ABSTRACT

KAPPIAH, NANDINI: Just-in-Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs (Under the direction of Assistant Professor Dr. Vincent W. Freeh).

Although users of high-performance computing are most interested in raw performance, both energy and power consumption have become critical concerns. As a result improving energy efficiency of nodes on HPC machines has become important and the importance of *power-scalable clusters*, where the frequency and voltage can be dynamically modified, has increased.

This thesis investigates the energy consumption and execution time of applications on a power-scalable cluster. It studies intra-node and inter-node effects of memory and communication bottlenecks. Results show that a power-scalable cluster has the potential to save energy by scaling the processor down to lower energy levels. This thesis presents a model that predicts the energy-time trade-off for larger clusters.

On power-scalable clusters, one opportunity for saving energy with little or no loss of performance exists when the computational load is not perfectly balanced. This situation occurs frequently, as keeping the load balanced between nodes is one of the long standing fundamental problems in parallel and distributed computing. However, despite the large body of research aimed at balancing load both statically and dynamically, this problem is quite difficult to solve. This thesis presents a system called *Jitter* that reduces the frequency on nodes that are assigned less computation and therefore have idle time or *slack time*. This saves energy on these nodes, and the goal of Jitter is to attempt to ensure that they arrive "just in time" so that they avoid increasing overall execution time. Specifically, we dynamically determine which nodes have enough *slack* time so that they can be slowed down—which will greatly reduce the consumed energy on that node. Thus a superior energy-time trade-off can be achieved.

This thesis studies a suite of MPI benchmarks, which are profiled, gathering information about the computation and communication occuring in the application. This information is used to analyse various energy-time trade-offs of benchmark suite. This thesis also proposes an algorithm that exploits load-imbalance to reduce energy consumption and minimize delays for parallel applications. This algorithm is validated on a large variety of benchmarks.

Just-in-Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs

by

Nandini Kappiah

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

Raleigh

2005

Approved By:

Dr. Gregory T. Byrd	Dr. Frank Mueller

Dr. Vincent W. Freeh
Chair of Advisory Committee

Biography

Nandini Kappiah was born on 4^{th} December 1980. She is from Bangalore, India, and she received her Bachelor of Engineering in Information Science from Visweswaraiah Technological University located in the same city, in 2003. She started her graduate studies at North Carolina State University in Fall 2003. With the defense of this thesis, she receives the Master of Science degree in Computer Science from NCSU in August 2005.

Acknowledgements

I would like to thank my advisor Dr. Vincent W. Freeh for his guidance and support throughout my research and his assistance in writing this thesis. I would like to thank Dr. Frank Mueller and Dr. Gregory T. Byrd for being on my advisory committee. I would like to thank Dr. David K. Lowenthal, Associate Professor, University of Georgia, for his invaluable help and guidance during my research. I would like to thank Feng Pan and Mark E. Femal, my fellow students in the AMPERE research group for all the help they provided me in the duration of this project. I would like to thank Nandan Tammineedi for all his help during my thesis preparation. Last but not the least, I would like to thank my parents for their many words of encouragement throughout my Masters education.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The tremendous increase in computer performance has come with an even greater increase in power usage. As a result, power consumption is a primary concern. According to Eric Schmidt, CEO of Google, what matters most to Google "is not speed but power—low power, because data centers can consume as much electricity as a city" [34]. This does not imply speed is not important, rather that excessive power limits performance. Such a limit might exist due to either a limited power supply or a limited capacity to dissipate and remove heat. Additionally, reducing the energy and cooling costs can be a high priority. Regardless of the reason, a power constraint is a *performance-limiting* factor.

As a result, power-aware computing has gained traction in the high-performance computing (HPC) community. Recently, low-power, high-performance systems have been developed to stem the ever-increasing demand for energy. We are most interested in clusters composed of microprocessors that support frequency *and* voltage scaling. Such systems increase the energy efficiency of nodes at lower frequency-voltage settings, which we term lower *energy gears* in this thesis. This either reduces the energy required to complete a task, or conversely increases the number of tasks that can be performed with a given amount of energy. Thus, one can dynamically adjust the trade-offs between performance and energy savings.

This thesis first investigates the tradeoff between energy and performance (execution time) for HPC systems. We evaluate trade-offs on a real power-scalable cluster. Each node is equipped with an AMD Athlon-64 processor that supports frequency and voltage scaling. Towards this goal, we analyzed benchmarks, both serial and parallel, to determine the relationship between voltage and frequency settings and execution time. Our results show that all of our tests have an *energy-time tradeoff*, meaning that a decrease in energy is possible but it comes at the cost of increased time. Not every energy-time tradeoff is desirable, as some offer little energy savings and large time penalties. However, more than half of these tests show a savings that is equal to the penalty (*e.g.*, 10% less energy and 10% more time), and some show an energy savings much greater than the time penalty (*e.g.*, 10% less energy and 7% more time).

While the emperical data gathered on a 10-node cluster is encouraging, it is unclear what performance in time and energy can be expected for larger power-scalable cluster. As we do not have access to more than ten nodes, we address this issue by developing a simulation model. We model computation and communication of parallel programs on large clusters by using a combination of Amdahl's law, trace gathering and source code inspection and use this model to predict energy-time trade-offs for large clusters.

This thesis also describes the design and implementation of a system that saves energy on parallel or distributed programs which suffer from load imbalance. Such an application usually contains one or more instances where two or more nodes synchronize with each other to exchange data or status information. Nodes that complete their work early and wait at these synchronization points provide an opportunity to save energy. With frequency scaling, such a node can shift into a reduced power and performance state so that computation is potentially completed *just in time* for the unblocking event. We present a dynamic, adaptive system for just-in-time dynamic voltage scaling (DVS), called *Jitter*. It monitors the time a program waits for external events and then adjusts the CPU frequency (and voltage) to minimize wait time and energy consumption.

The Jitter system saves 8% energy, with a 2.6% time penalty, on a unbalanced program, and it does this without modification to the application or the communication library. Furthermore, our system is within 5% of the hand-tuned, *i.e.*, optimal, solution. Additionally, we show that our solution adapts transparently to changes in load, which a hand-tuned solution or a premeditated solution cannot do.

1.1 The Case for Power-Aware High Performance Computing

Power-aware computing has flourished over the last decade in many ways, especially in mobile devices. However, it has gained little traction in the High Performance Computing (HPC) community because the HPC community has vastly different goals than those of mobile users. Since computational scientist are using HPC to gain maximum performance, many may resist any mechanism that increase execution time.

We believe that a significant subset of HPC programmers will be forced to accept less performance in order to conserve power. This will occur because the large and growing power consumption of current clusters at supercomputer centers—which are not immune to economic pressure—will eventually come to bear on the clusters. In addition, a sustained high performance, hence high power, consumption produces excessive heat, which may cause failure of the CPU and other hardware components. Emperical data from two leading vendors indicates that the failure rate of a computer node doubles with every $10^o\ C$ increment [16]. The total product cost also increases since more complex cooling and packaging systems are required to serve high-energy systems. Intel estimates that more than \$1/W will be incurred if the CPU's power dissipation exceeds 35-40W [46]. We argue that the above cited reasons will cause the programmers to move to lower-power machines such as Green Destiny [39, 50].

Green Destiny, which consists of a cluster of Transmeta processors, uses less power and energy than a conventional supercomputer. However, it also provides less performance. In particular, Green Destiny consumes about one third less energy per unit performance than the ASCI Q machine. However, because Green Destiny uses a slower (and cooler) microprocessor, ASCI Q is about 15 times faster per node (200 times overall) [50]. A reduction in performance by such a factor would surely be unacceptable from the point of view of many HPC programmers.

We believe that computational scientists would be more likely to consider energy reduction if we start from a baseline of a high performance, scalable microprocessor. Then from there we attempt to reduce power with only a small performance loss. The emergence of high-performance scalable microprocessors, such as Athlon-64/Opteron [2] makes this possible. This microprocessor has performance that is on a par with high-end, fixed frequency desktop microprocessors, but its power consumption can be reduced to 25% of

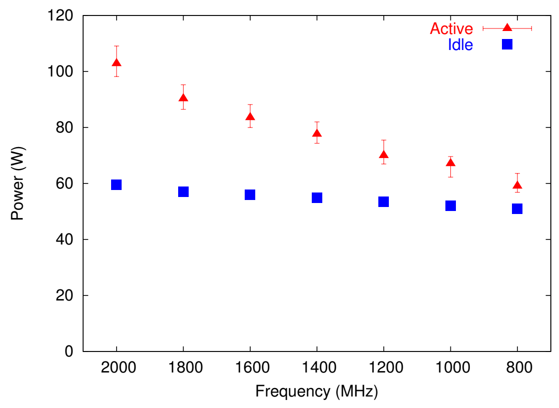


Figure 1.1: Cost Benefit Analysis of frequency reduction

the maximum using frequency and voltage scaling [51].

1.2 Saving Power and Energy with Gear Selection

High-performance computing (HPC) applications and machines are inefficient. For example, the 2004 Gordon Bell Prize [6], which is given annually to the application with the greatest performance, was awarded to one that sustained 46% parallel efficiency. This means that on average, a microprocessor in a supercomputer executing the world champion application was blocked and idling more than half of the time. In such a situation, a traditional microprocessor races forward at full speed to the next blocking point and waits. While blocked, the machine is consuming energy, but not performing any useful work. Furthermore, when a program has good parallel efficiency, it may not be efficient sequentially (*e.g.*, it may stall waiting for the memory subsystem).

As an analogy, consider a car speeding between stop lights. Because a traditional microprocessor has only one gear, which uses full power and provides the maximum performance, it must race between metaphorical stop lights. Recently, manufacturers have begun producing microprocessors with multiple energy gears. With such a microprocessor, a computer can shift into a reduced power and performance gear so that computation is potentially completed just in time for the unblocking event—*i.e.,* arriving just as the light turns green.

This method of execution saves energy in two ways. First, (wasted) energy consumed while waiting is eliminated. Second, a microprocessor uses less energy per operation at lower energy gears. This is because while performance is linearly proportional to clock frequency, power consumption is proportional to frequency *and* voltage squared. Thus, at a slower gear, a microprocessor will consume less energy per operation than at a faster gear [36]. Again, this follows the analogy of an automobile—it takes less gasoline to travel at 50 MPH for 2 hours than at 100 MPH for one hour, partly because drag is proportional to velocity squared. Figure 1.1 shows the active and idle power consumption of the Athlon-64, for all the gears. It may appear that node performance should be reduced solely based on its percentage of idle time, but this not the case. Doing so blindly will likely mean that events that are on the critical path are delayed, which increases execution time by the amount of that delay. In this thesis, we investigate the trade-off between energy and performance (execution time) for a wide range of applications and specifically look at programs that have load-imbalance as an opportunity to save energy.

1.3 Contributions

One contribution is a profiler for MPI. This profiler interposes between the application and the MPI library, transparently to both, and gathers information about the computation and communication occuring in the application. We have also created an analyser which obtains trace information from the profiler and analyzes various energy-time trade-offs of each application. We have validated this profiler and analyser on a suite of benchmarks, which have provided encouraging energy-time trade-offs. We have also created a software infrastructure which can alter online the performance of an Athlon-64 processor by shifting to different frequency and voltage settings. This thesis also proposes an algorithm that exploits load-imbalance and to reduce energy consumption and minimize delays for parallel applications. We have validated the software infrastructure and algorithm on a large variety of benchmarks.

The remainder of this thesis is organized as follows. Chapter 2 discusses the related work. Chapter 3 discusses our approach to saving power in high performance computing. Chapter 4 describes our model for predicting energy-time trade-offs for large clusters. Chapter 5 discusses the design and implementation of the Jitter system. Chapter 6 describes performance results of the Jitter systems on a wide variety of benchmarks. Finally, Section 7 summarizes this thesis.

Chapter 2

Related Work

There has been a voluminous amount of research performed in the general area of energy management, in both server/desktop and mobile systems. In addition, there has been a lot of work on dynamic load balancing. Another allied field is real-time DVS that applies the principles of voltage and frequency scaling to time-critical applications,

2.1 Server/Desktop Systems

Several researchers have investigated saving energy in server-class systems. The basic idea is that if there is a large enough cluster of machines, such as in hosting centers, energy management can become an issue. Chase *et al.* [10] illustrate a method to determine the aggregate system load and then determine the minimal set of servers that can handle that load. All other servers are transitioned to a low-energy state. A similar idea leverages work in cluster load balancing to determine when to turn machines on or off to handle a given load [42, 43]. Elnozahy *et al.* [15] investigated the policy in [42] as well as several others in a server farm. They found that when each node independently sets its voltage, the performance was almost as good as more complicated schemes that required coordination between server nodes. Complementary work shows that power and energy management are critical for commercial workloads, especially web servers [8, 32]. Additional approaches have been taken to include DVS [14, 45] and request batching [14]. Sharma *et al.* [45] apply real-time techniques to web servers in order to conserve energy while maintaining quality of service.

Our work differs from most prior research because it focuses on HPC applications and installations, rather than commercial ones. A commercial installation tries to reduce cost while servicing client requests. On the other hand, an HPC installation exists to speedup an application, which is often highly regular and predictable. One HPC effort that addresses the memory bottleneck is given in [26]; however, this is a purely static approach.

In server farms, disk energy consumption is also significant. Much work has been done on saving disk energy, including reducing spindle speed [9], modulating the speed of the disk [22, 23], improving cache performance so that the disk can be kept in a low power state more often [58], and using program counter techniques to infer the disk access pattern [20]. In addition, there are schemes to aggregate disk accesses, which again allows the disk to be kept in a lower power state more often; one of these uses an integrated compiler/run time approach [24], and the other uses prefetching [40].

Our work is complementary to techniques that save disk energy. Specifically, while the disk can incur a significant amount of power in HPC applications, the CPU is typically a much larger power consumer. Furthermore, the CPU is the only commercially available component that has multiple energy gears or performance states. Therefore we focus solely on scaling the CPU.

There are also a few high-performance computing clusters designed with energy in mind. One is BlueGene/L [1], which uses a "system on a chip" to reduce energy. Another is Green Destiny [50], which uses low-power Transmeta nodes. A related approach is the Orion Multisystem machines [39], though these are targeted at desktop users. The latter two approaches sacrifice performance in order to save energy by using less powerful machines.

Finally, our prior work was an evaluation-based study that focused on exploring the energy/time tradeoff in the NAS suite [19]. Specifically, we found that using a *single* slower gear was in some cases able to save energy with little time delay. We also found a significant benefit of varying the gear per phase, and developed an algorithm for choosing the assignment of gear to phase [18].

2.2 Mobile Systems

There is also a large body of work in saving energy in mobile systems; most of the early research in energy-aware computing was on these systems. Here we detail some of these projects.

At the system level, there is work in trying to make the OS energy-aware through making energy a first class resource [47, 13, 11]. Our approach differs in that we are concerned with saving energy in a *single* program, not a set of processes. One important avenue of application-level research on mobile devices focuses on collaboration with the OS (e.g., [53, 55, 3]). Such application-related approaches are complementary to our approach.

In terms of research on device-specific energy savings, there is work in the CPU via dynamic voltage scaling (e.g., [17, 41, 21]), the disk via spindown (e.g., [25, 12]), and on the memory or network [31, 29]. The primary distinction between these projects and ours is that energy saving is typically the primary concern in mobile devices. In HPC applications, performance is still the primary concern.

2.3 Dynamic Load Balancing

There has been a large volume of work in load balancing in parallel programs. It should be noted that application programmers themselves often employ a specific load balancing scheme. Here, we focus on runtime system techniques to balance the workload. A few of these include shared-memory systems such as SUIF-Adapt [37] as well as MPI-based ones such as Adaptive MPI [7, 30], Dyn-MPI [52], and Tern [27].

The important point here is that dynamic load balancing, whether it is employed at the application or system level, decreases the need for Jitter. However, many applications are at least somewhat resistant to dynamic load balancing; this is proven by the fact that the balancing must be done repeatedly throughout the lifetime of the application. In any case, Jitter could be integrated with the load balancing systems to gain information about the estimated amount of work per node.

2.4 Real Time DVS

The closest work to ours comes from the real-time community, where the goal is to meet a given deadline while running each task at the lowest possible energy gear. One example is in [56], where an approach to slack sharing in multiprocessor real-time systems is presented. Another is [57], which extends this work to handle AND/OR graphs. In [35] power-aware scheduling of periodic hard real-time tasks using dynamic voltage scaling by creating a static (off-line) solution to compute the optimal speed is described, assuming worst-case workload for each arrival and an on-line speed reduction mechanism to reclaim energy by adapting to the actual workload and a speculative speed adjustment mechanism to anticipate early completions of future executions by using the average-case workload information. The problem of static and dynamic variable voltage scheduling of tasks in heterogeneous distributed real-time embedded systems is discussed in [33]. The static scheduling algorithm constructs a variable voltage schedule via heuristics based on critical path analysis. The algorithm redistributes the slack in the initial schedule and refines task execution order in an efficient manner.

Chapter 3

Saving Energy in HPC applications with DVS

The energy management issue can be approached from either the hardware or software perspective. In the hardware perspective, multiple power states are integrated in the micro-architecture, circuit or device levels to take advantage of hardware power-saving features. Some industry standard have been developed such as the Advanced Power Management *APM* specification and the Advanced Configuration and Power Interface *ACPI* [11]. Today's major microprocessor manufacturers develop their own power management schemes in processor products such as AMD PowerNow technology, Intel's SpeedStep technology and Transmeta's Longrun technology. From the software layer, energy management has traditionally focussed on shutdown strategies implemented in operating system kernels, which puts the computer into a sleep or suspend state whenever the system is idle. There is a fixed cost in time and energy to enter sleep mode. Since the shutdown and wake-up operations themselves cost energy, these approaches usually require the existence of a long-enough idle period in the system to compensate for such overhead. In addition, since the shutdown and wake-up operations take time, the duration of sleep state must be large enough to amortize the latency, or one must predict the end of the sleep state and wake up in time for the next active state. In the absence of long idle periods in a system, such a coarse grained power management system becomes infeasible and a dynamic voltage scaling system seems more attractive.

To study the power consumption model, the following power dissipation model for

CMOS-based processors is widely used [38]:

$$P = p_t C_L V_{dd}^2 f_{clk} + I_{leak} V_{dd} + P_{short}$$

where P is the power dissipation, V_{dd} is the supply voltage, f_{clk} is the clock frequency, p_t and C_l are the probability of switching in power transition and the load capacitance, which can be regarded as constants. The first term $p_t C_L V_{dd}^2 f_{clk}$ is called dynamic power which is the power consumption of charging and discharging the load on each gate's output. I_{leak} is called leakage current. Leakage current contributes to static power. The third component is P_{short}, the short circuit power which reflects the power dissipation due to short-circuit current from the supply voltage to the ground briefly during a switching process. Among the three components, the dynamic power is the dominant factor and the other two can be mostly ignored. This is the assumption through most of our research. According to the above equation, different solutions have been proposed to reduce the total energy consumption of a CMOS-based processor.

One of the methods is dynamic voltage scaling (DVS) which lowers the supply voltage V_{dd} to reduce dynamic power of the processor according to system workload variations. Because of the quadratic relationship between the supply voltage and power consumption, lowering the voltage results in significant power reductions for CMOS circuits. Voltage limits the maximum clock frequency f_{max} a processor can operate on, as shown in the following relationship: $f_{max} \propto (V_{dd}$ -$V_t)/V_{dd}$. Therefore, DVS generally implies dynamic frequency scaling as well.

DVS capable processors usually have a set of special control registers to select the CPU clock frequency and supply voltage. Although these features are provided by hardware, it is the software that decides when to adjust the frequency and voltage and which level they should be scaled to. In contrast to coarse-grained energy saving solutions such as turning off the processor or I/O devices, DVS is a fine-grained energy saving mechanism. Frequency and voltage scaling incurs much less performance overhead than processor shutdown operations, which makes it possible to use much more aggressive power management systems.

3.1 Exploiting Bottlenecks

The primary method we adopt in our research is to identify code sections that have a significant amount of non-critical operations, which occur when the CPU is not on

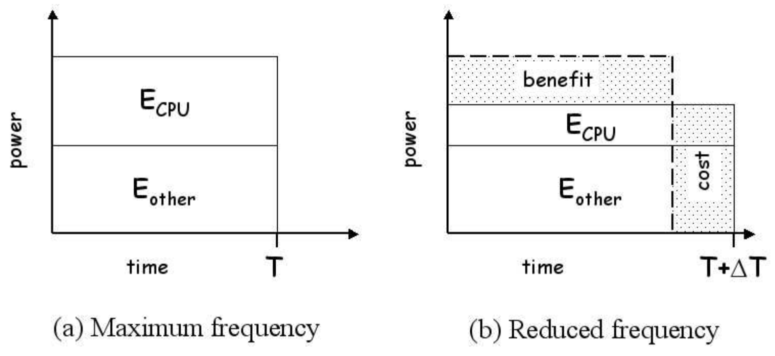


Figure 3.1: Cost Benefit Analysis of frequency reduction

the critical path due to bottlenecks at other components, and execute these sections of code at lower performance settings. The system power is seen as a function of frequency and is modeled as $P(f) = P_{CPU}(f) + P_{other}$. However, energy is the product of (average) power and time, so we consider the effect of frequency reduction has on elapsed time. Figure 3.1 shows the situation pictorially. When the program is run at the fastest gear, it takes time T. On the right is a situation where the program is run in reduced gear taking time $T + \Delta T$. The power savings is primarily a function of the frequency. Whether the lower frequency saves significant energy depends on whether the benefit due to power reduction is significantly greater than than the cost due to additional time. Essentially, this depends on $\Delta T/T$: when it is small, there will be energy savings. Broadly speaking, $\Delta T/T$ is small when whole programs or enough phases of programs are memory, I/O, or communication bound. On the other hand, when most of the program is CPU bound, $\Delta T/T$ is large and little energy can be saved. In a parallel or distributed program, each node performs some work and usually contains one or more instances where two or more nodes synchronize with each other to exchange data or status information. Such instances are called synchronization points. If the node finishes its work by time T, and the synchronization happens at time $T + \Delta T$, then the node idles, wasting energy. If the performance of such a node can be reduced, such that it completes it work by $T + \Delta T$, it saves energy, with no time penalty.

To save energy in distributed, high-performance programs we take advantage of bottlenecks that exist in most, if not all, such applications. The goal of this research is to identify such *reducible* code sections, which can be run slower without significant time penalty, which we examine in greater detail in the following sections.

Intra-Node Bottleneck

Even though a superscalar processor can issue many instructions per cycle, peak utilization is hard to sustain for even short durations. A major source of this inefficiency is memory stalls: the microprocessor is underutilized because the memory system cannot immediately provide the data. In this case, the microprocessor is executing at a faster speed than necessary because of a *memory bottleneck.* The memory bottleneck exists largely because of the great improvement in microprocessors. Furthermore, attempts to eliminate this bottleneck, such as instruction scheduling, memory access reordering, and prefetching, have been met with limited success. This is also an issue with HPC, as memory can throttle the performance of an individual node. Therefore, reducing the performance of sections of the application which suffer from the memory bottlenecks can result in a significant energy savings with a small decrease in performance.

Another intra-node bottleneck is the I/O bottleneck. The I/O bottleneck occurs when computation is stalled waiting for I/O. This bottleneck provides two opportunities for saving energy. The first occurs because programs that are I/O bound have relatively low computation workloads. In this case, similar to the memory bottleneck, significant energy savings are possible by executing at reduced performance with potentially only a small performance decrease. However, there is an additional opportunity different from that provided by the memory bottleneck. Specifically, during blocked periods, it may be possible to achieve energy savings by switching to the lowest operating point—or even sleep mode—with no performance loss, because the node can make no progress anyway and the computational demand is low. Both these are considered intra-node bottlenecks since they exist independent of other nodes.

Inter-Node Bottleneck

The inter-node bottleneck exists between multiple nodes in a distributed program. Typically, such programs have multiple phases, where the execution time of the nodes is due to several reasons, such as load imbalance or differing message latencies. Any node that is not on the critical path can be executed at reduced performance without decreasing the overall time of the node on the critical path. The communication and node bottlenecks exist because *parallel efficiency*, which is defined as the ratio of speedup to the number of processors used, generally decreases as the number of nodes increases. This is despite a

Frequency (MHz)	*Voltage* (V)	*Power* (W) idle	*Power* (W) active
2000	1.5	59.6	98.8–108.6
1800	1.4	57.0	86.5–95.2
1600	1.35	55.9	80.1–88.1
1400	1.3	54.8	74.3–81.9
1200	1.2	53.4	67.1–75.4
1000	1.1	52.0	62.2–69.6
800	1.0	51.0	56.8–63.5

Table 3.1: Idle and active power for Athlon-64.

large body of work in parallelizing compilers as well as adaptation by parallel systems.

Rather than eliminate these bottlenecks or increase parallel efficiency, our goal is to utilize the inevitable idle time to reduce energy consumed and heat generated. While it is difficult to so while incurring no increase in execution time, a significant reduction is possible with only a small performance degradation.

3.2 Methodology

Our experimental platform is a cluster of 10 nodes each with a frequency- and voltage-scalable AMD Athlon-64. Its available operating points are in the range of 800–2000MHz and 0.9–1.5V, see Table 3.1. Each node has 1GB main memory, a 128KB L1 cache (split), and a 512KB L2 cache, and the nodes are connected by 100Mb/s network. In this paper, we vary the CPU power and measure overall system energy. Although there are other components, throttling the CPU is effective in saving energy because the CPU is a major power consumer. In particular, the Athlon-64 CPU used in this study consumes approximately 45–55% of overall system energy.[1]

The programs we studied included two of the ASCI Purple benchmarks [4] as well as all of the NAS suite [5]. Presumably, such mature benchmarks have been thoroughly analyzed and are well-written (*e.g.*, see [54])—so some work has been done to balance the computational load. Therefore, well-tuned programs like those in ASCI and NAS should result in an approximate lower bound in terms of saving energy due to load imbalance. We

[1]CPU power is not measured directly. However, the system power at the fastest energy gear is 90–120 W. While the AMD datasheet states that the absolute maximum CPU power dissipation is 89 W, these benchmarks do not run that hot. We estimate the peak power of the CPU for our application is in the range of 40–50 W.

used as many nodes as possible on each test; this number is typically 8, though BT and SP run on 9 nodes.

For each program we measure execution time and energy consumed. Execution time is elapsed wall clock time. Total system power consumed by each node is measured by Watts-Up power meters at the wall outlet, which are connected to the serial ports of each node. These meters report average power consumption (in Watts) since last queried. Each node reads its associated meter every second and integrates power over time to determine the energy it consumes.

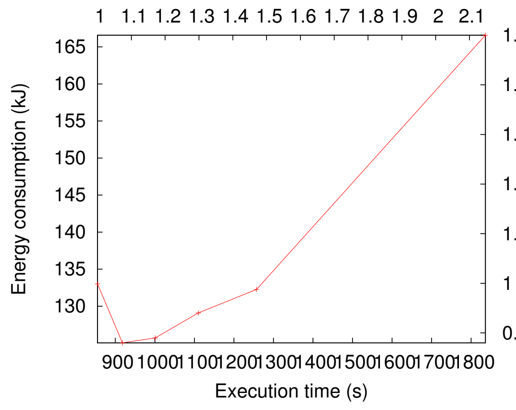
3.3 Results on the AMD cluster

In the results shown below, we used a cluster of ten nodes each with a frequency-scalable AMD Athlon-64. Table 3.1 shows the per-node system power consumption of different gears. We control the CPU power (by shifting gears) and measure overall system energy. This is effective in saving energy because the CPU—a major power consumer—uses less power. In particular, the Athlon CPU used in this study consumes approximately 55% of overall system power at 2000MHz and about 23% at 800MHz.

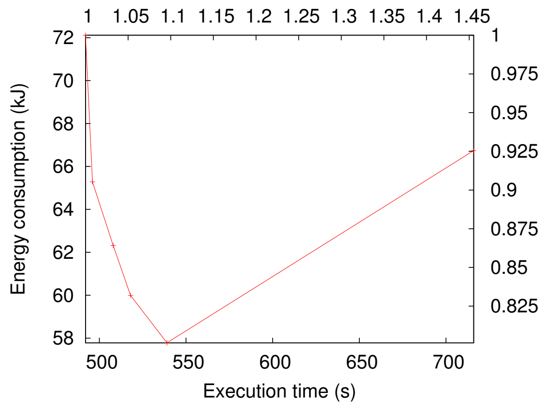
Single Processor Results

Figure 3.2 shows the results of NAS programs on a single Athlon-64 processor. For each graph, the *total system energy* consumed at each gear is plotted on the y-axis and the total execution time is plotted on the x-axis. The higher of two points uses more energy, and the further right of two points takes more time. Therefore, a near-vertical slope indicates an energy savings with little time delay between adjacent gears, whereas a horizontal slope indicates a time penalty and no energy savings. For readability, the origin of the graphs is not $(0, 0)$. Therefore, the alternate axes show the time and energy relative to the top gear (leftmost point). Because we tested the in-core version of NAS (*i.e.,* class B), these programs do not have significant I/O.

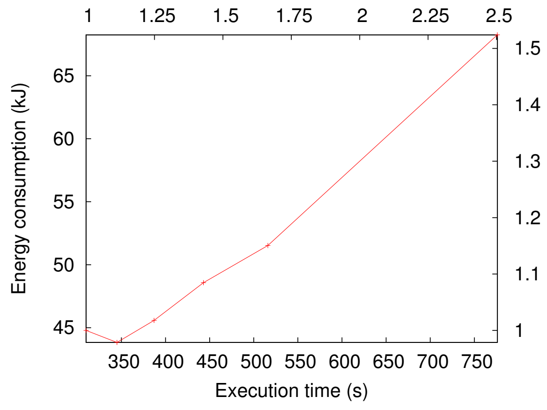
All of our tests show (unsurprisingly) that for a given program, using the highest gear takes the least time (*i.e.*, it is the leftmost point on the graph). The greatest relative savings in energy is 25%, which occurs in CG operating at 1000MHz. This savings incurs a delay (increase in execution time) of almost 7%. The best savings relative to performance occurs at 1400MHz, which saves more than 17% energy with a delay less than 1%. On the



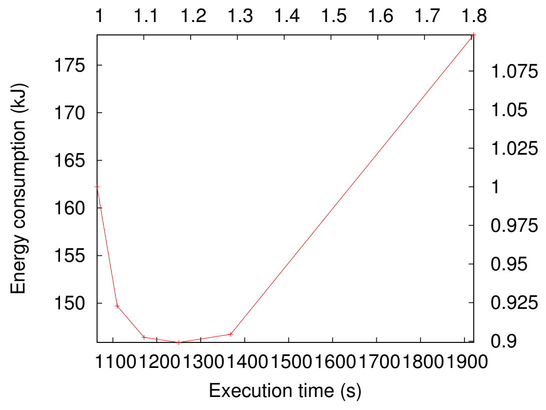
(a) BT B



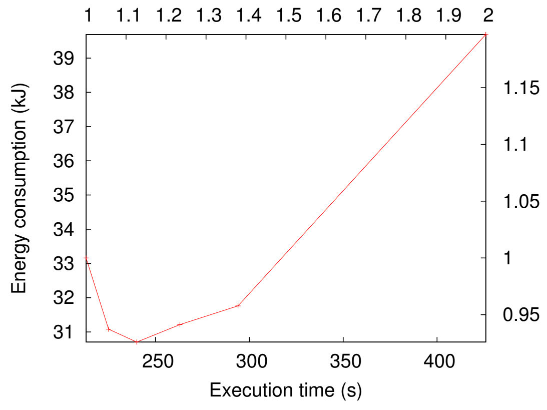
(b) CG B



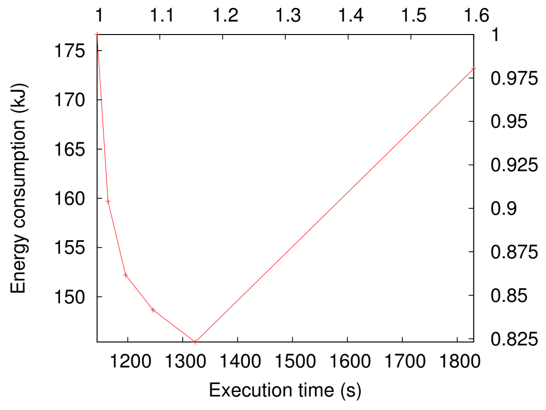
(c) EP B



(d) LU B



(e) MG B



(f) SP B

Figure 3.2: Energy consumption vs. execution time for NAS benchmarks on a single AMD machine.

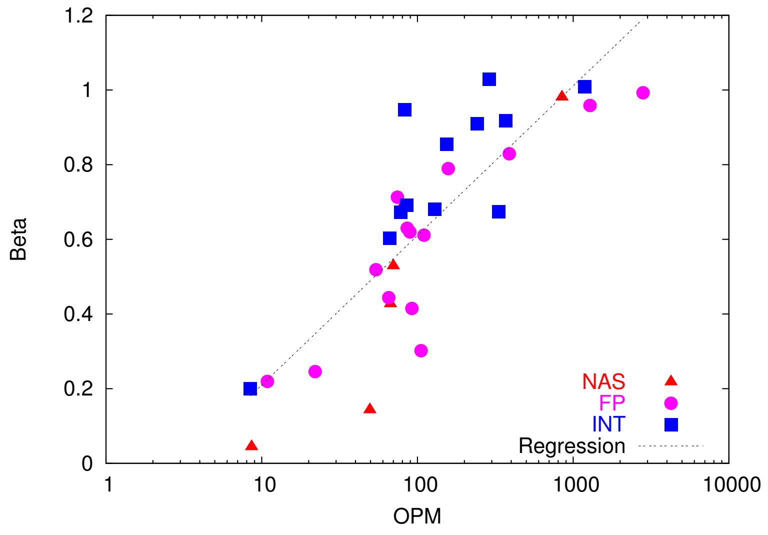


Figure 3.3: OPM v/s Beta for all benchmarks

other hand, EP at 1800MHz saves 2% energy with an 11% delay. This delay is the same as the increase in CPU clock cycle. EP is almost entirely CPU bound. Hence, inspection of CG and EP indicates that the challenge for system software is determining when to reduce the gear.

To quantify the degree of CPU (or memory) boundness, we use a metric called β, introduced in [26]. β is defined as a ratio between 0 and 1:

$$\beta = \frac{(T/T_{max} - 1)}{(f_{max}/f - 1)}$$

where T and T_{max} are execution times at a lower frequency and maximum frequency (2000MHz in this case), respectively, and f and f_{max} are the corresponding frequencies. $\beta = 1$ if an application's performance is completely dependent on the CPU frequency, and it equals 0 if completely independent. Unfortunately, β cannot be measured online. Figure 3.3 shows a scatter plot of β versus operations per cache miss, ops/miss(OPM), which can be measured on line. Regression analysis shows that log(OPM) is an excellent predictor of β, with a correlation coefficient of 0.84 and p-value of 0.002. Our extensive analysis of the NAS suite and other benchmarks leads us to conclude the memory pressure (as indicated by OPM) tends to predict the energy-time tradeoff in sequential programs [19].

Phases (MHz)		**Time** (s)	**Energy** (KJ)
2000	2000	481	64.1
1800	1800	500 (+4%)	60.1 (-6.2%)
1600	1600	563 (+17%)	58.8 (-8.2%)
2000	1800	482 (+0%)	62.5 (-2.5%)
2000	1600	486 (+1%)	61.6 (-4.0%)
2000	1400	492 (+2%)	60.6 (-5.4%)
1800	1600	505 (+5%)	57.9 (-9.6%)

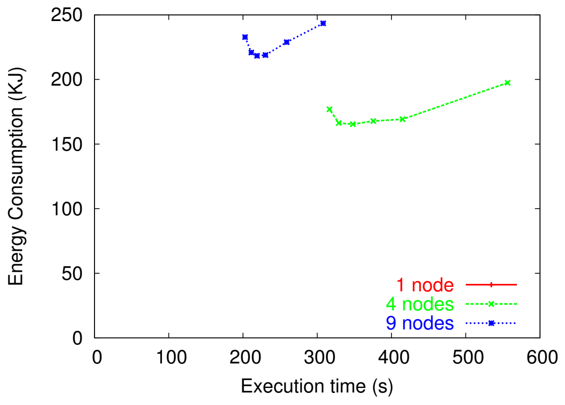
Table 3.2: Time and energy for different gears in different phases.

Multiple Processor Results

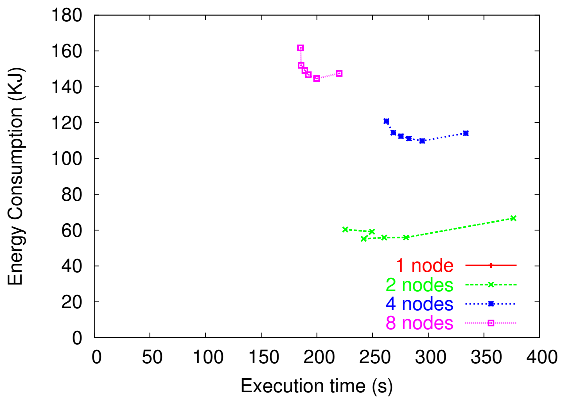
This section studies the effect of different gears on distributed programs. Figure 3.3 shows results for the NAS benchmarks. Each graph has the same general layout as in Figure 3.2, except that it shows the results from different node configurations. The energy plotted is the (measured) cumulative energy of all nodes used. These graphs show results using the same *single* gear on all nodes at all times. This shows that there are two tradeoffs to consider. One, which is similar to that on a single node, is the energy-time tradeoff *within* a curve (different gears, same number of nodes). The second consideration, which is unique to multi-node configurations, is the energy-time tradeoff *between* curves (different number of nodes). Thus, there are now two dimensions to consider when trading performance for energy savings.

Figure 3.4 offers several examples where speedup is poor. In particular, this case is illustrated in SP from 2 to 4 nodes. When speedup is poor, cumulative energy increases because the number of nodes increases, so the main consideration is within a curve. The figure also shows an example where speedup is good (but not perfect or superlinear). Consider LU at 4 and 8 nodes. Gear 4 on 8 nodes uses approximately the same energy gear 1 on 4 nodes, but executes 50% faster. This is an important result, as it shows that there exist cases in which using more nodes with each node at a reduced gear can execute faster *and* use less energy.

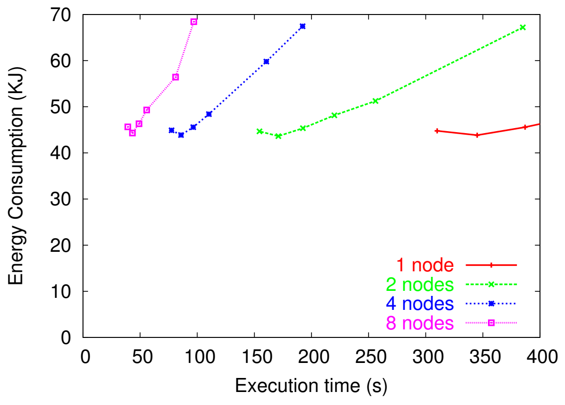
The results above use the same gear at all times on all nodes. First, we investigate shifting gears over time. Table 3.2 shows the advantage of dynamically changing gears between program phases. We placed phase boundaries manually based on MPI blocking operations as well as significant changes in μop/miss. Then, to determine the proper gear



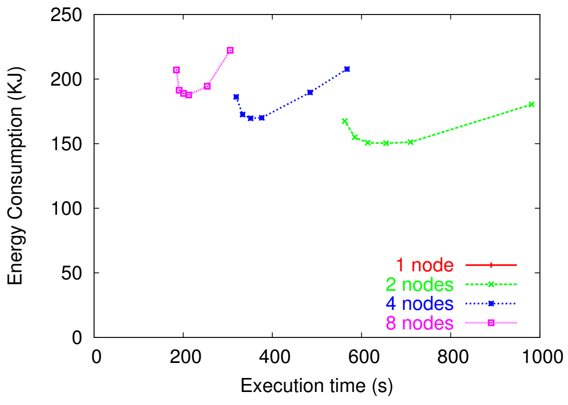
(a) BT B



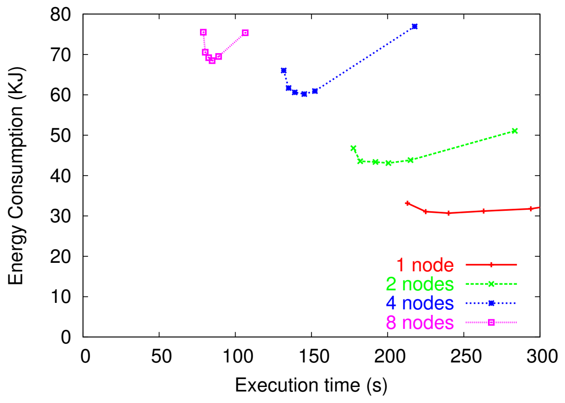
(b) CG B



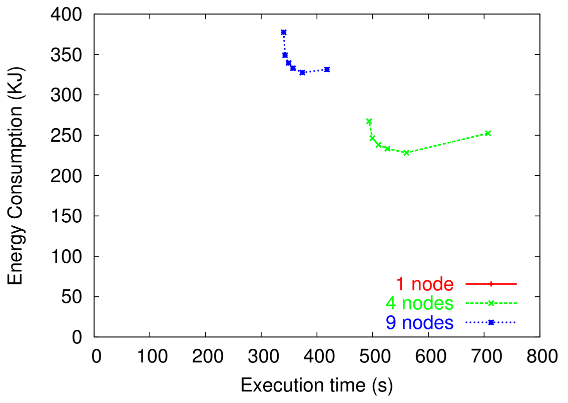
(c) EP B



(d) LU B



(e) MG B



(f) SP B

Figure 3.4: Energy vs. time for several benchmarks on 2 to 8 nodes: single, uniform gear.

per phase we used a profile-directed technique described in detail in our prior work [19]. The table shows, for example, that we can save 5.4% energy with only a 2% increase in time with the two phases run at 2000MHz and 1400MHz, respectively. This is better in time than any fixed-gear solution (with the exception, of course, of running solely at 2000MHz) and saves nearly as much energy as any of the other fixed gear solutions.

This chapter gives an overview of the tradeoff between energy and performance in MPI programs. Trends on both single processor and multiple processor programs are described. The reason why energy saving is possible is because of delays in the processor, where executing at a high frequency and voltage does not make the program execute faster, but wastes energy. This delay is due to the processor waiting for the memory system to fetch a value or the processor blocking awaiting a message from a remote processor. Exploiting such bottlenecks provides a unique opportunity to save energy without a significant time penalty.

Chapter 4

Simulation Model for Large Clusters

The previous chapter presented results on our small power-scalable cluster. While the results are encouraging, it is left unclear what performance (in time and energy) can be expected for larger power-scalable clusters. Most likely, before building or buying a large power-scalable cluster, one would like to determine the performance potential. As we do not have access to more than ten nodes, this section seeks to address this issue by developing a simulation model.

4.1 Model

Understanding scalability of parallel programs is of course a difficult problem; indeed, it is one of the fundamental problems in parallel computing [48, 49]. Researchers try to predict scalability using a range of techniques, from analytical to execution-based. We will use a combination of both.

Our methodology for predicting the behavior of larger power-aware clusters (that we cannot run programs on) is as follows. We model computation and communication using a combination of Amdahl's law, trace gathering, and source code inspection. To assist in understanding scalability, we use results from a 32-node (non-power-scalable) cluster. After that, we use the model to predict execution time on up to 32 power-scalable nodes at the fastest gear. (Our methodology can be applied to larger instances, but we do not have reliable results from a larger non-power-scalable cluster.) To determine time and energy on slower gears, we measure power consumption on a cluster node and use a straightforward algebraic formula. Below we describe our five-step methodology in full detail.

Step 1: Gather time traces. The first step is to gather active and idle times on n nodes ($T^A(n)$ and $T^I(n)$, respectively, where $T^I(n)$ includes the actual communication time) on each of our clusters.

Step 2: Model computation and communication. The second step is to develop a model of computation and communication that is based on $T^A(n)$ and $T^I(n)$. This will help us predict (in step 3) $T^A(m)$ and $T^I(m)$ where $m > 10$, *i.e*, for power-scalable configurations with more than ten nodes. Our approach here is distinct for each quantity, out of necessity: no matter what the gear, the power consumed is different when computing than when blocking awaiting data.

Determining F_p and F_s. Here, we use Amdahl's law to estimate F_p and F_s, which denote the parallelizable and inherently sequential fractions of an application, respectively. For a test with i nodes, we estimate F_p and F_s as follows:

$$\begin{aligned} T^A(i) &= T^A(1)(F_p/i + F_s) \\ F_p &= 1 - F_s \end{aligned}$$

We obtain a family of F_p and F_s values. We will use these to determine F_p and F_s on large power-scalable clusters. Also, $T^A(n)$ represents the *maximum* computation time over all nodes.

Classifying communication. Here, we recall that T^I includes idle time and communication time. While idle time (due to load imbalance) can be directly derived from T^A, the communication cost cannot. Hence, our approach is to categorize communication of each NAS program into one of three groups: logarithmic, linear, or quadratic. These are three common scaling behaviors for communication. To do this, we rely on three complementary methods: (1) inspection of the behavior of our measured T^I on up to nine power-scalable nodes, (2) dynamic measurements of each MPI call as well as inspection of corresponding source code, and (3) the literature in the field (*e.g.*, [54]). Specifically, we classified communication in BT, EP, MG, and SP as logarithmic; CG as quadratic, and LU as linear.

Step 3: Extrapolation of $T^A(m)$ and $T^I(m)$ at fastest gear. Third, we extrapolate to 16, 25, and 32 power-scalable nodes, *i.e.*, $m > 10$. For a given number of nodes, m, the sum of $T^A(m)$ and $T^I(m)$ yields the execution time.

Predicting active time: Predicting $T^A(m)$, requires an appropriate F_p and F_s for 16 and 32 nodes on the power-scalable cluster. Using our measured values on up to 32 nodes on the Sun cluster and up to 9 nodes on our power-scalable cluster, we fit F_p and F_s for 16, 25, and 32 nodes on the power-scalable cluster using a linear regression.

Predicting idle time: Given the classification of communication behavior (logarithmic, linear, or quadratic) for each application, we use regression to fit a curve to the communication using measured data on power-scalable nodes. This gives us communication time on 16, 25, and 32 nodes.

Validation. Our technique is validated in the following way. For $T^A(m)$, we compared F_p and F_s on up to 9 nodes on both clusters. With only 1 exception, it was identical; the outlier was CG, where the parallelism actually increases from 4 to 8 nodes on our power-scalable cluster, but is constant on the Sun cluster. For $T^I(m)$, each communication shape that we chose for our power-scalable cluster is identical on the Sun cluster up to 32 nodes. We also note that [54] supports our conclusion on five of the six programs. The exception is LU; for this program, we found that communication was best modeled as a constant; our traces showed that when nodes are added, each node sends more messages, but the average message size decreases.

Armed with estimates of $T^A(m)$ and $T^I(m)$ on larger configurations (at the fastest gear), we now turn our attention to determining the effect of different energy gears on execution time and energy consumption. The last two steps in this methodology are concerned with this issue.

Step 4: Determine S_g, P_g, and I_g. The next step is to gather power data from a single power-scalable node. Two values will be needed on a per-application and per-gear basis: application slowdown (S_g) and average power consumption (P_g). Separately, the power consumption for an idle (*i.e.*, inactive) system is determined for each gear (I_g).

This data determines the increase in time and the decrease in power. The execution time for a sequential program is wall clock time. This is done for *each* (sequential) program at each energy gear. The ratio S_g is determined as follows: $S_g = \frac{T_g(1)-T_1(1)}{T_1(1)}$. Now that we are discussing gear, we modify our notation: $T_g(1)$ is the time on one node at gear g.

The values P_g and I_g are obtained by measuring overall system power. The voltage and current consumed by the entire system is measured at the wall outlet to determine the instantaneous power (in Watts). This experimental setup determines the values, P_g, for each application and for each gear. The same setup, except this time with no application running, was used to determine the power usage of an idle system (I_g) at each gear.

Step 5: Determine $T_g(m)$ and $E_g(m)$. The final step is to estimate the time and energy consumption of a power-scalable cluster using the information developed so far. The time for a lower gear is computed by increasing the active time by the appropriate ratio, S_g. We assume that executing in a reduced gear does not itself increase the idle time, as our experimentation has shown that the time for communication is independent of the energy gear—the computational load during MPI communication is quite low. Using the values of P_g and I_g, we can estimate energy consumption at each lower gear for the MPI program. In this straightforward case, the time and energy estimates, on m nodes, for each gear are:

$$T_g(m) = S_g T^A(m) + T^I(m) \tag{4.1}$$

$$E_g(m) = P_g S_g T^A(m) + I_g T^I(m). \tag{4.2}$$

At slower gears the compute time is greater than that of the fastest gear, whereas the idle time is independent of the gear. Thus the time executing in gear g increases to $S_g T^A$. Communication latency is independent of gear, so this is assumed to remain the same.

However, this naive case above is too simple because it assumes all computation is on the critical path. In many programs, not all computation is on the critical path. Of course, reducing the energy gear of any computation on the critical path will delay other nodes, who must wait for data sent from the (now slower) node. In the refined model, T^A is classified into *critical* and *reducible* work (T^C and T^R), and in our estimates of computation we separate these and estimate each. In short, executing reducible work in a slower gear might *not* increase overall execution time, because an increase in the time of reducible work will decrease the idle time. On the other hand, executing critical work in a slower gear always increases execution time. The communication latency, which is unaffected by CPU frequency, is delayed only by the slowdown applied to the critical work. The idle time is *slack* for the reducible work. However, if reducible work is slowed such that all slack is consumed, then the time will be extended. This point of inflection is when $T^R + T^I = S_g T^R$.

The post-processing analysis conservatively determines the reducible work to be computation between the *last send*[1] and a blocking point. In between those two points there is no interaction between nodes, so the work is not on the critical communication path. With this refinement, Equation (4.1) changes as shown below. Note that for notational simplicity, we omit number of nodes (m).

$$T_g = \begin{cases} S_g(T^C + T^R), & \text{if } T^I + T^R \leq S_g T^R \\ S_g(T^C + T^R) + T^I + T^R - S_g T^R, & \text{otherwise} \end{cases}$$

Then, Equation (4.2) becomes

$$E_g = \begin{cases} P_g S_g(T^C + T^R), & \text{if } T^I + T^R \leq S_g T^R \\ P_g S_g(T^C + T^R) + I_g(T^I + T^R - S_g T^R), & \text{otherwise} \end{cases}$$

4.2 Evaluation

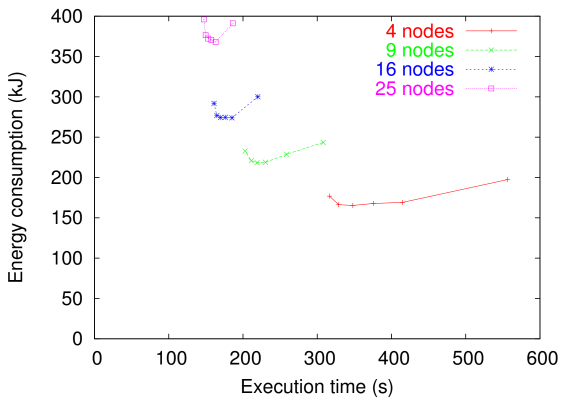
Figure 4.1 shows the results of our simulation on each application ranging from 2–32 nodes. All node configurations up to and including 9 nodes are actual runs on the cluster, and configurations of 16, 25, and 32 nodes are simulated using the model discussed previously. (CG has a speedup of less than one on 32 nodes, so that curve is not plotted.)

In the same way that the eight- and nine-node tests tend to be more "vertical" than the two- and four-node tests, as with the runs up to nine nodes, the shapes of the graphs tend to become more "vertical" when using 16, 25, or 32 nodes; *i.e.*, using lower gears becomes a better idea. As an example, consider SP. On four nodes, second gear consumes the least energy. On the other hand, on 16 nodes, fourth gear consumes the least energy.

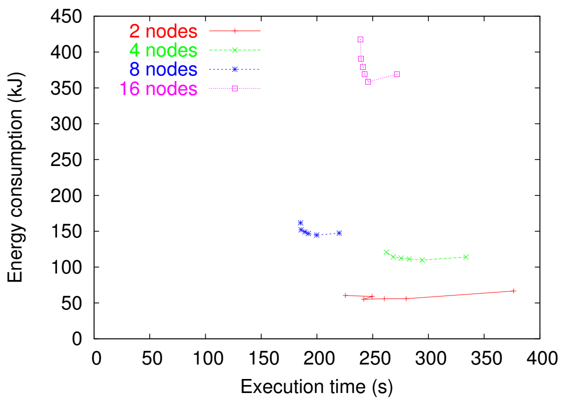
One possible implication of this is that for massively parallel power-scalable clusters, the individual nodes can be placed in a relatively low energy gear with only a modest time penalty. As discussed in the previous section, this may potentially allow for supercomputing centers to fit more nodes in a rack while staying within a given power budget. On the other hand, this could degrade performance significantly if many applications for such machines are embarrassingly parallel.

Second, speedup on the NAS suite generally starts to tail off around 25 or 32 nodes. Again, this is because this benchmark suite uses non-scaled speedup. The result of this is that the total cluster energy consumed starts to increase dramatically. Essentially, continuously increasing the number of nodes causes an application to be placed in the poor

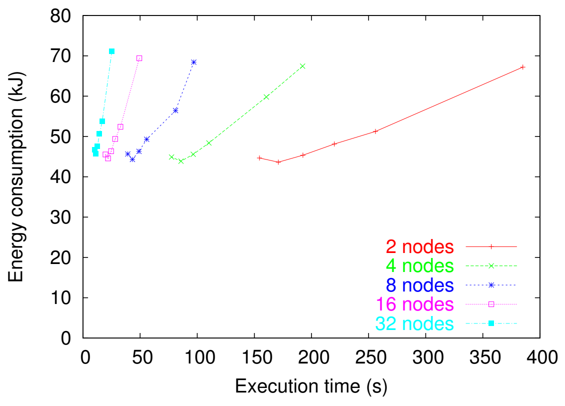
[1]We assume that the send is asynchronous.



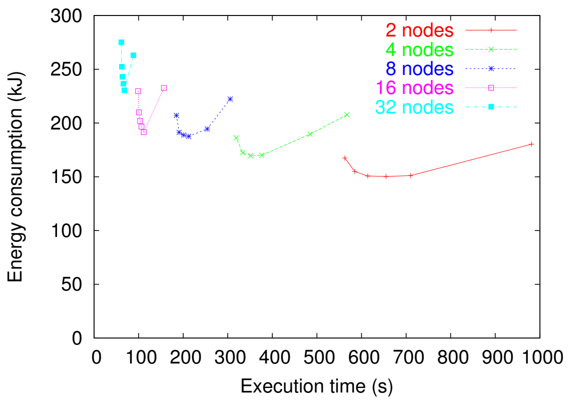
(a) BT B



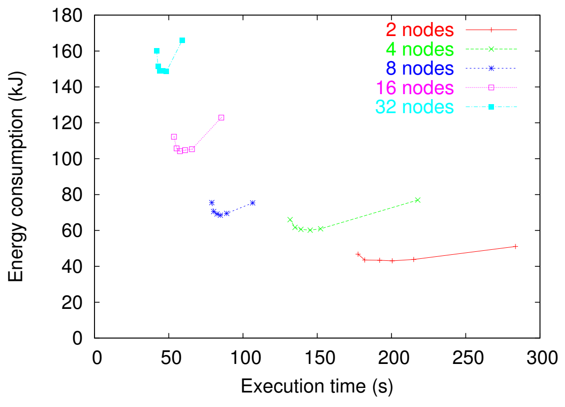
(b) CG B



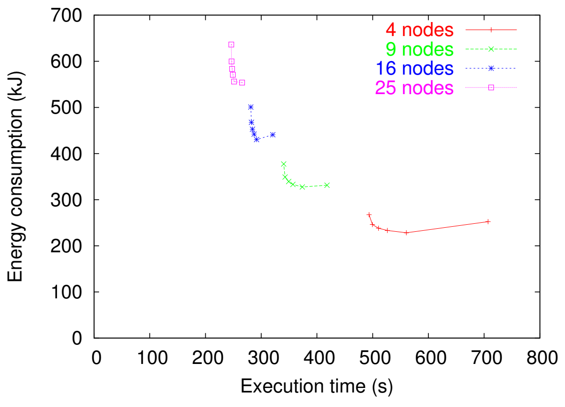
(c) EP B



(d) LU B



(e) MG B



(f) SP B

Figure 4.1: Simulated energy consumption vs execution time for NAS benchmarks on up to 32 nodes.

speedup classification (see previous section), which we know is energy inefficient. Also, for each application, there exists a certain number of nodes that, if exceeded, will cause program slowdown. It appears that that point is around 32 nodes for the NAS suite on our power-scalable Athlon-64 cluster.

This problem is not unique to power-aware computing; indeed, it is a problem with roots in the scalability field. However, it is clear that when this phenomenon occurs, it is necessarily the case that communication dominates computation. This means that in fact a lower gear is almost certain to be better. Hence, if one does not know what the parallel efficiency for a given application is, using a lower energy gear is a safeguard against excessive energy consumption.

This chapter builds on our understanding of bottlenecks and energy-time trade-offs of the previous chapter to develop a simulation model. Based on the empirical results from the 10 node AMD cluster, this model predicts energy-time tradeoff for big clusters.

Chapter 5

Just-in-Time Scaling

A parallel or distributed program usually contains one or more instances where two or more nodes synchronize with each other to exchange data or status information. We call these instances synchronization points. The node that is slowest to reach a synchronization point is considered the bottleneck node and is said to be on the critical path since it forces the remaining nodes involved in the synchronization to wait and is largely responsible for the duration of the synchronization. All other nodes are said to be non-critical, i.e. they have less or no influence on the duration of the synchronization. Typically, these non-critical nodes complete their work and wait for an event (*e.g.*, a message) from another node. During this *wait* period the node is idle. With *just-in-time* (JIT) scaling, such a node executes at reduced performance and completes its work just before the message arrives from the remote node. Typically, a load imbalanced parallel program is one in which the workload of the program is not equally distributed among all the nodes or parallel processes. Since the workload is not equitably distributed, some nodes finish their allocated work earlier and wait at the above mentioned synchronization points for the slower nodes to catch up. This provides an opportunity to reduce performance on the non-critical nodes and save energy without a significant time-penalty to the entire application.

The motivation of just-in-time scaling is to save energy by reducing the performance of nodes that are not on the critical path. Ideally, this reduction in performance of the non-critical nodes does not increase the completion time of the program as long as the increased completion time of the non-critical nodes does not exceed the running time of the bottleneck node. To achieve this goal, we must (1) identify the bottleneck node and its completion time, (2) identify the nodes that are not on the critical path, and then (3) determine the optimal CPU speed for the latter such that their completion time is still less than that of the bottleneck node.

It is important to note that there is a trivial way to address this problem: for any early-arriving node, once it blocks at a synchronization operation, reduce its gear until the end of the synchronization operation. This is a simple scheme, yet it does not nearly produce optimal results in terms of energy savings. The reason is that the difference between consumed energy at different gears when a node is idle is much less than that difference when the node is active. For example, on our Athlon-64 nodes, the difference in power between the fastest and slowest gears is less than 10W when a node is idle and more than 30W when the node is active.

From the previous chapter we know that each communication and computation burst is $T^A + T^I$, where T^A is the active time spent in computation and T^I is the idle time spent in waiting for a remote node to respond. The active computation time consists of computation time that is critical and reducible, $T^A = T^C + T^R$. The reducible work can be slowed down as it does not significantly affect the overall time taken by the computation-communication burst. We can slow down the processor such that T^A increases, reducing T^I, keeping in mind not to increase the overall time of the burst. However, reducing the performance of the processor for every burst is infeasible since they are very small in duration and the overhead of determining if a gear shift should be made and the cost of shifting is significant compared to the duration of the burst. Therefore, we aggregate these computation-communication bursts for a significant time period and then decide if it allows for any reducible work.

Our programming model is as follows. We assume iterative programs, which comprise the vast majority of scientific programs; furthermore, we assume that the iterations are relatively stable. This allows us to use past behavior to predict future behavior. For each node i, each iteration takes time T_i, and consists of one or more compute-communicate bursts. We denote the total compute time within one iteration as C_i and the wait time or slack time as W_i, where i is the node number (so $T_i = C_i + W_i$). We monitor the wait time of each node by measuring the time spent by that node in blocking communication calls.

There are some considerations with this approach. First, ΔC, the increase in the compute time when performance is reduced, is application dependent, which cannot be known *a priori*. Next, there can be hundreds of compute-communicate bursts per second. Moreover, these bursts are only synchronized across all nodes when they use global communication operations. Consequently, reacting on a per-burst basis is too fine-grain. Therefore, Jitter considers all bursts in an entire iteration. By aggregating the the individual wait times or each burst over the entire interval, Jitter determines the *slack* for each node over the entire interval, S_i=W$_i/T_i$. The slack metric solves both issues.

We then calculate minimum slack of all the nodes — $MinimumSlack = minimum(S_i)$. Net Slack is calculated as $NetSlack$ = S_i - $MinimumSlack$. If $NetSlack$ = 0, then the node is the bottleneck node and is on the critical path. As discussed above, the remaining nodes, which are not on the critical path, provide an opportunity to reduce gears if the resulting increase in compute time is less than the wait time, *i.e.*, $\Delta C < S$. A large amount of *Net Slack* indicates clearly that a node is not on the critical path and is a candidate for reduced performance. No or very little *Net Slack* indicates that the corresponding node provides no opportunity for frequency or voltage scaling.

Net slack is more useful than gross slack because a node with a net slack of 20% is clearly not a bottleneck whereas a node with 20% gross slack may actually be a bottleneck node. Also, the amount of slack per node can vary widely between different applications. For example, in the NAS suite average slack varies from less than 10% to over 90%. Second, the amount of slack per node can vary widely within an application and in practice is never zero on any node (because it is highly unlikely that any single node is always last to arrive at a large number of communication calls within an iteration).

The jitter algorithm only works for programs which are iterative and the behavior of the past iteration is a predictor for the next iteration. The jitter algorithm gathers information such as slack and iteration length over an iteration and uses this information to choose which gear the node should execute in for the next iteration. The decision it makes will be correct only if the next iteration has the behavior of the next iteration. Therefore, this discussion is limited to iterative programs (which is the vast majority of programs) where the past (usually) predicts the future.

Unfortunately, the existence of slack is not a sufficient condition to reduce the gear. There must be enough slack that reducing the performance will not increase the execution time too much. Armed with the normalized slack per node, our system reduces the performance of each node that has enough slack such that reducing the gear is unlikely to increase the execution time. Our algorithm is then adaptive in that if the gear reduction was too much, we increase the gear; if it was too little, we further reduce the gear.

We have created a library called *MPI-Jack* which is interposed between the application and the MPI Library. It is an interface that exploits PMPI [44] the profiling layer of MPI. MPI-Jack enables a user transparently to intercept (hijack) any MPI call. A user can execute arbitrary code before and/or after an intercepted call using *pre* and *post* hooks. In this work, we use MPI-Jack to determine the time spent in blocking MPI calls, such as *MPI_Recv*, *MPI_Wait*, and *MPI_Barrier.* The pre hook records the time the routine began. The post hook records when it ends, computes the wait time, and updates the node's overall wait time. Our MPI-Jack library is dynamically linked to the application during the process of compilation. When the application executes, all MPI calls made by the application are trapped by the MPI-Jack library.

To make an effective decision for reducing performance of non-critical nodes, there are several items that Jitter must determine:

- the iteration boundaries,
- the net slack of each node,
- when to reduce the performance,
- when to increase the performance,
- when to remain in the same gear,
- when to reset algorithm parameters, and
- how to determine the threshold slack

We describe each of these in turn. The Jitter Algorithm is given in pseudocode in Figure 5.1.

Determining the iteration boundaries Currently, we manually insert a special Jitter MPI call, *MPI_Jitter*, at the top of the main loop in the program; however, this is easily automated [28]. Such a loop must exist, as we are assuming iterative programs. When iterations are too short, Jitter limits the overhead by waiting several iterations before *MPI_Jitter* takes action. Thus is also done in order to amortize the overhead of Jitter and because there is little benefit to reducing performance for only a short period. The current implementation combines iterations until the time exceeds half a second. Jitter performs several actions in *MPI_Jitter*, which are described next.

Determining the net slack of each node Slack in Jitter is determined as follows. First, to determine wait time, we use our MPI-Jack tool as described above. Second, we

```
Current_Iteration_Length = Length of the current Iteration
Gross_Slack = Slack / Current_Iteration_Length
Min_Slack = Minimum Slack of all nodes
Net_Slack = Gross_Slack – Min_Slack

If Current_Iteration_Length  is much larger than previous iteration
        Reset to gear 0
        Reset upshift and downshift bias to initial values
Else if Current_Iteration_Length > Avg_Iteration_Length
              Increase Performance  (Decrease Gear)
              Increase Slack_Threshold
              Increase upshift_bias[Current_Gear]
         Else If Net_Slack > SLACK_THRESHOLD *  downshift_bias [Current_Gear]
                        Decrease Performance    (Increase Gear)
                        Increase   downshift_bias[Current_Gear]
              Else if Net_Slack < 0.25 * SLACK_THRESHOLD / upshift_bias [Current_Gear]
                        Increase Performance
                        Increase upshift_bias[Current_Gear]
```

Figure 5.1: The *Jitter* algorithm

then compute the absolute or gross slack as the ratio of the *wait time* divided by the *iteration time*. The global minimum slack among all nodes is determined using a reduction (*MPI_Allreduce*). Then the node's *net slack* is computed as the difference between its *wait time* and the global minimum. The need to use *net slack* has been explained in Chapter 5.

Determining when to reduce performance Jitter reduces a node's performance if there is enough net slack. The Jitter prototype uses the following relationship to determine whether there is enough slack to reduce the gear:

$$\text{net\_slack} > S \cdot d_g \rightarrow \text{enough slack.}$$

The term S is the *base slack threshold.* It represents the amount of net slack needed to reduce the gear. We set the initial value of *base slack threshold* and the algorithm learns the correct value of *base slack threshold* through the course of the iterations. For example, we dynamically tune the definition of "enough slack" over the first few iterations of the application.

While net slack is better than gross slack at identifying a bottleneck node, it is not enough. If Jitter chooses to reduce and it turns out to be a bad choice (which is learned when the node later observes a large increase in execution time and so increases performance) the threshold to reduce again is raised. This threshold increase is captured in the term d_g, which we call the *downshift factor*. Each time a node reduces from gear g it increases d_g using the formula $d_g = d_g * \text{bias}$. Through experimentation, we found that a bias = 1.5 worked well for all applications. We have not found that the benchmarks are very sensitive to the bias value; however, further experiments are ongoing.

Determining when to increase performance Each node determines if it is a bottleneck node according to the following relationship.

$$\text{net\_slack} < \alpha \cdot S/u_g \rightarrow \text{bottleneck node.}$$

The term α is used to ensure the possibility that Jitter can inform a node to remain in the same gear. Although it could be a user-provided parameter, we use $\alpha = 0.25$ in our tests. This simplifies use and has little adverse effect on Jitter's performance. We use an *upshift factor*, u_g, that is similar in spirit to the downshift factor above. It is adjusted each time there is an increase in performance using bias in the same way as d_g. This lowers the threshold slack required to shift up.

Because there is always at least one bottleneck node every iteration, being a bottleneck node is not a sufficient condition for increasing performance, for two reasons. First, a node can be a bottleneck in a lower gear without slowing down the computation. This happens when there is another node that is as slow or slower but is in the top gear. Second, there is variance in the times between iterations even in the best situation. Therefore, some conditions are transient and we do not want Jitter to react to them.

Consequently, a bottleneck node will increase its performance if either of the following conditions hold: (1) the iteration time has increased, and the node reduced its gear on the previous iteration, or (2) the node has been a bottleneck for three consecutive iterations.

Determining when to remain in the same gear Jitter remains in the same gear if it chooses not to reduce or increase, as described above. Initially, this range is $\alpha S <$

net_slack $< S$. The more the algorithm shifts gears, the larger this range is, due to the biasing of the upshift and downshift factors. Thus the algorithm tends to stabilize.

To prevent a form of “thrashing”, where the gear constantly changes, we use thresholds in the following way. If a node reduces the gear too far, we increase the threshold value for reducing the gear. That threshold is decreased if there is extra slack time still remaining after a gear reduction. However, we increase the threshold by more when the gear is reduced too far than we decrease it when the gear is not reduced too much—this is because we bias towards high performance for HPC applications. Our system learns the proper thresholds for each application at runtime.

Determining when to reset algorithm parameters In addition to the above three *standard* actions (reduce, increase, remain), there is a fourth, *extraordinary* action, which is to reset the algorithm parameters. This action is triggered by a dramatic change in the iteration time. Currently, a reset occurs if the time between adjacent iterations changes by 50% or more. The reset is not because of a problem within Jitter, but rather because because the application has changed in a significant way. (Jitter only otherwise changes by a single gear per iteration, which is not likely to cause a large change because frequencies change by 20% or less between adjacent gears.) None of the tested NAS or ASCI benchmarks algorithms causes a reset; however, we force a reset with our synthetic benchmark (see the next section).

How to determine the threshold slack Our Jitter algorithm chooses to reduce performance for a non-critical node if that node has “enough slack”, such that the decrease in performance does not increase the overall time of the application. Our study of load-imbalanced applications shows that what constitutes “enough slack” or the slack threshold is an highly application dependent quantity. For, some applications such as sweep3d, a benchmark from the ASCII purple suite, a slack threshold of 5% provides the best energy-time trade-off, whereas for aztec, yet another benchmark from the ASCII purple suite, a slack threshold of 15% provides the best energy time trade-off. The slack threshold depends not only on the net slack but also the gross slack in an application. For example, the slack in sweep3d varies from 20% to 35%, while the slack in aztec varies from 0% to 40%. We have seen for applications that have a large gross slack for all nodes require a lesser slack threshold than applications that do not. A naive way to determine the slack threshold would be fix at the rate of change of the frequency.

Hence, the slack threshold is difficult to determine without prior knowledge of the application. Therefore it was necessary to design Jitter such that is would dynamically determine the slack threshold for each of the applications. The algorithm begins with an initial slack threshold of 0.05, which signifies that if a node has 5% net slack, Jitter should consider it as a candidate to reduce performance. This slack threshold is only incremented when the algorithm decides to decrease performance for a particular node due to overall increase of the program time. When the threshold slack is incremented, *e.g* to 0.07, Jitter will only consider this node as a candidate to decrease performance only if its net slack is greater than 7%. Since the algorithm is designed in such a way that that it stabilizes after initial few iterations, the slack threshold also stabilizes to provide optimum energy-time trade-off. The slack threshold is further examined in the next chapter. .

In summary, the inter-node bottleneck exists between multiple nodes in a distributed, HPC program. Slack indicates the relative communication delay on a node. The node with the least slack is considered the bottleneck node. Net slack is the amount of slack on a node after subtracting the slack of the bottleneck node. A net slack of zero or nearly zero indicates that the node is critical, nodes with a large amount of net slack provide opportunity to reduce performance. The Jitter algorithm determines the amount of net slack per iteration of an application and reduces performance of the non-critical nodes. This saves energy on these nodes, and the goal of Jitter is to attempt to ensure that they arrive “just in time” so that they avoid increasing overall execution time.

Chapter 6

Jitter Results

In this section, we present the Jitter results. We show the benefits of Jitter on imbalanced applications. Then, using a synthetic benchmark, we show the full capabilities of Jitter. Last, we show that Jitter does not slow down a program unnecessarily when its load is balanced. We also evaluate the overhead and sensitivity of the Jitter implementation.

6.1 Non-Uniform Loads

This section shows the results of two benchmark programs that have load imbalance. The first one, Aztec, is a parallel iterative solver for sparse linear systems from the Purple suite. Table 6.1 shows results for Aztec using four different methods. All results are the average of at least 3 runs, with little variance. The first method is *Full* power, where all nodes execute in top gear (2000MHz). It is used as the baseline. The next uses a *hand-tuned* set of per-node gear settings. Using the slack on each node at Full as a guide, we tested many solutions to find the "best." Because we are biasing towards performance, the goal was to save as much energy as possible while allowing only small performance impact, which

	Time (s)	**Energy** (KJ)
Full	64.8	44.4
Hand-tuned	65.0 (0.3%)	38.6 (-13.1%)
Jitter	65.1 (0.4%)	38.7 (-12.8%)
Reduced	67.1 (3.0%)	40.6 (-8.5%)

Table 6.1: Aztec results

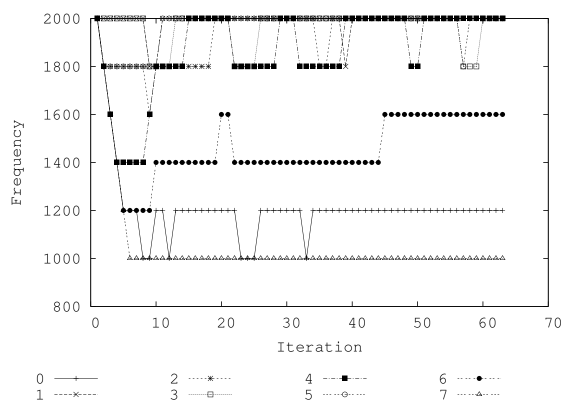


Figure 6.1: Gears for chosen by Jitter for Aztec.

we somewhat arbitrarily define as less than 2.5% increase in time. While our search was not exhaustive, it was extensive. The third method is *Jitter*, where all nodes begin in top gear and dynamically shift according to the algorithm described in the previous chapter. In the last method (*Reduced*), every node executes at 1800MHz—performance is reduced by one gear. This method serves as another baseline.

The hand-tuned run saves 13% energy with no time penalty. Each node executes in a single, but possibly different gear: one node runs in top gear (node 4), four in 1800MHz (nodes 1, 2, 3, and 5), and one each in the next three gears. (Frequency is used to name the gear, but both frequency and voltage are scaled.)

The Jitter run takes more time than Full, but saves nearly 8% energy. As expected, it does not save as much energy as the hand-tuned method. The primary reason for this is that the gears selected by the algorithm are higher than in hand-tuned. Five of the nodes execute more than half of the iterations in top gear. The secondary reason is the cost of constantly shifting gears, as it takes Jitter a handful of iterations to determine a solution. During this time it continually refines the gear settings. It should be noted that Aztec is the least stable of all benchmarks—*e.g.*, it still shifts to some extent after 50 iterations—which we believe is likely typical of a production application than the other benchmarks.

The four nodes that execute at 1800MHz in the hand-tuned case execute at 2000MHz in Jitter. Jitter mis-predicts this gear because the slack varies due to constant gear shifting. This variance causes these nodes to shift back to top gear. They often reduce performance again, but will eventually shift back to top gear. Each time through this

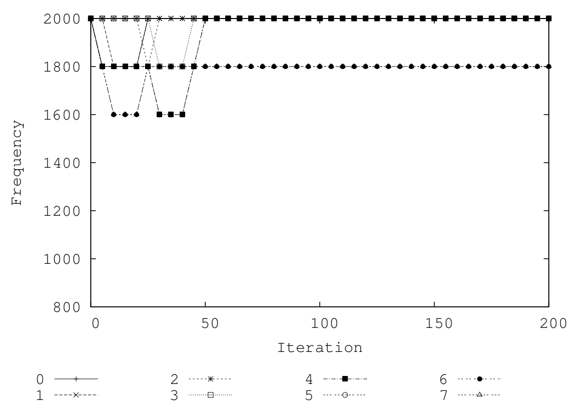


Figure 6.2: Gears for each node for each iteration in Sweep3d.

cycle it becomes harder to reduce due to the increasing downshift factor d_g. Our goal of favoring performance over energy forces these nodes into the top gear. Nevertheless, Jitter performs well. It is possible to extract better numbers from Jitter by tuning the parameters to optimize it for Aztec; however, this paper presents a more general usage of Jitter.

Figure 6.1 shows the gears per node per iteration for Aztec. It shows a single, representative run. Because there are more nodes than gears, the lines overlap significantly. At the end there are 5 nodes that execute at 2000MHz (nodes 1–5). Node 7 reduces one gear each iteration down to 1000MHz, where it remains for the duration of the program's execution. Node 0 predominantly executes at 1200MHz and node 6 at 1400MHz.

The second program that shows load imbalance is *Sweep3d*, which is also a Purple benchmark. Sweep3d solves a time-independent discrete geometry neutron transport problem in 3-dimensions. Table 6.2 shows the results. In the hand-tuned method two nodes, 6 and 7, are in 1800MHz, while the rest are in top gear. There is essentially no time penalty for hand-tuned. Even with this little difference from full performance, there is a noticeable energy savings.

	Time (s)	**Energy** (KJ)
Full	26.2	19.1
Hand-tuned	26.3 (0.3%)	18.1 (-5.3%)
Jitter	26.3 (0.3%)	18.1 (-5.3%)
Reduced	28.2 (7.0%)	17.9 (-6.3%)

Table 6.2: Sweep3d results

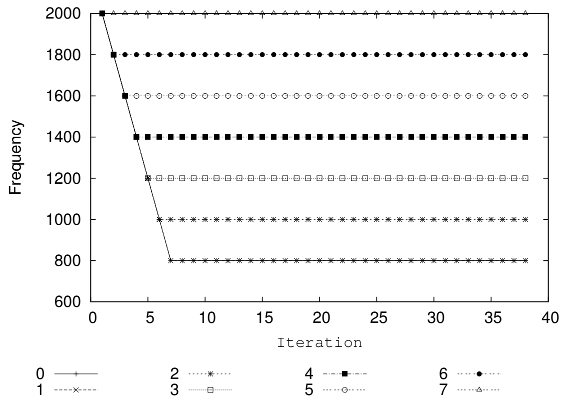


Figure 6.3: Gears for each node for each iteration in synthetic benchmark with stable, non-uniform load.

	Time (s)	**Energy** (KJ)
Full	80.0	55.1
Hand-tuned	80.1 (0.1%)	47.5 (-13.8%)
Jitter	80.8 (1.0%)	48.3 (-12.4%)
Reduced	88.6 (10.7%)	54.8 (-0.1%)

Table 6.3: Synthetic benchmark results

Figure 6.2 shows the gears used by Jitter in Sweep3d. Because the application iteration length is about 0.1 seconds, Jitter takes action only every 5 iterations; therefore, the figure plots every 5 iterations. It stabilizes much more quickly than Aztec, reaching stability within 50 iterations (10 Jitter iterations). After iteration 50, the nodes are in the same gears as hand-tuned. Before that, 4 different nodes reduce to 1600MHz, two of which climb back to top gear. Overall, 89% of the time Jitter is in the same gear as hand-tuned, and only 2% of the time are any nodes more that one gear away from hand-tuned. Therefore, Jitter performs nearly the same as hand-tuned.

6.2 Synthetic Benchmark Results

This section presents results from a synthetic benchmark, which was created to fully exercise Jitter. It is an iterative PDE solver, where we added (artificial) parameters that make the amount of load per node configurable. Essentially, each node does an customized amount of work each iteration. Then it sends data to each of two neighbors and

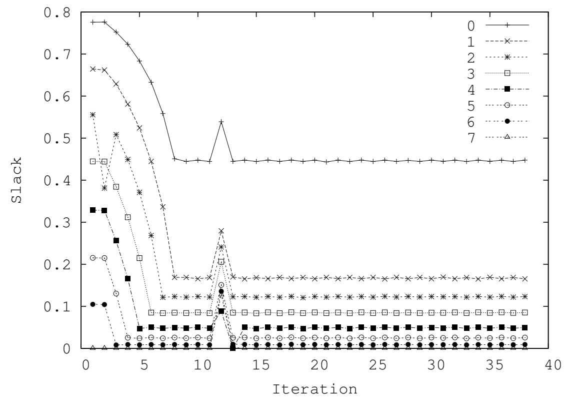


Figure 6.4: Slack for each node and each iteration in synthetic benchmark with stable, non-uniform load.

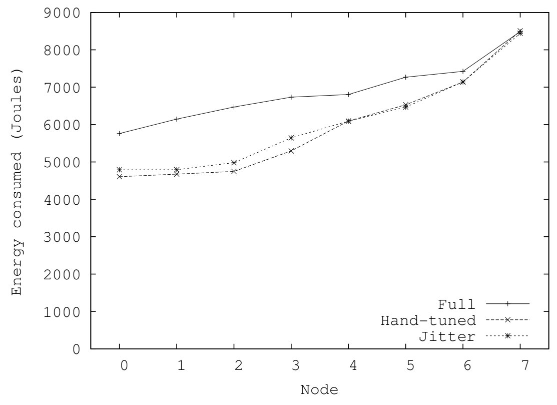


Figure 6.5: Energy consumed for each node in synthetic benchmark with stable, non-uniform load.

executes a barrier.

Figure 6.3 shows gears for each node. In this example, each node has an increasing amount of work, so node 0 has the least work and node 7 the most. Through separate experimentation, we determined these loads so that each node should select a different gear. (Because there are only 7 gears, 0 and 1 should select the same gear).

Table 6.3 shows the overall results. As expected, when using Jitter, there is significant energy savings and little time penalty. Jitter is nearly the same as the hand-tuned case because it stabilizes quickly to the same gear assignments as hand-tuned.

Figure 6.4 shows the slack for each node. There is a large spread initially, from almost zero to nearly 80%, but this range rapidly compresses. Node 0, which has half as

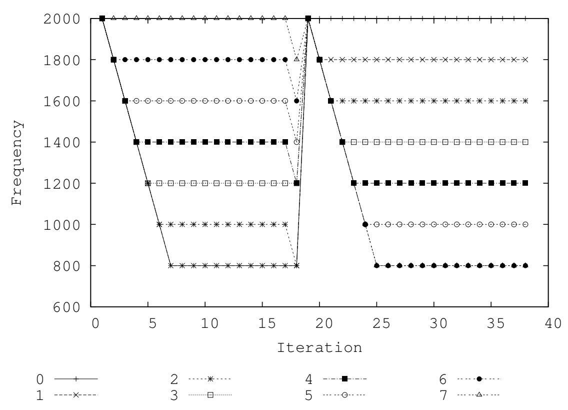


Figure 6.6: Gears for each node for each iteration in synthetic benchmark with unstable, non-uniform load.

much load as node 1, still has a significant amount of slack. However, five nodes have less than 10% slack. There is spike at iteration 12 that we cannot explain. Apparently, all communication takes longer in this iteration. Every node's total wait time increases about the same amount (15ms). This spike is seen in every run (but different iterations) of the synthetic benchmark for all methods and every run of MG (from the NAS benchmark suite). These are the only programs that have several nodes with less than 10% slack. It is possible that there is a transient delay in the network that programs with more slack are able to absorb, which is why we do not see spikes in other programs.

Figure 6.5 shows the energy consumed by each node for the three methods. The less loaded nodes use significantly less energy when using Full because Linux issues a *HALT* instruction during idle. The figure shows that the most energy is saved in these lower loaded nodes. Importantly, for this program, Jitter achieves nearly the same results as the hand-tuned method.

The next test shows that Jitter can adapt to a changing workload. The work assigned to each node changes at iteration 17. Initially, the work is assigned as above, with 0 having the least and 7 the most. In the second, half the workload distribution is reversed with 0 having the most and 7 the least. Figure 6.6 shows the gears selected at every iteration. Because node 0 is in the slowest gear and it has a large increase in work, iteration 18 takes a long time (5 seconds instead of 2 seconds). This causes a reset. After that the algorithm acts as before, only the nodes select gears in a "mirror-image" fashion. In Full this test takes 77.5 seconds and consumes 56.8 KJ. Jitter is 4.1% slower, but uses

Benchmark	**Method**	**Time** (s)	**Energy** (KJ)
BT	Full	72.2	53.7
	Hand-tuned	72.2 (—)	53.7 (—)
	Jitter	72.6 (0.3%)	53.5 (-0.4%)
	Reduced	74.1 (2.6%)	50.34 (-6.2%)
CG	Full	118	74.1
	Hand-tuned	121 (2.5%)	68.5 (-7.6%)
	Jitter	121 (2.5%)	68.9 (-7.0%)
	Reduced	121 (2.5%)	68.6 (-7.4%)
IS	Full	127	74.9
	Hand-tuned	129 (1.6%)	69.6 (-7.6%)
	Jitter	127 (0.0%)	75.6 (+0.9%)
	Reduced	130 (2.3%)	70.0 (-6.5%)
LU	Full	62.0	48.4
	Hand-tuned	63.5 (2.4%)	47.0 (-2.9%)
	Jitter	63.5 (2.4%)	47.1 (-2.6%)
	Reduced	66.2 (6.7%)	45.5 (-5.9%)
MG	Full	92.9	64.7
	Hand-tuned	92.9 (—)	64.7 (—)
	Jitter	93.2 (0.3%)	65.1 (+0.6%)
	Reduced	94.3 (1.0%)	60.4 (-6.6%)
SP	Full	55.6	40.0
	Hand-tuned	55.9 (0.5%)	36.4 (-8.9%)
	Jitter	55.6 (0.0%)	40.0 (0.0%)
	Reduced	56.8 (2.1%)	36.6 (-8.5%)

Table 6.4: NAS benchmark programs using Full, Hand-tuned, and Jitter. Percentage differences are given relative to Full; a line (—) means that same gear vector as Full was used.

11.1% less energy.

6.3 Uniform Loads

This section presents the performance of the NAS benchmark suite. Most of the NAS programs have well-balanced workloads, with the exception of CG. Therefore, generally speaking, the actions taken by Jitter should be to choose a uniform gear vector. In other words, we perform these tests primarily to ensure that Jitter does no harm to the application when the load is *balanced*.

We first discuss BT, LU, and MG (see Table 6.4). For these programs, the hand-tuned version is on average 0.6% slower than Full and saves an average of 0.8% energy.

Those numbers for Jitter are 1.0% and 0.6%, respectively. Clearly, both hand-tuned and Jitter perform similarly to Full, which justifies that Jitter does not adversely affect load balanced applications.

Finally, we consider the cases of CG, IS, and SP. For CG, both hand-tuned and Jitter save significant energy with a small time penalty. Essentially, the reason for the impressive performance of CG is because it is highly memory bound, as indicated by the performance of Reduced. This result is consistent with our previous results that studied exploiting the *intra-node* bottleneck [19]. Nevertheless, Jitter and hand-tuning are able to take advantage of the small amount of load imbalance present.

For IS and SP, there is a large amount of slack time, but it is constant over all nodes. Hence, the net slack is zero on all nodes, so Jitter takes no action. Note that in this case, the hand-tuned version reduces the gear such that all nodes execute in first gear (1800 MHz). IS and SP represent programs that do not have an inter-node bottleneck because the load is well balanced. We have different techniques, that leverage the memory bottleneck, to save energy in such cases [18].

6.4 Sensitivity and Overhead of Jitter

Now that we have examined the energy savings caused by *Jitter*, we further study *Jitter* to understand its overhead, sensitivity to various parameters.

Overhead of Jitter The overhead of the Jitter systems consists of the overhead caused by MPI-Jack, the Jitter algorithm, and gear shifts. The overhead of MPI-Jack is overhead caused by the MPI-Jack wrapper which interposes between the application and the library measures the time taken by each MPI call. This overhead was measured by linking only the MPI-Jack library with the application and measuring the difference in the time taken for the execution of the benchmark with the wrapper and without. This essentially determines the overhead of profiling all MPI calls at the top gear. The overhead of the Jitter algorithm is measured as the time taken by the Jitter algorithm to make a decision regarding the performance of a node.

Table 6.5 shows that the MPI-Jack overhead is less 1% for the benchmarks sweep3d, aztec and the synthetic benchmark. The overhead is largest for sweep3d because it makes significantly more MPI calls in each iteration than the other benchmarks. Table 6.5 also

	MPI-Jack Overhead	Jitter/iteration	Iteration length
aztec	0.1 %	0.41ms (0.04%)	970 ms
sweep3d	0.69 %	0.39ms (0.06%)	650ms
jacobi	0.2%	0.45ms (0.02%)	1990ms

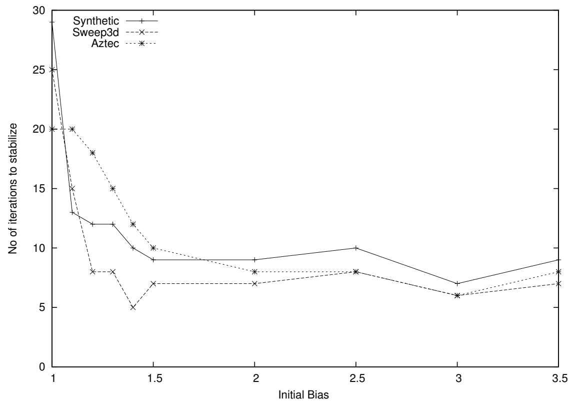
Table 6.5: Overhead of Jitter and MPI-Jack

shows the overhead caused by the Jitter system. This overhead is miniscule, *i.e.,* less than 0.1%. As implemented, Jitter inserts an additional reduction (MPI_Allreduce) in the code that determines the minimum slack of all the participating nodes. A single MPI_Allreduce on 8 nodes takes between 0.3 milliseconds to 0.9 milliseconds, with an average of 0.6 milliseconds.

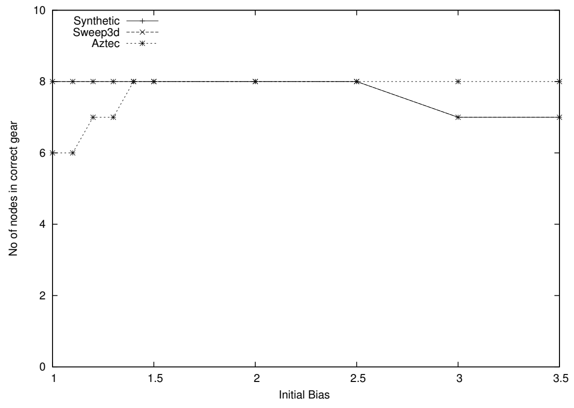
Table 6.6 gives the cost, in microseconds, of shifting gears for the AMD 3000+ CPU. Since our implementation of jitter only shift up or down by one gear, we have only provided the costs to perform such shifts. The cost of each shift is less than 1 millisecond except in the transition from 800MHz to 1000 Mhz. Therefore, the overhead caused by the MPI-Jack, the Jitter algorithm and the gear changes is not significant for iteration lengths of typical duration. However, if the iteration lengths are very small (i.e. few milliseconds), the *Jitter* system would impose noticable overhead.

From (MHz)	To (MHz)	Average	Minimum	Maximum
2000	1800	355.55	340	375
1800	2000	785.46	767	814
1800	1600	278.68	263	300
1600	1800	566.70	551	588
1600	1400	279.47	264	295
1400	1600	279.32	262	299
1400	1200	279.61	264	302
1200	1400	280.20	264	301
1200	1000	362.66	348	383
1000	1200	280.72	265	302
1000	800	297.68	281	316
800	1000	2141.23	2107	2205

Table 6.6: Performance state kernel transition cost (microseconds) between different gears for an AMD 3000+ CPU.



(a) Convergence Speed

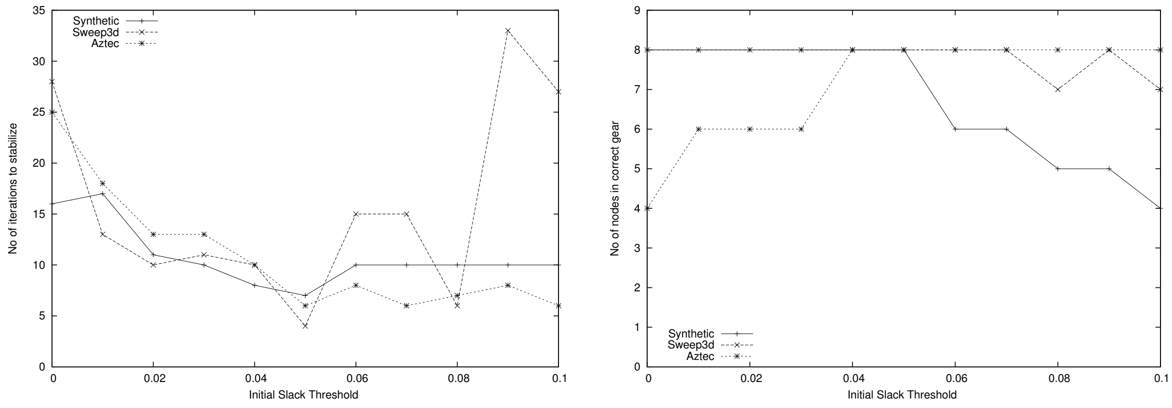


(b) Accuracy

Figure 6.7: Selecting correct BIAS

Sensitivity to BIAS While net slack is used to identify the correct gear for a node, it is not sufficient to ensure that the nodes stay in the correct gear. Jitter sometimes switches rapidly between successive gears. If Jitter chooses to reduce and it turns out to be a bad choice (which is learned when the node later observes a large increase in execution time and so increases performance), the threshold to reduce again is raised by the bias factor. Similarly, we use an upshift factor which is adjusted each time there is an increase in performance using bias in the same way as downshift. This lowers the threshold required to shift up by the bias factor. Figure 6.7 shows Jitter's behavior with bias values varying from 1 to 3.5. While a low bias value causes the nodes to fluctuate in their gear selection, a very high bias value prevents a node which has undergone some initial fluctuations from reaching the correct gear.

Figure 6.7 shows the sensitivity of Jitter to bias. The bias value is varied from 1.0 to 3.5 and its effect on the benchmark is studied. *Accuracy* is defined as the number of nodes that ultimately execute in the correct gears, with reference to the hand-tuned version. *Convergence speed* is defined as the number of iterations required to reach the correct gear. Since minor fluctuations are inherent in Jitter, convergence speed is calculated by considering 5 consecutive iteration, *i.e.,* 40 possible gear shifts. If less than 4 gear shifts (or 10 %) occur during this interval, then the benchmark is considered to have stabilized in its gear selection. Figure 6.7(a) shows that convergence speed does not change much



(a) Convergence Speed (b) Accuracy

Figure 6.8: Selecting correct Slack Threshold

for bias ≥ 1.5. The accuracy is not particularly dependant on bias value (Figure 6.7(b)). Therefore, an initial bias value of 1.5 or 2.5 appears to be best.

Sensitivity to Initial Slack Threshold In early versions of Jitter, our implementation did not learn the slack threshold adaptively. The user had to hand-tune slack threshold for each application as they were very sensitive to the slack threshold. Hence, the implementation of Jitter was modified to learn slack threshold adaptively throughout the duration of the application. Slack threshold has been discussed in greater detail in Chapter 5.

Figure 6.8 shows the sensitivity of Jitter to the initial slack threshold. Jitter starts with an initial slack threshold and adaptively learns the correct threshold. However, the initial slack threshold determines the speed to convergence and the accuracy of gear selection. It appears that a low slack threshold causes the nodes to converge to the correct gears slowly, while a high threshold cause two of the benchmarks to select incorrect gears. However, in the case of Aztec, a low initial slack threshold results in incorrect selection of gears while in the case of sweep3d a high threshold causes the application to converge slowly. From these graphs, we cannot argue for a particular slack threshold. We have found that a slack threshold of 0.05 worked well in the case of these benchmarks.

Chapter 7

Conclusion and Future Work

This thesis describes briefly our approach to saving power in High Performance Computing. We exploit inter-node and intra-node bottlenecks inherently present in high performance programs to reduce performance in the non-critical sections, thereby reducing the energy consumed, without an significant increase in time. This thesis presents a method to extrapolate our findings with our small cluster to predict energy-time trade-offs for large clusters.

Additionally, we have designed and implemented a system called Jitter, which leverages inter-node bottlenecks in MPI programs to save energy. The basic idea behind Jitter is to exploit *slack* time spent by nodes at synchronization points by reducing the energy gear on those nodes, which in turn significantly reduces the consumed energy. Jitter is designed so that nodes will arrive at a synchronization as close as possible to "just in time", so that there will be little or no execution time increase.

Performance results showed that Jitter saves as much as 8% energy, with as little as a 2% time penalty, on a unbalanced program. Furthermore, it has almost no effect on programs that it cannot help—ones where the load is already balanced. We believe that as scientific applications become more complex and adaptive, making it more difficult to balance the load, the usefulness of Jitter will only increase.

Future Work Future work for this research involves testing the Jitter system on different HPC benchmarks. We need to experiment with large-scale programs; while we believe the NAS and ASCI programs are representative, they are not industrial codes that number in the tens or hundreds of thousands of lines. We would like to experiment with large codes which have load-imbalanced and unstable behavior *i.e.,* the loads on the individual nodes vary throughout the lifetime of the program. Behavior of Jitter with unstable loads has only been tested with the synthetic benchmark, for which Jitter has performed admirably. We would also like to study the performance of Jitter on large clusters.

A more complete study can be performed to study the effect of bias and slack on Jitter. We would also like to explore algorithm improvements. We have employed net slack as our metric to determine load-imbalance in a program and reduce performance. Possibly, other methods exist to determine any load imbalance and opportunity to reduce performance which allows better energy-time trade-offs than the current Jitter system. Jitter currently considers reduction of performance only at iteration boundaries. Considering a smaller granularity than iterations might provide more opportunities to save energy. Jitter currently does not exploit memory bottlenecks. The current system could be extended to consider memory bottlenecks as well as load-imbalance as an opportunity to reduce performance. We would also like to consider other components, such as disk, for opportunities to save energy. Overall, the end goal of our research is the development of an energy- and performance-efficient parallel computing infrastructure.

Bibliography

[1] N.D. Adiga et al. An overview of the BlueGene/L supercomputer. In *Supercomputing 2002*, November 2002.

[2] AMD Athlon 64 processor data sheet, February 2004.

[3] Manish Anand, Edmund Nightingale, and Jason Flinn. Self-tuning wireless network power management. In *Mobicom*, September 2003.

[4] ASCI Purple Benchmark Suite. http://www.llnl.gov/asci/platforms/purple/rfp/-benchmarks/.

[5] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. RNR-91-002, NASA Ames Research Center, August 1991.

[6] Awards and prizes, SC 2004 Conference.

[7] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Adaptive load balancing for MPI programs. In *International Conference on Computational Science*, pages 108–117, San Francisco, CA, May 2001.

[8] P. Bohrer, E. Elnozahy, T. Keller, M. Kistler, C. Lefurgy, C. McDowell, and R. Rajamony. The case of power management in web servers. In Robert Graybill and Rami Melham, editors, *Power Aware Computing*. Kluwer/Plenum, 2002.

[9] Enrique V. Carrera, Eduardo Pinheiro, and Ricardo Bianchini. Conserving disk energy in network servers. In *Proceedings of International Conference on Supercomputing*, pages 86–97, San Fransisco, CA, 2003.

[10] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centres. In *Symposium on Operating Systems Principles*, pages 103–116, 2001.

[11] Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. Advanced configuration and power interface specification, revision 2.0. July 2000.

[12] F. Douglis, P. Krishnan, and B. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proc. 2nd USENIX Symp. on Mobile and Location-Independent Computing*, 1995.

[13] C.S. Ellis. The case for higher-level power management. *Proceedings of the 7th Workshop on Hot Topics in Operating Systems*, March 1999.

[14] E. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *USITS '03*, 2003.

[15] E.N. (Mootaz) Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy-efficient server clusters. In *Workshop on Mobile Computing Systems and Applications*, Feb 2002.

[16] W. Feng, M. Warren, and E. Weigle. The bladed Beowulf: A cost-effective alternative to traditional Beowulfs. In *IEEE International Conference on Cluster Computing Cluster 2002*, September 2002.

[17] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *Proceedings of the 7th Conference on Mobile Computing and Networking MOBICOM '01*, July 2001.

[18] Vincent W. Freeh, David K. Lowenthal, Feng Pan, and Nandini Kappiah. Using multiple energy gears in MPI programs on a power-scalable cluster. In *PPOPP 2005*, Chicago, IL, June 2005.

[19] Vincent W. Freeh, David K. Lowenthal, Rob Springer, Feng Pan, and Nandini Kappiah. Exploring the energy-time tradeoff in MPI programs on a power-scalable cluster. In *IPDPS 2005*, Denver, CO, April 2005.

[20] Chris Gniady, Y. Charlie Hu, and Yung-Hsiang Lu. Program counter based techniques for dynamic power management. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture*, February 2004.

[21] D. Grunwald, P. Levis, K. Farkas, C. Morrey, and M. Neufeld. Policies for dynamic clock scheduling. In *Proceedings of 4th Symposium on Operating System Design and Implementation*, October 2000.

[22] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. Dynamic speed control for power management in server class disks. In *Proceedings of International Symposium on Computer Architecture*, pages 169–179, June 2003.

[23] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mahmut Kandemir, and Hubertus Franke. Reducing disk power consumption in servers with DRPM. *IEEE Computer*, pages 41–48, December 2003.

[24] Taliver Heath, Eduardo Pinheiro, Jerry Hom, Ulrich Kremer, and Ricardo Bianchini. Application transformations for energy and performance-aware device management. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, September 2002.

[25] D. P. Helmbold, D. D. E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. In *Mobile Computing and Networking*, pages 130–142, 1996.

[26] C-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *ACM SIGPLAN Conference on Programming Languages, Design, and Implementation*, June 2003.

[27] Jian Ke and Evan Speight. Tern: Migrating threads in an MPI runtime environment. Technical Report CSL-TR-2001-1016, Cornell, November 2001.

[28] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 20(4):869–916, 1998.

[29] Ronny Krashinsky and Hari Balakrishnan. Minimizing energy for wireless web access with bounded slowdown. In *Mobicom 2002*, Atlanta, GA, September 2002.

[30] Orion Lawlor, Milind Bhandarkar, and L. V. Kale. Adaptive MPI. TR 02-05, University of Illinois, 2002.

[31] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Architectural Support for Programming Languages and Operating Systems*, pages 105–116, 2000.

[32] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commerical servers. *IEEE Computer*, pages 39–48, December 2003.

[33] Jiong Luo and Niraj K. Jha. Static and dynamic variable voltage scheduling algorithms for real-time heterogeneous distributed embedded systems. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 719, Washington, DC, USA, 2002. IEEE Computer Society.

[34] John Markoff and Steve Lohr. Intel's huge bet turns iffy. New York Times Technology Section, September 29, 2002. Section 3, Page 1, Coumn 2.

[35] Rami Melhem, Nevine AbouGhazaleh, Hakan Aydin, and Daniel Moss. Power management points in power-aware real-time systems. pages 127–152, 2002.

[36] Robert J. Minerick, Vincent W. Freeh, and Peter M. Kogge. Dynamic power management using feedback. In *Workshop on Compilers and Operating Systems for Low Power*, pages 6–1–6–10, Charlottesville, Va, September 2002.

[37] Donald G. Morris and David K. Lowenthal. Accurate data redistribution cost estimation in software distributed shared memory systems. In *Principles and Practice of Parallel Programming*, pages 62–71, June 2001.

[38] T. N. Mudge. Power: A first class design constraint for future architecture and automation. In *HiPC '00: Proceedings of the 7th Interational Conference on High Performance Computing*, London, UK, 2000.

[39] Orion Multisystems. http://www.orionmulti.com/.

[40] Athanasios E. Papathanasiou and Michael L. Scott. Energy efficiency through burstiness. In *WMCSA*, October 2003.

[41] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the International Symposium on Low-Power Electronics and Design ISPLED '98*, pages 76–81, August 1998.

[42] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. Dynamic cluster reconfiguration for power and performance. In *Compilers and Operating Systems for Low Power*, September 2001.

[43] Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Workshop on Compilers and Operating Systems for Low Power*, September 2001.

[44] Rolf Rabenseifner. Automatic profiling of MPI applications with hardware performance counters. In *PVM/MPI*, pages 35–42, 1999.

[45] Vivek Sharma, Arun Thomas, Tarek Abdelzaher, and Kevin Skadron. Power-aware QoS management in web servers. In *24th Annual IEEE Real-Time Systems Symposium*, Cancun, Mexico, December 2003.

[46] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. Reducing power in high performance microprocessors. *DAC '98, Proceedings of the 35th annual conference on Design automation*, 1998.

[47] A. Vahdat, A. Lebeck, and C. Ellis. Every joule is precious: The case for revisiting operating system design for energy efficiency. *SIGOPS European Workshop*, 2000.

[48] J. S. Vetter. Performance analysis of distributed applications using automatic classification of communication inefficiencies. In *International Conference on Supercomputing*, pages 245–254, May 2000.

[49] J. S. Vetter and M.O. McCracken. Statistical scalability analysis of communication operations in distributed applications. In *Principles and Practice of Parallel Programming*, pages 123–132, June 2001.

[50] Michael S. Warren, Eric H. Weigle, and Wu-Chun Feng. High-density computing: A 240-processor Beowulf in one cubic meter. In *SC 2002*, November 2002.

[51] Scott Wasson. Amd's athlon 64 3000+ processor. http://techreport.com/reviews/-2004q1/athlon64-3000/index.x?pg=1, January 2004. TechReport.Com.

[52] D. Brent Weatherly, David K. Lowenthal, Mario Nakazawa, and Franklin Lowenthal. Dyn-MPI: Supporting MPI on non dedicated clusters. In *Supercomputing 2003*, November 2003.

[53] M. Weiser, B. Welch, A. J. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Operating Systems Design and Implementation (OSDI '94)*, pages 13–23, 1994.

[54] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *Proceedings of Supercomputing '99*, Portland, OR, November 1999.

[55] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Currentcy: Unifying policies for resource management. In *USENIX 2003 Annual Technical Conference*, June 2003.

[56] D. Zhu, R. Melhem, and B. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):686–700, July 2003.

[57] D. Zhu, D. Mosse, and R. Melhem. Power-aware scheduling for and/or graphs in real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):849–864, September 2004.

[58] Qingbo Zhu, Francis M. David, Christo Devaraj, Zhenmin Li, Yuanyuan Zhou, and Pei Cao. Reducing energy consumption of disk storage using power-aware cache management. In *Proceedings of the 10th International Symposium on High-Performance Computer Architecture (HPCA-10)*, February 2004.