure of computational problems. Nevertheless, PRAM is over-idealized in that it assumes any communication can be finished in unit time. It does not capture the costs of contention, latency, and synchronization. To make PRAM more practical, several variants [6, 5, 70] are proposed to overcome the idealized assumptions regarding memory access or communication by adding parameters to model contention, latency, and asynchrony.

Several PRAM variants address memory contention that occurs when more than one processor accesses the same memory location at the same time. Exclusive read, exclusive write (EREW) PRAM disallows simultaneous reading and writing. Concurrent read, exclusive write (CREW) PRAM allows simultaneous reading but disallows simultaneous writing. Concurrent read, concurrent write (CRCW) PRAM allows both simultaneous reading and writing. There are two submodels of CRCW PRAM: priority and arbitrary [25]. On a priority CRCW PRAM, the processor with the highest priority number succeeds and has its value stored. In contrast, on an arbitrary CRCW PRAM, it is unspecified which one of the processors that simultaneously writes to the same shared memory cell will be successful. Gibbons [70] argues that neither the exclusive nor the concurrent rules accurately reflects the contention capabilities of most commercial and research

machines, and proposes the queue rule and QRQW PRAM for the contention. On a QRQW PRAM with queue rule, each location can be read and written by any number of processors in each step. Concurrent reads or writes to a location are serviced one-at-a-time.

Other variants [4, 5, 121] address the communication delay between processors on PRAM machines. Papadimitriou et al [121] claimed communication delay exists. Aggarwal et al [5] exploited spatial locality to reduce communication delay. They proposed the Block PRAM (BPRAM) model. On a BPRAM machine, a processor may access a word from its local memory in unit time. It may also read/write a block of contiguous locations from/to global memory. The cost of such an operation is $(l + b)$ where l is the startup time or the latency, and b is the length of the block.

APRAM [39, 70] addresses the effects of synchronization on PRAM machines. It incorporates asynchrony into the PRAM model and explicitly charges for synchronization. On an APRAM machine, there are four types of instructions: global reads, local operations, global writes, and synchronization steps. A synchronization step among a set S of processors is a logical point in a computation where each processor in S waits for all the processors in S to arrive before continuing in its local program.

BSP model and variants

The bulk-synchronous parallel (BSP) model [141] is a bridging model between hardware and programming for parallel computation. A BSP machine consists of a set of processors with local memory, a router that delivers point-to-point messages, and facilities for barrier synchronization. The execution of a BSP algorithm is a sequence of supersteps. Each superstep consists of

local computation, communication, and is concluded by the barrier synchronization. The hardware parameters on a BSP machine are: the number of available processors p, the bandwidth inefficiency or gap g, and the time the machine needs for the barrier synchronization L.

There are several BSP variants. BSP* [14] encourages block-wise communication by introducing an additional machine parameter, the critical block size B. In this extension, the communication cost of a single packet is $max(gh\lceil\frac{s}{B}\rceil, L)$, where s denotes the number of bytes of the packet. The Decomposable BSP model (D-BSP) [139] divides the BSP machine into several partitions or submachines: each acts like an independent BSP machine of smaller size. The hardware parameters for a submachine are BSP* parameters. D-BSP model has two advantages: 1) it can exploit locality, 2) each submachine can execute different algorithms independently. E-BSP [96] deals with unbalanced communication patterns in which the processors send or receive different amounts of data.

LogP model and variants

LogP [40] is another widely used bridging model. It uses four parameters to characterize the performance of the interconnection network: L, the upper bound on the latency or delay for a word (or small number of words) communication; o, the overhead the processor needs to prepare for the communication; g, the gap or the minimum time interval between consecutive message communication at a processor; and P, the number of processor/memory modules. Compared to the BSP model, LogP additionally incorporates asynchronous behavior, communication overhead and latency.

Some LogP variants are proposed to support additional important characteristics by adding parameters to LogP. Among these, LogGP [7] introduces parameter G, the time per byte to support long messages. Sending a k byte message from one processor to another takes $(o + (k - 1)max(g, o) + L + o)$ cycles under the LogGP model, and $(k - 1)(o + L + o)$ cycles under the LogP model. LogGPS [91] extends LogGP to capture the synchronization needed for long messages communications by some middleware. LoPC [62] and LoGPC [116] add a parameter C to model resource and network contention.

Memory logP model

Memory logP [30] models point-to-point memory communication in shared memory platforms. It characterizes memory communication cost using four parameters: effective latency l, overhead o, gap or the minimum time interval between contiguous message reception g, and number of processor/memory modules P. Overhead o is the communication cost for messages of size s with consecutive distribution, while effective latency l is the extra communication cost for message of size s due to non-continuous distribution. When the performance gap between memory and processor is large, effective latency l often exceeds overhead o.

Motivation for our work

The existing models (excluding memory logP) either describe the communication cost using hardware parameters or constant values. These models ignore the increasing effects of middleware on communication cost.

However, there are compelling reasons to incorporate middleware costs into models of distributed communication. First, cost occurring in middleware due to data distribution can dominate

communication cost. The hardware-parameterized models can only capture the effects of message size other than their distribution. Second, more accurate models of communication encourage efficient algorithm design. Existing hardware-parameterized models of communication ignore middleware as a potential performance bottleneck. This implies algorithms designed using these models will not be optimal. For example, an algorithm designed under LogP has no incentive to reduce the number of strided communications. Additionally, more accurate cost models encourage overlap in communications.

Since existing parallel programs often do not exhibit good performance on distributed systems, a large class of scientific applications (e.g. simulations) stands to benefit from the development of predictive models of distributed communication that incorporate system software characteristics and encourage reductions in middleware communication cost.

In this work, we develop a communication model [25, 29] that includes the effects of middleware on communication cost. This model uses software parameters such as message size and their distribution to reflect the cost of data movement, which is ignored by hardware parameterized models. Our model enables improvements in algorithm design and analysis which in turn lead to more efficient parallel applications and systems.

2.2.2 Power Profiling and Estimation

Three primary approaches: simulators, direct measurements and performance counter based models, are used to profile power of systems and components.

Simulator-based power estimation

We begin our discussion with architecture level simulators and categorize them across system components, i.e. microprocessor and memory, disk and network. These power simulators are largely built upon or used in conjunction with performance simulators that provide resource usage counts, and estimate energy consumption using power models for the resources.

Microprocessor power simulators. Wattch [20] is a microprocessor power simulator interfaced with a performance simulator, SimpleScalar[21]. Wattch models power consumption using an analytical formula $P_d = ACV_{dd}^2 f$ for CMOS chips, where C is the load capacitance, V_{dd} is the supply voltage, f is the clock frequency, and A is the activity factor between 0 and 1. Parameters V_{dd}, f and A are identified using empirical data. The load capacitance C is estimated using the circuit and the transistor sizes in four categories: array structure (i.e. caches and register files), CAM structures (e.g. TLBs), complex logic blocks, and clocking. When the application is simulated on SimpleScalar, the cycle-accurate hardware access counts are used as input to the power models to estimate energy consumption.

SimplePower [144, 150] is another microprocessor power simulator built upon SimpleScalar. It estimates both microprocessor and memory power consumption. Unlike Wattch which estimates circuit and transistor capacitance using their sizes, SimplePower uses a capacitance lookup table indexed by input vector transition. SimplePower differs from Wattch in two ways. First, it integrates SimpleScalar into its software, rather than interfaces with SimpleScalar. Second, it uses a capacitance lookup table instead of an empirical estimation of capacitance. The capacitance lookup table could lead to better accuracy in power simulation. However, it comes at the expense of flex-

ibility as any change in circuit and transistors would require changes in the capacitance lookup table.

TEM2P2EST [42] and the Cai-Lim model [23] are similar power models and both are built upon SimpleScalar. These two models add complexity to the power model and functional unit classification as done in Wattch. First, these two models can toggle between an empirical mode and an analytical mode. Second, both approaches model dynamic and leakage power. Third, both models include a temperature model based on power dissipation.

PowerTimer [18] is a power-performance modeling toolkit. It runs a simplified version of Wattch for PowerPC processors and includes a web-based interface to characterize the tradeoff between performance and power.

Network power simulators. Orion [145] is an interconnection network power simulator at the architectural-level based on the performance simulator LSE [140]. It models power analytically for CMOS chips using architectural-level parameters, thus reducing simulation time compared to circuit-level simulators while providing reasonable accuracy.

System power simulators. Softwatt [73] is a complete system power simulator that models the microprocessor, memory systems and disks based on SimOS [127]. Softwatt calculates the power values for microprocessor and memory systems using analytical power models and the simulation data from log-files. The disk energy consumption is measured during simulation based on assumptions that full power is consumed if any of the ports of a unit is accessed, otherwise it assumes no power is consumed.

Powerscope [60] is a tool for profiling the energy usage of mobile applications. Powerscope consists of three components: the system monitor samples system activity by periodically recording the program counter (PC) and process identifier (PID) of the currently executing process; the energy monitor collects and stores current samples; and the energy analyzer maps the energy to specific processes and procedures.

Direct measurements

There are two basic approaches to measuring processor power directly. The first approach [15, 95] inserts a precision resistor into the power supply line and use a multi-meter to measure its voltage drop. The power dissipation by the processor is the product of power supply voltage and current flow, which is equal to the voltage drop over the resistor divided by its resistance. The second approach [93, 136] uses an ammeter to measure the current flow of the power supply line directly. This approach is less intrusive as it does not need to cut wires in the circuits.

Tiwari et al [136] used ammeters to measure the current drawn by a processor while running programs on an embedded system and developed a power model to estimate power cost. Isci et al [93] use ammeters to measure the power for P4 processors to derive their event-count based power model. Bellosa et al [15] derive CPU power by measuring current on a precision resistor inserted between the power line and supply for a Pentium II CPU; they use this power to validate their event-count based power model and save energy. Joseph et al [95] use precision resistor to measure power for a Pentium Pro processor. These approaches can be extended to measure single processor system power. Flinn et al [59] used a multimeter to sample the current being drawn by a laptop from its external power source.

Event-based modeling

Most high-end CPUs have a set of hardware counters to count performance events such as cache hit/miss, memory load, etc. If power is mainly dissipated by these performance events, power can be estimated based on performance counters. Isci et al [93] developed a runtime power monitoring model which correlates performance event counts with CPU subunit power dissipation on real machines. CASTLE [95] did similar work on performance simulators (SimpleScalar) instead of real machines. Joule Watcher [15] also correlates power with performance events, the difference is that it measures the energy consumption for a single event such as a floating point operation, L2 cache access, and uses this energy consumption for energy-aware scheduling.

Temperature simulation and emulation, measurement, and modeling

Thermal studies at VLIW and RTL levels focus on simulation and modeling. 3-D thermal-ADI [145] simulates the temperature in a 3-D environment based on the alternating direction implicit method. Other work [103] presents a multigrid iterative approach for full-chip thermal modeling and analysis. Hotspot models temperature based on a stacked-layer packaging scheme in modern very large-scale integration systems [88]. The direct measurement efforts include infrared thermal measurement using infrared cameras [114].

At the architecture level, Donald et al [44] uses HotSpot [88] to study the thermal-aware design issues for simultaneous multi-threading (SMT) and chip multiprocessing (CMP) architectures for superscalar architectures. Li et al [104] uses HotSpot [88] and performance/power simulators to evaluate the performance, power, and thermal considerations for SMT and CMP

for a POWER4/POWER5-like core. Thermal Herding [125] uses micro-architecture techniques to control hotspots in high-performance 3D-integrated processors.

At the system level, Mercury and Freon [77] is proposed to emulate and manage temperature. Mercury is a software suit that emulates temperatures based on layout, hardware, and component-utilization data, while Freon is a system for server farms that manages thermal emergencies by workload scheduling or turning on/off nodes. Tempest [24] provides techniques to profile temperature for functions and programs on server systems. ThermoStat [36] is a 3-dimensional computational fluid dynamics based thermal modeling tool for rack-mounted server systems.

Motivation for our work

Previous studies of power consumption on a high performance distributed system focus on building-wide power usage [100]. Previous single node power measurements focus on isolating the power consumption of the processor only [15, 95, 93, 136]. Such studies neither separate individual nodes or components, nor reveal the power profiling for functions. Other attempts to estimate power consumption for systems such as ASCI Terascale facilities use rule-of-thumb estimates (e.g. 20% peak power)[10]. Based on past experience, this approach could be completely inaccurate for future systems as power usage increases exponentially for some components.

There are two compelling reasons for in-depth study at component and function-level granularity of the power usage of distributed applications. First, there is need for a systematic approach to quantify the energy cost of typical high-performance systems. Such cost estimates could be used to accurately estimate future machine operation costs for common application types. Second, a component-level study may reveal opportunities for power and energy savings. For example,

component-level profiles could suggest schedules for powering down equipment not being used over time.

One of my research objectives is to provide a power/energy profiling framework for distributed systems at the component level and function granularity. Previous approaches focus on AC power measurements that include power consumption of all components, the power supply, fans, etc. Previous approaches for DC power only provide data for the CPU. We simultaneously measure the power consumption at multiple points and components, as well as provide a user API for mapping the power to functions. In addition, we build an empirical system power model that can estimate system component power where direct measurement is not possible.

2.2.3 Performance Scalability

Several speedup models are widely accepted for evaluating application performance scalabilities of high-performance parallel applications. These models have been extremely important in the extant literature and are typically used to bound the theoretical performance limits of applications.

Amdahl's law (fixed-problem speedup). Amdahl's law [9] refers to a scalability study of the number of processing nodes versus speedup assuming a fixed workload. Amdahl's law assumes the workload is fixed-size and only composed of two parts: a serial part which can only be executed on one processor, and a perfectly parallel part which can be executed on any number of processors. When the workload is executed on n processors, the speedup is $1/(s+p/n)$, where s and $p = 1-s$ are the amount of time spent on serial parts and parallel parts under sequential execution. The maximum speedup is $1/s$ for an infinite number of processors. Amdahl's law is also known as

strong-scaling speedup, and it reveals the relation between the number of processors and execution time. It is used to evaluate techniques that reduce execution time.

Gustafson's law (fixed-time speedup). Gustsfson's law [74] refers to a scalability study of the number of processing nodes versus speedup with a variable workload size. Gustafson argues that it is more realistic to assume run time, not the problem size, is constant given a more powerful system. Under fixed-time execution, the problem size can scale up to n processors, and the speedup obtained is $n-s(n-1)$. Gustafson's law is also known as weak-scaling speedup. The goal of weak-scaling is to reveal the relationship between workload, the number of processors, and execution time.

Sun-Ni's law (memory-bound speedup). Sun-Ni's law [135] refers to a scalability study of the number of processing nodes versus speedup with a variable workload size constrained by the capacity of main memory. Sun et al [135] argued that the problem size could be scaled further in large memory systems to gain more speedup and improve accuracy.

Isoefficiency. Gustafson's law and Sun-Ni's law investigate how the problem size scales under the constraints of execution time or main memory, isoefficiency by Grama et al [71] studies how the problem size scales to maintain the same performance efficiency when more processors are available, where efficiency is defined as the ratio of speedup to the number of processors used.

Motivation for our work

While existing scalability models are useful for quantifying the performance improvement of a parallel application in conventional high-end computers, they are not applicable to emergent power-aware high-end computers for two reasons.

First, existing scalability models are not able to capture the performance effect of power consumption under different power modes. Second, traditional speedup performance metrics follow the "performance at any cost" logic since they do not consider the impact of performance improvements on system energy efficiency. When using existing scalability models to evaluate two system configurations with similar performance but considerably different power consumption, the configuration with the higher speedup is optimal even though it may require twice the power and energy. Such configuration is not the best in power-performance efficiency.

Any power-aware model of parallel performance must accurately quantify the amount of execution time affected by power modes. For example, parallel overhead influences the percentage of total execution time affected by parallelism. Similarly, parallel overhead affects the percentage of total execution time affected by processor frequency. Furthermore, the percentage of total execution time due to parallel overhead changes with the application and the snumber of nodes. Thus the execution time effects of frequency and parallelism are interdependent. As we will see in the next section, this complicates modeling the power-performance of power-aware clusters.

The scalability model presented in this work aims to characterize power-aware high-end computers. This model accurately quantifies the combined effects of power and system size on application performance, and predicts the most efficient system configuration.

2.2.4 Power Reduction and Energy Conservation

Power reduction and energy conservation have recently gained traction in HEC community. Two kinds of approaches have been investigated for this purpose: low-power approach and power-aware approach.

Low-Power Approach

To address operating cost and reliability concerns, large-scale systems are being built with low power components. For example, Green Destiny [55] is built with Transmeta Crusoe mobile processors, Argus [57] with PowerPC 750CXe embedded processors, and IBM BlueGene/L [3] with PowerPC 440 embedded processors. All of these processors consume much less power than high performance processors for servers and workstations; many do not even use active cooling [55].

Low-power processors are designed for power efficiency. As a result, the low power approach requires changes in architectural design to improve performance. For example, Green Destiny relies on the development of the Transmeta Crusoe processor, and BlueGene/L uses a version of the embedded PowerPC chip modified with additional floating point support. The resulting components are no longer strictly commodity parts, and the high-end computing systems are no longer strictly composed of commodity parts -- making this approach very expensive to sustain. Such difficulty is well reflected by the history of Transmeta[1].

[1]Transmeta had hoped to design low-power processors that are comparable in performance with X86 Intel or AMD server processors, but failed in both power budget and performance projection. Although Transmeta processor performance in later versions improved with more complexity in hardware, power consumption increased accordingly. After several years in the market, Transmeta announced that it would no longer develop and sell hardware, but would focus on the development and licensing of intellectual property.

Power-Aware Approach

Power-aware components have a set of power/performance modes available, where the mode with higher performance normally consumes more power. Power-aware computing uses power-aware components to build systems and dynamically switches components among different performance/power modes according to processing needs to reduce power consumption.

The power-aware approach originated from energy-constrained, low power, real time and mobile systems [53, 109, 110, 147, 79, 33, 94, 107, 58]. This research exploits multiple performance/power modes, or power-aware techniques available on components such as processor [53, 109, 110, 147], memory [51, 52], disk [31, 49, 48, 102], and communication links [32, 41, 90, 130, 133]. When components are idle or not fully utilized, they are turned to lower power modes or even turned off to save energy. The challenge for a power-aware approach is to sustain application performance and meet task deadlines as 1) switching components between modes introduces overhead; and 2) lower power modes usually reduce performance.

More recently, as power has become a critical issue, the power-aware approach has been used in Internet processors and servers, and data centers. The efforts can be largely divided into four categories. The first category studies the power consumed by processors in server farms. Work [16, 50, 54] found dynamic voltage and frequency scaling (DVFS) for processors is effective for saving energy. The second category focuses on disk subsystems. Carrera et al [31] found multi-speed disks could save energy up to 23% for network servers. Zhu et al [151, 152] combines several techniques of multi-speed disks, data migration, and performance boost to reduce energy consumption while meeting performance goals. Son et al [131] presents a proactive approach that

pre-activates a disk from low-power mode to avoid potential performance impacts. The third category exploits the opportunities in main memory and caches. Representative work includes DMA-aware memory energy management [120] and power-aware page allocation [101]. The fourth category is the studies in networking protocols and devices [111, 63, 97, 126].

The power-aware approach has recently been exploited in the HEC community as power-aware technology migrates to commodity high performance server products. Most work focuses on reducing the power consumption of the processor using DVFS technology and developing techniques to exploit low CPU utilization during memory access [149, 80], communication [64, 68], load imbalance [34], or their combination [69, 83] for a given workload. Some use DVFS technology to reduce power as well as manage thermal emergencies [24, 76, 99]. Others attempt to reduce the power consumption of components [137, 112, 43] or thermal management [106] in memory subsystem.

HEC DVFS Approach

We focus the remainder of this discussion on power reduction and energy savings using DVFS due to its close relation to our work.

Off-line, trace-based DVFS techniques were initially proposed [27, 68, 84] to reduce processor power consumption in HEC. The basic off-line approach involves (1) source code instrumentation for performance profiling, (2) execution with profiling, (3) determination of appropriate processor frequencies for each phase, and (4) source code instrumentation for DVFS scheduling. Ge et al. [27, 68] use PMPI to profile MPI communications. Hsu et al. [85, 86, **?**] use compiler instru-

mentation to profile and insert DVFS scheduling functions for sequential codes. Freeh et al. [64] use PMPI to time MPI calls and then insert DVFS scheduling calls based on duration. Off-line approaches typically require manual intervention to determine the target frequency for inefficient phases.

Run-time DVFS scheduling techniques are automated and transparent to end users. Hsu and Feng [82] proposed the β-adaption algorithm to automatically adapt the voltage and frequency for energy savings at run-time. Lim et al. [105] implemented a run-time scheduler that intercepts MPI calls to identify communication-bound phases in MPI programs. Wu et al. [148] made use of a dynamic compiler to monitor memory-bound regions in sequential codes for power reduction. In addition, CPUSPEED[2] provides an interval-based DVFS scheduler for Linux distributions. CPUSPEED adjusts CPU power/performance modes based on the CPU utilization during the past interval.

Motivation for our work

We believe that the power-aware approach is appropriate for power reduction and energy savings in high performance computing. Processing needs of high performance applications are not constant, but vary with execution phases over time. By adaptively switching processor to performance/power modes that meet processing needs with minimal power consumption, we can reduce power while maintaining performance.

The key to any successful power-aware approach in HEC then lies in the ability to understand the impact of power and predict parallel application performance. Predicting parallel perfor-

[2]http://carlthompson.net/software/cpuspeed

mance accurately is an open problem. To ensure HEC systems can leverage power-aware technologies, we need a theory that accurately quantifies parallel application performance under various power/performance modes of DVFS processors. Otherwise, we can not satisfy the tight performance constraints specified by end users of high-end computing systems. Additionally, to increase adoption by the HEC community, we need techniques that automatically and transparently maximize power reductions and energy savings. In high-end computing, neither user intervention or marginal power reductions and energy savings are practical or attractive.

The methodology for embedded and real time systems works under different context or serves different purposes, and therefore can not simply be ported to high performance computing. For example, embedded systems do not typically communicate synchronously as is common in parallel scientific applications. In addition, embedded systems are rarely equipped with multiprocessors or multi-cores. The techniques optimal for a single-processor system might not be optimal for multiprocessor systems. Furthermore, power-aware embedded or mobile systems techniques focus on prolonging the battery life, while high performance computing considers performance as the primary constraint.

The techniques for interactive workloads (e.g. web services) in data centers cannot simply be ported to HEC. Such techniques react to and schedule independent, i.e., embarrassingly parallel, process workloads that vary in time. Scientific applications are non-interactive, often dependent processes that vary according to algorithm.

Our work [28] was the first published to specifically propose power-aware techniques that are practical for HEC systems. Later, we studied the feasibility of power-aware high performance

computing and developed phase based scheduling and offline phase detection [67]. Our latest work includes an analytical performance model for power-aware clusters [65] and techniques [69] that automatically and transparently exploit all the opportunities in parallel applications and systems to reduce power and save energy under performance constraints.

Chapter 3

Models of Point-to-Point Communication in Distributed Systems

In this chapter we present log_nP, a general software-parameterized model of point-to-point communication in distributed systems. Realizing the growing gap between memory and CPU performance combined with the trend toward large-scale, clustered share memory platforms, the log_nP model considers the impact of memory and communication middleware on distributed systems and achieves more accurate performance prediction than existing communication models. We describe how to use this model on real systems and illustrate the benefits of this model on efficient parallel algorithm design.

3.1 Introduction

Scientific distributed applications typically involves frequent data transfers among a group of dependent processors. Though there exist many patterns, in general all data transfers can be imple-

mented with a series of point-to-point communications, which require moving data from the local memory tied to a source process to the local memory tied to a target process. As the cost of communication often dominates the overall execution time for many scientific distributed applications, optimizing the point-to-point communication will increase the applications' performance and their energy efficiencies.

For scientific distributed applications, algorithm designers and application developers usually use explicit communications that specify the source and target processes, memory locations, and amount (or types) of data being transferred. These explicit communications are abstractions for a series of implicit communications that are hidden from the programmers by the underlying hardware or communication middleware. For example, in a typical load-store architecture for a loop assigning $A[i] = B[i]$ for $0 < i < n - 1$, a series of block transfers between memory hierarchy levels brings data from main memory to cache to registers to complete this task. The user explicitly specifies source and target locations, but the assignment implicitly causes movement of data from memory to registers and back to memory. Implicit communications are the transmissions that occur "behind the scenes" to complete an explicit communication. This requires hardware (e.g. data replication from memory to cache) and system software support (e.g. demand paging) when the data does not reside in memory. The details of the implicit communication are hidden to ease programming efforts.

In this chapter, we focus our discussion on message passing (e.g. MPI), a common computing model in large-scale clusters. For message passing in a distributed system, explicit communications like sends and receives are accomplished using implicit communication mechanisms provided in

communication middleware such as system software and communication libraries. For example, an MPI_Send() of a strided message describes a point to point transfer explicitly. To ensure packed data is sent across the network, MPI middleware performs a series of implicit communications to complete the transfer (i.e. packing strided data at the source and unpacking data by stride at the target). Some transmissions occur in user space, others via the operating system in kernel space.

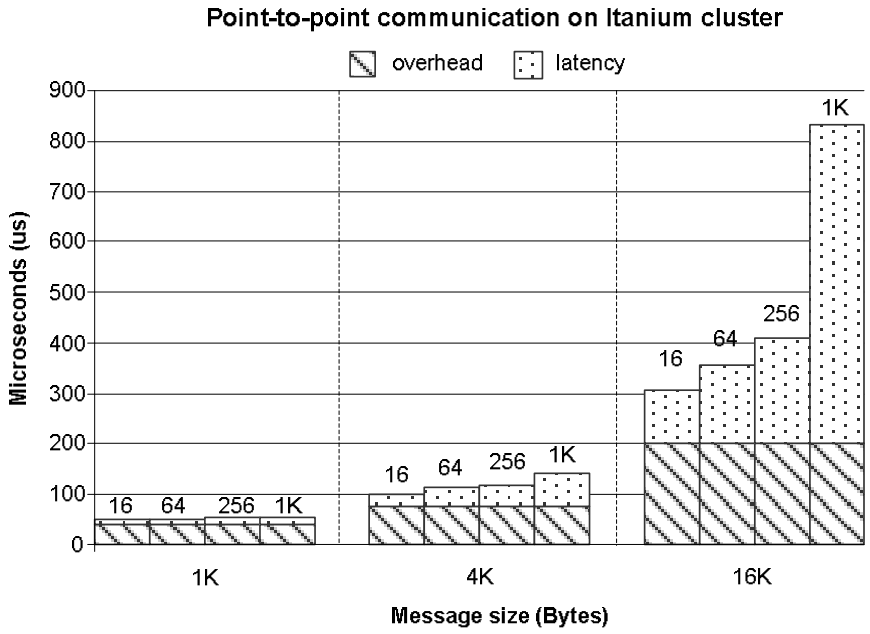


Figure 3.1: Half round-trip time for point-to-point communication. Overhead is the communication cost of non-strided message transfers. Latency is the additional time for strided message transfers. Latency can dominate transmission costs for strided comunications.

Middleware can dominate communication cost. For example, Fig. 3.1 shows point-to-point communication cost on an Itanium-based cluster. The lower stack of each bar (overhead) is the total unit-stride transfer cost in microseconds between source and target nodes for various data message sizes (1K, 4K, and 16K bytes). This cost, an upper bound of the hardware transfer cost, does not change with a messages stride size (16, 64, 256, and 1K bytes). The communication cost

is quickly dominated by the upper stack of each bar (latency), or the additional cost of strided data packing performed by communication middleware. The impact of latency on communication varies with data size, data stride, and system implementation.

However, existing communication performance models using only hardware parameters (e.g. network bandwidth) like $LogP$ tend to ignore the incurred costs in communication middleware for simplicity. This implies algorithms designed using these models is not optimal. For example, algorithms may include more than required strided communications. Nonetheless, Fig. 3.1 shows such communications can easily grow to 4x the cost of unit-stride communications.

In this chapter, we present an accurate yet practical communication performance model, the log_nP model and one of its descendants, the log_3P model. These two models include the cost of middleware by separating the costs of unit-stride and strided accesses at various points along the communication critical path. The log_nP model is general, accurate, and enough robust to apply to any point-to-point communication, yet it is cumbersome to use in practice. Hence, we applied reduction techniques to log_nP model to create the log_3P model, which is more practical to use in a wide range of clusters. We validate log_3P on a real system and show its usage in performance analysis and prediction, as well as algorithm design to optimize performance.

3.2 The log_nP and log_3P Models of Point-to-point Communication

In this section, we describe the log_nP and log_3P models of point-to-point communication. The log_nP model is a general model that incorporates middleware cost in estimating the cost of distributed communication, and the log_3P model is simplified log_nP model.

3.2.1 The log_nP model

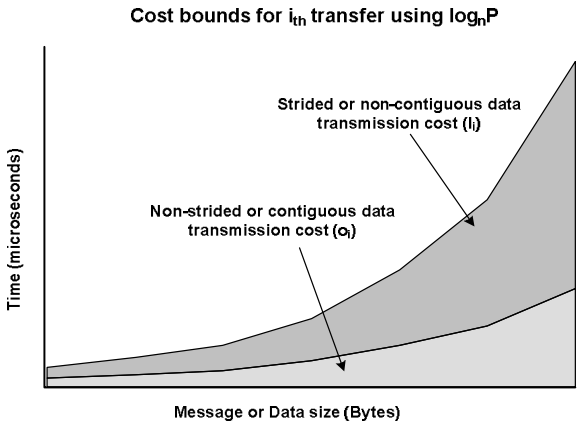


Figure 3.2: Performance bounds with the log_nP model parameters. o and l are both functions of message size. l is additionally subject to variations due to stride size. l is shown for a single, fixed stride.

Fig. 3.2 provides an illustrative view of the parameters in the log_nP model. In this view, we explicitly separate the cost of data transfer into two parts: communication cost for transferring continuous (unit-stride) data, and additional communication cost due to strided data. Formally we characterize the cost of each data transfer using five parameters:

l: the effective latency (the letter "ell"), defined as the effective delay in the transmission or reception of a strided message over and above the cost of a unit-stride transfer. The system-dependent l cost is a function of the message data size (s) under a variable stride or distribution (d). We denote this function as $l = f(s, d)$, where variable s corresponds to a series of discrete message sizes in bytes, variable d corresponds to a series of discrete stride distances in bytes between array elements, and function f is the additional time for transmission in microseconds over and above the unit-stride cost for variable message data size s and stride d. This cost is bounded above by the cost of data transfers without computational overlap and bounded below by 0 or full computational overlap.

o: the effective overhead, defined as the effective delay in the transmission or reception of a unit-stride message. The system-dependent o cost is a function of the message data size (s) under a fixed unit-stride (i.e. when $d = 1$ array element). We denote this function as $f(s, d) = f(s, 1) = o$, where variable s corresponds to a series of discrete message sizes in bytes , variable $d = 1$ array element corresponds to the unit-stride between adjacent array elements, and function f is the time for transmission in microseconds for variable message data size s and stride $d = 1$ array element. This average, unavoidable overhead represents the best case for data transfer on a target system. This cost is bounded below by the data size divided by the hardware bandwidth.

g: the gap, is unit-stride point-to-point effective communication cost including additional system delays. o is the cost of a unit-stride point to point transfer without resource contention. $g - o$ is the additional cost of contention. g provides flexibility for expansion of our model to consider effects of multiple messages not covered by o and l. For now, we assume this parameter

has no impact on communication cost, effectively using $o = g$. At times, we use $max(o, g)$ for completeness, but this cost simply reduces to o under our assumption.

n: the number of implicit transfers along the data transfer path between two endpoints of communication. Endpoints can be as simple as two distinct local memory arrays or as complex as a remote transfer between source and target memories across a network. o_i or l_i are the average costs for the i^{th} implicit transfer along the data transfer path where $0 < i < n - 1$. As n increases, so does the accuracy and complexity of the model of implicit communication.

P: the number of processor/memory modules. This parameter is used when determining the cost of collective communications estimated as a series of point-to-point transfers.

All parameters are measured as multiples of processor clock cycles converted to microseconds. Conversion to rates of cycles or microseconds per byte is straightforward. In our discussion, we assume typical load/store architectures with hierarchical memory implementations. Clusters may be composed of single processor or multiprocessor nodes communicating on a shared bus or through a network interface card (NIC) attached to interconnect. Our analyses and predictions are at the application level, so nondeterministic characteristics of memory access delay at the microarchitecture level are not considered. We assume deterministic access delay and use minimums of average values as inputs to our model. This assumption is validated if our predictions are accurate. In this chapter, our predictions for common collective communications are typically within 3%. As is customary, we assume the receiving processor may access a message only after the entire message has arrived. At any given time a processor can either be sending or receiving a single message.

3.2.2 Communication cost estimation using log_nP

For an explicit end-to-end communication consisting of n implicit transfers numbered 0 to $n-1$, the log_nP model estimates its communication cost (or time), T as:

$$T = \sum_{i=0}^{n-1} \{\max(o_i, g_i) + l_i\}, \tag{3.1}$$

here n is the number of implicit data transfers. Each transfer has data characteristics of size (s) and stride (d). We denote o_i the cost for unit-stride transfers, and denote l_i the additional cost for strided transfers; both are for the i^{th} implicit communication. As o_i is a function of the size (s) and unit-stride ($d = 1$), we write $o_i = f(s, d)_i = f(s, 1)_i$. Similarly, as l_i is a function of the size and stride of a message, we write $l_i = f(s, d)_i$. In practical use, we also assume $g_i - o_i = 0$ since in practical use we can ignore system contention while maintaining sufficient model accuracy. Following these discussions, we rewrite Equation (3.1) as:

$$T = \sum_{i=0}^{n-1} \{o_i + l_i\} = \sum_{i=0}^{n-1} \{f(s, 1)_i + f(s, d)_i\}. \tag{3.2}$$

The log_nP model allows consideration of communication costs previously ignored by hardware models of communication. It is also flexible enough to apply to any point-to-point transfer. However, directly using log_nP as described in Eq(3.2) requires substantial efforts to measure all the model parameters.

3.2.3 The log_3P model

The convergence of distributed architectures to clusters of SMPs implies we can make reasonable assumptions to reduce the complexity of log_nP .

First, as our initial intent is to model point-to-point and collective communications instead of non-deterministic effects of resource contention, we ignore the extraneous effects of multiple messages competing for limited system resources. In short, we assume $o = g$.

Second, we break each end-to-end communication into three implicit communication points:

- Point 0: middleware communication from user space to the network interface buffer;
- Point 1: communication across the inter-connect;
- Point 2: middleware communication from the network interface buffer to user space.

As described in section 3.2.1, we break each implicit communication into costs ($o + l$), and take into account of the effects of memory hierarchy for both points 0 and 2. In short, we assume $n = 3$.

Based on these two assumptions, Equation (3.2) can be reduced to:

$$\begin{aligned} T &= \sum_{i=0}^{2} \{f(s,1)_i + f(s,d)_i\} \\ &= \{f(s,1)_0 + f(s,d)_0\} + \{f(s,1)_1 + f(s,d)_1\} + \{f(s,1)_2 + f(s,d)_2\} \end{aligned} \quad , \tag{3.3}$$

or

$$T = \{o_0 + l_0\} + \{o_1 + l_1\} + \{o_2 + l_2\} \ . \tag{3.4}$$

Following the common practice adopted by most communication models, we combine the source and target overhead. We notice that since any point-to-point communication inherently causes source and target overhead, separating point 0 and point 2 does not provide information of enough interest to warrant further complexity in the model. On the other hand, separating the overhead (unit-stride cost) and latency (additional strided data costs) from the communication cost is necessary given the significance of the middleware costs and the impact of memory distribution on

execution time. Thus, we can rewrite Equation (3.4) as,

$$T = \{o_0 + o_2\} + \{l_0 + l_2\} + \{o_1 + l_1\} \ . \tag{3.5}$$

Further, we group the implicit communications at point 0 and point 2 as an aggregated communication and label it $\mathrm{middleware}$. Hence, we have:

$$\text{middleware overhead} = o_{mw} = o_0 + o_2 = f(s,1)_0 + f(s,1)_2 \, ,$$

and

$$\text{middleware latency} = l_{mw} = l_0 + l_2 = f(s,d)_0 + f(s,d)_2 \, .$$

We also label point 1 as network communication. We assume $o_1 = f(s,1)_1$ is a linear function of a fixed packet size transfer cost across the interconnect, and $l_1 = f(s,d)_1$ is zero as packets are unit-stride and fixed size. These assumptions results in a linear function

$$\text{network overhead} = o_{net} = f(s,1)_1 \, .$$

From the above discussion we semantically express the $log_3 P$ model as:

$$T = o_{mw} + l_{mw} + o_{net} \, . \tag{3.6}$$

3.3 $log_3 P$ Model Parameter Derivation

In this section, we show how to derive the parameters in the $log_3 P$ model. For illustration purposes, we use the MPICH implementation for the standard send operation (MPI_Send) on Linux clusters as an example.

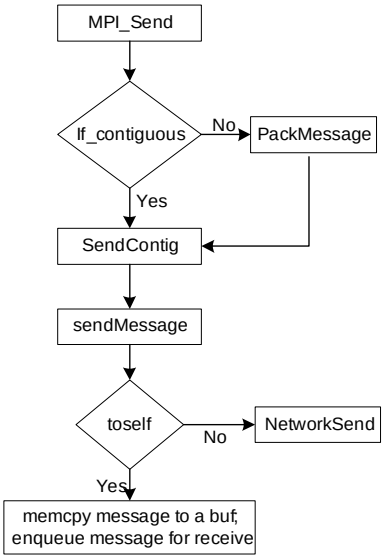


Figure 3.3: Sender distributed communication. This flow chart shows the MPICH implementation of a blocking MPI_SEND for long messages. Any strided message is packed prior to transmission. Messages sent in shared memory (or to self) avoid the use of sockets. Messages sent across the network use sockets and require additional size-dependent buffering.

Fig. 3.3 provides an abstract flow chart of the implicit communications on the sender side for long messages. Unit-stride messages do not require packing. Strided data is packed into a contiguous buffer and sent across the network to its destination. In either case, the "send contiguous" function is invoked.

The MPICH implementation selects one of three size-dependent protocols to ensure good performance. Messages are classified as short ($s < 1Kbytes$), long ($1Kbytes \leq s \leq 128Kbytes$) and very long ($s > 128Kbytes$). For short and long messages, no message handshakes or acknowledgements are needed to establish communication between sender and receiver at MPI level as the data are saved in an intermediate local buffer allocated by MPI on sender or receiver side. For very

long messages, handshakes or acknowledgements at MPI level are required to directly transfer data between sender's and receiver's application buffers. In short, two kinds of buffer managements are deployed at MPICH for message passing based on message size.

- Short/Long messages. Data copy from sender's application buffer to intermediate buffer allocated by MPI to receiver's application buffer.
- Very long messages. Streaming data copy between sender's application buffer and receiver's application buffer.

For model parameterizations, we measure the communication time of MPI_Send for two cases:

Case 1: sender and receiver are the same process. (Send to self)

Case 2: sender and receiver are different processes. (Remote Send)

Fig. 3.4 shows our model parameters for sender and receiver (generally) and costs for long messages (16 Kbyte) and long, strided messages (1 Kbyte stride) on the IA-64 Linux cluster (Titan). Fig. 3.4a and 3.4b show costs for unit-stride send and receive pairs. For simplicity, we use symmetrical parameters (e.g. the values for o_0 and o_2 from Equation (3.5) are averaged for sender and receiver and expressed as a single value such as $o_0 = o_{mw}/2$ and $o_2 = o_{mw}/2$). The o_{mw} term used in later graphs refers to the total overhead ($o_{mw} = o_{mw}/2 + o_{mw}/2$) on sender and receiver as described by Equation (3.6). Fig. 3.4c and 3.4d show the additional latency for strided communications (see"pack message" in Fig. 3.3). The l_{mw} term is the total middleware latency at sender and receiver. As discussed earlier, network transfer costs have $l_{net} = 0$, so it is not necessary to break o_{net} down further (since sender and receiver network latency is intuitively a single cost).

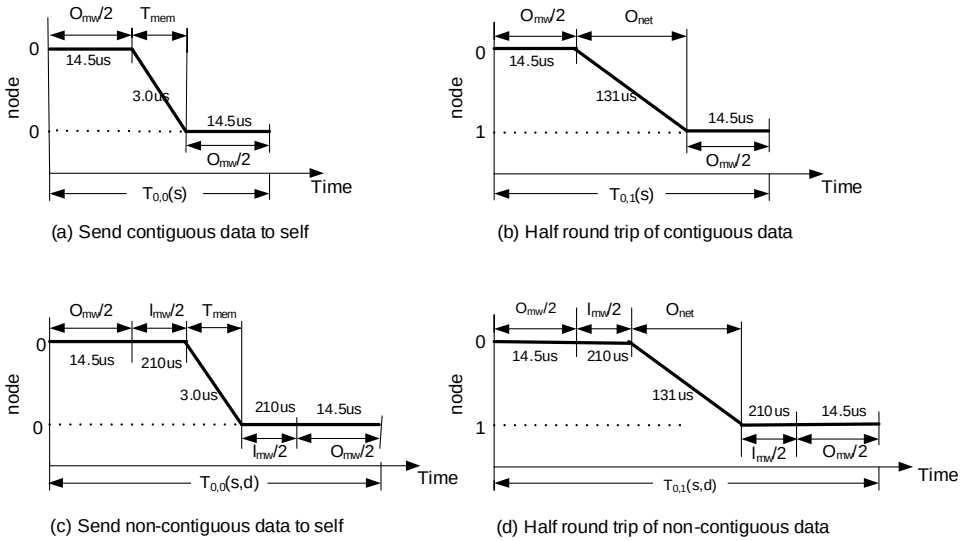


Figure 3.4: Half-round trip sender/receiver communication cost. Case 1 (Same source and destination or send to self) shown for non-strided and strided costs in (a) and (c) respectively. Case 2 (Different source and destination) shown for non-strided and strided costs in (b) and (d) respectively. Actual costs (in microseconds) shown for message size of 16 Kbytes, stride of 1 Kbytes for (c) and (d) on IA-64 cluster. Note: costs not drawn to scale for illustrative purposes.

Now, we can identify each term of the $log_3 P$ model shown in Fig. 3.4 as follows. First, we obtain the round trip costs of contiguous transfers for two cases (send to self or 0 sends to 0 and remote send or 0 sends to 1, respectively) as a function of size (s): send to self ($2T_{0,0}(s)$) and remote send ($2T_{0,1}(s)$). Next, we measure T_{mem} (the cost of memory copy) for different message sizes. Then, we solve the send to self equation ($T_{0,0}(s) = o_{mw}/2 + T_{mem} + o_{mw}/2$) to obtain o_{mw}. Lastly, we use the remote send equation ($T_{0,1}(s) = o_{mw}/2 + o_{net} + o_{mw}/2$) to solve for o_{net}.

To separate the costs l_{mw} for non-contiguous data, we perform similar operations as above except using non-contiguous data instead of using contiguous data. We first obtain the round trip cost of non-contiguous transfers as a function of size (s) and stride (d) for send to self ($2T_{0,0}(s,d)$).

Next we use the previously measured and derived costs to solve the send to self equation for non-contiguous data ($T_{0,0}(s,d) = o_{mw}/2 + l_{mw}/2 + T_{mem} + l_{mw}/2 + o_{mw}/2$) to obtain l_{mw}.

At this point, we have derived all individual costs for the log_3P model. To correctly use these costs, we have to be aware that both o_{mw} and o_{net} are functions of size (s), and l_{mw} is a function of size (s) and data stride (d).

Using the above parameter derivation approach and a modified version of the mpptest toolkit [72], we have obtained the values of the log_3P model parameters for 16KB message and 1K stride on the Titan (IA-64) machine as shown in Fig. 3.4: $o_{mw} = 29us$, $l_{mw} = 420us$, $T_{mem} = 3us$, and $o_{net} = 131us$. We stress that all values are repeatable during our experimental evaluation on real systems.

3.4 Experimental Model Validation

In this section, we predict the cost of point-to-point communications on an IA-64 Linux cluster using the log_3P model. If the prediction is accurate, then we can experimentally validate the correctness of the log_3P model. To show the benefit of the log_3P model, we also compare the results against predictions using the LogP/LogGP model.

3.4.1 Experimental Methodology

In our experiments, we use an IA-64 cluster named Titan. Each node of Titan has two 800 MHz Intel Itanium I processors and 2GB ECC SDRAM. Each processor is equipped with L1, L2, and

L3 caches of 32KB, 96KB, and 4MB respectively. The nodes are connected using Myrinet 2000 technology. The cluster is Linux-based and each node runs a copy of Red Hat Linux Version 7.1.

Though the $log_3 P$ model and the parameter derivation approach are general, we focus our analysis on MPI and use MPICH as the implementation of the MPI standard. The open-source characteristics of MPICH allow us to examine the implementation details and thereby enable explanation of performance trends.

We created a set of micro benchmarks using a modified version of mpptest [11]. The mpptest tool provides platform independent, reproducible measurement of message passing experiments such as ping-pong and memory copy. It is part of the MPICH distribution. It can be used to benchmark systems for determining MPICH platform dependent parameters. To ensure reproducible results we 1) pre-load data sets to "warmup" the cache to avoid measuring start-up costs; 2) repeat an explicit communication operation n times (n is an input parameter usually $>$100) and take the average as one sample to ensure we are measuring a steady state; 3) we take m samples (m is an input parameter set to 100) and choose the minimum of these m samples as the measured value to select the best case transmission; 4) we repeat this full set of $m \cdot n$ measurements at least two different times at varied hours to ensure system loads do not perturb results.

The mpptest tool provides various functions of use in our experiments. Specifically, we control the message size and type, call type (e.g. blocking or nonblocking send), and the precision or tolerance desired. We modified the tool to provide further granularity such as specifying the stride of a message. We use the resulting control to vary the data type (char, integer, and double), message size (s) and stride (d). The modified tool is portable to all systems under study (and any system

Table 3.1: $LogP/LogGP$ parameters

Parameters	Values(us)
L	13.62
o	5.9
G	0.00448
g	14.6

running MPI). For simplicity, we only present results for data type double and common communication functions MPI_Send and MPI_Recv unless mentioned explicitly. For all measurements related to strided data, we consider only regular access patterns and if using derived data types use MPI_Type_vector.

3.4.2 Experimental Results

In this section, we predict the performance of point-to-point communications using the derived log_3P model parameters (from Section 3.3) and compare this prediction to directly measured value and the prediction using the LogP/LogGP model. We calculate cost predictions per message using the LogGP model as $2o + L + (k-1)G$, where k is the message size in bytes. In all of our direct comparisons with LogP/LogGP, we consider changes between rates (cycles per message in LogGP) and direct cost (microseconds in log_3P) when predicting transfer time. We use the MPI LogP/LogGP benchmark tool [98] to gather the parameters presented in Table 3.1.

Fig. 3.5 shows the point-to-point communication prediction using the log_3P model and LogGP model on the IA-64 Linux cluster (Titan). For a given message size, data stride does not affect LogGP predictions. LogGP captures hardware characteristics and ignores the effects of middle-

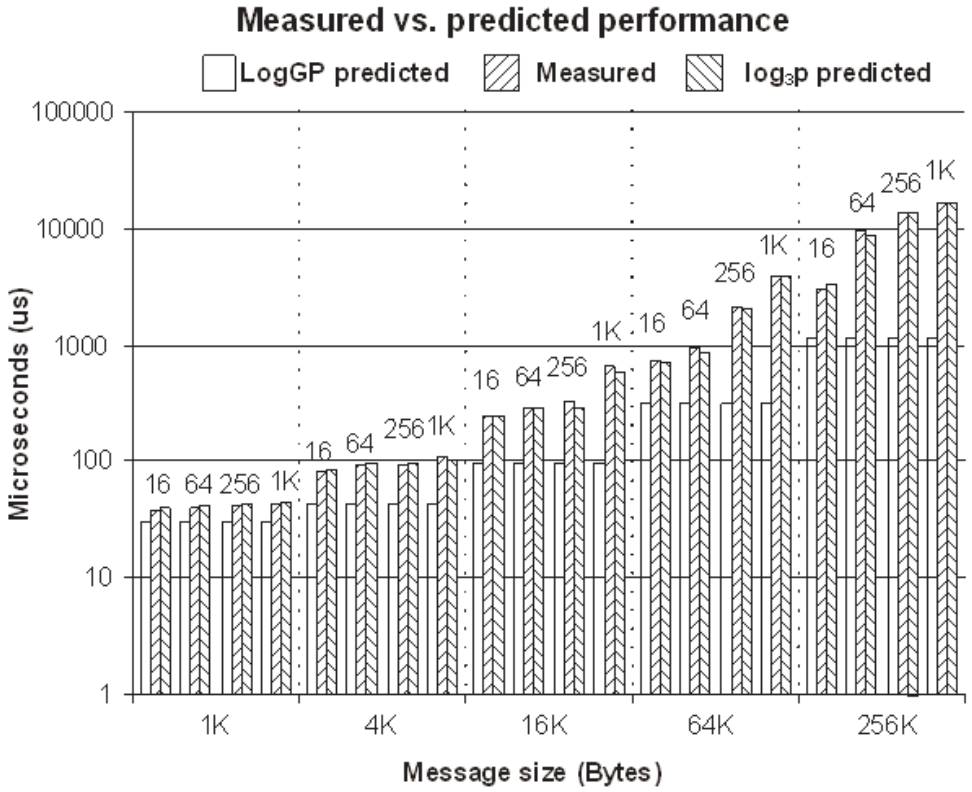


Figure 3.5: Prediction comparison between $LogGP$ and log_3P on Itanium cluster. Measured vs. predicted cost of half round trip for derived data type on Itanium cluster using LogGP (first bar) and log3P (third bar) are presented. The x-axis is message size in bytes and y-axis is cost in microseconds on a log scale. For each message size, we measure and predict regular stride of 16, 64, 256 and 1024 bytes.

ware. As shown (note the y axis is in log), middleware has a significant effect on the communication cost. The bigger the stride size, the larger this extra cost. The average relative error of LogGP prediction for contiguous data communication is 28%. The proposed log_3P model can predict the cost with average error of 5% for all the measurements. Our model prediction is slightly more accurate for short messages less than 256 bytes and large messages bigger than 128K bytes. This

observation can be explained by the fact that under these situations data can be fit in cache totally or out of cache, and data transfer cost is slightly more stable and predictable.

From the experiments results, we have shown that as the log_3P model captures the cost of memory communication through parameters such as middleware overhead (o_{mw}) and middleware latency (l_{mw}), the predictions from the log_3P are more accurate than that using the LogGP model. These results experimentally validate the correctness of the log_3P model and the corresponding parameter derivation approach for point-to-point communications.

3.5 The Practical Use of the log_3P Model

There are many practical uses for the log_3P model. We show how to apply log_3P to system analysis, communication cost prediction, and algorithm design.

3.5.1 System performance analysis

The left hand picture in Fig. 3.6 shows the measured middleware overhead (o_{mw}) and network overhead (o_{net}) on the Itanium cluster described in section 3.4.1. From the figure we see both middleware overhead (o_{mw}) and network overhead (o_{net}) increase with all message size, while middleware overhead (o_{mw}) increases faster for large message sizes. For small message sizes, network overhead (o_{net}) dominates the cost. With the increase in message size, middleware overhead (o_{mw}) or the memory/middleware communication cost dominates communication cost. For small messages that fits in the cache, hit rate is high, and middleware overhead (o_{mw}) is small. However, once the message size exceeds the cache size, capacity misses increase the average memory access

time. Additional costs in middleware determine a system-specific intersection of the two curves. This crossover point is the point at which memory or middleware delays on the source and target nodes dominate overall communication cost.

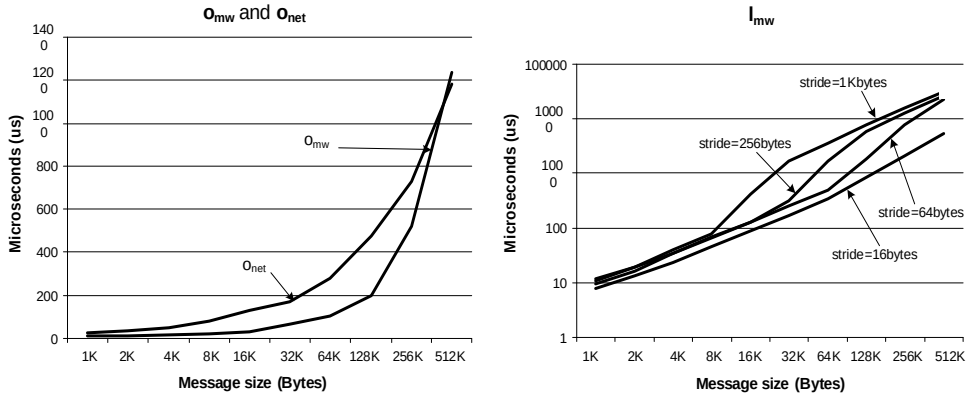


Figure 3.6: Measured overhead and latency on the IA-64 (Titan) cluster. The left picture shows how middleware overhead (o_{mw}) and network overhead (o_{net}) vary with size. Though the trends are linear as expected (x axis is in log) the slopes are different indicating tradeoffs occur at some crossover point that varies with data size. The right picture illustrates how the latency (additional costs for strides) varies substantially (x and y axis are in log_2 and log respectively) with size and stride. The varied magnitude of this cost implies crossover points in the left figure will vary with size and stride.

Latency costs, depicted on the right side of Fig. 3.6, are particularly susceptible to cache characteristics such as associativity. This figure depicts various strides over increasing message sizes. The larger the stride and message size, the more this cost dominates communication. Note the x- and y-axis are expressed in log_2 and log respectively. Cache characteristics are evident in the large differences between various strides and the relationship between (size x stride) and cost. As distances between accesses increase, average memory access times increase.

The left hand graph of Fig. 3.6 illustrates an important use of our $log_3 P$ model for application and system analysis. In the graph the crossover point occurs at about 512K. Message transmissions

larger than 512K are dominated by memory/middleware communication cost. We have additional data that shows o_{net} versus (o_{mw} + l_{mw}) for various stride sizes. As the costs due to data strides increase (reflected in the l_{mw} parameters on the right side of Fig. 3.6), crossover points will move steadily to the left (in the graph on the left in Fig. 3.6) to smaller data sizes. For example, stride=16 bytes results in a crossover point at message size of 32K and stride=256 bytes results in cross-over point at message size of 8K.

Applications that limit message sizes falling to the right of a crossover point may improve performance. If such messages are unavoidable, then system improvements in the middleware or hardware should target reducing memory communication costs. In such applications and systems, decreasing network transmission latency will not address the dominant bottleneck of the communication. A corollary to this observation is that our analyses could influence machine design to support a single type of application that only exhibits characteristics on one side of this crossover point. Next, we analyze the performance of the middleware implementation. Fig. 3.7 shows the cost separation by our model parameters for strided message transfers into three parameters: middleware overhead (o_{mw}), network overhead (o_{net}) and middleware latency (l_{mw}). All three parameters increase with message size, but middleware latency (l_{mw}) increases the fastest. Moreover, middleware latency (l_{mw}) varies with stride as well as data size.

Fig. 3.7. Parameterized strided costs broken down for the NCSA IA-64 cluster (Titan): middleware overhead (o_{mw}), network overhead (o_{net}) and middleware latency (l_{mw}). Data characteristics determine which parameter dominates communication cost and should be targeted for optimization. For each message size (1K, 4K, 16K), costs for four different stride sizes (16, 64, 256 and

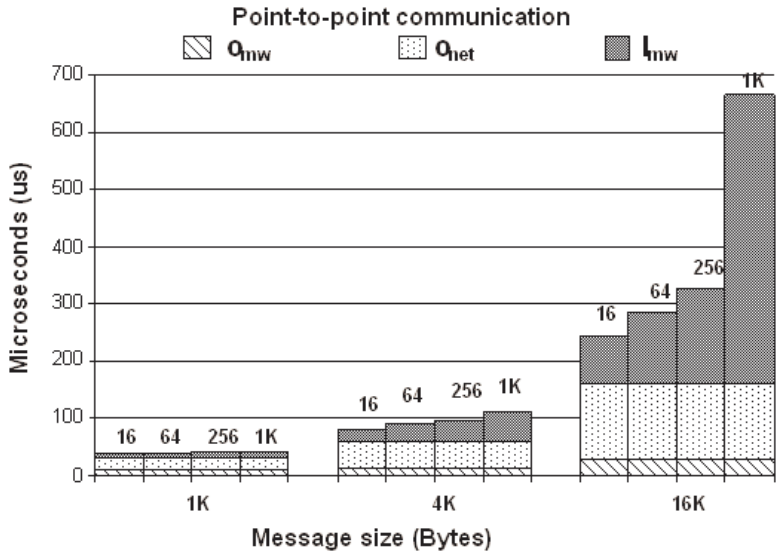


Figure 3.7: Parameterized strided costs broken down for the NCSA IA-64 cluster (Titan). Parameters are middleware overhead (o_{mw}), network overhead (o_{net}) and middleware latency (l_{mw}). Data characteristics determine which parameter dominates communication cost and should be targeted for optimization. For each message size (1K, 4K, 16K), costs for four different stride sizes (16, 64, 256 and 1K) are measured.

1K) are measured. The larger the stride size, the larger the middleware latency (l_{mw}) due to plateau cache performance already discussed. For large message size with large stride size, middleware latency (l_{mw}) dominates communication time. MPICH is responsible for the middle latency (l_{mw}), allocating extra buffers for pack and unpack operations on the sender and receiver. These additional memory copies impact performance severely. This indicates where MPICH performance can be targeted for optimization [1].

[1]We do not mean to imply that middleware latency is the fault of the MPI implementation. Our model isolates the costs resulting from the interaction of application and middleware. An application may require strided accesses resulting in significant middleware latency. Our model quantifies the impact and identifies a possible culprit (MPI). However, it may be more appropriate in some cases to modify the application.

3.5.2 Communication cost prediction

In section 3.4.2, we showed that the log_3P model can accurately predict the cost of point-to-point communication. In this section, we show how to predict the costs of communication for derived data types and collective communication.

Derived Data Types

Although derived data types (DDT) provide an abstraction to ease programming, some implementations (e.g. MPICH) may suffer poor performance when DDTs are employed. An alternative often embraced by users is to pack and unpack data manually (using simple optimizations for block size, loop unrolling, etc.). One implementation of packing and unpacking can be simulated by copying indexed items in a buffer to a contiguous buffer, for instance:

```
for (i=0, j=0; j<count; i+=stride, j++) a[j]=b[i].
```

Unpacking is copying items from a contiguous buffer to a non-contiguous buffer by index. The sum of packing and unpacking is the cost of the explicit communication.

We use our log_3P model to predict the cost of packing and unpacking for various sizes and strides of data. Fig. 3.8 shows the measured and predicted latency using our model. The average relative error of prediction is 3.5%. The prediction is slightly more accurate for short messages less than 256 bytes and large messages bigger than 128K bytes for all the strides. One interesting observation is that contrary to our expectation, manual packing and unpacking does not always guarantee much better performance than DDT on this IA-64 cluster. The average improvement

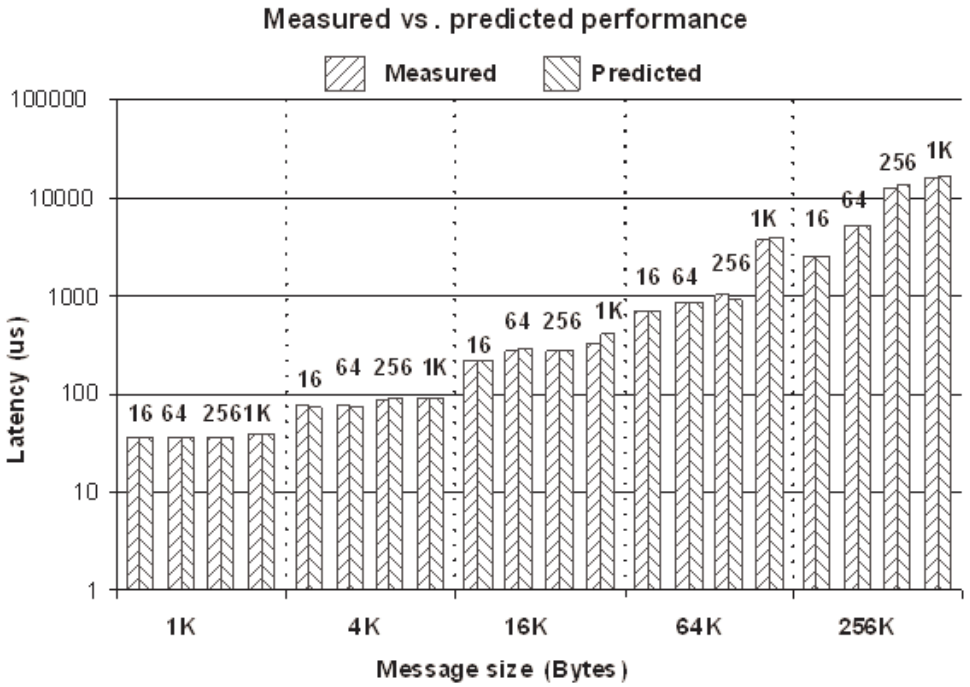


Figure 3.8: Measured vs. predicted half round trip time for packing and unpacking. The x-axis is message size in bytes and y-axis is time in microseconds (in log). For each message size, we measure and predict regular strides of 16, 64, 256 and 1024 bytes. The first bar is measured cost, while the second bar is the predicted cost.

over derived data types is about 15%. The maximum improvement is as much as 50%, while in some cases, the improvement is just below 2%.

Researchers at Argonne National laboratory have used the $log_3 P$ model to improve the general performance of derived data types as follows. When a derived data type is used, the size and stride information is embedded in the DDT representation. At run-time, the size and stride information is used as input to our model to predict the performance of various algorithm implementations. The prediction is used to suggest the best algorithm implementations for various blocking and array padding factors. By selecting the best performing algorithm at runtime, derived data type perfor-

mance was improved significantly (at times more than 50%) over both MPICH and proprietary IBM MPI implementations for various systems. Further details can be found in a related paper [22].

Collective Communication

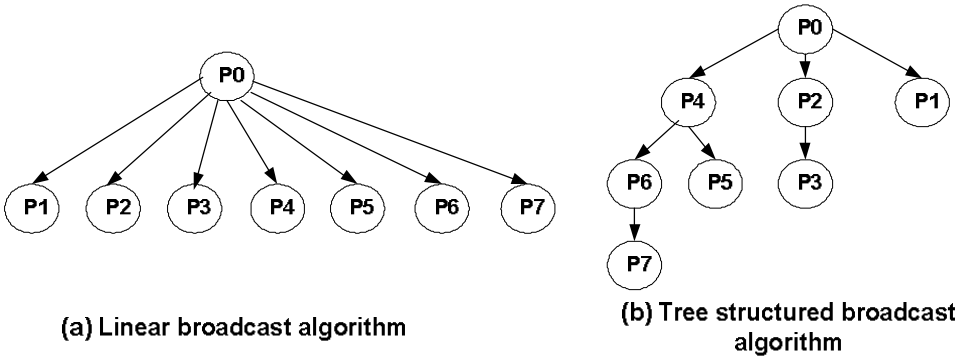


Figure 3.9: Communication patterns of 8-way broadcast. Here the numbering denotes the order of the communications. A typical MPICH broadcast implements a hybrid version of these algorithms where processes are grouped. Linear broadcast is used within the groups and tree-structured broadcast is used across groups. We show linear and tree-structured broadcast only since the hybrid scheme is a mixture of these two extremes.

In this section, we illustrate use of the point-to-point $log_3 P$ communication model to analyze two collective communication algorithms: linear broadcast and tree structured broadcast. The communication patterns are depicted in Fig. 3.9. The actual MPI broadcast is implemented in MPICH by integrating these two algorithms. For example, for an 8-node broadcast, MPICH uses linear broadcast (Fig. 3.9a) for group size = 8, and tree structured broadcast (Fig. 3.9b) for group size = 1. For other group sizes, a tree structured algorithm is used to broadcast a message between groups of processes, and then the linear algorithm is used to broadcast the message from the first

process in a group to all other processes. These examples serve several purposes: 1) they quantify the impact of middleware costs for simple algorithm cost models; 2) they illustrate how to apply the model to algorithm cost analysis for comparison.

The linear broadcast algorithm is based on point-to-point communication, in which $(P - 1)$ individual consecutive MPI_Send's are used at the source/root node to transfer data to each remaining node, where P is the number of processors. The cost of this implementation of broadcast includes the overhead at the source, the cost of network transmission, and the cost of delays until the last node receives the message. We implement a linear broadcast, as one would implement an algorithm, predict the cost analytically, and compare this prediction to the measured cost. For data transmission, the cost should be the sum of contiguous data communication and the extra latency introduced by strided data. Using $log_3 P$, the cost is $P \cdot (o_{mw}/2 + l_{mw}/2)$ + o_{net}, where $P \cdot (o_{mw}/2 + l_{mw}/2)$ is the middleware overhead and middleware latency occurring at the source node for sending data to other $(P - 1)$ nodes and the last receiving node, and o_{net} is the network overhead. The prediction of broadcasting a message size of $(k + 1)$ bytes with LogGP is $2o + L + (P - 1)Gk + (P - 2)g$ where $2o$ is overhead at the source node and the last receiving node, L is the network latency, $(P - 1)Gk$ is the cycles to send $(P - 1)$ messages with each of them taking Gk cycles, and $(P - 2)g$ is the cost of $(P - 2)$ gaps between $(P - 1)$ messages. The values of parameters o, L, G, and g are given in Table 3.1.

Fig. 3.10 shows predictions for linear broadcast using the $log_3 P$ model and LogGP model on the IA-64 Linux cluster. For small message sizes and small strides, the LogGP prediction is accurate. But for large message sizes and larger strides, LogGP prediction error is considerable,

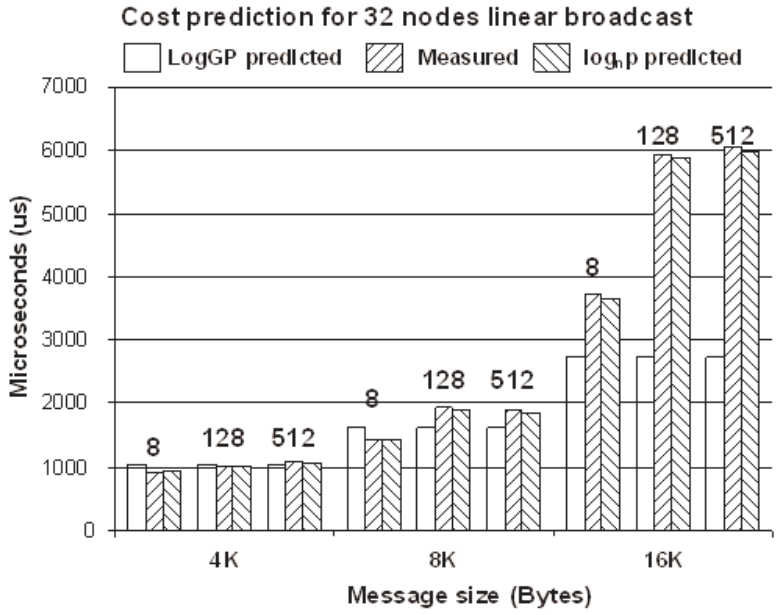


Figure 3.10: Cost prediction of linear broadcast. X-axis represents message size in bytes and y-axis represents time in microseconds. For each message size, we measure and predict regular strides of 8, 128, and 512 bytes. The first bar is LogGP predicted cost, the second bar is the measured cost, while the third bar is the predicted cost by the $log_3 P$ model.

and it increases with data size and stride. For the data points measured, maximum relative error of LogGP is 54% and average relative error is 20.3%. The average error of $log_3 P$ predictions is about 3%, and the maximum relative error is 11% for the data points measured. For the tree structured broadcast algorithm, each node sends data to its children after receiving from its parent. The root node is the source. This algorithm has a characteristic that the message latency is determined by the height of the tree. Using the $log_3 P$ model, the latency of this algorithm is (o_{mw} + l_{mw}+o_{net}) times the height of the tree, which is $h = log_2(P)$, where P is the number of processors. The prediction for LogGP is $h(2o + L + kG) + (h - 1)g$, where $(k + 1)$ is message size in bytes and other parameters are given in Table 3.1.

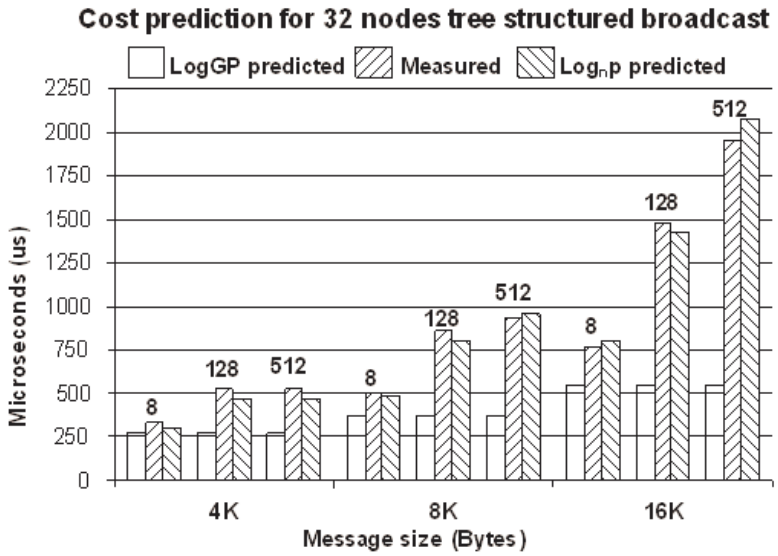


Figure 3.11: Cost prediction of tree structured broadcast. The x-axis is message size in bytes and y-axis is time in microseconds. For each message size, we measure and predict regular strides of 8, 128, and 512 bytes. The first bar is LogGP predicted cost, the second bar is the measured cost, while the third bar is the predicted cost by the $log_3 P$ model.

Fig. 3.11 shows the predictions for tree structured broadcast using the $log_3 P$ model and LogGP model on the IA-64 Linux cluster. The average relative error of LogGP prediction is 46% for the data points measured. The error increases with data size and stride. The minimum relative error is 16% for contiguous data with size of 4Kbytes, and maximum relative error is 72% at size 16Kbytes with stride of 512bytes. The average relative error of $log_3 P$ prediction is about 6% for the measured data points. The maximum error is 18% for 16 nodes broadcast, and 11% for 32 nodes broadcast.

3.6 log_3P Model Guided Algorithm Design

As mentioned in section 3.1, an accurate but practical model of communication can help software developers design more efficient parallel algorithms. In this section we use a layered 3-D FFT application as an example to show how to use the log_3P model for efficient parallel algorithm design.

The 3-D FFT algorithm partitions a 3-D array of data in the z direction and performs three 1-D FFT operations in x, y and z dimensions. The 1-D FFT in the x and y dimensions can be completed locally on each node, but the 1-D FFT in the z dimension requires all-to-all exchanges between nodes and a transpose between endpoints.

We first consider communication using a derived data type (*ddt*) algorithm to exchange strided data and perform the transpose. The *ddt* algorithm relies on middleware to pack the strided data and map strided data to contiguous locations at the destination. This results in middleware latency (l_{mw}). A second algorithm design (*pack*) manually packs and transposes the matrix and then exchanges the packed message data with other processors. Both designs are naive [22] in that they operate on entire rows or columns and introduce significant latency due to strided memory communication. A third optimized (*opt*) algorithm design uses blocking to manually pack and transpose the matrix. The NAS PB FT benchmark uses a similar implementation and blocking.

We used the NAS Parallel Benchmark (FT) for the *opt* algorithm and created our own versions of FT for the *ddt* and *pack* algorithms. These codes were executed on a NERSC IBM 1.9GHz p575 POWER 5 system of 122 8-processor nodes, each with 32GB shared memory connected by a high-bandwidth low latency switching network. Each processor has a 64KB/32KB Instruction/Data L1

cache, 1.92MB L2 cache and 36MB L3 cache. Fig. 3.12 shows costs for the *ddt*, *pack*, and *opt* algorithms for three sets of problem-size and processor combinations (FT.B.4, FT.B.8, and FT.C.16). Within a single set there are 3 groups of 3 bars. The groups refer to predicted communication cost combined with measured computation cost for $LogP$ (i.e. LogGP) and log_3P, and actual measured values respectively. Each bar in a group provides values for the three algorithms under study.

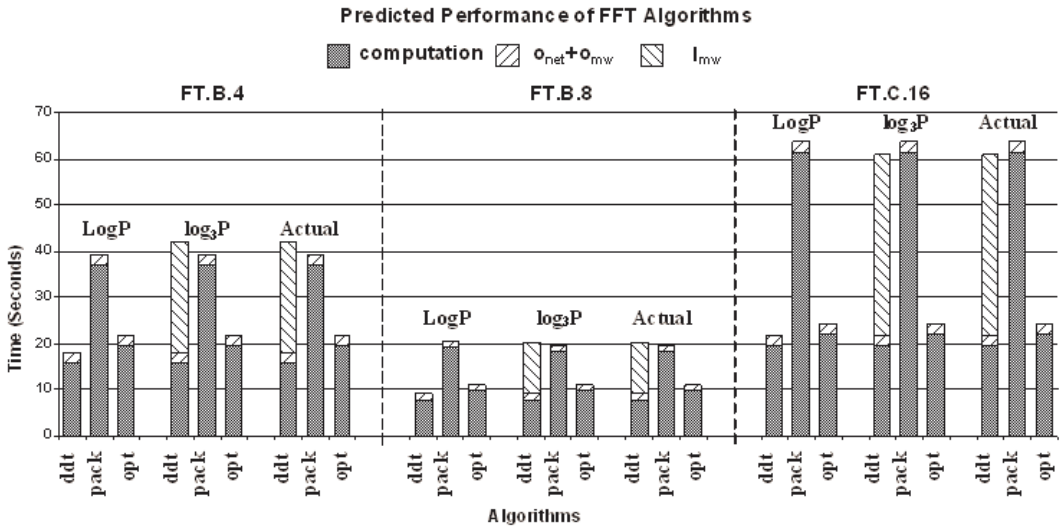


Figure 3.12: Algorithm design using log_3P. We compare $LogP$ and log_3P predicted and actual performance for ddt, pack and opt algorithms on 4, 8, and 16 processors. $LogP$ assumes middleware latency (l_{mw}) is negligible and suggests the ddt algorithm always performs best. log_3P suggests the opt algorithm will perform best and suggests optimizing middleware cost may result in cost savings. Memory communication for these FFT codes is as much as 59.3% of total time.

For each bar in Fig. 3.12, we divide the actual or predicted execution time into three costs as appropriate: FFT computation time , contiguous data communication time ($o_{net}+o_{mw}$), and strided data communication time (l_{mw}). We also measured FFT setup, checksum and synchronization time, but omit these in our graphs since they represent a small fraction of total time and are constant across all algorithm implementations.

We first observe that memory communication cost in these implementations is significant. Fig. 3.12 shows actual measurements in all three data sets for *ddt* and *pack* algorithms. The actual cost of packing strided data in middleware (l_{mw}) in the *ddt* algorithm is 51.6% of the total execution time for FT.B running on 4 processors. The actual cost of manually packing strided data in our *pack* algorithm is 48.5% of total execution time on 4 processors - this cost is included in the "computation" cost. The *opt* algorithm improves the *ddt* algorithm performance by 43.5% on 4 nodes. The percentages of packing cost to total cost are 59.3% for the *ddt* algorithm and 11.3% for the *opt* algorithm on 16 processors. In all cases, the best (i.e. shortest) actual execution time is found using the *opt* algorithm.

Now that we have identified the best cost for the actual measurements on a real system, we can use $LogP$ and log_3P to identify the best algorithms suggested by model prediction. Fig. 3.12 shows the predicted execution times for $LogP$ and log_3P. For FT.B.4, $LogP$ suggests the best execution time is obtained using the *ddt* algorithm while log_3P suggests the best execution time is obtained using the *opt* algorithm. Since *opt* is actually best in all cases, log_3P suggests the appropriate algorithm. In the *ddt* case, $LogP$ under-predicts since it ignores the middleware costs and log_3P predicts accurately and quantifies the costs of middleware that can be reduced with optimization. In the *pack* case, $LogP$ and log_3P provide good estimates of actual cost since packing costs are absorbed in computational cost . In the *opt* case, $LogP$ and log_3P provide accurate estimates since middleware costs have been minimized via blocking.

The results for FT.B.8 and FT.C.16 are similar. In both cases, $LogP$ suggests the *ddt* algorithm performs best while the log_3P model suggests the *opt* algorithm performs best. Again $LogP$ and

log_3P are accurate for the *pack* algorithm and the *opt* algorithm but the lack of middleware estimates by $LogP$ significantly underestimates the actual cost of the *ddt* algorithm which leads to an incorrect conclusion.

3.7 Chapter Summary

We presented in this chapter simple and practical yet accurate point-to-point communication performance models (log_nP and log_3P) that include the impact of communication middleware on system and application performance. On a real system, log_3P delivers very accurate predictions for both point-to-point (within 5% error) and collective broadcast communications (within 11% error). It has been used to improve MPICH performance, and guide optimal algorithm design for realistic applications.

Although the log_nP model and related analysis techniques show promise in performance evaluation and prediction, there are some limitations. For example, in this chapter our analyses were limited to regular access patterns. Prediction is more cumbersome for irregular patterns present in some codes that use sparse matrices. As future work, we are exploring techniques used by the copy-transfer model [134] to handle irregular accesses, though it is not clear at present whether this is applicable to middleware cost estimation. For more realistic communication schemes embedded in full applications, analyses will be additionally complicated. For instance we are exploring incorporation of contention in the g parameter of our model. We are attempting to refine our approach for improved accuracy in such a context.

Chapter 4

Evaluating Power/Energy Efficiency for Distributed Systems and Applications

This chapter presents PowerPack, a software and hardware toolkit for profiling, evaluating, and characterizing the power and energy consumption of distributed parallel systems and applications. Through the combination of direct measurement, performance counter-based estimation, and flexible software, PowerPack provides fast and accurate power-performance evaluation of large scale systems at component and function-level granularity. Typical applications of PowerPack include but are not limited to: 1) quantifying the power, energy, and power-performance efficiency of given distributed systems and applications; 2) understanding the interactions between power and performance at fine granularity; 3) validating the effectiveness of candidate low-power and power-aware technology. In this dissertation, PowerPack serves as the measurement infrastructure of the performance-directed power-aware high performance computing approach.

4.1 The PowerPack Framework

As shown in Figure 4.1 and 4.2, the PowerPack framework consists of three major components: hardware power/energy profiling, data acquisition/processing, and profiling control software. The PowerPack framework supports component-level power profiling and mapping between power profiles and source code.

4.1.1 Direct Component-Level Power Measurements

For direct power measurements, PowerPack uses three kinds of hardware: multimeters, smart power strips, and ACPI (advanced configuration and power interface)-enabled power supplies. While smart power strips and ACPI-enabled power supplies sample the total nodal power and energy consumption, multimeters provide directly power measurements at component granularity.

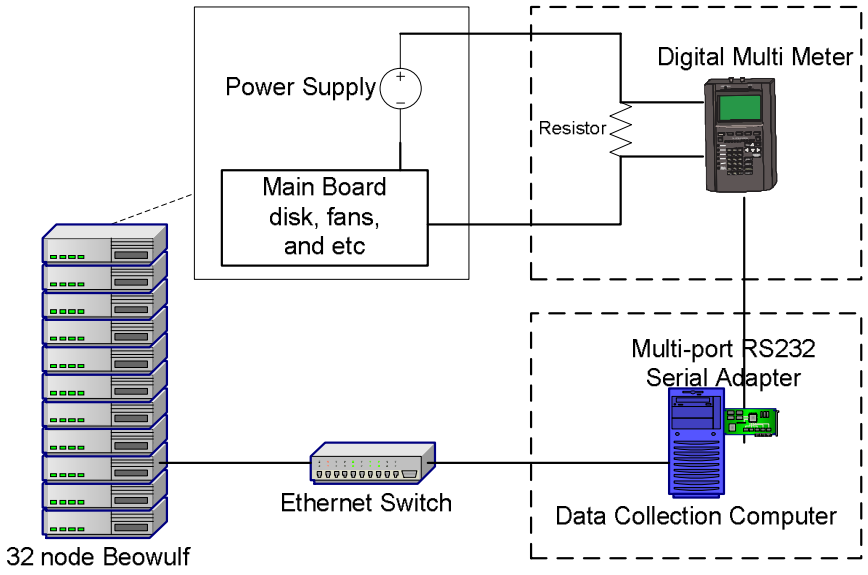


Figure 4.1: The prototype for direct power measurement

In Figure 4.1, we show a prototype system that measures a 32-node Beowulf cluster. Each slave node on the cluster has one 933 MHz Intel Pentium III processor, four 256M SDRAM modules, one 15.3 Gbyte IBM DTLA-307015 DeskStar hard drive, and one Intel 82559 Ethernet Pro 100 onboard Ethernet controller. While our discussion is specific, this approach is portable to many cluster architectures composed of commodity parts.

In this prototype, ATX extension cables connect the tested node to a group of 0.1 ohm sensor resistors on a circuit board. The voltage on each resistor is measured with one RadioShack 46-range digital multimeter 22-812. All digital multimeters are attached to a multi port RS232 serial adapter plugged into a data collection computer running Linux. We measure 10 power points using 10 independent multimeters between the power supply DC output and node components simultaneously. We also measure the AC power to the power supply using an additional multimeter.

The prototype currently measures one node at a time. To obtain the in-depth power consumption of a whole cluster, we use a node remapping approach. Node remapping works as follows. Suppose we are running a parallel application on M nodes, we fix the measurement equipment to one physical node (e.g. node #1) and repeatedly run the same workload M times. Each time we map the tested physical node to a different virtual node. Since all slave nodes are identical (as they should be and we experimentally confirmed), we use the M independent measurements on one node to emulate one measurement on M nodes. As shown later, we can accelerate profiling process by replacing node remapping with performance counter-based power estimation.

4.1.2 Power Breakdown by Component

PowerPack uses direct or derived measurement to break down nodal power profiles into four major components: CPU, memory, disk and network interface (NIC). The remaining components are treated as "others", which includes video card, power supply, fans, floppy drive, keyboard, mouse, etc.

Our measurement approach is as follows: if a component is powered through individual pins, we measure power consumption through every pin and use the sum as the component power; if two or more components are powered through shared pins, we observe the changes on all pins while adding/removing components and running different micro benchmarks to infer the mapping between components and pins. Specifically, here is the technique used for each component on our prototype system.

CPU Power: According to our experiments and confirmed by the ATX power supply design guide, the CPU is powered through four +5VDC pins. Thus we can profile CPU power consumption directly by measuring all +5VDC pins directly.

Disk Power: The disk is connected to a peripheral power connection independently and power by one +12VDC pin and one +5VDC pin. By directly measuring both +12VDC and +5VDC pins, we can profile disk power consumption directly.

NIC Power: The slave nodes in the prototype are configured with onboard NIC. It is hard to separate its power consumption from memory and other onboard components directly. As the total system power consumption changes only slightly between disabled NIC and saturated network

card bandwidth, after consulting the documentation of the NIC (Intel 82559 Ethernet Pro 100), we approximate it with a constant value of 0.41 watt.

Memory Power: Memory, NIC and other onboard components are powered through +3.3VDC pins. We measure the idle part of memory power consumption (idle power is defined as the power consumption when there is no workload running on the slave node) using an extrapolation-based approach. As each slave node in the prototype has four 256MB memory modules, we measure the power consumption of the slave node configured with 1, 2, 3, and 4 memory modules separately, then use the measured data to estimate the idle power consumed by the whole memory system. Hence, we can get the power consumption from other onboard components by subtracting memory idle power and NIC power from the total power consumption through all +3.3VDC pins. For simplicity, we treat the power consumption of other onboard components as constant. We introduce this simplification since parallel scientific applications on computational clusters rarely access most onboard components (such as video card) on the slave node. Following the above simplification, we can profile the memory power through directly measuring all +3.3VDC pins on the main power connector and subtracting a constant value.

4.1.3 Automatic Power Profiling and Code Synchronization

To automate the entire profiling process and correlate the power profile with application code, PowerPack provides a suite of library calls for the application to control and communicate with a multimeter control process. The structure of the profiling software is shown in Figure 4.2.

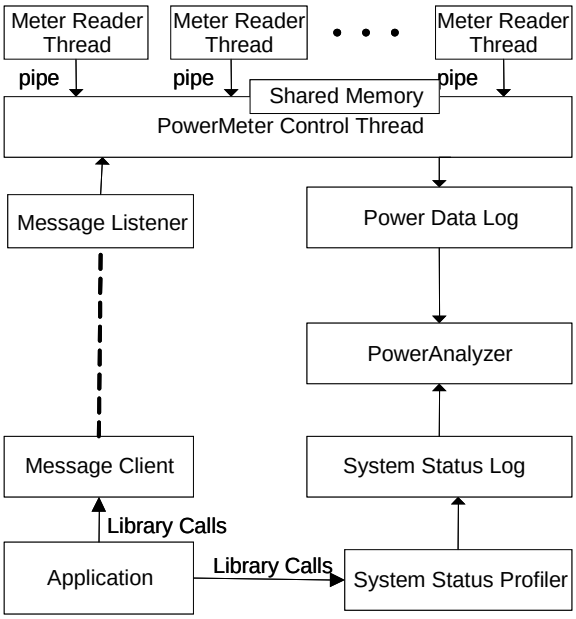


Figure 4.2: Software structure for automatic profiling

The data collection computer runs a meter control thread and a group of meter read threads, each meter reader thread corresponds to one multimeter. The meter reader threads collect readings from the multimeters and send them to the meter control thread. All the meter readers are controlled by globally shared variables. The meter control thread listens to messages from applications running on the cluster and modifies the shared variables according to messages received.

To synchronize the live power profiling process with the application, the profiled applications running on the cluster trigger message operations through a set of library calls, informing the meter control thread to take corresponding actions to annotate the power profile. Thus, by inserting the power profile API **pmeter_start_session** and **pmeter_end_session** before and after the code region

of interest, we are able to map the power profile to the source code. In Figure 4.3, we list the mostly used power profile API in PowerPack.

```
pmeter_init ( char *ip_address, int *port );
     //connect to meter control thread
pmeter_start_log ( char *log_file );
     //start a new profile log file
pmeter_stop_log ( );
     //close current log file
pmeter_start_session( char *session_label );
     //start a new profile session and label it
pmeter_end_seesion ( );
     //close current profile session
pmeter_finalize( );
     //disconnect from the meter control thread
```

Figure 4.3: The commonly used power profile API

4.2 Experimental Validation

We use three methods to validate the correctness of PowerPack in direct power measurement and power breakdown by components. First, we measure the total power consumption on a test node using multimeters and smart power strip and cross validate the measured results. Second, we compare the measured component power against the reference values provided in the component specification. Third, we profile and analyze the power consumption of a set of benchmarks that access a subset of the components.

Figure 4.4 shows the CPU and memory power profiles of an modified version of the Saavedra-Smith benchmark [128], a memory micro benchmark that accesses the cache and memory in a

regular pattern with different sizes of arrays and strides. This figure shows direct power measurement for a benchmark run with a stride size of 128 bytes. From this figure, we have observed three patterns: 1) when the array size is less than 16K bytes, CPU consumes about 30 watts and memory consumes about 3.7 watts; 2) when the array size is between 16K bytes and 128K bytes, CPU power increases to 32 watts but memory power still holds at 3.7 watts; 3) when the array size is larger than 128K bytes, CPU power goes down to 24 watts while memory power goes up to 8 watts.

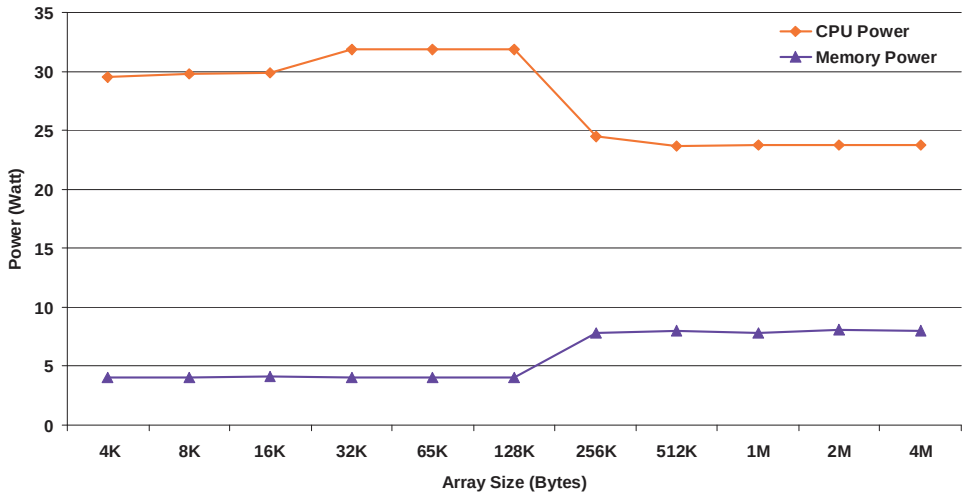


Figure 4.4: CPU and memory power consumption of memory accesses. The benchmark is Saavedra-Smith Benchmark.

Recalling that the 933-MHZ Intel Pentium III processor has a 16 KB L1 data cache and a 256 KB L2 cache, the power profile shown in Figure 4.4 matches our expectations well: when the workload does not access memory, the memory power should remain constant; when memory accesses increase, the cache misses increase and the CPU computes less; as a result, the memory power consumption goes up and the CPU power consumption goes down.

Further, the Intel documentation shows the 933-MHZ Intel Pentium III processor consumes 29 watts power, which is about the same value measured by PowerPack.

4.3 System-wide Power Distribution

Before studying the power characteristics of distributed systems and applications, we first investigate system wide power distribution of sequential applications on a single compute node. Figure 4.5 shows the snapshots of power distribution when the system is idle and when the system is running the **164.gzip**, **171.swim**, and **cp** programs. Here, **164.gzip** and **171.swim** are two benchmarks included in the SPEC CPU2000 benchmark suite [87]; **cp** is the standard Linux command for data movement. This figure exposes several important points:

1. Both system power and component power vary with workload. Different workloads stress different components in a system. Component usage is reflected in the power profile.

2. The system power under zero workload is more than 65% of the system power under workload. Reducing power consumption of non-active components could save significant energy.

3. Non-computing components such as power supply and fans contribute more than 1/2 system power when idle or 1/3 system power when busy. Improving power efficiency for those components could result in considerable power savings.

4. When the system is under load, CPU power dominates (e.g. for 164.gzip, it is 47% of system power). However, depending on the workload characteristics, disk and memory may also become the significant contributors to system power.

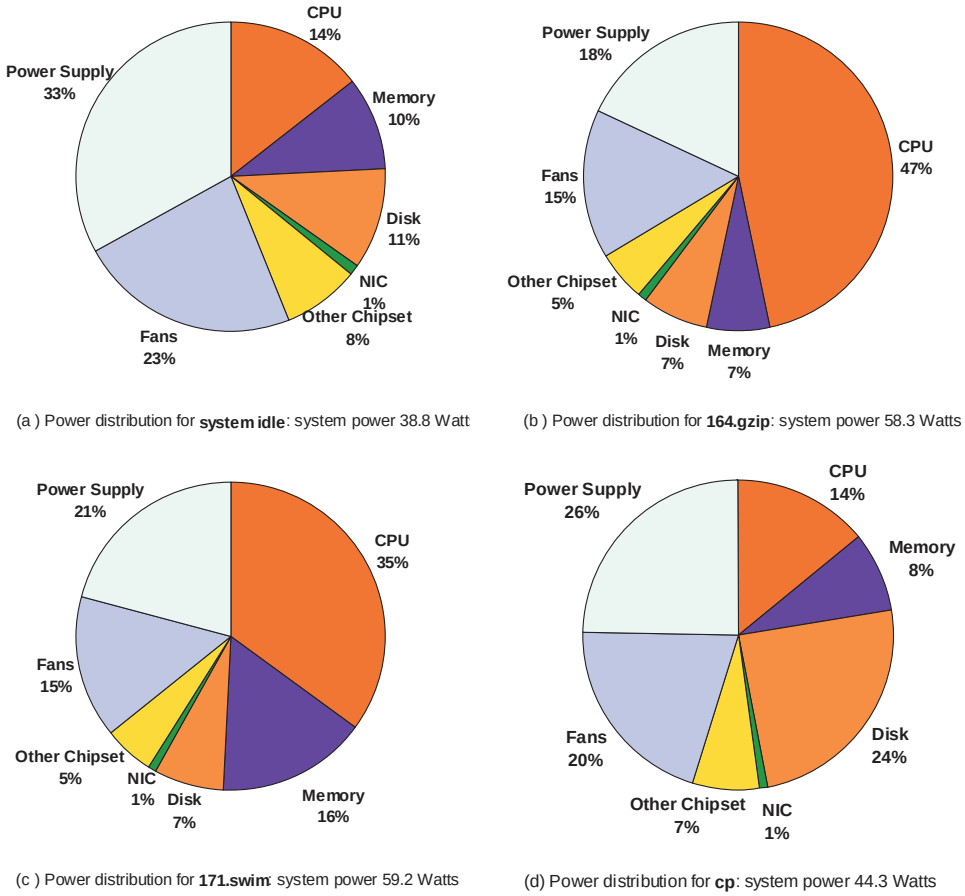


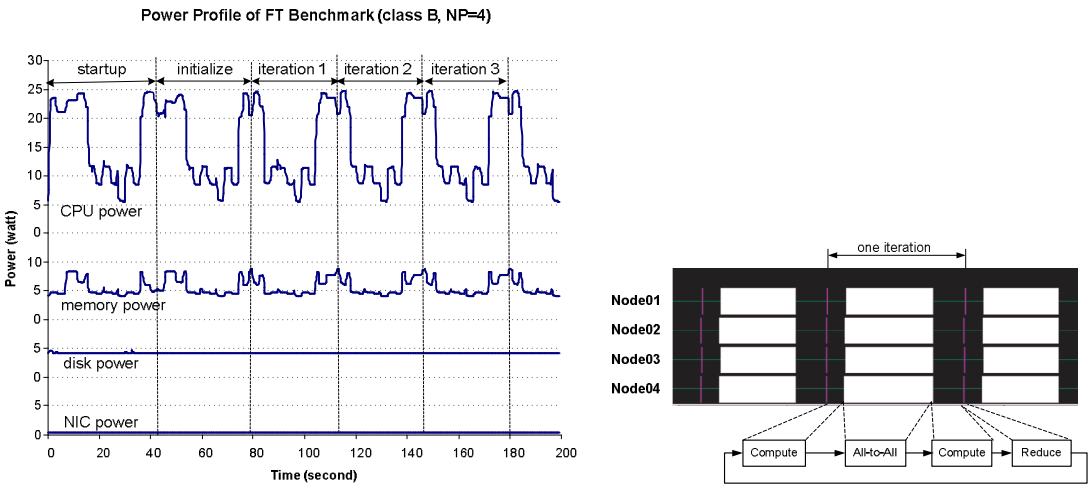
Figure 4.5: Power distribution for a single work node under different workloads. (a) zero workload (system is in idle state); (b) CPU bounded workload; (c) memory bounded workload; (d) disk bounded workload.

4.4 Power Profiles of Distributed Applications

As a case study and proof of concept, we profile the power-energy consumption of the NAS parallel benchmarks (Version 2.4.1) on the 32-node Beowulf cluster using the PowerPack prototype. The NAS parallel benchmarks [12] consist of 5 kernels and 3 pseudo-applications that mimic the computation and data movement characteristics of parallel computational fluid dynamics (CFD)

applications. We measured CPU, memory, NIC and disk power consumption over time for different benchmarks running on different numbers of compute nodes. We ignore power consumed by the power supply and the cooling system since they are roughly constant and machine dependent.

4.4.1 Nodal Power Profile of the FT Benchmark



(a) (b)

Figure 4.6: Power and performance profile of FT benchmark

The FT benchmark begins with a warm up phase and an initialization phase followed by a certain number of iterations, each iteration consisting of computation (fft), all-to-all communication, computation, and reduce communication.

In Figure 4.6 (a), we plot the first 200 seconds power profile of the NPB FT benchmark with problem size B when running on 4 nodes, and in Figure 4.6 (b) we show the annotated perfor-

mance profile generated from MPI profile tools. For ease of presentation, the x-axis is overlaid in Figure 4.6 (a).

The power profiles are identical for all iterations in which spikes and valleys occur with regular patterns coinciding with the characteristics of different computation stages. In other words, there exist apparent "power phases" corresponding to the workload phases (or stages). The CPU power consumption varies from 25 watts in the computation stage to 6 watts in all-to-all communication stage. The memory power consumption has the same trend with CPU power consumption, varying from 9 watts in the computation stage to 4 watts in communication-stage. The power profiles of CPU and memory are related in that when memory power goes up, CPU power goes down and the inverse is also observed.

We also measured constant power consumption for the disk since the FT benchmark requires few disk accesses. As discussed earlier, the power consumed by the NIC is constant (0.41 watt under our assumption). For simplification, we ignore the disk and NIC power consumption in discussions and figures where they do not change.

4.4.2 Mapping Power Profile to Source Code

PowerPack can correlate an application's power profile to its source code, thereby allowing us to study the power behavior of a specific function or code segment. Figure 4.7 shows the mapping between the power profile and the major functions of the FT benchmark. From this figure, we observe the power variations for functions that are computing intensive or memory intensive or communication intensive. Using code analysis and code-power profile synchronization mecha-

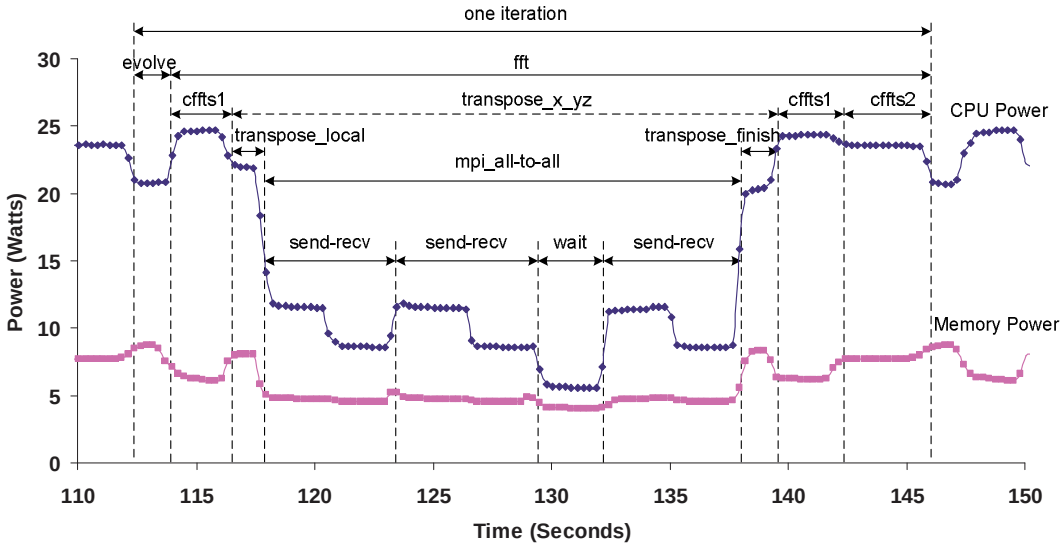


Figure 4.7: Mapping between power profile and code segments for FT benchmark

nisms provided in PowerPack, we can map the power phases to each individual function and perform detailed power-efficiency analysis on selected code segments. This is useful when exploring function-level power-performance optimization.

Using this power-to-code mapping, we can collect power statistics (such as average power, total energy consumption) for each function and perform detailed power-efficiency analysis on selected code segments. This is useful when exploring function-level power-performance optimization. For example, we can pinpoint which function needs to be optimized for better power-performance efficiency and to evaluate how much benefit can be obtained from such an optimization.

4.4.3 Power Profile Variation with Node and System Size

Using the node remapping technique described earlier, PowerPack can provide power profiles for all nodes running the parallel applications in the cluster. For the FT benchmark, as workload is distributed evenly across all working nodes, there are no significant differences for the profile among different nodes. However, for applications with imbalanced workload distribution, the power profiles for different nodes may vary in terms of power phases and values.

The power profile of parallel applications also varies with the number of nodes used in the execution when we fix the problem size. Scaling a fixed workload to an increasing number of nodes may change the workload characteristics (the percentage of CPU computation, memory access and message communication) and the change is reflected in the power profile. We have profiled the power consumption for all NPB benchmarks with different combinations of number of computing processors (up to 32) and problem sizes. In Figure 4.8(a)-(c), we provide an overview of the profile variations under different system scales for benchmarks FT, EP, and MG. These figures show segments of synchronized power profiles for different numbers of nodes; all power profiles correspond to the same computing phase in the application on the same node. For FT and MG, the profiles are similar for different system scale except the average power decreases with the number of execution nodes; for EP, the power profile is identical for all execution nodes.

4.5 Energy Efficiency of Parallel Applications

In this section, we apply PowerPack to analyze the energy efficiency of parallel applications. While power (P) describes the rate of energy consumption at a discrete point in time, energy (E) specifies the total number of joules spent in time interval (t_1, t_2), as a product of the average power ($\bar{P}$) and delay ($D = t_2 - t_1$):

$$E = \int_{t_1}^{t_2} P(t)dt = \bar{P} \times D\,. \tag{4.1}$$

4.5.1 Energy Scaling

Equation (4.1) specifies the relation between power, delay and energy. To reduce energy, we need to reduce the delay, the average power, or both. In the context of parallel processing, by increasing the number of processors, we can speedup the application but also increase the total power consumption. Depending on the parallel scalability of the application, the energy consumed by an application may be constant, grow slowly or grow very quickly with the number of processors.

For distributed parallel applications, we would like to use (E) to reflect energy efficiency, and use (D) to reflect the performance efficiency. To compare the energy-performance behavior of different parallel applications such as NPB benchmarks, we use two metrics: a) the speedup (D_1/D_N) where D_1 is the execution time running on 1 processor, and D_N is the execution time running on N processors in parallel; and b) normalized system energy (E_N/E_1), or the ratio of energy for single node configuration and multi-node configuration. Plotting these two metrics on the same graph with x-axis as the number of nodes, we identify three energy-performance categories for the code we measured.

Type I: energy remains constant or approximately constant while performance increases linearly. EP, SP, LU and BT belong to this type (see Figure 4.9a).

Type II: both energy and performance increase but performance increases faster. MG and CG belong to this type (see Figure 4.9b).

Type III: both energy and performance increase but energy consumption increases faster. FT and IS belong to this type. For small problem sizes, the IS benchmark gains little in performance speedup by using more nodes but consumes much more energy (see Figure 4.9c).

Our further analysis indicates that that energy scaling (i.e. efficiency) of parallel applications is strongly tied to parallel scalability. In other words, as applications have good scalability, they also make more efficient use of the energy where using more nodes. For example, given an embarrassingly parallel application such as EP, total energy consumption remains constant as we scale the number of nodes to improve the performance.

4.5.2 Resource Scheduling

An application's energy efficiency is dependent on its speedup or parallel efficiency. For certain applications such as FT and MG, we can achieve speedup by running on more processors while increasing total energy consumption. The question remains whether the performance gain was worth the additional resource requirement. Our measurements indicate there are tradeoffs between power, energy, and performance that should be considered to determine the best resource "operating points" or the best configurations in number of nodes (NP) based on the user's needs.

For performance-constrained systems, the best operating points will be those that minimize delay (D). For power-constrained systems, the best operating points will be those that minimize power (P) or energy (E). For systems where power-performance must be balanced, the choice of appropriate metric is subjective. The energy-delay product ED^{α} (α is real number and $\alpha \geq -1$) is commonly used as a single metric to weight the effects of power and performance for given application under different configurations.

Figure (4.10) presents the relationship between four metrics (normalized E and D, EDP, $ED2P$) and the number of nodes for NPB MG benchmark (class A). To minimize energy (E), the system should schedule only one node to run the application which corresponds in this case to the worst performance. To minimize delay (D), the system should schedule 32 nodes to run the application which achieves 6 times speedup and consumes 4 times the energy. For power-performance efficiency, the EDP metric suggests 8 nodes for a speedup of 2.7 and an energy cost of 1.7 times the energy of 1 node. Using the ED2P metric suggests 16 nodes for a speedup of 4.1 and an energy cost of 2.4 times the energy of 1 node. For accuracy, the average delay and energy consumption obtained from multiple runs are used in Figure (4.10).

4.6 Performance Profile Based Power Estimation

Direct power measurement is fast and accurate. However it is cumbersome and expensive when applied to large systems with thousands of nodes. On the other hand, performance counter based performance profiling is relatively easier and cheaper. Thus, we attempt to approximate the power profile of large systems from performance counter measurements.

4.6.1 Empirical Power Model

From an architectural perspective, we can divide a system component such as CPU, memory and disk into a set of lower level functional units. For example, we can view the CPU as system component consisting of integer register files, floating point register files, instruction fetch, instruction queue, instruction decode, L1 cache, L2 cache, TBL, bus control and other units.

Empirically, we can calculate the component power as the sum of the power consumed by all units that belong to the component c, i.e.,

$$P(c) = \sum_{i=1}^{S} P_i \,, \tag{4.2}$$

and estimate each unit's power (P_i) from its access rate (R_i), i.e.,

$$P_i = P_i^{idle} + P_i^{active} = P_i^{idle} \,. \tag{4.3}$$

Here, access rate (R_i) is the total number of accesses to the i^{th} unit per unit time interval. The relations between unit power P_i and unit access rate (R_i) may be linear or nonlinear depending on the circuit design. Similar to the method adopted by Isci and Martonosi [93], we approximate a nonlinear power-access rate relation using a piecewise linear function, i.e.,

$$P_i = \sum_{k} \left(\alpha_{i,k} + \beta_{i,k} \cdot R_{i,k}\right) \cdot \delta_k \,, \tag{4.4}$$

where k is time index, α and β are coefficients, and δ is either 0 or 1 depending on whether unit i is accessed or not for the time interval $[k, k+1)$.

$$\delta_k = \begin{cases} 1 & R_{i,k} \geq 0 \\ 0 & otherwise \end{cases} \,. \tag{4.5}$$

For a linear power-access relation, Equation (4.4) becomes

$$P_i = \alpha_i + \beta_i \cdot R_i \,. \tag{4.6}$$

4.6.2 Power Estimation Methodology

Equation 4.2 and 4.3 form the basis of performance profile based power estimation. This power estimation method can be implemented as two steps:

First, we quantify the relation between power and unit access rate. We directly measure the component power using the PowerPack prototype and record performance events using performance counters provided by the computer system for a same set of benchmarks. As a result, we obtain a stream of power and performance data $(P_c^t, R_{c,1}^t, R_{c,2}^t, \ldots, R_{c,m}^t)$ for each component c, where P_c^t is the power consumption of component c at time t, and $R_{c,i}^t$ is the access rate of the i^{th} unit of component c at time t. Then we can derive the values for the parameter in the empirical power model represented by Equation 4.4 by applying statistical learning methods such as least square estimation.

Second, we profile the performance events using hardware counters on all compute nodes of the distributed system and then use the performance profile to approximate the component power profile by applying the empirical model and parameters determined in the first step.

For simplification, we refer to both power consumption and performance activities as system-wide which include the contributions from the application side and the operating system side as well.

A practical limitation is that currently most systems only support a limited number of performance counters and some events can not be combined during profiling. As a solution, we run the application multiple times and profile a few events each time. For online power estimation using performance counters, we can multiplex events on available performance counters to approximate the performance profile with a single run.

4.6.3 Experimental Validation

Figure (4.11) and (4.12) show the power profile comparison between direct PowerPack measurement and performance profile based estimation for the NPB FT benchmark on one compute node in the Beowulf cluster. We profile in total 6 performance events: number of instructioins, floating points per seconds, L1 access, L1 Data Read, L2 Data Write,and Memory Reference using hardware counters. The sample interval is 1 second. We run the same benchmark 6 times and each time we profile one event. Since the sample interval for the power profile is different from the sample interval for the performance profile, we approximate the average power at a time interval from the measured power samples. We assume a linear relation between access rate and the power consumption for the selected performance event, and calculate the model parameters using the least square estimation method.

Figure (4.11) shows that for memory power, the estimated power profile matches the measured power profile well during both memory access intensive phases and memory access inactive phases. In Figure (4.12), we observed that for CPU power, there is about 5% difference between the estimated power profile and the measured power profile. We hypothesize that these differences

are due to profile samples alignment variations of multiple runs and the linear assumption of the power-access relation. Overall, the experimental results support performance profile based power estimation as being both feasible and accurate.

4.7 Chapter Summary

Power-energy measurement and profiling is a key component in power-aware computing. In this chapter we presented PowerPack for power-performance profiling and evaluation of distributed systems and applications. The PowerPack framework supports power profiling at component level and function granularity. By combining power estimation from performance events, we are able to scale PowerPack to very large distributed parallel systems. We have applied a PowerPack prototype for several case studies for profiling parallel benchmarks on traditional Beowulf clusters.

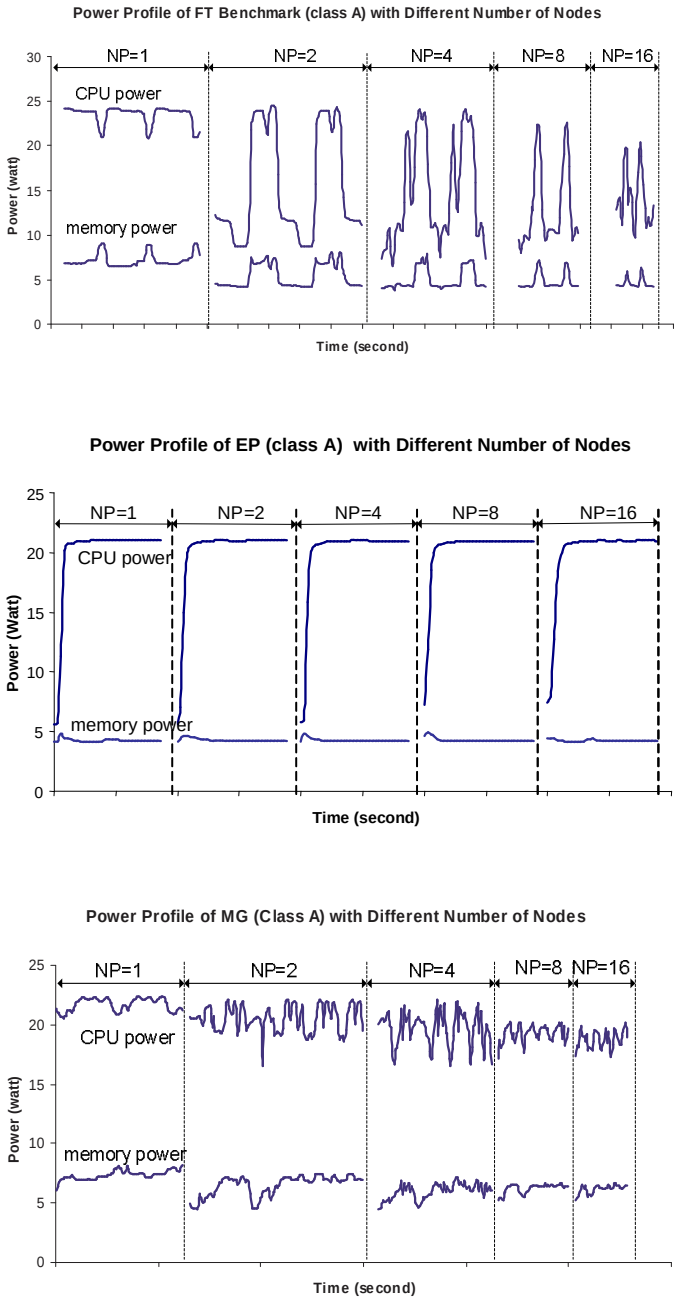


Figure 4.8: Power profile of representative NPB code. The power profiles for FT, EP and MG on different numbers of work nodes are presented.

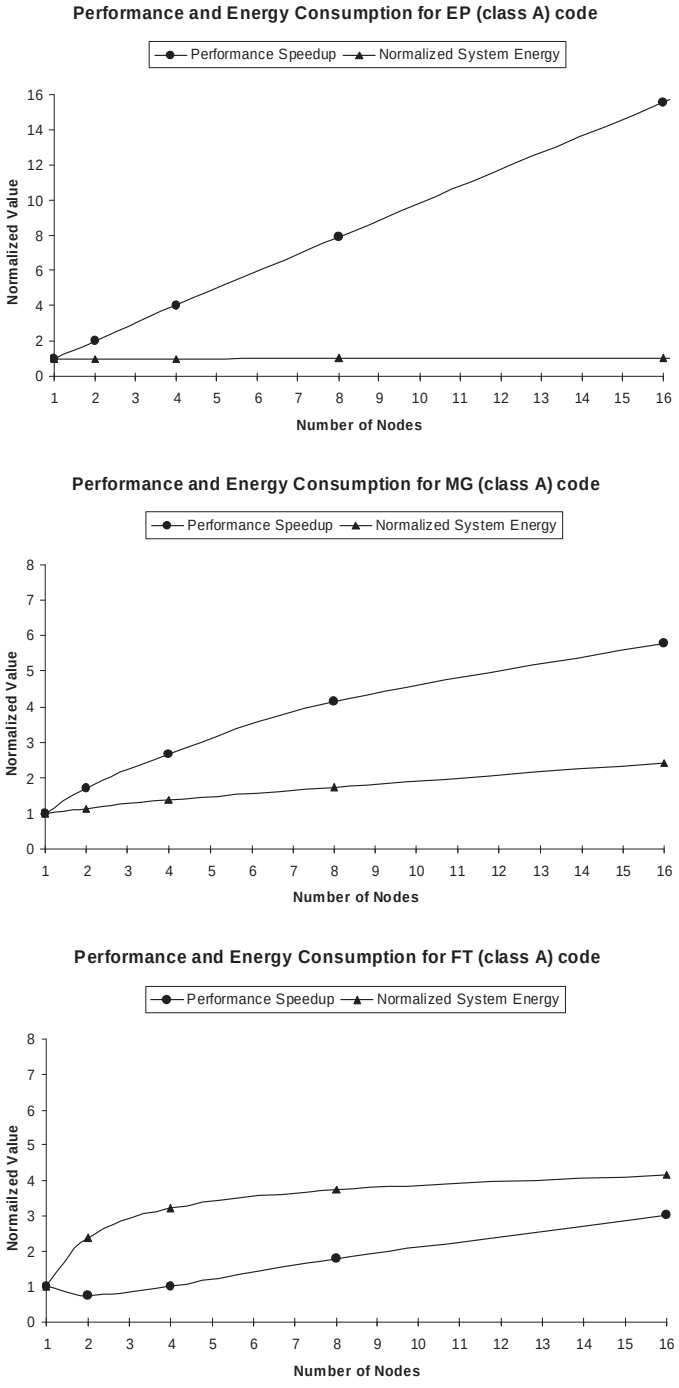


Figure 4.9: Energy-performance efficiency. These graphs use normalized values for performance (i.e. speedup) and total system energy. (a) EP shows linear performance improvement with constant energy consumption. (b) MG is capable of some speedup with the number of nodes with a corresponding increase in the amount of total system energy. (c) FT shows only minor performance improvement but significant increase in total system energy.

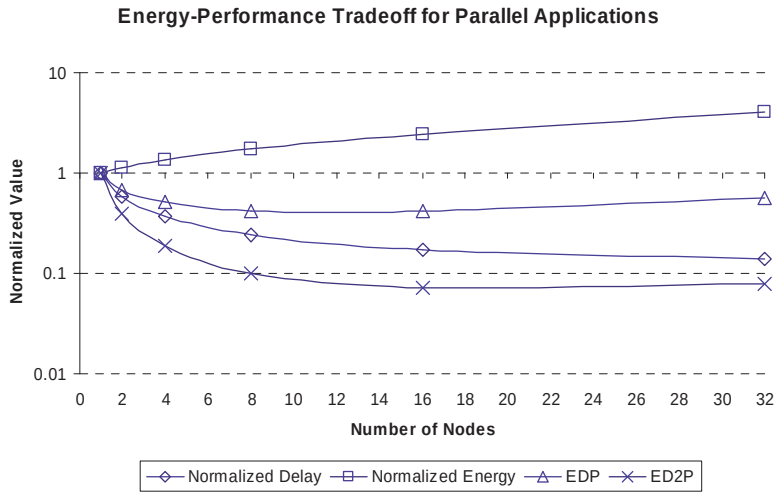


Figure 4.10: Energy-performance tradeoffs. Note: logarithm scale is used for y-axis.

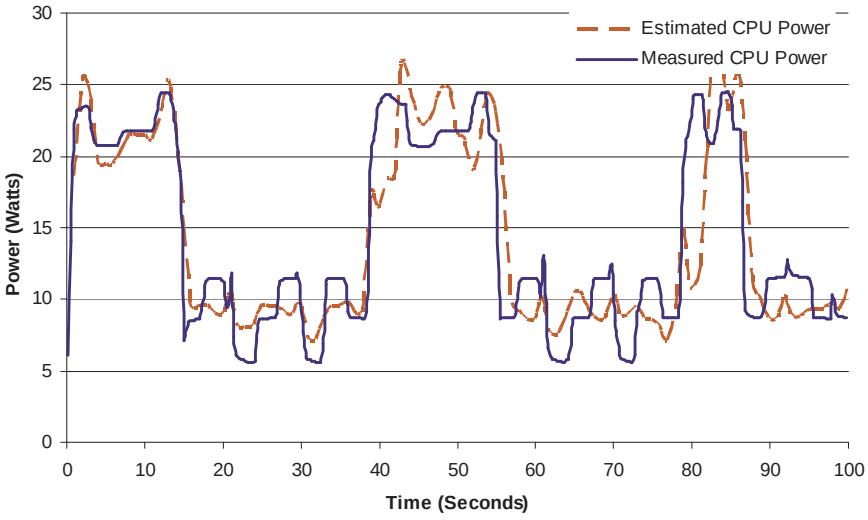


Figure 4.11: Estimated CPU power from performance events

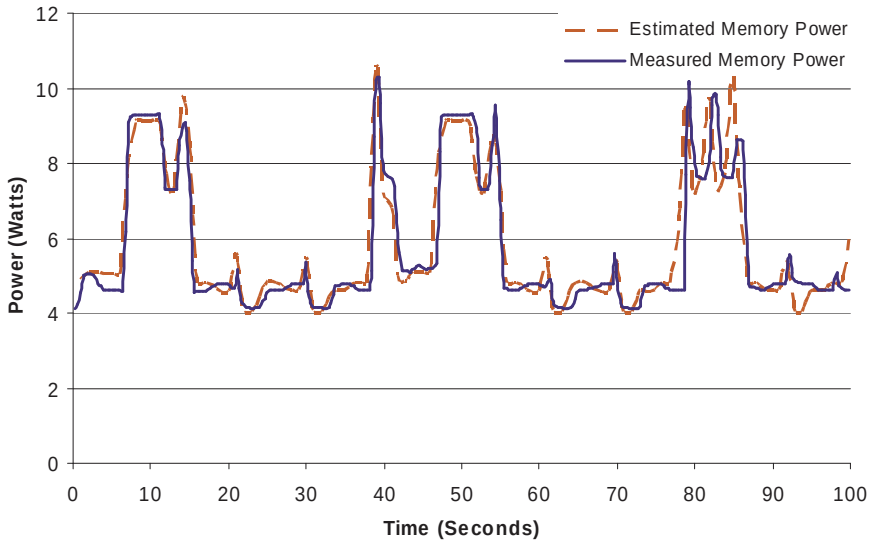


Figure 4.12: Estimated memory power from performance events

Chapter 5

Performance Analysis of Distributed Applications on Power Scalable Clusters

This chapter presents a performance model for applications running on power scalable high-end computing systems. By analytically quantifying the performance effects of parallelism, power/performance modes, and their combination, this model forms the theoretical foundation for performance and efficiency evaluation. In this chapter, we focus on the model and its application to performance and efficiency evaluation. We show how to use the model to design power-aware schedulers in Chapter 7.

5.1 Introduction

Amdahl's Law, or the law of diminishing returns, is a parallel speedup model commonly used by the research community. The basic idea is that any system enhancement is only applicable to a certain portion of a workload. For parallel computing, the increased number of nodes is considered

the enhancement and speedup S is often defined as the ratio of sequential to parallel execution time:

$$S_N(w) = \frac{T_1(w)}{T_N(w)}\,, \tag{5.1}$$

where,

w: the workload or total amount of work (in instructions or computations),

$T_1(w)$: the sequential execution time or the amount of time to complete workload w on 1 processor, and

$T_N(w)$: the parallel execution time or the amount of time to complete workload w on N processors.

If the fraction of enhanced workload (FE) is the portion of the total workload that is parallelizable, and the enhancement will reduce execution time of the parallelizable workload portion by a speedup factor (SE), the parallel speedup for the entire workload can be expressed [9, 122] as:

$$S_N(w) = \frac{T_1(w)}{T_n(w)} = \left[(1 - FE) + \frac{FE}{SE}\right]^{-1}, \tag{5.2}$$

For e enhancements where $e >= 1$, we can generalize Equation 5.2 [122] as:

$$S_N(w) = \prod_e \left[(1 - FE_e) + \frac{FE_e}{SE_e}\right]^{-1}, \tag{5.3}$$

Equation 5.3 states that the speedup for a workload using e simultaneous enhancements is the product of the individual speedups for each enhancement. This generalization of Amdahl's Law is the only available speedup model that considers multiple enhancements simultaneously. Thus, we investigate its suitability for power scalable parallel systems.

Table 5.1: Performance evaluation using Amdahl's Law. To determine the best system configuration for FT for all combinations of frequency and processor count, we need pairwise speedup comparisons to the slowest frequency (600 MHz) and the smallest number of nodes ($N = 1$) as the base sequential execution time. To predict speedup, we use Equation 5.3 for $e = 2$ variables. Each table entry is the relative error. 600 MHz is used as the basis for comparison, so its column shows no error since it effectively varies only with number of nodes, exemplifying traditional speedup. Errors occur when trying to compare the results for two enhancements simultaneously since their effects are interdependent and not modeled by Equation 5.3.

N	Frequency (MHz)				
	600	800	1000	1200	1400
2	0%	30%	31%	49%	66%
4	0%	18%	36%	42%	58%
8	0%	30%	41%	59%	78%
16	0%	26%	40%	54%	72%

In power scalable clusters, a typical goal is to maximize performance while minimizing power consumption. Speedup models can be used to predict performance, and thus identify "sweet spot" system configurations of processor count and frequency that meet these constraints. If the performance or speedup prediction is accurate, we can either select the best speedup across all the data, or use the execution time predictions in an energy-delay metric [19] to determine the tradeoffs between performance and energy.

We use the speedup model in Equation 5.3 to predict the simultaneous effects of processor count and frequency on speedup relative to the lowest processor frequency (600 MHz) and smallest number of processors ($N = 1$). Table 5.1 shows the speedup prediction errors of a parallel Fourier transform (FT) application. Each table entry is the relative errors for prediction against actual measured speedup. Due to the large errors between predicted and measured speedup, identifying "sweet spot" system configurations using Equation 5.3 for multiple enhancements is problematic.

Equation 5.3 overpredicts speedup on power scalable clusters since it assumes the effects of multiple enhancements are independent. Power scalable clusters and applications violate this assumption since parallel overhead depends on processor count and influences the effects of frequency scaling. Use of Equation 5.3 to model FT on a 16-node power scalable cluster gives errors as large as 78%; 45% on average. In our power scalable cluster work, we combine the effects of processor count and frequency into a metric that captures and explains their simultaneous effects on execution time. Ultimately, we would like to predict these effects for a given processor count and frequency. To this end, we propose power-aware speedup and denote it as:

$$S_N(w, f) = \frac{T_1(w, f = f_0)}{T_N(w, f)}\,, \tag{5.4}$$

where

w: the *workload* or total amount of work (in instructions or computations),

f: the clock *frequency* in clock cycles per second and f_0 is the base frequency,

$T_1(w, f)$: the *sequential execution time* or the amount of time to complete workload w on 1 processor for frequency f, and

$T_N(w, f)$: the *parallel execution time* or the amount of time to complete workload w on N processors for frequency f.

Power-aware speedup is the ratio of sequential execution time for a workload (w) and frequency (f) on 1 processor to the parallel execution time for a workload running on N processors. In this work we focus on the situation that all processors run at the same frequency. In the next section we detail the additional equations necessary to quantify the execution times of Equation 5.4. In

succeeding sections, we show how our model improves the error rates shown in Table 5.1. We identify the key differences between power-aware speedup and equations 5.1-5.3 (Amdahl's Law).

5.2 A Performance Model for Power-Aware Clusters

In this section, our goal is to describe power-aware speedup as simply as we can. We will use the terms defined by Equation 5.4 and introduce definitions as needed to understand each derivation step and then use the defined terms to express equations that build on one another.

Sequential execution time for a single workload $(T_1(w, f))$

CPI: the average number of clock *cycles per workload*.

Using this definition and others from Equation 5.4, sequential execution time is

$$T_1(w, f) = w\frac{CPI}{f}\,. \tag{5.5}$$

This is a variant of the CPU performance equation [122]. The time to execute a program on 1 processor is the product of the workload (w) and the rate at which workloads execute (CPI/f or seconds per workload). For now, we assume f is a fixed value, noting that $T_1(w, f)$ depends on the processor frequency.

Sequential execution time for an ON-chip/OFF-chip workload $(T_1(w^{ON}, f^{ON}))$**,** $T_1(w^{OFF}, f^{OFF})$

w^{ON}: ON-chip *workload*, or the workload portion that does not require data residing OFF-chip at the time of execution.

w^{OFF}: OFF-chip *workload*, or the workload portion that requires OFF-chip data accesses at the time of execution.

f^{ON}: ON-chip clock *frequency* in clock cycles per second. Affected by processor DVFS.

f^{OFF}: OFF-chip clock *frequency* in clock cycles per second. Not affected by processor DVFS.

CPI^{ON}, CPI^{OFF}: the average number of clock cycles per ON-chip (CPI^{ON}) or OFF-chip (CPI^{OFF}) workload.

Others have shown [37] that a given workload (w) can be divided into ON-chip (w^{ON}) workload and OFF-chip (w^{OFF}) workload. Under these constraints, the total amount of work (in instructions or computations) is given as $w = w^{ON} + w^{OFF}$. We can modify our simple representation of sequential execution time[1] as:

$$T_1(w, f) = T_1(w^{ON}, f^{ON}) + T_1(w^{OFF}, f^{OFF}) = w^{ON}\frac{CPI^{ON}}{f^{ON}} + w^{OFF}\frac{CPI^{OFF}}{f^{OFF}}\,. \qquad (5.6)$$

Assuming ON-chip and OFF-chip frequencies are equal ($f^{ON} = f^{OFF}$), and $CPI = \frac{(CPI^{ON}+CPI^{O}FF)}{2}$, this equation reduces to Equation 5.5. We observe that generally for ON-chip and OFF-chip workloads $f^{ON} \neq f^{OFF}$, meaning CPU and memory bus frequencies differ, and $CPI^{ON} \neq CPI^{OFF}$, meaning the workload throughput is different for ON- and OFF-chip workloads.

Parallel execution time on N processors for an ON-/OFF-chip workload with $DOP = i$ ($T_N({w_i}^{ON})$, $T_N({w_i}^{OFF})$)

i: the degree of parallelism or DOP, defined as the maximum number of processors that can be busy computing a workload for an observation period given an unbounded number of processors.

m: the maximum DOP for an application encompassing workloads with various DOP.

w_i: the amount of work (in instructions or computations) with i as the DOP.

${w_i}^{ON}$: the number of ON-chip workloads with $DOP = i$.

[1] This does not account for out-of-order execution and overlap between memory access and computation, simplifying the discussion for now.

${w_i}^{OFF}$: the number of OFF-chip workloads with $DOP = i$.

N: the number of homogeneous processors available for computing the workloads.

w_{PO}: the parallel overhead workload due to extra work for communication, synchronization, etc.

$T(w_{PO}, f)$: the *execution time* for parallel overhead w_{PO} for frequency f.

$T_N(w, f)$: the *parallel execution time* or the amount of time to complete workload w on N processors for frequency f.

The total amount of work (in instructions or computations) is given as $w = \sum_{1 \le i \le m}({w_i}^{ON} + w_i^{OFF})$, where$1 \le i \le m$. Thus,

$$\begin{aligned} T_N(w_i, f) &= T_N(w_i^{ON}, f^{ON}) + T_N(w_i^{OFF}, f^{OFF}) \\ &= \frac{w_i^{ON}}{i} \cdot \frac{CPI^{ON}}{f^{ON}} + \frac{w_i^{OFF}}{i} \cdot \frac{CPI^{OFF}}{f^{OFF}}\,, \end{aligned} \tag{5.7}$$

where $m \le N$.[2] Next, we include the additional execution time $T(w_{PO}, f)$ for parallel overhead. We assume parallel overhead workload cannot be parallelized, but that it is divisible into ON-chip (${w_{PO}}^{ON}$) and OFF-chip (w_{PO}^{OFF}) workloads. Thus

$$T_N(w, f) = \sum_{i=1}^{m} \left(T_N(w_i^{ON}, f^{ON}) + T_N(w_i^{OFF}, f^{OFF}) \right) + T(w_{PO}, f)\,, \tag{5.8}$$

and

$$\begin{aligned} T_N(w, f) = &\sum_{i=1}^{m} \left(\frac{w_i^{ON}}{i} \cdot \frac{CPI^{ON}}{f^{ON}} + \frac{w_i^{OFF}}{i} \cdot \frac{CPI^{OFF}}{f^{OFF}} \right) \\ &+ \left(T(w_{PO}^{ON}, f^{ON}) + T(w_{PO}^{OFF}, f^{OFF}) \right). \end{aligned} \tag{5.9}$$

[2]Strictly speaking, this limitation is not required. For $m > N$, we can add an $\lceil i/N \rceil$ term to Equation 5.5 and succeeding equations to limit achievable speedup to the number of available processors, N. We omit this term to simplify the discussion and resulting formulae.

Power-aware speedup for DOP and ON-/OFF-chip workloads ($S_N(w, f)$)

f_0^{ON}: the lowest available ON-chip frequency.

$S_N(w, f)$: the ratio of sequential execution time ($T_1(w, f)$) to parallel execution time ($T_N(w, f)$).

On power-aware parallel systems, ON-chip frequency f^{ON} may change due to DVFS scheduling of the processor. As a consequence, power-aware speedup has two key variables: ON-chip clock frequency (f^{ON}) and the number of available processors (N) computing workload w. Speedup is computed relative to the sequential execution time to complete workload w on 1 processor at the lowest available ON-chip frequency, f_0^{ON}. Power-aware speedup is defined using Equations 5.6 and 5.9 as:

$$\begin{aligned} S_N(w, f) &= \frac{T_1(w, f)}{T_N(w, f)} \\ &= \left[w^{ON} \frac{CPI^{ON}}{f_0^{ON}} + w^{OFF} \frac{CPI^{OFF}}{f^{OFF}} \right] \Bigg/ \\ &\quad \left[\sum_{i=1}^{m} \left(\frac{w_i^{ON}}{i} \cdot \frac{CPI^{ON}}{f^{ON}} + \frac{w_i^{OFF}}{i} \cdot \frac{CPI^{OFF}}{f^{OFF}} \right) + \left(T(w_{PO}^{ON}, f^{ON}) + T(w_{PO}^{OFF}, f^{OFF}) \right) \right]. \end{aligned} \tag{5.10}$$

Usage of power-aware speedup ($S_N(w, f)$)

Equation 5.10 illustrates how to calculate power-aware speedup. For a more intuitive description, assume the workload is broken into a serial portion (w_1) and a perfect parallelizable portion (w_N) such that $w = w_1 + w_N$, $N = m$, and $w_i = 0$ for $i \neq 1$, $i \neq m$. Then, allowing for flexibility

in our execution time notation, we can express the power-aware speedup under these conditions as:

$$S_N(w,f) = \Big[T_1(w^{ON}, f_0^{ON}) + T_1(w^{OFF}, f^{OFF})\Big] \Big/ \Big[\big[T_N(w_1^{ON}, f^{ON}) + T_N(w_1^{OFF}, f^{OFF})\big] + \big[T_N(w_N^{ON}, f^{ON}) + T_N(w_N^{OFF}, f^{OFF})\big] + \big[T(w_{PO}^{ON}, f^{ON}) + T(w_{PO}^{OFF}, f^{OFF})\big]\Big], \tag{5.11}$$

here, $T_1(w^{ON}, f_0^{ON}) + T_1(w^{OFF}, f^{OFF})$ is the base line sequential execution time unaffected by CPU frequency scaling or parallelism. $T_N(w_1^{ON}, f^{ON})$ is the sequential portion of the workload affected by CPU frequency scaling, but not affected by parallelism. $T_N(w_1^{OFF}, f^{OFF})$ is the sequential portion of the workload not affected by CPU frequency scaling or parallelism. $T_N(w_N^{ON}, f^{ON})$ is the parallelizable portion of the workload also affected by CPU frequency. $T_N(w_N^{OFF}, f^{OFF})$ is the parallelizable portion of the workload not affected by CPU frequency. $T(w_{PO}^{ON}, f^{ON})$ is the parallel overhead affected by CPU frequency. $T(w_{PO}^{OFF}, f^{OFF})$ is the parallel overhead not affected by CPU frequency.

5.3 Model Validation

In this section, we analyze the power-aware speedup for two classes of applications: computation-bound applications with negligible parallel overhead and communication-bound applications with significant parallel overhead. We use the embarrassingly parallel (EP) and Fourier transform (FT) benchmarks from the NAS Parallel Benchmark suite [11] for each category respectively. We note that our intention here is to show the accuracy of our approach for analytically quantifying the impact of power-aware features on execution time and speedup. We start with EP since the results

Table 5.2: Operating points in frequency and supply voltage for the Pentium M 1.4GHz processor.

Frequency	Supply Voltage
1.4GHz	1.484V
1.2GHz	1.436V
1.0GHz	1.308V
800MHz	1.180V
600MHz	0.956V

are straightforward and as proof of concept for power-aware speedup. We describe results for more interesting codes such as FT in the succeeding subsection.

5.3.1 Experimental Platform

The power-aware system used in these experiments is a 16-node DVS-enabled cluster. It is constructed with 16 Dell Inspiron 8600s connected by a 100M Cisco System Catalyst 2950 switch. Each node is equipped with a 1.4 GHz Intel Pentium M processor using Centrino mobile technology to provide high-performance with reduced power consumption. The processor includes an on-die 32K L1 data cache, a on-die 1 MB L2 cache, and each node has 1 GB DDR SDRAM. Enhanced Intel Speedstep technology allows software to dynamically adjust the processor among five supply voltage and clock frequency settings given by Table 5.2. We installed open-source Linux Fedora Core and MPICH for communication on each node.

5.3.2 Computation-Bound Benchmark EP

Figure 5.1 shows the measured parallel execution time and power-aware speedup for the EP benchmark. EP evaluates an integral using a pseudorandom trial. Cluster-wide computations require virtually no inter-processor communication. The ratio of memory operations to computations on each node is very low. Figure 5.1 indicates the following for the EP workload:

1. Execution time (Figure 5.1a) for a fixed frequency can be reduced by increasing the number of nodes used in computations.
2. Execution time (Figure 5.1a) for a fixed processor count can be reduced by increasing the CPU clock rate.
3. Speedup (Figure 5.1b) for a fixed base frequency (600MHz) increases linearly with the number of processors. For instance, speedup increases from 1 at 1 processor to 15.9 at 16 processors.
4. Speedup (Figure 5.1b) for 1 processor increases linearly with processor frequency. For instance, speedup increases from 1.0 at 600MHz to 2.34 at 1400MHz.
5. The overall speedup using simultaneous enhancements of processor count and frequency is nearly the product of the individual speedups for each enhancement. For instance, the maximum speedup (36.5) measured on 16 processors for 1400MHz is almost equal to the product of measured parallel speedup (15.9) and frequency speedup (2.34).

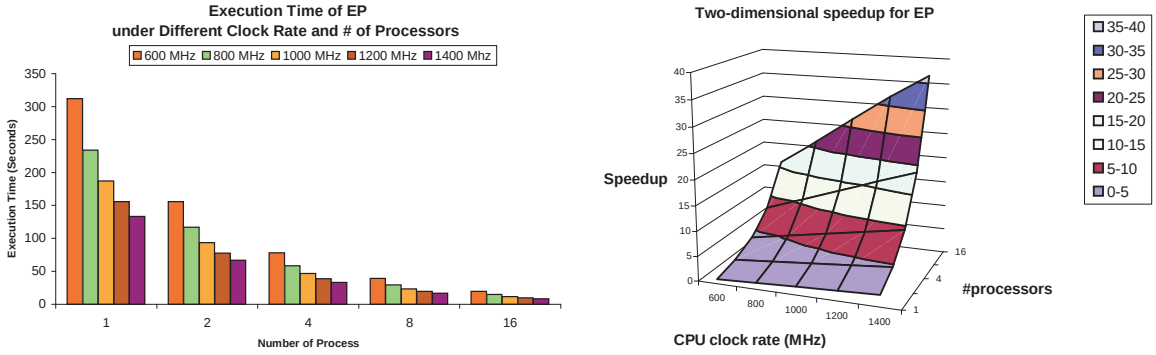
We now use our power-aware speedup formulation to explain these observations analytically. A computation-bound application such as EP spends the majority of its execution time doing calculations on the CPU, and the time spent performing OFF-chip (i.e. memory) accesses is negligible.

Thus, the workload w is only an ON-chip workload, or more formally $w = \sum_{i=1}^{m} w_i^{ON}$. The OFF-chip portion of the workload is negligible, or $\sum_{i=1}^{m} w_i^{OFF} = 0$. The characteristics of EP also indicate a majority of the workload can be completely parallelized, such that $w = \sum_{i=1}^{m} w_i = w_N$, where $N = m$, and $w_i = 0$ for $i \neq m$. Hence, $w = \sum_{i=1}^{m} w_i^{ON} = w_N^{ON}$, and since EP exhibits almost no inter-processor communication, $w_{PO}^{ON} = w_{PO}^{OFF} = 0$. Under these assumptions, the analytical power-aware speedup for EP using Equation 5.11 is

$$S_N(w, f) = \frac{T_1(w, f_0)}{T_N(w, f)} = \frac{w_N^{ON} \frac{CPI^{ON}}{f_0^{ON}}}{\frac{w_N^{ON}}{N} \cdot \frac{CPI^{ON}}{f^{ON}}} = N \cdot \frac{f^{ON}}{f_0^{ON}} \,. \tag{5.12}$$

The EP application exhibits near perfect performance: easily parallelized workload, no overhead for communication, and nearly ideal memory behavior. Thus, the speedup predicted by Equation 5.12 is a simple product of the individual speedups for parallelism (N), and for frequency (f^{ON}/f_0^{ON}), where we are comparing a faster frequency (e.g. $f^{ON} = 1400$) to the base frequency ($f_0^{ON} = 600$). The predicted speedup for 16 processors (37.3) is within 2.3% of the measured speedup (36.5), and this error is the maximum error over all the predictions for EP.

Predicting the power-aware speedup for EP makes a reasonable case for using the product of individual speedups described by Equation 5.3 (Amdahl's Law generalization). The speedup of embarrassingly parallel (EP) applications with small memory footprints will always improve with increased processor count and frequency. Though EP prediction shows our methods are as accurate and useful as Amdahl's Law, this behavior is not typical of many parallel scientific applications such as FT. In the next subsections we use power-aware speedup techniques to analyze codes with significant parallel overhead (FT) and more complex memory behavior.



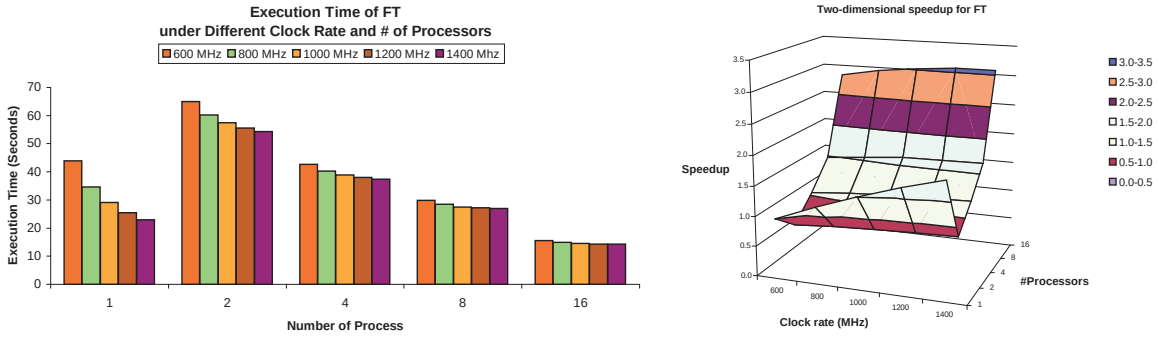
(a) (b)

Figure 5.1: Execution time and two-dimensional speedup of EP. (a) shows measured parallel execution time with varying clock rate; (b) shows the speedup for scaled processor counts and frequencies.

5.3.3 Communication-bound Benchmark FT

Figure 5.2 shows the measured parallel execution time and power-aware speedup for the FT benchmark. FT computes a 3-D partial differential equation solution using fast Fourier Transforms. Parallel FT iterates through four phases: computation phase 1, reduction phase, computation phase 2, and all-to-all communication phase. Both computation phases spend most of their time performing calculations but with a larger memory footprint than EP. The parallel overhead of the reduction and all-to-all communication phases dominate execution time. Figure 5.2 indicates the following for the FT workload:

1. Execution time (Figure 5.2a) for 2 or more processors is reduced by increasing the number of processors used in computations. However, the rate of improvement is sub-linear.
2. Execution time (Figure 5.2a) for 1 processor can be reduced by increasing CPU clock rate. However, the rate of improvement is sub-linear; from 1.0 at 600MHz to 1.9 at 1400MHz.



(a) **(b)**

Figure 5.2: Execution time and two-dimensional speedup of FT. (a) shows measured parallel execution time with varying clock rate; (b) shows the speedup for scaled processor counts and frequencies.

3. Speedup (Figure 5.2b) for a fixed base frequency (600 MHz) decreases from 1 to 2 processors. Speedup increases from 2 to 16 processors. For instance, for 600MHz speedup increases from 1.0 on 1 processor to 2.9 on 16 processors.
4. Speedup (Figure 5.2b) for 1 processor increases sub-linearly with processor frequency. For instance, speedup increases from 1.0 at 600MHz to 1.6 at 1400MHz.
5. The overall speedup using simultaneous enhancements of processor count and frequency is a complicated function. For example, the effects of frequency scaling on execution time (Figure 5.2a) diminish as the number of nodes increase.

We now use our power-aware speedup formulation to explain these observations analytically. First, we consider the workload run sequentially at various CPU clock frequencies. As mentioned, execution time decreases sub-linearly. This behavior differs from EP where the effects were linear. The memory behavior of FT, computing transforms on sizable matrices, takes more time and more

OFF-chip accesses than EP. Thus, we cannot simplify the numerator of Equation 5.11 since we must consider both ON-chip and OFF-chip delays in the workload.

Next, we consider the parallel overhead. A communication-bound application such as FT spends the majority of its execution time performing communications or parallel overhead, w_{PO}. From 1 to 2 processors, the execution time increases for all frequencies. This indicates the parallel overhead is significant and we must determine its effects on execution time. The parallel overhead for FT is actually dominated by all-to-all communications and synchronization. Such communication overhead is not affected significantly by CPU clock frequency. Thus, we claim ${w_{PO}}^{ON} = 0$, but ${w_{PO}}^{OFF}$ is a significant portion of parallel execution time.

Last, we consider the effects of parallelism. As the number of nodes increases, execution time decreases. This indicates that a good portion of the workload is parallelizable. However, the return from parallelism decreases and speedup tends to flatten out as the number of nodes increases. We observed speedup does not change significantly from 16 to 32 nodes. Given this observation, we assume DOP of ft $m = 16 = N$ reasonable assumption [3] . Under this assumption, the analytical power-aware speedup for FT using Equation 5.11 is

$$S_N(w,f) = \frac{T_1(w,f)}{T_N(w,f)} = \frac{w^{ON}\frac{CPI^{ON}}{f_0^{ON}} + w^{OFF}\frac{CPI^{OFF}}{f^{OFF}}}{\sum_{i=1}^{16}\left(\frac{w_i^{ON}}{i}\cdot\frac{CPI^{ON}}{f^{ON}} + \frac{w_i^{OFF}}{i}\cdot\frac{CPI^{OFF}}{f^{OFF}}\right) + T(w_{PO}^{OFF}, f^{OFF})}. \quad (5.13)$$

This application exhibits less than perfect performance: workload with limited parallelization, significant overhead for communication, and time consuming memory behavior. Thus, the speedup

[3] Admittedly, it would be nice to confirm this result on a larger power-aware cluster. However, at the time of this work, ours was one of only a few power-aware clusters in the US and there are few (if any) larger than 16 or 32 nodes. We are attempting to acquire a larger machine presently.

Table 5.3: Speedup prediction for FT using power-aware speedup. To predict speedup, we use Equation 5.13. Each table entry is the error or the difference between the measured and predicted speedup divided by the measured speedup. 600 MHz is used as the basis for comparison, so its column shows no error since it effectively varies only with number of nodes, exemplifying traditional speedup.

N	Frequency (MHz)				
	600	800	1000	1200	1400
2	0%	0.2%	0.2%	0.2%	0.3%
4	0%	0%	0.1%	0.1%	0.2%
8	0%	0.4%	2.0%	1.2%	0.7%
16	0%	2.1%	2.2%	2.3%	1.4%

predicted by Equation 5.13 is not a simple product of the individual speedups for parallelism and frequency as it was for EP.

Table 5.3 shows the errors for prediction of FT power-aware speedup using Equation 5.13. Here, the errors are reduced to a maximum of 3%, compared to errors in table 5.1 from Amdahl's law. The power-aware speedup for FT captures all of the empirical observations we noted. For example, the diminishing effects of frequency scaling as number of nodes scale is due primarily to the increasing impact of parallel overhead ($T(w_{PO}^{OFF}, f^{OFF})$). For small numbers of nodes, the effects are lessened since the ON-chip workload $\sum_{i=1}^{16} \left(\frac{w_i^{ON}}{i} \cdot \frac{CPI^{ON}}{f^{ON}} \right)$ makes up a large portion of total execution time. However, as the number of nodes increases, this portion decreases. With this decrease, parallel overhead eventually dominates. Thus the effects of frequency diminish since $w_{PO}^{ON} = 0$.

5.4 Model Usage in Performance Prediction

5.4.1 Coarse-Grain Parameterizations

Though we have shown that our power-aware speedup model is accurate, to this point we have purposely hidden the details of how to obtain our model parameters on real systems. In this section, we show how to derive model parameters and apply them in both equations to predict power-aware speedup. We primarily use versions of Equations 5.10 and 5.11 to obtain speedup predictions. For this simplified parameterization, we make two assumptions.

Assumption 1: a majority of the workload can be completely parallelized, such that $w = \sum_{i=1}^{m} w_i = w_N$, where $N = m$, and $w_i = 0$ for $i \neq m$. Under this assumption [4], sequential execution time is simplified as

$$\begin{aligned} T_1(w,f) &= \left[T_1(w_N^{ON}, f^{ON}) + T_1(w_N^{OFF}, f^{OFF})\right] \\ &= w_N^{ON} \cdot \frac{CPI^{ON}}{f^{ON}} + w_N^{OFF} \cdot \frac{CPI^{OFF}}{f^{OFF}}, \end{aligned} \tag{5.14}$$

and parallel execution time is simplified as

$$\begin{aligned} T_N(w,f) &= \left[T_N(w_N^{ON}, f^{ON}) + T_N(w_N^{OFF}, f^{OFF})\right] \\ &+ \left(T(w_{PO}^{ON}, f^{ON}) + T(w_{PO}^{OFF}, f^{OFF})\right) \\ &= \frac{T_1(w,f)}{N} + \left(T(w_{PO}^{ON}, f^{ON}) + T(w_{PO}^{OFF}, f^{OFF})\right). \end{aligned} \tag{5.15}$$

[4] Most speedup models are for bound analysis. It is common to make the assumption that workload consists of only serial portion w_1 and parallelizable portion w_N. In practice, speedup analysis focuses solely on the parallelizable portion of the code and w_1 is considered negligible. We follow this common practice, though we are exploring ways to measure w_1 directly.

Assumption 2: parallel overhead is not affected by ON-chip frequency [29], i.e. $w_{PO}^{ON} = 0$. Under Assumption 2, Equation 5.15 is reduced to

$$T_N(w, f) = \frac{T_1(w, f)}{N} + T(w_{PO}^{OFF}, f^{OFF}) . \tag{5.16}$$

Equation 5.16 holds for all the frequencies. Given the relationship shown in Equation 5.16, we now describe how to predict power-aware performance given a processor count and frequency.

Step 1. Measure the sequential execution time $T_1(w, f_0^{ON})$ and parallel execution time $T_N(w, f_0^{ON})$ for workload w when ON-chip frequency f^{ON} is set as base frequency f_0^{ON}.

Step 2. Derive the parallel overhead time using the measured times from Step 1 and Equation 5.16 such that $T_N({w_{PO}}^{OFF}, f^{OFF})$ is the parallel overhead $T(w_{PO}^{OFF}, f^{OFF})$ for processor count N:

$$T_N(w_{PO}^{OFF}, f^{OFF}) = T_N(w, f_0^{ON}) - \frac{T_1(w, f_0^{ON})}{N} . \tag{5.17}$$

Step 3. Measure the sequential execution time $T_1(w, f)$ for the same workload w on 1 processor for each available frequency.

Step 4. Use the derived parallel overhead in Step 2 and measured sequential execution time from Step 3 to predict the parallel execution time $T_N(w, f)$ for any given combination of processor count ($N > 1$) and frequency ($f > f_0^{ON}$).

$$\begin{aligned} T_N(w, f) &= \frac{T_1(w, f)}{N} + T_N(w_{PO}^{OFF}, f^{OFF}) \\ &= \frac{T_1(w, f)}{N} + \left[T_N(w, f_0^{ON}) - \frac{T_1(w, f_0^{ON})}{N} \right] . \end{aligned} \tag{5.18}$$

Table 5.3 shows prediction errors for FT are less than 3% using this technique. With these results, our assumptions appear reasonable for FT. In fact, the assumption is a very practical means

of obtaining power-aware speedup. Nonetheless, there are drawbacks to this approach. First, this technique requires measurements for the sequential ($T_1(w, f)$) and parallel ($T_N(w, f_0^{ON})$) execution time. Second, this technique does not separately consider the ON-chip and OFF-chip portions of the workload. Thus, the effects of frequency are accounted for but inseparable from the execution time. Third, the assumptions used are the root cause of the observable error. Assuming perfect parallelism means over-estimating the effects of increasing the number of processors. Assuming parallel overhead is not affected by frequency means underestimating the effects of increasing processor frequency. The aforementioned problems can be partly resolved by fine-grain parameterizations with the aid of tools including hardware counters via PAPI [28], mpptest [19], and LMbenchmark [26].

5.4.2 Fine-Grain Parameterizations

In this section, we show how to derive and use detailed power-aware speedup parameters to predict performance. We use Equation 5.10 as the basis for our discussion. We have applied this technique to FT with error rates similar to those in Table 5.3. For diversity, we use the lower-upper diagonal (LU) benchmark from the NAS Parallel Benchmark suite as a case study. LU uses a symmetric, successive overrelaxation numerical scheme to solve a regular-sparse, block lower and upper triangular system. LU is an iterative solver with a limited amount of parallelism and a memory footprint comparable to FFT. LU exhibits a regular communication pattern and makes use of the memory hierarchy.

Table 5.4: Workload measurement and decomposition

Workload	Memory Level	Derivation	#ins(10^9)
ON-chip	CPU/Register	PAPI_TOT_INS-PAPI_L1_DCA	145
	l1	PAPI_L1_DCA-PAPI_L1_DCM	175
	L2	PAPI_L2_TCA-PAPI_L2_TCM	4.71
OFF-chip	Main Memory	PAPI_L2_TCM	3.97

This technique consists of three steps: workload distribution, unit workload execution time, and application execution time prediction.

Step 1: Workload distribution (w^{ON}, w^{OFF})

The goal for this step is to obtain the distribution of the ON-/OFF-chip portions of the workload. On the system measured, ON-chip portion of the workload consists of computations with data residing in the registers, and the L1 or L2 cache. OFF-chip portion of the workload consist of computations with data residing in main memory or on a disk. We use hardware counters to measure the workload parameters for LU. Hardware performance counters are special registers that accurately track low-level operations and events such as the number of executed instructions and cache misses with minimum overhead. Hardware limitations on the number and type of events counted simultaneously require us to run the application multiple times in order to record all the events we need. In this work, we use PAPI [124] to access the counters. We assume hardware event counts are similar across different processors for the same workload and obtain measurements on 1 processor [5].

[5]This technique is commonly used for regular SPMD codes such as LU. We observe the performance event counts are within 2% from sequential execution to parallel execution. For non SPMD codes, we could obtain results from individual processors and perform similar (albeit more cumbersome) analyses.

To quantify ON-chip workload distribution, we monitor the following PAPI events: total instructions (PAPI_TOT_INS), L1 data cache accesses (PAPI_L1_DCA), L1 data cache misses (PAPI_L1_DCM), L2 cache accesses (PAPI_L2_TCA), and L2 cache misses (PAPI_L2_TCM).

Table 5.4 shows the formulae [6] and the workload distributions for LU in number of instructions and percentage of ON-chip portion of the workload. ON-chip portion of the workload (w^{ON}) account for 98.8% of the total workload. Despite LU's significant memory footprint, most data (97.4%) is available in the L1 cache. OFF-chip portion of the workload (w^{OFF}) account for 1.2% of the total workload. We can also determine the distribution of the ON-chip workload: 44.66% CPU/Register instructions, 53.89% L1 cache instructions, 1.45% L2 cache instructions. The OFF-chip workload consists of only memory instructions.

Step 2: Unit workload execution time CPI^{ON}/f^{ON}, CPI^{OFF}/f^{OFF}, and $T_N(w_{PO}, f)$

Next, we measure the average amount of time (CPI_j/f) required for each of the four types of workload identified in the previous step (where j=[1,2,3,4]=[CPU/Register, L1 cache, L2 cache, memory]). We use the LMBENCH [113] toolset as it enables us to isolate the latency for each of these workload types. Using the weighted ON-chip workload distribution identified in the previous step, we can calculate the weighted average CPI/f for ON-chip workloads, $CPI^{ON}/f = 0.446CPI_1/f + 0.538CPI_2/f + 0.014CPI_3/f$ where f is any of the available frequencies [7]. Similarly, the weighted average CPI/f for OFF-chip workloads is $CPI^{OFF}/f = CPI_4/f$.

[6] We use event count of data cache access to approximate total cache access due to limited available event counters on the measured system.

[7] This assumes one floating point double (FPD) computation per memory operation. For the actual predictions, we adjust to account for instruction-level parallelism that enables about 2.42 FPD computations per memory operation.

Table 5.5: Seconds per Instruction (CPI/f) for ON-/OFF-chip workload

Frequency		600MHz	800MHz	1000MHz	1200MHz	1400MHz
w^{ON}	ON-chip CPI^{ON}	2.19	2.19	2.19	2.19	2.19
	$(CPI^{ON}/f^{ON}(10^{-9})$	3.65	2.74	2.19	1.83	1.56
w^{OFF}	$CPI^{OFF}/f^{OFF}(10^{-9})$	140	140	110	110	110
w_{PO}	For 155 doubles (10^{-6})	25	25	25	25	25
	For 310 doubles (10^{-6})	200	167	167	167	167

Table 5.5 presents the seconds per workload for ON-/OFF-chip workloads for each available processor frequency. Our premise is that ON-chip workloads are affected by frequency while OFF-chip workloads are not. The results in Table 5.5 show CPI^{ON}/f^{ON} or seconds per ON-chip workload decrease with frequency scaling. Table 5.5 shows OFF-chip workloads are basically constant with frequency scaling. On our system, we measured a slight increase in seconds per memory workload for slower CPU clock frequencies. We believe this is due to a hardware-driven decrease in the bus speed (f^{OFF}) for lower CPU clock frequencies. This system-specific behavior is captured by our parameter measurements. So, the effects are included in our predictions. Nonetheless, we are investigating this further to determine if it is common across platforms.

To measure communication workload time ($T_N(w_{PO}, f)$), we measure the seconds per communication for different message sizes using the MPPTEST [72] toolset. We observe that LU transmits 310 doubles per message between two nodes. For four nodes, LU transmits 155 doubles per message. Table 5.5 shows the transmission times for each of these cases. The trend is similar as the number of nodes increases. For the larger message sizes (310 doubles) on the slowest frequency, the communication time is influenced by the CPU (f^{ON}). For smaller message sizes on more than

Table 5.6: Performance prediction using power-aware speedup. The data shows the perdition errors for LU. FP uses fine-grain parameterization to perform predictions. SP uses simplified parameterization to perform predictions.

N	Frequency(MHz)									
	600		800		1000		1200		1400	
	FP	SP	FP	SP	FP	SP	FP	SP	FP	SP
1	5%	0%	7%	0%	3%	0%	4%	0%	1%	0%
2	6%	0%	6%	2%	5%	4%	6%	3%	8%	6%
4	2%	0%	6%	3%	8%	4%	10%	7%	7%	7%
8	3%	0%	1%	4%	8%	8%	11%	10%	7%	13%

2 nodes, CPU frequency has no noticeable effects. We use the product of number of messages and message time to compute $T_N(w_{PO}, f)$.

Step 3. Application execution time and speedup prediction

Now, we can predict the execution time of LU for combinations of processor count and frequency using Equations 5.14 and 5.15 and the parameter values from Steps 1 and 2. We use Equation 5.14 to predict sequential execution time, $T_1(w, f)$. This means we rely on Assumption 1, that the total workload is parallelizable. We use Equation 5.15 to predict parallel execution time, $T_N(w, f)$, where we use $T_N(w_{PO}, f)$ from the previous step for the number of messages obtained by profiling LU.

Table 5.6 presents the prediction error of the fine-grain parameterizations (FP) and a comparison with simplified parameterizations (SP). From this table, we observe that the errors for SP parameterizations increase steadily with both number of nodes and frequency. Errors for FP increase with number of nodes but appear to be leveling off with frequency. SP outperforms FP for some cases because SP requires more information to evaluate effects of parallelism than FP.

Such extra information warrant better predictions. Our assumptions explain these observations. Assuming the workload is completely parallelizable in both techniques increases the error rates. We are working presently to obtain better estimates of DOP to help mitigate these errors - though all speedup models suffer this problem. In the FP case, we separate the ON- and OFF-chip workloads. Thus, we are able to improve the insight and accuracy of the SP method. Of course, FP requires additional parameterizations studies.

5.5 System Efficiency Evaluation

The system configuration with the minimum energy-delay product, denoted as $E \cdot D$, is optimal in energy-performance efficiency if energy and delay have equal weight (see earlier discussions). For applications with large workloads running on large-scale systems, identifying the optimal system configurations ahead of the actual executions conserves energy and guarantees performance.

The identification of the optimal system configuration consists of three steps. First, we predict the execution time for each system configuration. Second, we estimate the system-wide energy consumption. Third, we calculate and evaluate energy-performance efficiencies and identify the optimal configuration.

We predict the performance using the methodology presented in the preceding section, and estimate the energy consumption using the same methodology presented by Springer el al [132] based on empirical data on our system. The power consumption on a single node varies between operations with different access level and CPU frequency. However, for simplicity, we assume there are only two different power consumptions for a fixed CPU frequency: one is the power

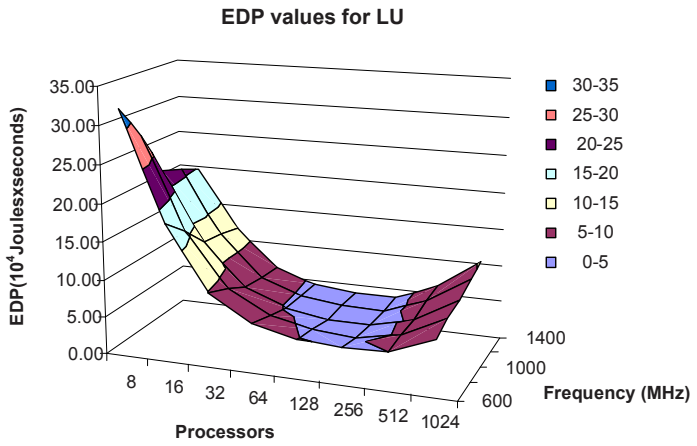


Figure 5.3: Energy-performance efficiencies in EDP for LU benchmark. The efficiency is evaluated with EDP for different system configurations. Optimal System Configuration is 256 processors with 1200MHz CPU Frequency.

consumption when the system is dedicated to computation, and the other is the power consumption when the system is dedicated to communication. The former varies with CPU frequency, while the latter is independent of CPU frequency, and both are independent with the number of processors.

Figure 5.3 shows the EDP values for system configurations in combinations of processors count and CPU frequency for LU benchmark. We observe EDP decreases with processor count and CPU frequency increases when processors count is less than 128. When processor count reaches 256, EDP decreases and then increases as CPU frequency increases. Furthermore, EDP increases with processors count after it exceeds 256. The optimal system configuration is the one with 256 processors and 1200MHz. At this point the cost of parallel overhead dominates computation time, and the communication/computation ratio is 1.73:1. The interesting observation is that the system slackness is reflected in both processor count and CPU frequency. Increasing either CPU frequency

or processors count further beyond the optimal configuration incurs larger EDP values. Not only our model can easily identify for users the maximum speedup and system configuration given an energy budget.

5.6 Chapter Summary

We presented in this chapter a performance model for emergent power-aware distributed systems. By decomposing the workload with DOP and ON-/OFF-chip characteristics, this model takes into account the effects of both parallelism and power aware techniques on speedup. Our study of NPB codes on a DVS-enabled power aware cluster shows that the proposed model is able to capture application characteristics and their effects on performance. Coupled with an energy-delay metric, this new speedup model can predict both the performance and the energy/power consumption.

The next two chapters will apply this model for power and performance management in power-aware distributed systems. Chapter 6 exploits inter-process communications, while Chapter 7 exploits both communications and memory access for power reduction and efficiency improvement. Both chapters consider fixed system size, and thus focus on power-aware techniques.

Chapter 6

Phase-Based Power Aware Approach for High End Computing

In this chapter, we propose phase-based DVFS scheduling and study its effectiveness for improving the energy-performance efficiency of scientific applications. By analyzing the energy-performance profiles of NAS parallel benchmarks on a power aware cluster, we identify code regions where power mode scheduling can reduce energy while performance loss is minimized. The findings in this chapter motivate the use of a phase-based DVFS power aware approach in parallel scientific computing.

6.1 Introduction

Dynamic Voltage Frequency Scaling (DVFS) is a technology now present in high-performance microprocessors. DVFS works on a very simple principle: decreasing CPU supply voltage or frequency can dramatically reduce CPU power consumption.

There are compelling reasons for using DVFS in HPC server clusters. The first reason is to exploit the dominance of CPU power consumption on system node (and thus cluster) power consumption. Our early work from PowerPack (see Figure 4.5) shows the breakdown of system node power obtained using direct measurement. The percentage of total system power for a Pentium III CPU is 35% under load. This percentage is lower (15%) but still significant when the CPU is idle. While the Pentium III can consume nearly 45 watts, recent processors such as Itanium 2 consume over 100 watts with a growing percentage of total system power. Reducing the average power consumed by the CPU can result in significant server energy saving that is magnified in cluster systems.

The second reason to use DVFS in HEC is to save energy without increasing execution time. Distributed applications suffering from poor performance efficiency despite aggressive optimizations exhibit CPU idle or slack times. During these times, the CPU is waiting on slower components such as memory, disk, or the network interface, and we can switch the CPU to low power/performance mode for energy conservation without affecting performance drastically.

To minimize the impact on application execution time, we must ensure DVFS scheduling corresponds to CPU intensiveness and application execution. The solution is phase based scheduling. Specifically, we categorize an execution pattern into phases, and study the energy-performance efficiency for each phase. Though previous DVFS work is phase-based in nature [80], the key difference and challenge in our work has been to identify all the different phases of parallel program execution which includes communication phases typically ignored by previous techniques.

For parallel applications, we analyze all types of phases then scale down CPU power modes during phases that promise energy conservation without performance impact. We use system-centric techniques to identify memory-, CPU-, and I/O-bound phase combined with parallel performance analysis techniques to identify communication-bound phases. Before we present the details of our phase-based power-aware scheduling, we will first evaluate the energy-performance efficiency of applications on DVFS-capable power-aware clusters.

6.2 Application Efficiency under DVFS

In this section, we study the energy-performance efficiency of parallel applications on DVFS based power-aware clusters. We use programs in the NAS parallel benchmark suite, run each program 5 times on the Nemo cluster introduced in Chapter 5, and record system energy consumption and application execution time. For each program, 5 runs are with single speed scheduling. Under single speed scheduling, the processor frequency is set as one of 5 available frequencies (600MHz, 800MHz, 1000MHz, 1200MHz, 1400MHz), and does not change over the application execution.

Table 6.1 gives raw figures for energy and delay for all the frequency operating points available on our system over all the codes in the NAS PB suite. In each cell, the number on the top is the normalized delay and the number at the bottom is the normalized energy, relative to those at 1400MHz. Selecting a good frequency operating point needs a metric to tradeoff execution time and energy. For instance, BT at 1200 MHz has 2% additional execution time (delay) with 7% energy savings. Is this better or worse than BT at 1000 MHz with 4% additional execution time

Table 6.1: Energy-performance profiles of NPB benchmarks. Delay (top # in each cell) and energy (bottom # in each cell) is normalized to fastest processor speed (1400MHz).

Code	Frequency (MHz)				
	600	800	1000	1200	1400
BT.C.9	1.52	1.27	1.14	1.05	1.00
	1.52	1.27	1.14	1.05	1.00
CG.C.8	1.14	1.08	1.04	1.02	1.00
	0.65	0.72	0.80	0.93	1.00
EP.C.8	2.35	1.75	1.40	1.17	1.00
	1.15	1.03	1.02	1.03	1.00
FT.C.8	1.13	1.07	1.04	1.02	1.00
	0.62	0.70	0.80	0.93	1.00
IS.C.8	1.04	1.01	0.91	1.03	1.00
	0.68	0.73	0.75	0.94	1.00
LU.C.8	1.58	1.32	1.18	1.07	1.00
	0.79	0.82	0.88	0.95	1.00
MG.C.8	1.39	1.21	1.10	1.04	1.00
	0.76	0.79	0.85	0.97	1.00
SP.C.8	1.18	1.08	1.03	0.99	1.00
	0.67	0.74	0.81	0.91	1.00

and 20% energy savings? For this type of comparison, we use the ED3P (ED3) metric to place a high weight performance compared to energy savings.

Figure 6.1 shows the energy-performance efficiency of NPB benchmarks under single speed with ED3P (ED^3) metric. This figure is obtained as follows: For each benchmark, compute the ED^3 value at each operating point using the corresponding normalized delay and normalized energy, and use the operating point which has the smallest ED^3 value as the scheduling point thereafter. If two points have the same ED^3 value, choose the point with the higher performance. The effect of single speed scheduling shown in Figure 6.1 reduces energy with minimum execution time increase and selects an operating frequency that is application dependent.

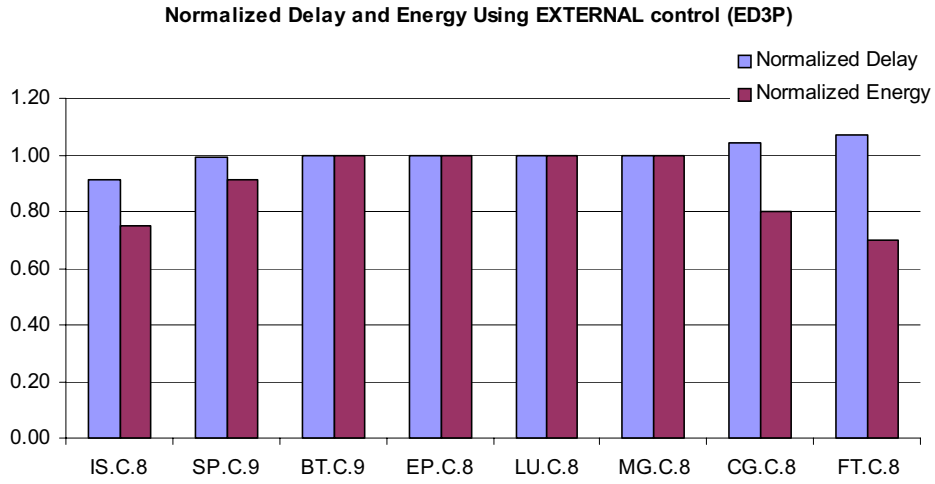


Figure 6.1: Energy-performance efficiency of NPB codes with ED3P.

The effects of single speed DVFS scheduling can be classified in three categories:

1. Energy reduction with minimal performance impact. For FT, this scheduling technique saves 30% energy with 7% delay increase in execution time. For CG, this scheduling technique saves 20% energy with 4% delay increase in execution time.

2. Energy reduction and performance improvement at the same time. For SP, this scheduling technique saves 9% energy and also improves execution time by 1%. For IS, this scheduling technique saves 25% energy with 9% performance improvement. This result is repeatable. We speculate this result is due to bus arbitration issues at the various frequencies. At these processor speeds for these codes, arbitration on the front-side-bus effectively results in a lower average memory access time.

3. No energy savings and no performance loss. BT, EP, LU, MG fall into this category.

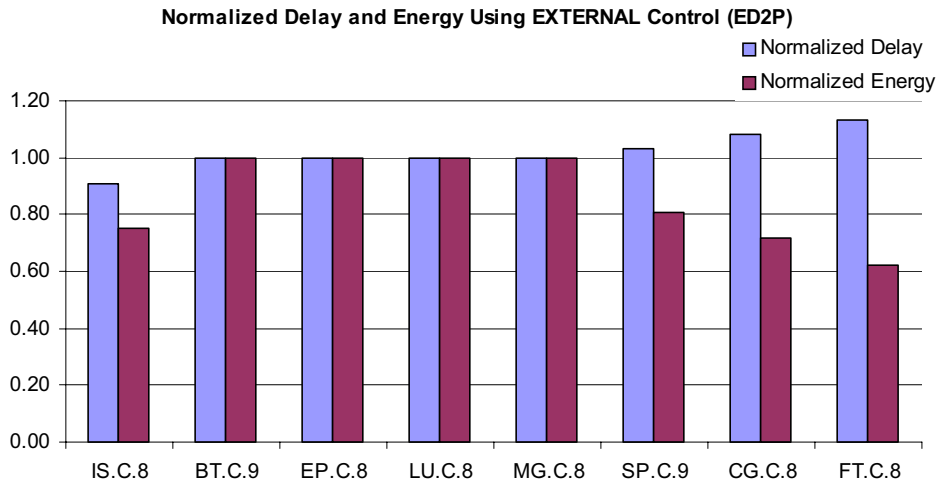


Figure 6.2: Energy-performance efficiency of NPB codes with ED2P.

Figure 6.1 uses ED3 as an energy-performance metric that favors performance over energy savings. If users are comfortable with slightly larger performance loss in exchange for more energy savings, ED2P (ED2) or EDP (ED) could be used as the energy-performance metric. Figure 6.2 shows the effects of ED2P metrics used with single speed scheduling. The trend is the same as Figure 6.1, but the metric may select frequency operating points where energy savings have slightly more weight than execution time delays. For example, ED2P would select different operating points for FT corresponding to energy savings of 38% with 13% delay increase; for CG, it selects 28% energy with 8% delay increase. For SP, it selects 19% energy with 3% delay increase.

The benefits of single speed scheduling are limited by three factors:

1. the influence of clock frequency on application performance;

2. the granularity of DVFS control;

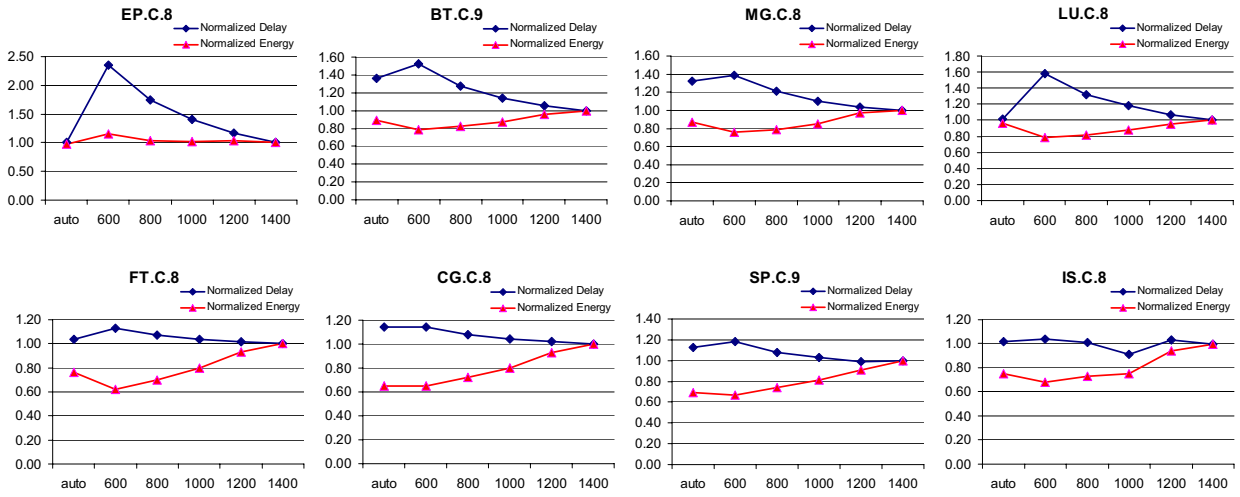


Figure 6.3: Energy-delay crescendos of NPB benchmarks. X-axis is CPU speed. Y-axis is the normalized value (delay and energy). The eight figures are grouped into four categories.

3. the homogeneity of DVFS control.

Figure 6.3 shows graphically the energy performance profiles under single speed scheduling using "energy-delay crescendos." These figures indicate that we can group the eight benchmarks into four categories:

Type I (EP): near zero energy benefit, linear performance decrease when scaling down CPU speed.

Type II (BT, MG and LU): near linear energy reduction and near linear delay increase, the rate of delay increase and energy reduction is about same.

Type III (FT, CG and SP): near linear energy reduction and linear delay increase, but the rate of delay increase is smaller than the rate of energy reduction.

Type IV (IS): near zero performance decrease, linear energy savings when scaling down CPU speed.

This classification matches the effects of external control shown in figure 6.1 and figure 6.2. In other words, the observed trends indicate basically that Type III and Type IV save energy while Type I and Type II do not save energy.

The second and third limitations are not shown in the figure but can be understood analytically: a real parallel application will consist of combinations of dependent computation modules which belong to two or more of the categories mentioned above. If we only schedule a single CPU speed for all nodes during the whole execution, benefits obtained from Type III and Type IV will be compromised by the impact of Type I and Type II. Therefore, we must consider application execution phases at granularity finer than the entire application and use phase-based control to overcome this limitation.

6.3 Phase Categorization

We categorize an application execution pattern into three phases: *computation bound phase* dominated by on-chip accesses, *memory bound phase* by main memory accesses, and *communication bound phase* by remote communications.

As shown in Chapter 5, reducing CPU frequency does not typically impact the performance of memory bound and communication bound phases, but reduces the CPU power consumption. Here we experimentally study how CPU scheduling affects system energy consumption and performance efficiency for these phases.

To quantify the impact of DVFS on each phase, we emulate phases using microbenchmarks. A computation bound phase is emulated by a synthetic benchmark with register computation and on-chip cache accesses, a memory bound phase is emulated by a benchmark with only main memory access, and a computation bound phase is emulated by a benchmark with MPI communications.

Memory-bound phase. Figure 6.4 presents the energy consumption and delay of memory access under different CPU frequencies. The measured code reads and writes elements from a 32MB buffer with a stride of 128Bytes. The buffer exceeds L2 cache size, thus each data reference is fetched from main memory. At 1.4 GHz, the energy consumption is maximal, while execution time is minimal. The energy consumption decreases with operating frequency, and it drops to 59.3% at the lowest operating point 600MHz. However, execution time is only minimally affected by the decreases in CPU frequency; the worst performance at 600 MHz shows a decrease of only 5.4% in performance. The conclusion is memory-bound applications offer good opportunity for energy savings since memory stalls reduce CPU efficiency. This confirms findings and experiences of other researchers [84].

CPU-bound phase. Figure 6.5 is energy consumption and delay under DVFS for a CPU-intensive micro benchmark. This benchmark reads and writes elements in a buffer of size 256Kbytes with stride of 128Bytes, where each calculation is an L2 cache access. Since L2 cache is on-die, we can consider this as CPU-intensive [1]. The energy consumption for a CPU-intensive phase is different from a memory access phase in that computation speed is directly affected by process frequency as shown in Figure 6.5.

[1] Although strictly speaking an L2 cache access is part of the memory hierarchy, we are primarily interested in data accesses not affected by core speed. Since L2 is on die, it is directly affected and thus included in our characterization of CPU-intensive code phases.

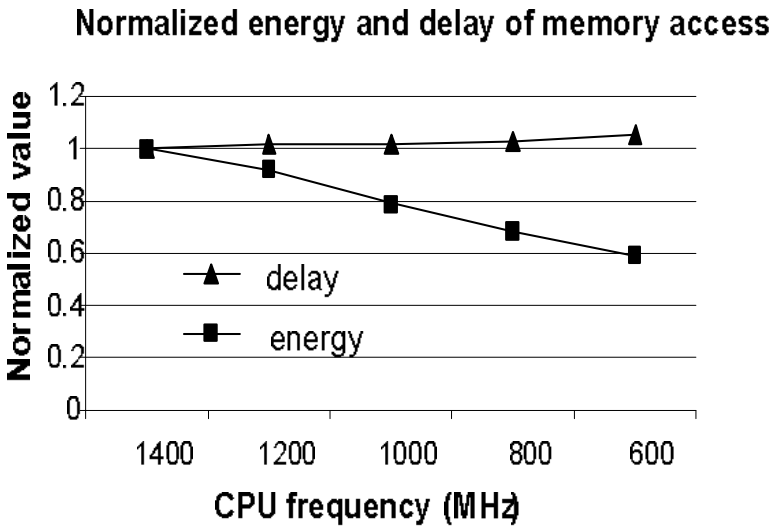


Figure 6.4: Normalized energy and delay of memory accesses

As we expect, the results in Figure 6.5 are unfavorable for energy conservation. Delay increases with CPU frequency near-linearly. At the lowest operating point, the performance loss can be 134%. On the other hand, energy consumption decreases first, and then goes up. Minimum energy consumption occurs at 800 MHz (10% decrease). Energy consumption then actually increases at 600 MHz. The dramatic performance decrease counteracts the potential energy savings from CPU power reduction. While average CPU power and the total system power decreases, execution time and the resulting system energy consumption increases. If we limit memory accesses to registers thereby eliminating the latency associated with L2 hits the results are even more striking. The lowest operating point consumes the most energy and increases execution time by 245%.

Communication-bound phase. Figure 6.6 shows the normalized energy and execution time for MPI primitives. Figure 6.6a is the round trip time for a 256 Kbyte message. Figure 6.6b is the

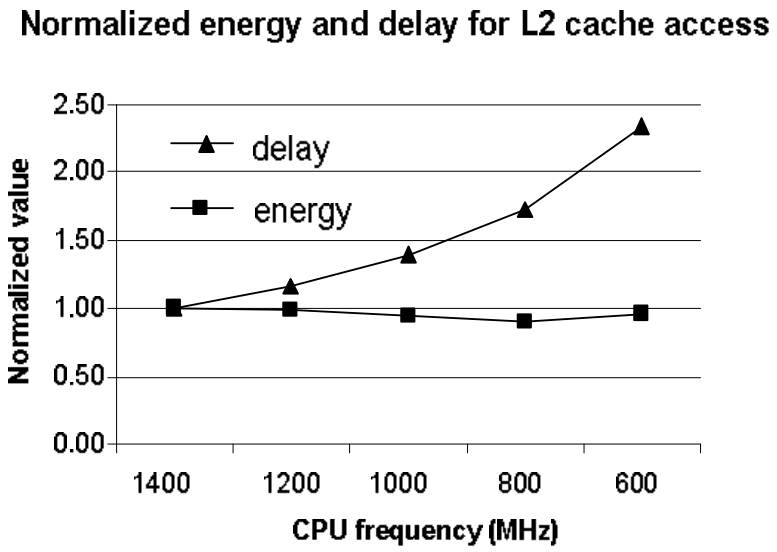
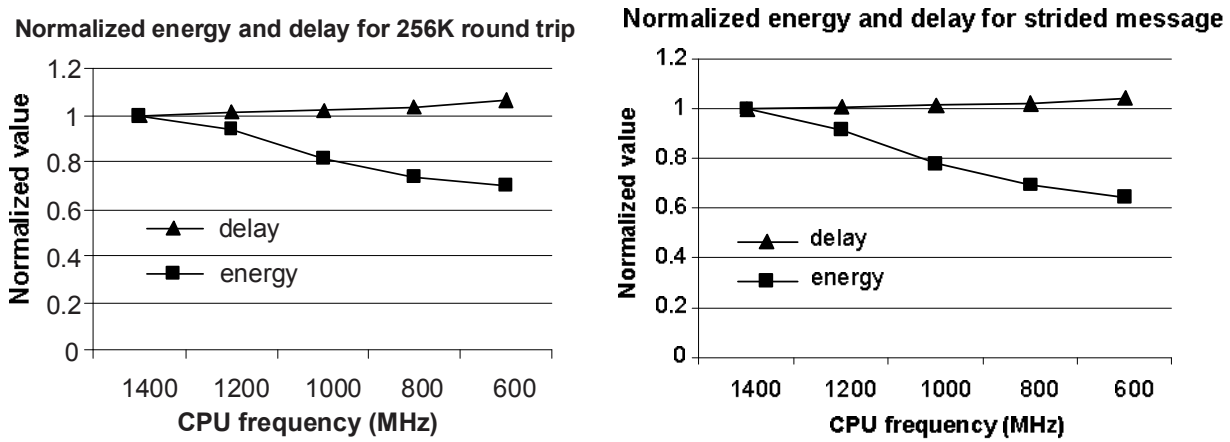


Figure 6.5: Normalized energy and delay of on-chip cache access

round trip time for a 4 Kbyte message with stride of 64bytes. Memory load latency on this system is 110ns. Basic point-to-point communications take dozens of microseconds. Collective communications take several hundreds of microseconds. All MPI communications present opportunity for CPU slackness, during which CPU is not fully utilized. Our $log_3 P$ model allows us to further quantify the impact of memory delay on communications (see Chapter 3).

As we expect, the crescendos in Figure 6.6a and Figure 6.6b are favorable to energy conservation for communications. The energy consumption decreases with CPU frequency drastically while execution times increase only slightly. For the 256K round trip, energy consumption at 600MHz decreases 30.1% and execution time increases 6%. For 4KB message with stride of 64Bytes, at 600 MHz the energy consumption decreases 36% and execution time increases 4%. We note these are the first studies of the impact of DVFS on communication cost.



(a) (b)

Figure 6.6: Normalized energy and delay of remote network access

6.4 Phase Based Power-Aware Scheduling

Our microbenchmark profiles indicate that: 1) code regions or phases which are memory bounded or communication bounded can use DVFS to reduce energy while maintaining performance. 2) code regions or phases which are computation bounded will lose significant performance if DVFS is used.

Considering a generic application, we can represent its execution as a sequence of M phases over time, i.e. (w_1, t_1), (w_2, t_2), ..., (w_M, t_M), where w_i is the workload in the i^{th} phase and t_i is the time duration to compute w_i at the highest frequency $f_{\max}$.

Different execution phases require different power-performance modes for power-performance efficiency. The goal of a DVFS scheduler for parallel codes is to identify all execution phases,

quantify DVFS impact on workload characteristics, and then switch the system to the most appropriate power/performance mode.

We use the NAS parallel benchmarks FT.C.8 and CG.C.8 as examples to illustrate how to implement phase based scheduling for applications. Each example starts with performance profiling and then a DVFS scheduling strategy is derived by analyzing the profiles. The effects of the scheduler are verified with experimental results.

Phase Based Scheduling for FT benchmark

Performance Profiling: Figure 6.7 shows the performance profile of FT generated with the MPICH trace utility by compiling the code with "-mpilog" option. We observe from the profile that:

- FT is communication-bound; its communication to computation ratio is about 2:1.
- Most execution time is consumed by all-to-all communication.
- The execution time per iteration is large enough so that the CPU speed transition overhead (10-40ms) can be ignored if switching occurs between iterations.
- The workload is almost balanced across all nodes.

Scheduler Design: Based on the above observations we divide each iteration into all-to-all communication phases and other phases. Scheduling is set to the lowest speed for all-to-all communication phases, and restored to the highest CPU speed thereafter. Figure 6.8 shows how DVFS control is inserted into the source code.

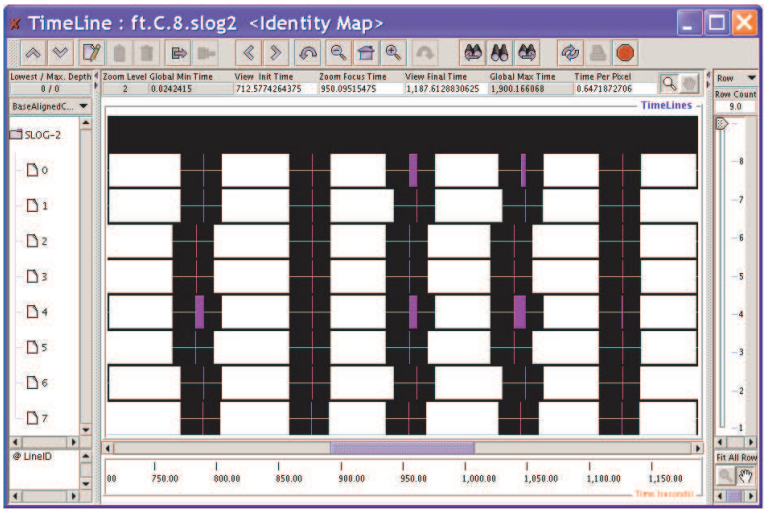


Figure 6.7: Performance trace of FT.C.8 using MPE tool provided with MPICH. The traces are visualized with jumpshots, which are graphical tools for understanding the performance of parallel programs.

```
...
call set_cpuspeed( low_speed)
call mpi_alltoall(  )
call set_cpuspeed( high_speed)
...
```

Figure 6.8: Phase-based DVFS scheduling for FT benchmark

Experiment Results: Figure 6.9 shows the energy savings and delay increase using phase based scheduling. By choosing high_speed as 1400 MHz and low_speed as 600 MHz, phase based scheduling can save 36% energy without noticable delay increase. This is significant improvement

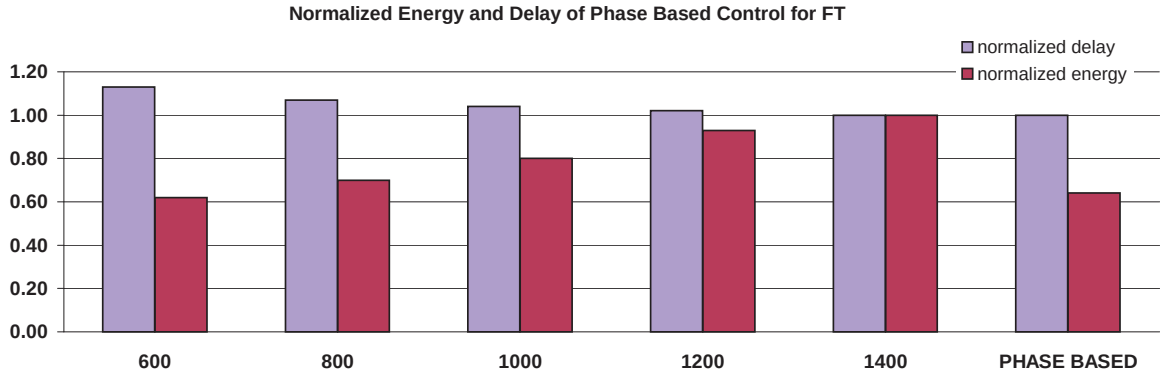


Figure 6.9: Energy and delay of FT benchmark under various DVFS scheduling strategies. In phase based control, high speed and low speed are set as 1400 and 600 MHz respectively. The traces are visualized with jumpshots.

over single speed control. Single speed control at 600MHz saves 38% energy but the cost is a 13% delay increase. Thus phase based control is appropriate when an application contains obvious CPU-bound phases and non-CPU bounded phases and each phase lasts long enough duration that it is sufficiently large in comparison to the CPU speed transition overhead.

Phase Based Scheduling for the CG benchmark

Performance Profiling: Figure 6.10 shows a performance profile of CG. We note the following observations:

- CG is communication intensive and the ratio of communication to computation is about or over 1:1.
- Different nodes exhibit different communication and computation behavior. First, nodes 4-7 have larger communication-to-computation ratios compared to nodes 0-3. Second, com-

puting nodes arrive at computation or communication phases at different time. While nodes 1-3 are in computation phases, nodes 4-7 are in communication phases, and vice versa.

- The execution time of each iteration is relatively small, message communications are frequent, and CPU speed transition overhead could be significant.

Scheduling Decision: Based on the above observations, we found it difficult to improve power-performance efficiency of CG using symmetric phased-based DVFS scheduling, i.e. setting the frequencies of all nodes at a same value. We implemented two symmetric phase-based dynamic scheduling schemes: 1) scale down CPU speed during MPI_wait and MPI_send; and 2) scale down CPU speed only during MPI_Wait. Both phase-based DVS scheduling increase energy and delay (1-3%).

Whereas, we can use asymmetric DVFS scheduling, i.e. set different speed for each execution node given their different behaviors. Our asymmetric DVFS scheduling scheme is shown in Figure 6.11.

Experiment Results: The results from the experiments are shown in Figure 6.12. We provide results for the two best configurations we found: phase based I which uses 1200 MHz as the high speed and 800 MHz as the low speed and phase based II which uses 1000 MHz as the high speed and 800 MHz as the low speed. Experiments show that phase based I saves 16% energy with 8% delay increase and phase based II saves 23% energy with 8% delay increase. Both phase based I and II scheduling for CG do not show an advantage over single speed scheduling at 800MHZ as phase based scheduling suffers time and energy overheads caused by frequency switches.

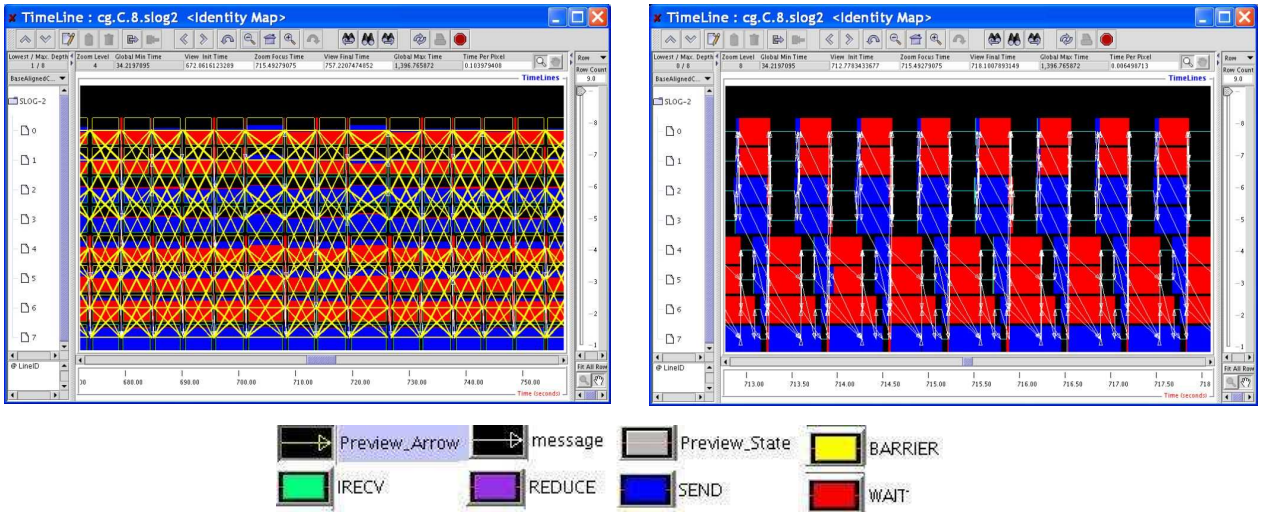


Figure 6.10: Performance trace of CG benchmark. The traces are visualized with jumpshot.

```
  ...
if ( myrank .ge. 0 .and. myrank .le. 3)
  call set_cpuspeed( high_speed)
else
  call set_cpuspeed( low_speed)
endif
  ...
```

Figure 6.11: Phase-based DVFS scheduling for CG benchmark

6.5 Chapter Summary

In this chapter we have presented phase-based DVFS scheduling for DVFS capable power-aware clusters. Phase-based DVFS scheduling adaptively switches processor performance/power modes

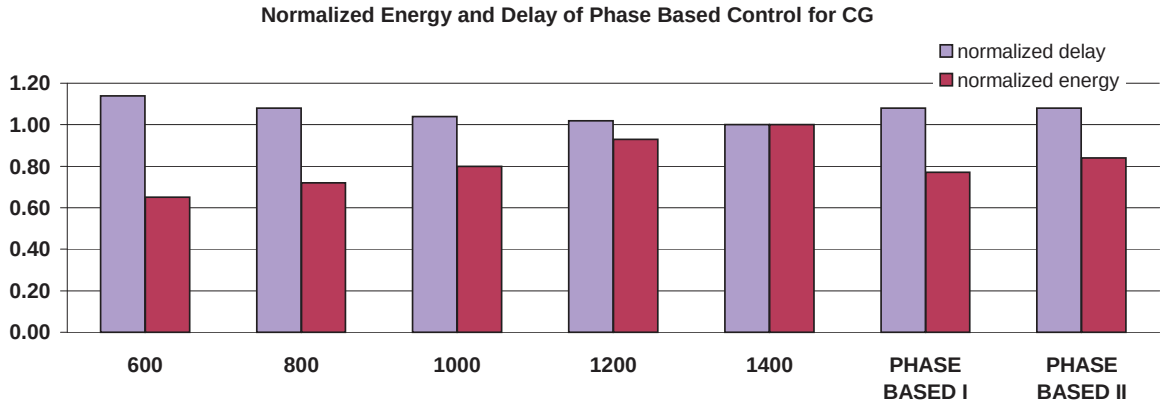


Figure 6.12: Energy and delay of CG benchmark under various DVFS scheduling strategies. For phase based I, high speed is 1200, and low speed is 800; for phase based II, high speed is 1000 and low speed is 800.

according to application execution phases, thus reducing energy consumption and improving efficiency. The key to save energy for parallel codes is to identify code regions where CPU slack is available including memory and communication phases. Using phase-based DVFS scheduling, we achieved total energy savings as large as 36% with no negative impact on performance. Our study also showed that energy savings vary greatly with application, workload, system, and DVFS strategy.

The deployed techniques for phase detection and DVFS instrumentation in this chapter are largely manual, though automatic intercepting MPI calls would be straightforward [64]. In next chapter we will present a run-time system that overcomes this limitation and automatically and transparently controls the power consumption and performance for computation, memory, I/O and communication phases.

Chapter 7

Performance-Directed Run-Time Power Management

In this chapter, we present a run-time system (CPU MISER) and an integrated performance model for performance-directed, power-aware cluster computing. CPU MISER supports system-wide, application-independent, fine-grain, dynamic voltage and frequency scaling (DVFS) based power management for a generic power-aware cluster. In addition to energy savings for typical parallel benchmarks, CPU MISER is able to constrain performance loss for most applications within user specified limits. These results are achieved through accurate performance modeling and prediction, coupled with advanced control techniques.

7.1 Introduction

In chapter 6, we have shown that off-line, trace-based DVFS scheduling is feasible for high performance computing. However, off-line, trace-based DVFS scheduling usually involves tedious tasks

like code instrumentation and performance profiling. Moreover, because an application's performance profile varies with data size and the underlying computing systems, a DVFS decision that saves energy for one execution does no guarantee energy savings for another execution unless circumstances are identical. Though off-line approaches provide a solid basis to improve the effectiveness of runtime techniques, run-time DVFS scheduling techniques are promising if they can save energy and remain automatic and transparent to end users.

However, run-time DVFS scheduling is more challenging since effective scheduling requires accurate prediction of the effects of power modes on future phases of the application without any a priori information. False prediction may have dire consequences for performance or energy efficiency.

Current run-time DVFS techniques for HEC have leveraged our early phase based findings [66] presented in Chapter 6 to motivate use of MIPS-based metrics [81, 82] or parser-driven identification of MPI calls to identify phases [64]. These techniques have been shown to reduce energy with reasonable performance loss. However, MIPS-based metrics use throughput as a performance measure which may not track actual parallel execution time and the performance impact of DVFS on parallel applications. On the other hand, intercepting MPI calls can identify communication phases accurately but this technique ignores other memory- or IO-bound phases that provide additional opportunities for power and energy savings.

In this chapter, we propose a performance-directed, run-time DVFS scheduler that is applicable to all execution phases and independent of specific programming models. Based on this methodology, we implemented a new run-time DVFS scheduler, named CPU MISER (which is short for

CPU Management Infra-Structure for Energy Reduction), that supports system-wide, application-independent, fine-grained, DVFS-based power management for generic power-aware clusters. The advantages of CPU MISER include:

- System-level management of power consumption and performance. CPU MISER can optimize for performance and power on multi-core, multi-processor systems.
- Exploitation of low CPU utilization phases including memory accesses, IO accesses, communication phases, and system idle under power and performance constraints. Communication phases have been ignored by most previous DVFS work.
- Complete automation of run-time DVFS scheduling. No user intervention is required.
- Integrated, accurate DVFS performance prediction model that allows users to specify acceptable performance loss for an application relative to application peak performance.

The remaining sections of this chapter are organized as follows. We first describe the target problem and theoretical foundation of CPU MISER, including the underlying performance model, workload prediction, and performance control. Then we briefly explain the system design of CPU MISER, followed by an evaluation of CPU MISER on a power-aware cluster. Finally, we summarize our findings and conclusions.

7.2 The δ-constrained DVFS Scheduling Problem

For a DVFS-based, power-aware cluster, we assume each of its compute nodes has N power/performance modes or processor frequencies available: $\{f_1, f_2, \ldots, f_N\}$ satisfying $f_1 < f_2 < \ldots < f_N = f_{\max}$.

Without loss of generality, we assume that the corresponding voltage V_i for $1 \leq i \leq n$ changes with f_i.

By changing the CPU from the highest frequency $f_{\max}$ to a lower frequency f, we can reduce the CPU's power consumption. However, if the workload is CPU-bound, reducing CPU frequency may also significantly reduce performance as well.

Considering a generic application, we can represent its entire workload as a sequence of M execution phases over time, i.e., (w_1, t_1), (w_2, t_2), ..., (w_M, t_M), where w_i is the workload in the i^{th} phase and t_i is the time duration to compute w_i at the highest frequency $f_{\max}$. As different workload characteristics require different power/performance modes for optimal power-performance efficiency, the goal of a system-wide DVFS scheduler is to identify each execution phase, quantify its workload characteristics, and then switch the system to the most appropriate power/performance mode.

To derive a generic methodology for designing an automatic, performance-directed, system-wide DVFS scheduler, we formulate the δ-constrained DVFS scheduling problem as follows:

Given a power-aware system and a workload W, schedule a sequence of CPU frequencies over time that is guaranteed to finish executing the workload within a time duration $(1 + \delta^*) \cdot T$ and minimizes the total energy consumption, where δ^* is a user-specified, performance-loss constraint (such as 5%) and T is the execution time when the system is continuously running at its highest frequency $f_{\max}$.

Co-scheduling power and performance is a complicated problem. However, empirical observations show that CPU power decreases as the CPU frequency decreases while the performance

decreases at a slower rate. This implies that as long as the performance loss is relatively small for the lower frequency energy savings result. Hence, heuristically, if we schedule a minimum frequency for every execution phase that satisfies the performance constraint, the end result is an approximate solution for the δ-constrained DVFS scheduling problem.

However, because it is difficult to detect the phases boundaries at run-time, we approximate each execution phase with a series of discrete time intervals and then schedule the power/performance modes based on the workload characteristics during each time interval. Therefore, we decompose the task of designing a performance-directed, system-wide DVFS scheduler into four subtasks: (1) instrumenting/characterizing the workload during each time interval; (2) estimating the time needed to compute a given workload at a specific frequency; (3) predicting the workload in the next time interval; and (4) scheduling an appropriate frequency for the next interval to minimize both energy consumption and performance loss.

To solve these subtasks, we first describe a performance model that captures the correlations between workload, frequency, and performance loss due to frequency scaling. Then, we describe techniques for workload prediction and performance control.

7.3 Performance Model and Phase Quantification

At the system level, any time duration t can conceptually be broken into two parts: t_w, the time the system is executing the workload w, and t_0, the time the system is idle. Thus we have,

$$t = t_w + t_0 \,. \tag{7.1}$$

Further, we can dissect t_w into two parts: $t_w(f_{\text{on}})$, the CPU frequency-dependent part, and $t_w(f_{\text{off}})$, the CPU frequency-independent part. In short, we express t_w as

$$t_w = t_w(f_{\text{on}}) + t_w(f_{\text{off}})\,. \tag{7.2}$$

Here, f_{on} and f_{off} refer to the on-chip and the off-chip instruction-execution frequencies, respectively.

In Equation (7.2), $t_w(f_{\text{on}})$ can be estimated by $t_w(f_{\text{on}}) = w_{\text{on}} \cdot \frac{CPI_{\text{on}}}{f}$, where w_{on} is the number of on-chip memory (including register and on-chip cache) accesses, and CPI_{on} is the average cycles per on-chip access [38, 65]. $t_w(f_{\text{off}})$ can be further decomposed into main-memory access time t_{mem} and I/O access time t_{IO}. We approximate the main-memory access time as $t_{mem} = w_{\text{mem}} \cdot \tau_{\text{mem}}$, where w_{mem} is the number of main-memory accesses and τ_{mem} is the average memory-access latency. Thus, we can quantify the correlations between t, w, and $f_{\max}$ as

$$t = w_{\text{on}} \cdot \frac{CPI_{\text{on}}}{f_{\max}} + w_{\text{mem}} \cdot \tau_{\text{mem}} + t_{\text{IO}} + t_0\,. \tag{7.3}$$

Since on-chip access is often overlapped with off-chip access on modern computer architectures [148], we introduce an overlapping factor α (such that $0 \leq \alpha \leq 1$) into Equation (7.3), i.e.,

$$t = \alpha \cdot w_{\text{on}} \cdot \frac{CPI_{\text{on}}}{f_{\max}} + w_{\text{mem}} \cdot \tau_{\text{mem}} + t_{\text{IO}} + t_0\,. \tag{7.4}$$

When the system is running at a lower frequency f, the time duration to finish the same workload w becomes:

$$t' = \alpha \cdot w_{\text{on}} \cdot \frac{CPI_{\text{on}}}{f} + w_{\text{mem}} \cdot \tau_{\text{mem}} + t_{\text{IO}} + t_0\,. \tag{7.5}$$

Assuming $f_{\max} \geq f$, normally $t \leq t'$ and a performance loss may occur. To quantify the performance loss, we use the normalized performance loss δ, which is defined as:

$$\delta(f) = \frac{t' - t}{t}\,, \tag{7.6}$$

and substitute t and t' from Equations (7.4) and (7.5), respectively, into Equation (7.6) to obtain

$$\delta(f) = (\alpha \cdot w_{\text{on}} \cdot \frac{CPI_{\text{on}}}{f_{\max}}) \cdot \frac{1}{t} \cdot \frac{f_{\max} - f}{f}\,. \tag{7.7}$$

Equation (7.7) indicates that performance loss is determined by both processor frequency and workload characteristics. Within the context of DVFS scheduling, we summarize the workload characteristics using κ, which is defined as

$$\kappa = (\alpha \cdot w_{\text{on}} \cdot \frac{CPI_{\text{on}}}{f_{\max}}) \cdot \frac{1}{t}\,. \tag{7.8}$$

We interpret κ as an index of CPU intensiveness. When $\kappa = 1$, the workload is CPU bounded, and when $\kappa \approx 0$, the system is either idle, memory-bound, or I/O-bound.

Given a user specified performance loss bound δ^*, we identify the optimal frequency as the lowest frequency f^* that satisfies

$$f^* \geq \frac{\kappa}{\kappa + \delta^*} \cdot f_{max}\,. \tag{7.9}$$

7.4 Online Performance Management

In Equation (7.8) and (7.9), we assume the workload is given when calculating the workload characteristic index and the optimal frequency. Unfortunately, we normally do *not* know the next workload at run-time. Thus, we must predict the workload on the fly. Meanwhile, we also have to minimize or offset the performance loss due to false prediction.

7.4.1 Workload Prediction Algorithms

In this work, we use history-based workload prediction. During each interval, we collect a set of performance events and summarize them with a single metric κ. Then we predict the κ value for the future workload using the history values of κ.

Various prediction algorithms can be used. The simplest but most commonly used technique is the PAST [146] algorithm:

$$\kappa'_{i+1} = \kappa_i \,, \tag{7.10}$$

Here κ'_{i+1} is the predicted workload at the $(i+1)^{th}$ interval and κ_i is the measured workload at the i^{th} interval. The PAST algorithm works well for slowly varying workloads but incurs large performance and energy penalties for volatile workloads. To better handle volatility, two kinds of enhancements have been suggested for the PAST algorithm. The first enhancement is to use the average of the history values across more intervals [143]. The second enhancement is to regress the workload either over time [38] or over the frequencies [82]. A more complicated prediction algorithm is the proportional-integral-derivative controller (PID controller) [108], which addresses prediction error responsiveness, prediction overshooting, and workload oscillation by carefully tuning its control parameters.

In this paper, we consider an alternative algorithm called exponential moving average (EMA) [89] which predicts the workload using both history values and run-time profiling. The EMA algorithm can be expressed as:

$$\kappa'_{i+1} = (1-\lambda) \cdot \kappa'_i + \lambda \cdot \kappa_i \,, \tag{7.11}$$

where κ_i' is the predicted workload at the i^{th} interval, and λ is a smoothing factor that controls how much the prediction will depend on the current measurement κ_i.

7.4.2 Performance Loss Control

Two major factors for performance loss of run-time DVFS schedulers include the DVFS scheduling overhead and the misprediction of workload characteristics. While the effects of DVFS scheduling overhead are deterministic and could be decreased by reducing the number of power/performance mode transitions (possible techniques include workload smoothing, faster scheduler execution and larger control intervals), workload misprediction is inevitable due to the stochastic nature of the workload. Consequently, performance loss occurs when misprediction happens. For example, given a system whose highest frequency is $f_{\max} = 2.6GHz$ and lowest frequency is $f_{\min} = 1.0GHz$, consider a case where the predicted workload is $\kappa = 0$ and the actual workload is $\kappa = 1.0$, the actual performance loss during the i^{th} interval would be as high as 160%.

We address this problem by adapting the sampling interval and decreasing the weight of the intervals with possible large performance loss. Specifically, we decrease the sampling interval when the processor switches to a lower frequency and increase the sampling interval when the processor runs at a higher frequency. Specifically, we set the sampling interval $T_s'(f)$ at frequency f as:

$$T_s'(f) = max\{\frac{\delta(f)}{\delta^*}T_s,\ T_{\text{s0}}\}\,. \tag{7.12}$$

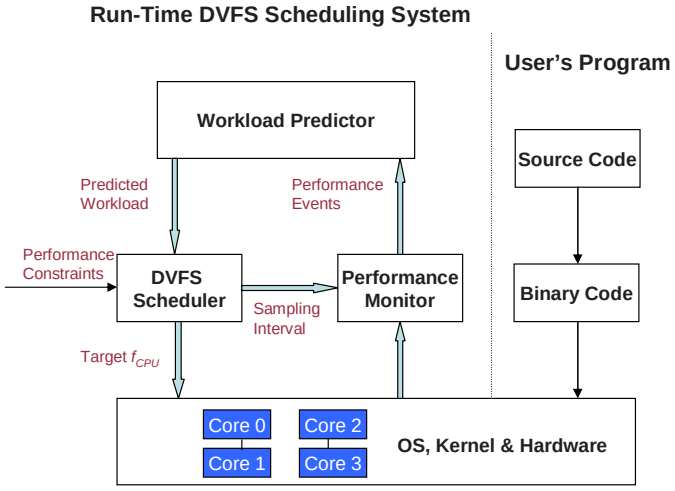


Figure 7.1: The implementation of CPU MISER

Here T_s is the standard sampling interval at $f_{\max}$; δ^* is the user-specified, performance-loss constraint; $\delta(f)$ is the potential performance loss at frequency f; and T_{s0} is an upper bound due to practical considerations.

7.5 System Design and Implementation of CPU MISER

Figure 7.1 shows the implementation of CPU MISER, a system-wide, run-time DVFS scheduler for multicore or SMP based power aware clusters. CPU MISER consists of three components: performance monitor, workload predictor, and DVFS scheduler.

The performance monitor periodically collects performance events using hardware counters provided by modern processors during each interval. The current version of CPU MISER monitors four performance events: retired instructions, L1 data cache accesses, L2 data cache accesses, and

memory data accesses.[1] The first three events capture the on-chip workload w_{on}, and the last event describes the off-chip memory access w_{mem}. Performance monitors are also used to approximate t_{IO} and t_0 from the statistics data provided by the Linux pseudo-files /proc/net/dev and /proc/stat.

The workload predictor first calculates κ using the performance data collected by performance monitors and then predicts κ with a workload prediction algorithm. In CPU MISER, the memory-access latency, τ_{mem}, is estimated using the *lat_mem_rd* tool provided in the the LMbench microbenchmark. We estimate α and CPI_{on} separately at run-time using Equation (7.5) and then use their product and Equation (7.8) to compute κ. Though CPU MISER supports several workload prediction algorithms, it uses the EMA algorithm by default. For the EMA algorithm, CPU MISER sets the smoothing factor to an empirical value of $\lambda = 0.5$. This is semantically equivalent to the proportional mode of a PID controller with a proportional gain $K_{\mathrm{P}} = 0.5$, i.e.,

$$\kappa'_{i+1} = \kappa'_i + 0.5 \cdot (\kappa_i - \kappa'_i)\,. \tag{7.13}$$

The DVFS scheduler determines the target frequency for each processor based on the predicted workload κ' and modifies processor frequency using the CPUFreq interface.[2] Since the processor only supports a finite set of frequencies, we empirically normalize the calculated frequency as follows:

$\forall f^* \in [f_1, f_2]$ where f_1 and f_2 is a pair of adjacent available CPU frequencies, we set $f^* = f_2$ if $f^* \in [f_1 + \frac{f_2 - f_1}{3}, f_2]$, and $f^* = f_1$ if $f^* \in [f_1, f_1 + \frac{f_2 - f_1}{3})$.

[1] We chose these performance events for AMD Athlon and Opteron processors. For other architectures with different numbers and types of counters, the performance events monitored may require adjustment.

[2] The CPUFreq Linux kernel subsystem allows users or applications to change processor frequency on the fly.

Current multicore processors are only capable of setting the same frequency for all cores. Thus, the DVFS scheduler chooses the highest calculated frequency among all cores for the targeted processor frequency.

One additional function of the DVFS scheduler is to adapt the sample frequency based on the current frequency as described in Section 7.1. In our current implementation, we use two sampling intervals: when the processor is using its lowest frequency, we empirically set the sampling interval to 50ms; otherwise we set the sampling interval as 250ms. We plan to study the effects of varying sampling frequency in future work.

7.6 Experimental Results and Discussions

7.6.1 Experimental Methodology

We evaluate CPU MISER on a 9-node power-aware cluster named ICE. Each ICE compute node has two dual-core AMD Opteron 2218 processors and 4GB main memory. Each core includes one 128KB split instruction and data L1 cache as well as one 1MB L2 cache. Each processor supports 6 power/performance modes as shown in Table 7.1. The nodes are interconnected with Gigabit Ethernet. We run SUSE Linux (kernel version 2.6.18) on each node. We use CPUFreq for the DVFS control interface and PERFCTR [1] for the hardware-counter access interface.

The programs we evaluate are the NAS Parallel Benchmarks. We use MPI (Message Passing Interface) as the programming model. The MPI implementation is MPICH Version 1.2.7. We note each experiment as XX.S.NP where XX refers to the code name, S refers to the problem size,

and NP refers to the number of processes. For example, FT.C.16 means running the FT code with problem size C on 16 processes. Since we used all cores on each node during the computation, only 4 nodes are needed to provide the 16 processors.

We measure the total system power (AC power) for each node using the *Watts Up? PRO ES* power meter. We record the power profile using an additional Linux machine. The power meter samples power every $1/4$ second and outputs the data to the Linux machine via an RS232 interface.

In all results, energy and performance values are normalized to the highest CPU speed (i.e., 2600MHz). In this section, we refer to energy as the total energy consumed by all the compute nodes, and to performance as the elapsed wall clock time. We repeat each experiment three times and report their average values.

7.6.2 Overall Energy and Performance Results

Table 7.2 presents the overall energy and performance results when running the NPB benchmarks. We run each code at each frequency shown in Table 7.1 (denoted as single speed control from this point), followed by one run with CPU MISER enabled, and another with CPUSPEED enabled.

Table 7.1: Power/performance modes available on a dual core dual processor cluster ICE

Frequency (MHz)	Voltage (V)
1000	1.10
1800	1.15
2000	1.15
2200	1.20
2400	1.25
2600	1.30

Table 7.2: Normalized performance and energy for the NAS benchmark suite. In each cell, the number on top is the normalized execution time, and the number on the bottom is the normalized energy. For CPU MISER, the user specified performance loss is $\delta^* = 5\%$.

Code	Frequency (MHz)						DVFS Scheduler	
	1000	1800	2000	2200	2400	2600	CPU MISER	CPUSPEED
BT.C.16	1.66	1.17	1.08	1.07	1.05	1.00	1.06	1.48
	1.06	0.88	0.84	0.90	0.96	1.00	0.95	1.07
CG.C.16	1.47	1.15	1.11	1.07	1.03	1.00	1.02	1.36
	0.98	0.88	0.88	0.91	0.94	1.00	0.93	0.95
EP.C.16	2.57	1.45	1.30	1.18	1.08	1.00	1.10	1.05
	1.57	1.07	1.00	0.98	0.98	1.00	0.99	1.01
FT.C.16	1.40	1.10	1.06	1.04	1.02	1.00	1.03	1.07
	0.92	0.84	0.83	0.88	0.94	1.00	0.89	0.92
IS.C.16	1.52	1.07	0.99	1.01	1.01	1.00	0.96	1.08
	1.01	0.82	0.79	0.85	0.93	1.00	0.80	0.78
LU.C.16	1.62	1.13	1.05	1.02	1.06	1.00	1.03	1.32
	1.03	0.86	0.83	0.86	0.96	1.00	0.94	1.01
MG.C.16	1.41	1.11	1.03	1.05	0.99	1.00	1.04	1.32
	0.92	0.84	0.81	0.87	0.90	0.98	0.92	0.92
SP.C.16	1.53	1.08	1.03	1.02	1.05	1.00	1.08	1.32
	1.00	0.84	0.81	0.87	0.96	1.00	0.98	0.97

CPUSPEED is a DVFS scheduler included in most Linux distributions. CPUSPEED periodically monitors CPU utilization from /proc/stat and sets the CPU frequency for the next period accordingly.

Table 7.2 shows CPU MISER can save significant energy without requiring any a priori information from the applications. The behavior of CPU MISER is captured by the theory discussed in Section7.1. The results also indicate that the benefits of CPU MISER vary significantly for different benchmarks. For codes with large amounts of communication and memory access, CPU MISER can save up to 20% energy with 4% performance loss. For codes that are CPU-bound (e.g., EP),

CPU MISER saves little energy since reducing processor frequency would impact performance significantly.

Figure 7.2 presents the results from Table 7.2 in graphical form. We observe that CPU MISER and single speed DVFS control result in similar performance slowdown and energy savings for BT, CG, and FT. For IS, CPU MISER performs better while single speed control performs better for LU, MG, SP, and EP. However, choosing the best single speed processor frequency requires either a priori information about the workload or significant training and profiling. Thus, the dynamic and transparent characteristics of CPU MISER are more amenable to use in systems with changing workloads.

In comparing CPU MISER to CPUSPEED, CPU MISER saves more energy than CPUSPEED, and its performance loss is controlled. In contrast, CPUSPEED may lose up to 40% performance with 7% energy increase. Thus, we conclude that CPUSPEED is *not* appropriate for system-wide DVFS scheduling in high-performance computing. To achieve optimal energy-performance efficiency, the rigorous theoretical analysis such as that used in CPU MISER is necessary for scheduler design.

7.6.3 Effects of Workload Prediction Algorithms

We have implemented several workload prediction algorithms in CPU MISER. Here we compare two of them: the PAST algorithm and the EMA algorithm. The results of these two algorithms for NPB benchmarks are shown in Figure 7.3. We observe that the EMA algorithm controls performance loss better than the PAST algorithm, while the PAST algorithm may save more energy.

The PAST algorithm responds to the current workload more quickly than the EMA algorithm, while the EMA has a tendency to delay its decision until having observed similar workload for several intervals. These decisions dampen reactions to dramatic workload changes that last for

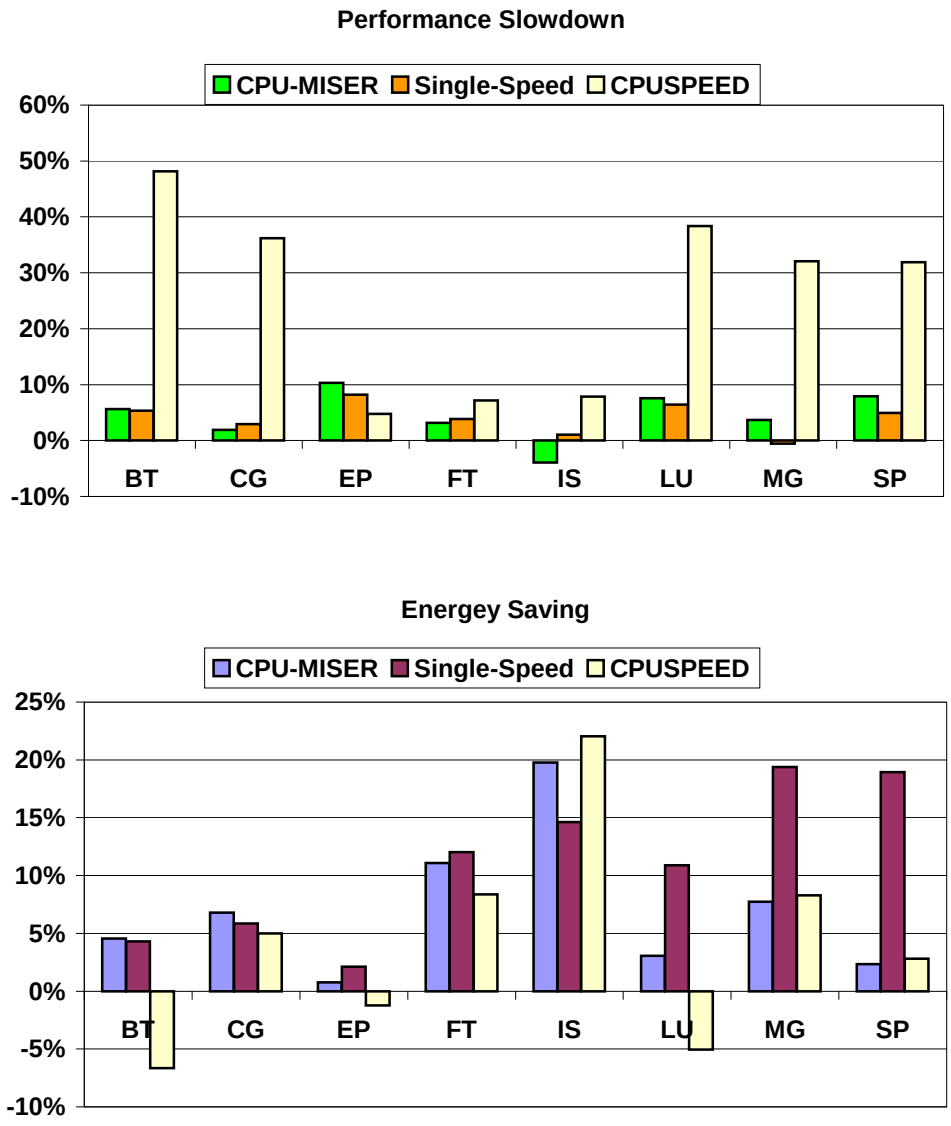


Figure 7.2: Performance slowdown and energy saving of CPU MISER, single speed control, and CPUSPEED. A negative performance slowdown indicates performance improvement and a negative energy saving indicates energy increases.

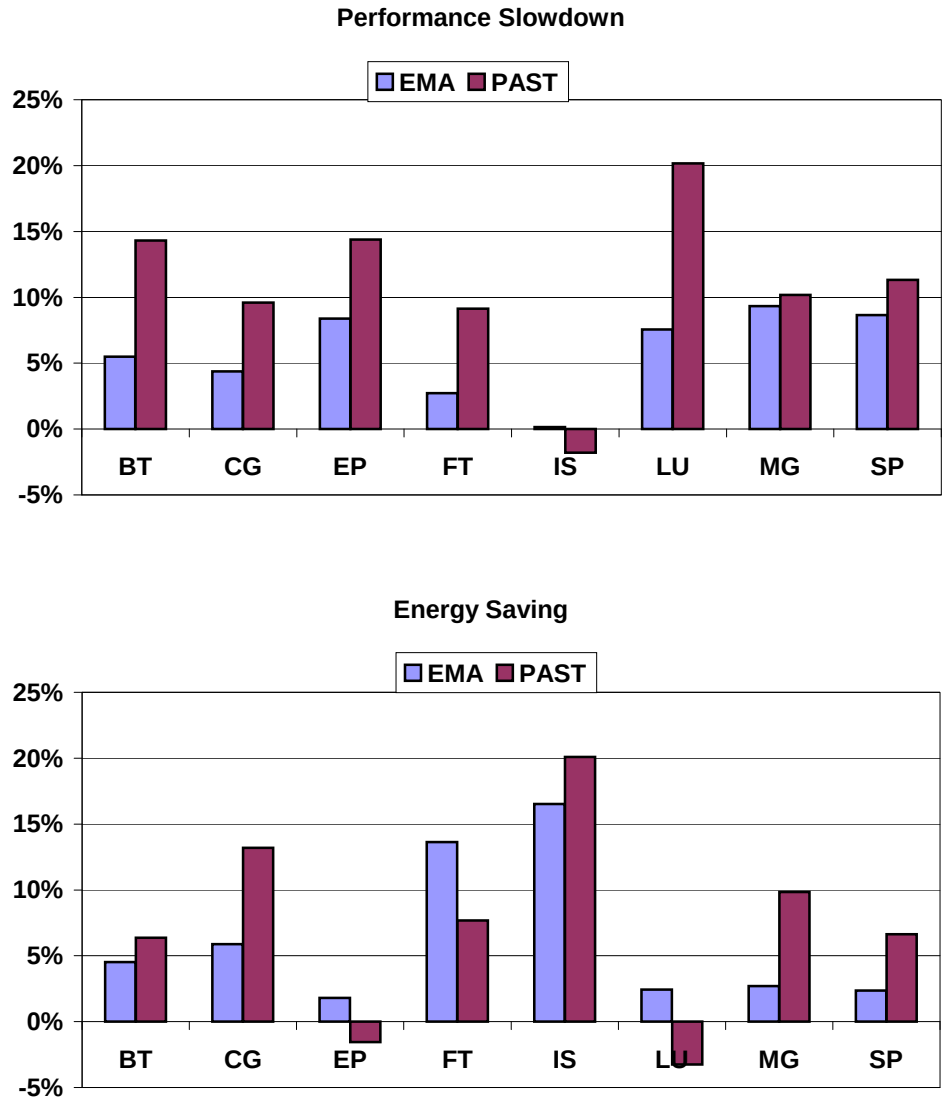


Figure 7.3: CPU MISER performance comparisons for the PAST algorithm and the EMA algorithm with $\lambda = 0.5$.

only very short durations, thereby reducing the chances of workload mispredictions. Furthermore, mispredictions are costly and often times lead to significant performance losses.

7.6.4 The Dynamic Behavior of CPU MISER

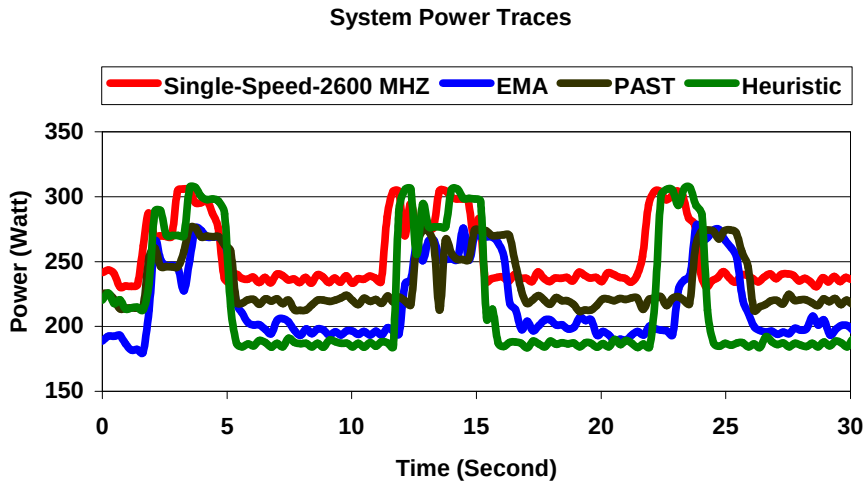
To better understand the behavior of CPU MISER, we trace the system power consumption and CPU frequency settings on one of the compute nodes. Figure 7.4 shows the traces for the FT benchmark.

Figure 7.4-(a) shows all tested DVFS schedulers can correctly capture workload phases but different DVFS schedulers may result in different system power consumptions. In contrast, CPU MISER not only scales down the processors during the communication phases, but also runs at a relatively lower frequency during the computation phases.

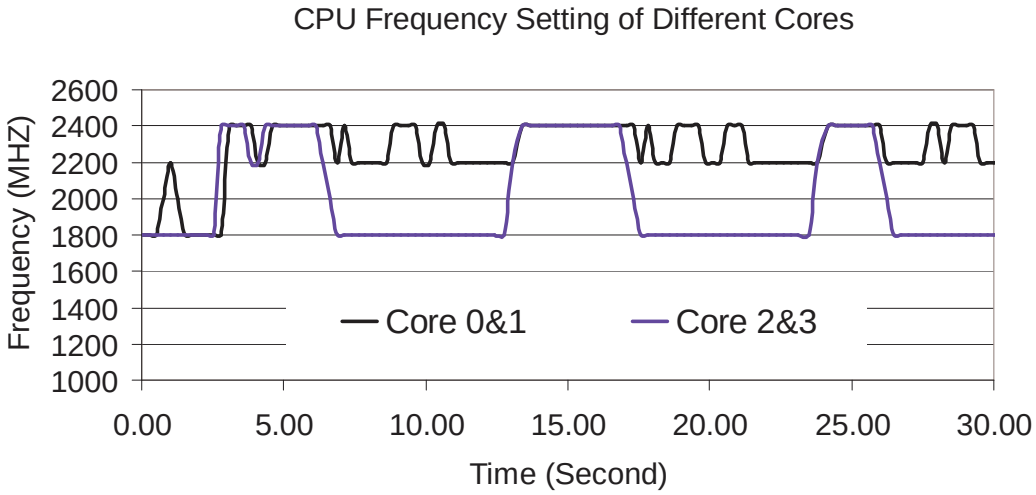
A detailed examination of CPU MISER in Figure 7.4-(b) shows that CPU MISER schedules CPU core 2 and core 3 to a lower frequency than core 0 and core 1. We believe that the major reason for this difference comes from the fact that core 0 runs operating system and performs communications between other nodes. Because two cores on the same processor have to be run at the same frequency, core 1 incurs some power inefficiency due to its co-scheduling with core 0, though CPU MISER does correctly predict the best frequencies for core 0 and core 1.

7.7 Chapter Summary

In summary, this chapter has presented the methodology, design, and implementation of a performance directed run-time power and performance management system (CPU MISER) for high performance computing.



(a) Traces of CPU power consumptions



(b) Traces of CPU frequencies at different cores

Figure 7.4: The power and frequency traces of CPU MISER with EMA algorithm

Our experimental results show that CPU MISER save up to 20% energy for NPB benchmarks and the performance loss for most applications is within the user-defined limit. This implies that the methodology we presented in this paper is very promising for large-scale deployment. We attribute these results to the underlying performance model and performance-loss management.

However, we also note further enhancement and tuning for CPU MISER is possible and we leave it the subject of future work.

Given that CPU MISER is built upon a generic framework and is transparent to both users and applications, we expect that it can be extended to many power-aware clusters for energy savings. In the future, we will refine the run-time parameter derivation and improve the prediction accuracy. We will also further investigate the impact of CPU MISER on more architectures and applications.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This thesis presents theories, techniques, and toolkits for analyzing, controlling, and improving the power-performance efficiency of high-end computing systems. Underlying this work is the observation that today's typical high-end computing systems are extraordinarily powerful in raw speed but unusually inefficient in terms of sustained performance and power consumption. To obtain an effective solution, we first study theories that describe the interactions between performance and power for applications, and then develop techniques that optimize them for higher level of efficiency. Specifically, our contributions and findings presented in this work include:

1. **Analytical models of point-to-point communication cost in distributed systems.** Memory and communication costs pose a major barrier to improve the computing efficiency of high-end systems. In our work, we found that the communication costs due to data movements in the middleware layer of scientific applications and data distributions across the memory

space are significant when compared to the total communication cost. Therefore, we developed log_nP and log_3P models to explicitly include the impacts of both middleware and data distributions into the calculation of point-to-point communication cost. Experiments on IA64 clusters indicate these software parameterized models can accurately predict distributed communication cost with less than 5% prediction errors. We also showed how our models can be practically applied to optimize algorithms and middleware designs for improved efficiency.

2. **Predictive models of performance scaling on power scalable clusters.** Efficient high-end computing requires accurate prediction of the impacts of different system configurations on application performance. For a power scalable cluster, a system configuration is defined by a combination of system size (i.e., number of compute nodes) and system speed (i.e., the processor frequency). In our work, we found that for certain applications it is possible to reduce power and energy without noticeable performance loss by using fewer computing nodes and/or running at a lower processor frequency. To explain such phenomena and also to form a theoretical foundation for efficient computing using power-aware clusters, we present a new power-aware speedup model that quantifies the performance effects of parallelism, power/performance modes, and their combinations. In this model, we decompose the workload by considering DOP (degree of parallelism) and the portion ofON-/OFF-chip memory access, and take into account the combined effects of both system and frequency on performance scaling. Our experimental study of several NPB codes on DVS-enabled power-aware clusters shows that our model can accurately capture application characteristics and predict

their performance under given system configurations. Coupled with a metrics for efficiency evaluation, this new speedup model can predict system configurations that result in power-performance efficiency.

3. **Power and energy profiling for distributed scientific applications.** As a measurement infrastructure for efficient high-end computing, we present a software and hardware toolkit named PowerPack for profiling, evaluating, and characterizing power and energy consumption of distributed parallel systems and applications. Through the combination of direct measurement, performance counter-based estimation, and flexible software control, PowerPack provides fast and accurate power-performance evaluation of large scale systems at component level and at function granularity. Typical applications of PowerPack include but are not limited to: 1) quantifying the power, energy, and power-performance efficiency of given distributed systems and applications; 2) understanding the interactions between power and performance at a fine granularity; 3) validating the effectiveness of candidate technology for efficiency improvement. In our work, we apply PowerPack to several case studies and obtain numerous insights on improving power-performance efficiencies of distributed scientific computing.

4. **Distributed DVFS scheduling and power-aware techniques for scientific computing.** Dynamic Voltage Frequency Scaling (DVFS) is a technology now present in high-performance microprocessors. DVFS works on a very simple principle: decreasing the supply voltage to the processor or processor frequency can reduce CPU power consumption. However, power and performance are two interdependent quantities; reducing power may simultaneously

decrease performance. Fortunately, in our work we found that different workload categories possess different power-performance behaviors and significant energy can be saved by adapting the power-performance modes to current workload phases. Specifically, we divided parallel scientific workloads' phases into three categories: CPU-bound, memory-bound, and IO-bound. We theoretically and experimentally show that by slowing down the professor frequency during memory-bound and IO-bound workload phases, we can reduce power, save energy and maintain performance. By applying the power-aware speedup model to localized, fine-grain performance modeling and prediction, we derive a methodology for performance-directed system-wide run-time power management. Based on this methodology, we designed a run-time DVFS schedulers (CPU MISER) to make DVFS-enabled power management for large-scale power-aware clusters automatic, transparent, and independent of application implementations.

Improving power-performance efficiency is important for future high-end computing systems. Though innovative computer architectures and revolutionary computing approaches may provide partial solutions to break efficiency barriers, we found that by exploiting the adaptive power during application execution, we can significantly improve the power-performance efficiency of high-end computing systems. Our approach required rigorously modeling the factors that account for inefficiency and intelligently adapting system configurations. Moreover, though the theories and techniques presented in this thesis are targeted at high-end computing, they should provide insight for power-performance improvement for other systems such as commercial data centers and their applications.

8.2 Future Work

In the near future, we will extend our work in the following two directions:

1. **Holistic performance and power management in high-end computing.** The contining demand for computing capability in scientific, economic, and engineering research drives high-end system design toward larger size and higher performance. Within less than three years, petascale systems will be a reality. How to make operation of these systems affordable, efficient and reliable is a challenging problem. Driven by market forces and technology convergence, we can predict that mainstream high-end systems will be built with technology and components that are available on computers targeted for consumers markets. Therefore we expect most components will include CPU, memory, disk, networking, and cooling systems that support power-aware features. Thus we can exploit all available power-aware components to improve the system's overall power-performance efficiency. Initially, we will consider the underlying theoretical model of each power-aware component, and then study performance and power optimization techniques when all major components become power-aware. Then we will study the holistic effects and coordination of adapting multiple components including processors, cores, threads, memories, and disks to the workload characteristics. Simultaneously, we will also study how to adapt our work to innovative architectures like multicore and heterogenous processors to bring even more efficiency to high-end computing.

2. **Performance, power, and temperature management for data centers.** Power-performance efficiency is critical to large data centers because: 1) utility costs spent on power and cooling are growing as a percentage of operating cost; 2) power and thermal densities place physical limits on the sustainable growth of data centers; and 3) of worldwide initiatives in energy conservation. Therefore, extending the work presented in this thesis to data centers is beneficial to both academic research and industry practice. For data centers, efficient computing requires considering physical constraints including performance, power, energy, thermals, and space. There are numerous research topics in this area. For example, an immediate problem is how to apply the performance-directed power-aware computing approach to data centers with the aim of saving energy while keeping the same level quality of service. Also, as data centers contain more power management controls like server consolidation and virtualization, additional innovation is possible to improve efficiency across the entire data center. As in the context of high-end computing, developing theories and tools to automatically and transparently exploit a wide varieties of power management techniques to achieve a higher level of efficiency for data center is highly desirable.

Bibliography

[1] http://user.it.uu.se/ mikpe/linux/perfctr/.

[2] http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufreq.html.

[3] N. Adiga, G. Almasi, and R. Barik. An Overview of the BlueGene/L Supercomputer. In *Supercomputing 2002*. Baltimore, MD, 2002.

[4] Adnan Agbaria, Yosi Ben-Asher, and Ilan Newman. Communication-Processor Tradeoffs in Limited Resources PRAM. *Algorithmica*, 43(3):276–297, 2002.

[5] A. Aggarwal, A. K. Chandra, and M. Snir. On Communication Latency in PRAM Computation. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 11–21. Santa Fe, New Mexico, United States, 1989.

[6] A. Aggarwal, A. K. Chandra, and M. Snir. Communication Complexity of PRAMs. *Theoretical Computer Science*, 71(1):3–28, 1990.

[7] Albert Alexandrov, Mihai F. Ionescu, K. Schauser, and Chris Scheiman. LogGP: Incorporating Long Messages into the LogP model. In *Seventh Annual Symposium on Parallel Algorithms and Architecture*, pages 95–105. Santa Barbara, CA, 1995.

[8] G. Allen, T. Dramlitsch, Ian Foster, T. Goodale, N. Karonis, Matei Ripeanu, Ed Seidel, and Brian Toonen. Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus. In *SC 2001*. Denver, CO, 2001.

[9] G.M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Spring Joint Computer Conference*, pages 483–485. Reston, VA, 1967.

[10] Anna Maria Bailey. Accelerated Strategic Computing Initiative (ASCI): Driving the Need for the Terascale Simulation Facility (TSF). In *Energy 2002 Workshop and Exposition*. Palm Springs, CA, 2002.

[11] D. H. Bailey. The Nas Parallel Benchmarks. *International Journal of Supercomputer Applications and High Performance Computing*, 5(3):63–73, 1991.

[12] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Technical report, NASA Ames Research Center Technical Report #NAS95020, December 1995.

[13] David H. Bailey. 21st Century High-End Computing. *In invited Talk Application, Algorithms and Architectures workshop for BlueGene/L*, 2002.

[14] Armin Baumker and Wolfgang Dittrich. Fully Dynamic Search Trees for An Extension of the BSP Model. In *the 8th SPAA*, pages 233–242. 1996.

[15] Frank Bellosa. The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems. In *Proceedings of 9th ACM SIGOPS European Workshop*. Kolding, Denmark, 2000.

[16] Pat Bohrer, Elmootazbellah N. Elnozahy, Tom Keller, Michael Kister, Charles Lefurgy, Chandler Mcdowell, and Ram Rajamony. The Case For Power Management in Web Servers. In R. Graybill and R. Melhem, editors, *Power Aware Computing*. Klewer Academic, IBM Research, Austin TX 78758, USA., 2002.

[17] Shekhar Borkar. Low Power Design Challenges for the Decade. In *Proceedings of the 2001 conference on Asia South Pacific design automation*, pages 293–296. ACM Press, Yokohama, Japan, 2001.

[18] David Brooks. *Computer Science 246: Advanced Computer Architecture*. 2003.

[19] David Brooks, Margaret Martonosi, John-David Wellman, and Pradip Bose. Power-Performance Modeling and Tradeoff Analysis for a High End Microprocessor. In *Workshop on Power-Aware Computer Systems (PACS2000, held in conjuction with ASPLOS-IX)*. Cambridge, MA, 2000.

[20] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *27th International Symposium on Computer Architecture*, pages P. 83–94. Vancouver, BC, 2000.

[21] D. C. Burger and Todd M. Austin. The SimpleScalar Toolset, Version 2.0. *Computer Architecture News*, 25(3):13–25, 1997.

[22] Surendra Byna, W. Gropp, Xian-He Sun, and R. Thakur. Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost. In *IEEE International Conference on Cluster Computing (Cluster 2003)*. Hong Kong, 2003.

[23] G. Cai and C. Lim. Architectural Level Power/Performance Optimization and Dynamic Power Optimization. In *Cool Chips Tutorial at 32nd ISCA*. 1999.

[24] K. W. Cameron, H. K. Pyla, and S. Varadarajan. Tempest: A Portable Tool to Identify Hot Spots in Parallel Code. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*. 2007.

[25] Kirk W. Cameron and Rong Ge. Predicting and Evaluating Distributed Communication Performance. In *2004 ACM/IEEE conference on Supercomputing (SC 2004)*. Pittsburgh, PA, 2004.

[26] Kirk W. Cameron, Rong Ge, and Xizhou Feng. High-Performance, Power-Aware Distributed Computing for Scientific Applications. *IEEE Computer*, 38(11):40–47, 2005.

[27] Kirk W. Cameron, Rong Ge, and Xizhou Feng. High-Performance, Power-Aware Distributed Computing for Scientific Applications. *IEEE Computer*, 38(11):40–47, 2005.

[28] Kirk W. Cameron, Rong Ge, Xizhou Feng, Drew Varner, and Chris Jones. POSTER: High-performance, Power-aware Distributed Computing Framework. In *Proceedings of 2004 ACM/IEEE conference on Supercomputing (SC 2004)*. 2004.

[29] Kirk W. Cameron, Rong Ge, and Xian-He Sun. log_nP and log_3P: Accurate Analytical Models of Point-to-Point Communication in Distributed Systems. *IEEE Transactions on Computers*, 56(3):314–327, 2007.

[30] Kirk W. Cameron and Xian-He Sun. Quantifying Locality Effect in Data Access Delay: Memory logP. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003)*. Nice, France, 2003.

[31] E. V. Carrera, E. Pinheiro, and R. Bianchini. Conserving Disk Energy in Network Servers. In *the 17th International Conference on Supercomputing*. 2003.

[32] Surendar Chandra. *Wireless Network Interface Energy Consumption Implications of Popular Streaming Formats*, volume 4673 of *Multimedia Computing and Networking (MMCN'02)*. The International Society of Optical Engineering, San Jose, CA, 2002.

[33] Jui-Ming Chang and Massoud Pedram. Energy Minimization Using Multiple Supply Voltages. *IEEE Trans. Very Large Scale Integr. Syst.*, 5(4):436–443, 1997.

[34] Guilin Chen, Konrad Malkowski, Mahmut Kandemir, and Padma Raghavan. Reducing Power with Performance Contraints for Parallel Sparse Applications. In *The First Workshop on High-Performance, Power-Aware Computing*. Denver, Colorado, 2005.

[35] Zhanping Chen, Mark Johnson, Liqiong Wei, and Kaushik Roy. Estimation of Standby Leakage Power in CMOS Circuits Considering Accurate Modeling of Transistor Stacks. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 239–244. ACM Press, New York, NY, USA, 1998.

[36] Jeonghwan Choi, Youngjae Kim, A. Sivasubramaniam, J. Srebric, Qian Wang, and Joonwon Lee. Modeling and Managing Thermal Profiles of Rack-mounted Servers with ThermoStat. In *IEEE 13th International Symposium on High Performance Computer Architecture*, pages 205–215. 2007.

[37] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. Dynamic Voltage and Frequency Scaling based on Workload Decomposition. In *the 2004 international symposium on Low power electronics and design*, pages 174 – 179. Newport Beach, California, USA, 2004.

[38] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. Fine-Grained Dynamic Voltage and Frequency Scaling for Precise Energy and Performance Trade-Off Based on the Ratio of Off-Chip Access to On-Chip Computation Times. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. 2004.

[39] Richard Cole and O. Zajicek. The APRAM: Incorporating Asynchronyinto the PRAM Model. In *the first annual ACM symposium on Parallel algorithms and architectures*, pages 25–28. Santa Fe, New Mexico, United States, 1989.

[40] David E. Culler, R. Karp, David A. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Fourth Symposium on Principles and Practices of Parallel Programming*, pages 1–12. ACM SIGPLAN, San Diego, CA, 1993.

[41] Anindya Datta, Aslihan Celik, Jeong G. Kim, Debra E. VanderMeer, and Vijay Kumar. Adaptive Broadcast Protocols to Support Power Conservant Retrieval by Mobile Users. In *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering*, pages 124–133. IEEE Computer Society, Washington, DC, USA, 1997.

[42] Ashutosh Dhodapkar, Chee How Lim, George Cai, and W. Robert Daasch. TEM2P2EST: A Thermal Enabled Multi-model Power/Performance ESTimator. In *the First International Workshop on Power-Aware Computer Systems*. Springer-Verlag London, UK, 2000.

[43] Bruno Diniz, Dorgival Guedes, Jr. Wagner Meira, and Ricardo Bianchini. Limiting the Power Consumption of Main Memory. *SIGARCH Comput. Archit. News*, 35(2):290–301, 2007.

[44] James Donald and Margaret Martonosi. Temperature-Aware Design Issues for SMT and CMP Architectures. In *Fifth Workshop on Complexity-Effective Design (WCED) held in conjunction with ISCA-31*. Munich, Germany, 2004.

[45] J. J. Dongarra, J. R. Bunch, C. B. Moller, and G. W. Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, PA, 1979.

[46] Jack Dongarra. Present and Future Supercomputer Architectures, 2004.

[47] Jack Dongarra. An Overview of High Performance Computing, 2005.

[48] Fred Douglis, P. Krishnan, and Brian Marsh. Thwarting the Power-Hungry Disk. In *USENIX Winter*, pages 292–306. 1994.

[49] Fred Douglis, Padmanabhan Krishnan, and Brian Bershad. Adaptive Disk Spin-down Policies for Mobile Computers. In *Proc. 2nd USENIX Symp. on Mobile and Location-Independent Computing*. 1995.

[50] Mootaz Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy Conservation Policies for Web Servers. In *USITS'03: Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems*, pages 8–8. USENIX Association, Berkeley, CA, USA, 2003.

[51] Xiaobo Fan, Carla S. Ellis, and Alvin R. Lebeck. Memory Controller Policies for DRAM Power Management. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 129–134. 2001.

[52] Xiaobo Fan, Carla S. Ellis, and Alvin R. Lebeck. The Synergy between Power-Aware Memory Systems and Processor Voltage Scaling. Technical Report TR CS-2002-12, Department of Computer Science Duke University, 2002.

[53] Keith Farkas, Jason Flinn, Godmar Back, Dirk Grunwald, and Jennifer Anderson. Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine. In *SIGMETRICS '00*. Santa Clara, CA, 2000.

[54] Mark E. Femal and Vincent W. Freeh. Boosting Data Center Performance Through Non-Uniform Power Allocation. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 250–261. IEEE Computer Society, Washington, DC, USA, 2005.

[55] W. Feng, M. Warren, and E. Weigle. Honey, I Shrunk the Beowulf! In *2002 International Conference on Parallel Processing (ICPP'02)*, pages 141–149. Vancouver, B.C., Canada, 2002.

[56] Xizhou Feng, Rong Ge, and Kirk W. Cameron. Power and Energy Profiling of Scientific Applications on Distributed Systems. In *IPDPS*. 2005.

[57] Xizhou Feng, Rong Ge, and Kirk W. Cameron. The Argus Prototype: Aggregate Use of Load Modules as a High density Supercomputer. *Concurrency and Computation: Practice and Experience*, 18, 2006.

[58] Krisztian Flautner, Steven K. Reinhardt, and Trevor N. Mudge. Automatic Performance Setting for Dynamic Voltage Scaling. In *Mobile Computing and Networking*, pages 260–271. 2001.

[59] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *17th ACM Symposium on Operating Systems Principles*. Kiawah Island Resort, SC, 1999.

[60] Jason Flinn and M. Satyanarayanan. PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications. In *the Second IEEE Workshop on Mobile Computer Systems and Applications*. 1999.

[61] S. Fortune and Wyllie. Parallelism in Random Access Machines. In *10th Annual ACM Symposium on Theory of Computing*, pages 114–118. ACM Press, San Diego, CA, 1978.

[62] Matthew I. Frank, A. Agarwal, and Mary K. Vernon. LoPC: Modeling Contention in Parallel Algorithms. In *Sixth Symposium on Principles and Practice of Parallel Programming*, pages 276–287. ACM SIGPLAN, Las Vegas, NV, 1997.

[63] M. Franklin and T. Wolf. Power Considerations in Network Processor Design. In *Network Processor Workshop in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, pages 10–22. 2003.

[64] Vincent W. Freeh, David K. Lowenthal, Feng Pan, and Nandani Kappiah. Using Multiple Energy Gears in MPI Programs on a Power-Scalable Cluster. In *10th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*. 2005.

[65] Rong Ge and Kirk W. Cameron. Power-Aware Speedup. In *The 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS'07)*. Long Beach, CA, 2007.

[66] Rong Ge, Xizhou Feng, and Kirk Cameron. Performance-constrained, Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters. In *2005 ACM/IEEE conference on Supercomputing (SC 2005)*. Seattle, WA, 2005.

[67] Rong Ge, Xizhou Feng, and Kirk W. Cameron. Improvement of Power-Performance Efficiency for High-End Computing. In *the first HPPAC workship in conjection with 19th IEEE/ACM International Parallel and Distributed Processing Symposium (IPDPS)*. Denver, Colorado, 2005.

[68] Rong Ge, Xizhou Feng, and Kirk W. Cameron. Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters. In *Proceedings of the ACM/IEEE Supercomputing 2005 (SC'05)*. 2005.

[69] Rong Ge, Xizhou Feng, Wu-Chun Feng, and Kirk W. Cameron. CPU MISER: a Performance-Directed, Run-Time System for Power-Aware Clusters. In *International Conference in Parallel Processing (ICPP) 2007 (to appear)*. Xian, China, 2007.

[70] P. B. Gibbons. A More Practical PRAM Model. In *the ACM Symposium on Parallel Algorithms and Architectures*, pages 158–168. Santa Fe, New Mexico, United States, 1989.

[71] Ananth Y. Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE concurrency*, 1(3):12–21, 1993.

[72] W. Gropp and E. Lusk. Reproducible Measurements of MPI Performance. In *PVM/MPI '99 User's Group Meeting*, pages 11–18. 1999.

[73] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mary Jane Irwin, N. Vijaykrishnan, and Mahmut Kandemir. Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach. In *Eighth International Symposium on High-Performance Computer Architecture (HECA'02)*, page P. 0141. Boston, Massachusettes, 2002.

[74] J. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31:532–533, 1988.

[75] J. Halter and F. Najm. A Gate-Level Leakage Power Reduction Method for Ultra-Low-Power CMOS Circuits. In *Proceedings of IEEE Custom Integrated Circuits Conference*, pages 475–478. 1997.

[76] H. Hanson, S.W. Keckler, Rajamani. K, S. Ghiasi, F. Rawson, and J. Rubio. Power, Performance, and Thermal Management for High-Performance Systems. In *HPPAC Workship in conjunction with IEEE International conference Parallel and Distributed Processing Symposium*, pages 1–8. Long Beach, LA, 2007.

[77] Taliver Heath, Ana Paula Centeno, Pradeep George, Luiz Ramos, and Yogesh Jaluria. Mercury and Freon: Temperature Emulation and Management for Server Systems. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 106–116. ACM Press, New York, NY, USA, 2006.

[78] HECRTF. Federal Plan for High-End Computing: Report of the High-End Computing Revitalization Task Force. Technical report, 2004.

[79] Inki Hong, Miodrag Potkonjak, and Mani B. Srivastava. On-Line Scheduling of Hard Real-Time Tasks on Variable Voltage Processor. In *ICCAD '98: Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 653–656. ACM Press, New York, NY, USA, 1998.

[80] Chung-Hsing Hsu. *Compiler-Directed Dynamic Voltage and Frequency Scaling for Cpu Power and Energy Reduction*. Ph.D. thesis, 2003. Director - Ulrich Kremer.

[81] Chung-Hsing Hsu and Wu chun Feng. Effective Dynamic Voltage Scaling Through CPU-Boundedness Detection. In *The 4th international workshop on Power-aware computer systems (PACS'04)*. 2004.

[82] Chung-Hsing Hsu and Wu chun Feng. A Power-Aware Run-Time System for High-Performance Computing. In *Proceedings of the ACM/IEEE Supercomputing 2005 (SC'05)*. 2005.

[83] Chung-Hsing Hsu and Wu-chun Feng. Towards Efficient Supercomputing: Choosing the Right Efficiency Metric. In *The First Workshop on High-Performance, Power-Aware Computing*. Denver, Colorado, 2005.

[84] Chung-Hsing Hsu and Ulrich Kremer. The Design, Implementation, and Evaluation of a Compiler Algorithm for CPU Energy Reduction. In *ACM SIGPLAN Conference on Programming Languages, Design, and Implementation (PLDI'03)*. San Diego, CA, 2003.

[85] Chung-Hsing Hsu, Ulrich Kremer, and Michael S. Hsiao. Compiler-Directed Dynamic Frequency and Voltage Scheduling. In *Power-aware computer systems (4th international workshop, PACS 2004*, pages 65–81. 2000.

[86] Chung-Hsing Hsu, Ulrich Kremer, and Michael S. Hsiao. Compiler-directed dynamic voltage/frequency scheduling for energy reduction in mircoprocessors. In *ISLPED*, pages 275–278. 2001.

[87] http://www.spec.org. The SPEC benchmark suite, 2002.

[88] Wei Huang, Shougata Ghosh, Siva Velusamy, Karthik Sankaranarayanan, and Kevin Skadron. HotSpot: a Compact Thermal Modeling Methodology for Early-Stage VLSI Design. *IEEE Trans. Very Large Scale Integr. Syst.*, 14(5):501–513, 2006.

[89] JS Hunter. The exponentially weighted moving average. *Journal of Quality Technology*, 18:203–210, 1986.

[90] Tomasz Imielinski, Monish Gupta, and Sarma Peyyeti. Energy Efficient Data Filtering and Communication in Mobile Wireless Computing. In *MLICS '95: Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, pages 109–120. USENIX Association, Berkeley, CA, USA, 1995.

[91] F. Ino, N. Fujimoto, and K. Hagihara. LogGPS: A Parallel Computational Model for Synchronization Analysis. In *PPoPP '01*, pages 133–142. Snowbird, Utah, 2001.

[92] Intel. Intel Pentium M Processor datasheet. 2004.

[93] Canturk Isci and Margaret Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 93. 2003.

[94] Mark C. Johnson and Kaushik Roy. Datapath Scheduling with Multiple Supply Voltages and Level Converters. *ACM Trans. Des. Autom. Electron. Syst.*, 2(3):227–248, 1997.

[95] Russ Joseph, David Brooks, and Margaret Martonosi. Live, runtime power measurements as a foundation for evaluating power/performance tradeoffs. In *Workshop on Complexity-effective Design*. Goteborg, Sweden, 2001.

[96] B.H.H. Juurlink and H.A.G. Wijshof. The E-BSP Model: Incorporating General Locality and Unbalanced Communication inti the BSP Model. In *2nd International Euro-Par Conference*, pages 339–347. Springer-Verlag, 1996.

[97] Stefanos Kaxiras and Georgios Keramidas. IPStash: a Power-Efficient Memory Architecture for IP-lookup. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 361. IEEE Computer Society, Washington, DC, USA, 2003.

[98] Thilo Kielmann, Henri E. Bal, and Kees Verstoep. Fast Measurement of LogP Parameters for Message Passing Platforms. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 1176–1183. Springer-Verlag, London, UK, 2000.

[99] M. Kondo, Y. Ikeda, and H. Nakamura. A High Performance Cluster System Design by Adaptive Power Control. In *HPPAC Workshop in conjunction with IEEE International conference Parallel and Distributed Processing Symposium*, pages 1–8. Long Beach, LA, 2007.

[100] LBNL. *Data Center Energy Benchmarking Case Study*. 2003.

[101] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. Power Aware Page Allocation. *SIGOPS Oper. Syst. Rev.*, 34(5):105–116, 2000.

[102] Kester Li, Roger Kumpf, Paul Horton, and Thomas Anderson. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. In *WTEC'94: Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, pages 22–22. USENIX Association, Berkeley, CA, USA, 1994.

[103] Peng Li, L. T. Pileggi, M. Asheghi, and R. Chandra. Efficient Full-Chip Thermal Modeling and Analysis. In *ICCAD '04: Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, pages 319–326. IEEE Computer Society, Washington, DC, USA, 2004.

[104] Yingmin Li, David Brooks, Zhigang Hu, and Kevin Skadron. Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 71–82. IEEE Computer Society, Washington, DC, USA, 2005.

[105] Min Yeol Lim, Vincent W. Freeh, and David K. Lowenthal. MPI and Communication - Adaptive, Transparent Frequency and Voltage Scaling of Communication Phases in MPI Programs. In *Proceedings of the ACM/IEEE Supercomputing 2006 (SC'06)*. 2006.

[106] Jiang Lin, Hongzhong Zheng, Zhichun Zhu, Howard David, and Zhao Zhang. Thermal Modeling and Management of DRAM Memory Systems. *SIGARCH Comput. Archit. News*, 35(2):312–322, 2007.

[107] Yann-Rue Lin, Cheng-Tsung Hwang, and Allen C.-H. Wu. Scheduling Techniques for Variable Voltage Low Power Designs. *ACM Trans. Des. Autom. Electron. Syst.*, 2(2):81–97, 1997.

[108] Bela Liptak. *Instrument Engineers' Handbook: Process Control*. Chilton Book Company, 1995.

[109] J. R. Lorch and A. J. Smith. PACE: A New Approach to Dynamic Voltage Scaling. *Ieee Transactions on Computers*, 53(7):856–869, 2004.

[110] Jacob R Lorch and Alan Jay Smith. Software Strategies for Portable Computer Energy Management. *IEEE Personal Communications Magazine*, 5(3):60–73, 1998 1998.

[111] Yan Luo, Jia Yu, Jun Yang, and Laxmi Bhuyan. Low Power Network Processor Design Using Clock Gating. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 712–715. ACM Press, New York, NY, USA, 2005.

[112] K. Malkowski, G. Link, Padma Raghavan, and M.J. Irwin. Load Miss Prediction - Exploiting Power Performance Trade-offs. In *HPPAC07 in conjunction with IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages pp.1–8. 2007.

[113] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX 1996 Annual Technical Conference*. San Diego, CA, 1996.

[114] Francisco Javier Mesa-Martinez, Joseph Nayfach-Battilana, and Jose Renau. Power Model Validation Through Thermal Measurements. *SIGARCH Comput. Archit. News*, 35(2):302–311, 2007.

[115] Gordon Moore. Cramming More Components onto Integrated Circuits. *Eletronics*, 38(8):114–117, 1965.

[116] Csaba A. Moritz and Matthew I. Frank. LoGPC: Modeling Network Contention in Message-Passing Programs. In *SIGMETRICS '98*, pages 254–263. Madison, WI, 1998.

[117] Srinath R. Naidu and E.T.A.F. Jacobs. Minimizing Stand-By Leakage Power in Static CMOS Circuits. In *Proceedings of Design, Automation, and Test in Europe (DATE '01)*, page 0370. 2001.

[118] NERSC. DOE Greenbook - Needs and Directions in High-Performance Computing for the Office of Science. Technical report, 2005.

[119] Vijay S. Pai and Sarita Adve. Code Transformations to Improve Memory Parallelism. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1999.

[120] Vivek Pandey, W. Jiang, Y. Zhou, and R. Bianchini. DMA-Aware Memory Energy Management. In *The Twelfth International Symposium on High-Performance Computer Architecture*, pages 133–144. 2006.

[121] Christos Papadimitriou and Mihalis Yannakakis. Towards an Architecture-Independent Analysis of Parallel Algorithms. In *the twentieth annual ACM symposium on Theory of computing*, pages 510–513. ACM Press, Chicago, Illinois, United States, 1988.

[122] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Fancisco, CA, 3rd edition, 2003. CPI formulas found in pages 35-38.

[123] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kale. NAMD: Biomolecular Simulation on Thousands of Processors. In *IEEE/ACM SC2002*, pages 36–49. Baltimore, Maryland, 2002.

[124] PTOOLS. Performance API Home Page, May 1999.

[125] K. Puttaswamy and G.H. Loh. Thermal Herding: Microarchitecture Techniques for Controlling Hotspots in High-Performance 3D-Integrated Processors. In *IEEE 13th International Symposium on High Performance Computer Architecture*, pages 193–204. 2007.

[126] V. C. Ravikumar, R.N. Mahapatra, and Laxmi Narayan Bhuyan. EaseCAM: An Energy and Storage Efficient TCAM-Based Router Architecture for IP Lookup. *IEEE Transactions on Computers*, 54(5):521–533, 2005.

[127] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, Fall 1995, 1995.

[128] Rafael H. Saavedra and Alan Jay Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Run Times. *IEEE Transactions on Computers*, 44(10):1223–1235, 1995. Dot-product microbenchmark extension to cache and TLB modeling.

[129] H. Sakagami, H. Murai, Y. Seo, and M. Yokokawa. TFLOPS three-dimensional fluid simulation for fusion science with HPF on the Earth Simulator. In *SC2002*. 2002.

[130] Suresh Singh, Mike Woo, and C. S. Raghavendra. Power-aware routing in mobile ad hoc networks. In *MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 181–190. ACM Press, New York, NY, USA, 1998.

[131] S. W. Son, M. Kandemir, and A. Choudhary. Software-Directed Disk Power Management for Scientific Applications. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, pages 4b–4b. IEEE Computer Society, Washington, DC, USA, 2005.

[132] Rob Springer, David K. Lowenthal, Barry Rountree, and Vincent W. Freeh. Minimizing Execution Time in MPI Programs on an Energy-Constrained, Power-Scalable Cluster. In *11th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 2006.

[133] M. Stemm and R. H. Katz. Measuring and Reducing Energy Consumption of Network Interfaces in Hand-Held Devices. *IEICE Transactions on Communications*, E80-B(8):1125–31, 1997.

[134] Thomas Stricker and Thomas Gross. Optimizing Memory System Performance for Communication in Parallel Computers. In *ISCA 95*, pages 308–319. Santa Margherita Ligure, Italy, 1995.

[135] Xian-He Sun and Lionel Ni. Scalable Problems and Memory-Bounded Speedup. *Journal of Parallel and Distributed Computing*, 19:27–37, 1993.

[136] V. Tiwari, D. Singh, S. Rajgopal, G.Mehta, R.Patel, and F.Baez. Reducing Power in High-Performance Microprocessors. In *Proceedings of the 35th Conference on Design Automation*, pages 732–737. ACM Press, San Francico, California, 1998.

[137] Matthew E. Tolentino, Joseph Turner, and Kirk W. Cameron. Memory-miser: a Performance-Constrained Runtime System for Power-Scalable Clusters. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 237–246. ACM Press, New York, NY, USA, 2007.

[138] top500. 27th Edition of TOP500 List of Worlds Fastest Supercomputers Released: DOE/LLNL BlueGene/L and IBM gain Top Positions, 2006.

[139] P. de la Torre and C. P. Kruskal. Submachine Locality in the Bulk Synchronous Setting. In *Euro-Par'96*, pages 1123–1124. Springer Verlag, 1996.

[140] Manish Vachharajani, Neil Vachharajani, David A. Penry, Jason A. Blome, and David I. August. Microarchitectural Exploration with Liberty. In *35th International Symposium on Microarchitecture (Micro-35)*. 2002.

[141] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[142] Enrique Vargas. High Availability Fundamentals, 2000.

[143] Ankush Varma, Brinda Ganesh, Mainak Sen, Suchismita Roy Choudhury, Lakshmi Srinivasan, and Jacob Bruce. A Control-Theoretic Approach to Dynamic Voltage Scheduling. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems (CASE'03)*. 2003.

[144] N. Vijaykrishnan, M. Kandemir, M. Irwin, H. Kim, and W. Ye. Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower. In *27th International Symposium on Computer Architecture*. Vancouver, British Columbia, 2000.

[145] T.-Y Wang and C. C.-P. Chen. 3-D Thermal-ADI: A Linear-Time Chip Level Transient Thermal Simulator. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, 21(12):14341445, 2002.

[146] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *Proceedings of the First Symposium on Operating System Design and Implementation (OSDI'94)*. November 1994.

[147] Andreas Weissel and Frank Bellosa. Process Cruise Control-Event-Driven Clock Scaling for Dynamic Power Management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002)*. Grenoble, France, 2002.

[148] Qiang Wu, Margaret Martonosi, Douglas W. Clark, Vijay Janapa Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. Dynamic-Compiler-Driven Control for Microprocessor Energy and Performance. *IEEE Micro*, 26(1):119–129, 2006.

[149] Qiang Wu, V.J. Reddi, Youfeng Wu, Jin Lee, Dan Connors, David Brooks, Margaret Martonosi, and Douglas W. Clark. A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In *the 38th IEEE/ACM International Symposium on Microarchitecture (MICRO-38)*, pages pp. 271–282. Barcelona, Spain, 2005.

[150] W. Ye, Narayanan Vijaykrishnan, Mahmut T. Kandemir, and Mary Jane Irwin. The Design and Use of Simplepower: a Cycle-Accurate Energy Estimation Tool. In *Design Automation Conference*, pages 340–345. 2000.

[151] Qingbo Zhu, Zhifeng Chen, Lin Tan, Yuanyuan Zhou, Kimberley Keeton, and John Wilkes. Hibernator: Helping Disk Array Sleep Through the Winter. In *the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. 2005.

[152] Qingbo Zhu and Yuanyuan Zhou. Power Aware Storage Cache Management. *IEEE Transactions on Computers (IEEE-TC)*, 54(5):587–602, 2005.