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

Nandini Kappiah
Vincent W. Freeh
Department of Computer Science
North Carolina State University
Raleigh, NC, USA
{nkappia,vwfreeh}@ncsu.edu

David K. Lowenthal
Department of Computer Science
The University of Georgia
Athens, GA, USA
dkl@cs.uga.edu

ABSTRACT

Recently, improving the energy efficiency of HPC machines has become important. As a result, interest in using *power-scalable clusters*, where frequency and voltage can be dynamically modified, has increased. 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 balancing load between nodes is one of the long standing problems in parallel and distributed computing.

In this paper we present a system called Jitter, which reduces the frequency on nodes that are assigned less computation and therefore have *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. For example, in Aztec, from the ASCI Purple suite, our algorithm uses 8% less energy while increasing execution time by only 2.6%.

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" [30]. 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 paper. 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 tradeoffs between performance and energy savings.

Previously, we have shown how to exploit this energy-time tradeoff using a single frequency-voltage setting [16] as well as multiple settings [15]. Both of these approaches primarily attack the *intra-node bottleneck*, where the CPU is not on the critical path. At such times, some other component (*e.g.*, the memory) is the bottleneck resource; therefore, reducing the performance of the CPU saves energy with little performance penalty.

This paper addresses the *inter-node bottleneck*, where at least one of the nodes is *not* on the critical path. In other words, some of the nodes arrive early at a synchronization point, meaning that one or more (different) *bottleneck nodes* determine program execution time. In such a situation, a non-bottleneck node will wait for a message (or other event) from another node, which wastes energy. 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. With frequency scaling, a node can shift into a reduced power and performance state so that computation is potentially completed *just in time* for the unblocking event—*i.e.*, arriving just as the light turns green.

In this paper, we present a dynamic, adaptive system for just-in-time performance scaling, called *Jitter*. Our system interposes itself between an application and the MPI library, making it generally transparent to both the application and

*This research was funded in part by a University Partnership award from IBM and NSF grants CCF-0429643 and CCF-0234285.

SC|05 November 12–18, 2005, Seattle, Washington, USA

Gear	Frequency (MHz)	Voltage (V)
0	2000	1.5
1	1800	1.4
2	1600	1.3
3	1400	1.2
4	1200	1.1
5	1000	1.0
6	800	0.9

Table 1: Frequency-voltage pairs for Athlon-64.

MPI. It monitors the time a program waits for external events and then uses *dynamic voltage scaling*—meaning that it dynamically adjusts the CPU frequency (and voltage)—to minimize wait time and energy consumption. (See Table 1; for convenience, in the remainder of this paper we describe this as simply varying the frequency.)

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 ("optimal") solution. Additionally, we show that our solution adapts to changes in load, which a hand-tuned solution cannot do.

The remainder of this paper is organized as follows. Section 2 discusses related work, and Section 3 describes the Jitter implementation. Next, Section 4 describes performance results on a wide variety of benchmarks. Finally, Section 5 summarizes this paper.

2. RELATED WORK

There has been a voluminous amount of research performed in the general area of energy management. 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 [45], where an approach to slack sharing in multiprocessor real-time systems is presented. Another is [46], which extends this work to handle AND/OR graphs.

In the rest of this section, we describe work in more general areas of energy management, including both server/desktop and mobile systems. In addition, there has been a significant amount of work on dynamic load balancing, which is relevant because the difficulty of balancing load dynamically gives rise to our Jitter system.

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 such machines, such as in hosting centers, energy management can become an issue. Chase *et al.* [8] 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 [35, 36]. Elnozahy *et al.* [13] investigated the policy in [35] 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. Such work shows that power and energy management are critical for commercial workloads, especially web servers [6, 29]. Additional approaches have been taken to include dynamic voltage scaling (DVS) [12, 38] and request batching [12]. The work in [38] applies 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 a highly-variable stream of client requests. On the other hand, an HPC installation exists to speedup an application, which is often regular and predictable. One HPC effort that addresses the memory bottleneck is given in [23]; 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 [7], modulating the speed of the disk energy [19, 20], improving cache performance so that the disk can be kept in a low power state more often [47], and using program counter techniques to infer the disk access pattern [17]. 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 [21], and the other uses prefetching [33].

Our work is complementary to techniques that save disk energy. Specifically, while the disk can consume a non-trivial amount of power in HPC applications, the CPU is typically a much larger power consumer. Moreover, unlike the CPU, disks with more than one power-performance setting (*i.e.*, gear) are not yet commercially available. 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 [40], which uses low-power Transmeta nodes. A related approach is the Orion Multisystem machines [32], 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 [16]. 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 to using *multiple* gears per iteration (varying the gear per phase), and developed an algorithm for choosing the assignment of gear to phase [15].

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 [39, 11, 9]. 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 de-

vices focuses on collaboration with the OS (*e.g.*, [42, 44, 2]). 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.*, [14, 34, 18]), the disk via spindown (*e.g.*, [22, 10]), and on the memory or network (*e.g.*, [28, 26]). 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 [31] as well as MPI-based ones such as Adaptive MPI [5, 27], Dyn-MPI [41], and Tern [24].

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.

3. JITTER IMPLEMENTATION

The general idea behind our implementation is to exploit a node that is not on the critical path. Such a node completes its work and idles waiting for a message from another node. With *just-in-time* (JIT) scaling, this node executes at reduced performance and completes its work just before the message arrives from the remote node. Ideally, with JIT scaling, the reduction in performance does not increase application completion time. This is the case as long as the node executing at a reduced gear finishes its computation before the bottleneck node.

In this paper 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. There can be hundreds of compute-communicate bursts per second. Thus, reacting on a per-burst basis is too fine-grain. Therefore, our model aggregates all bursts in an iteration together, rather than monitoring and reacting to each individually. To do this Jitter uses the notion of *slack*, defined as the sum of individual wait times for each burst in the iteration divided by the time of the iteration.

It is important to note that modern OSs such as Linux are already power-aware in that they issue a *HALT* instruction sending the CPU into a reduced power state during idle/wait periods. A trivial improvement possible in a scalable cluster is to shift to the lowest gear before issuing the *HALT*. However, this change adds a latency to transition into and out of the low gear due to the *HALT*, which can take more than 1 millisecond on our AMD. As there are many compute-communicate bursts per iteration, this latency can swamp any possible improvement.

We have created an implementation of just-in-time scaling called Jitter. The implementation is within a layer beneath the application that interacts with MPI, so that it is independent of any particular application. 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, and
- when to reset algorithm parameters.

We describe each of these in turn.

3.1 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 [25]. 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. The current implementation combines iterations until the time exceeds half a second. Jitter performs several actions in *MPI_Jitter*, which are described next.

3.2 Determining the Net Slack

Slack in Jitter is determined as follows. First, to determine wait time, we use our MPI-jack tool, which is an interface that exploits PMPI [37], 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.

Second, we 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 slack and the global minimum slack.

Net slack, rather than absolute slack, is used for two reasons. First, the amount of slack 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 can vary widely among nodes 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).

3.3 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 have tuned S over several applications, and currently choose S between 10 and 20%. Because a user must define S, Jitter is not completely transparent. Future work will develop techniques to derive S dynamically, which will make Jitter completely transparent.

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 increases performance, as described below) 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 $\text{bias} = 2$ and initial $d_g = 1$ works well for all applications. We have not found that the benchmarks are very sensitive to the bias value; however, further experiments are ongoing.

3.4 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, explained below, defines the initial range in which the gear remains the same and must be less than 1. 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.

3.5 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 $\alpha S < \text{net\_slack} < S$, because upshift and downshift factors are initialized to 1. 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. We use $\alpha = 0.5$ in our tests, but it could be a user-provided parameter. However, this simplifies use and has little adverse effect on Jitter's performance.

3.6 When to Reset Algorithm Parameters

In addition the above three *standard* actions (reduce, increase, remain), there is a the 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).

	Time (s)	**Energy** (KJ)
Full	64.8	44.4
Hand-tuned	65.0 (0.3%)	38.6 (-13.1%)
Jitter	66.5 (2.6%)	40.9 (-7.9%)
Reduced	67.1 (3.0%)	40.6 (-8.5%)

Table 2: Aztec results

4. RESULTS

This section present our results. First, we describe our methodology. Next, 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.

4.1 Methodology

Our experimental platform is a cluster of 10 frequency- and voltage-scalable AMD Athlon-64s. Its available operating points are in the range of 800–2000MHz and 0.9–1.5V, see Table 1 in Section 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 [3] as well as all of the NAS suite [4]. Presumably, such mature benchmarks have been thoroughly analyzed and are well-written (*e.g.*, see [43])—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 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 port their respective 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.

4.2 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 2 shows results for Aztec using four different methods.

[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.

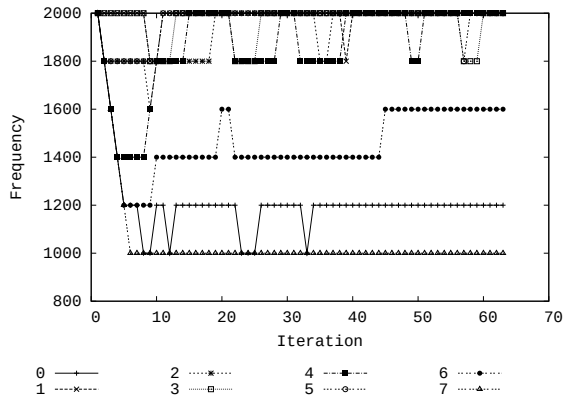

Figure 1: Gears for each node for each iteration in Aztec.

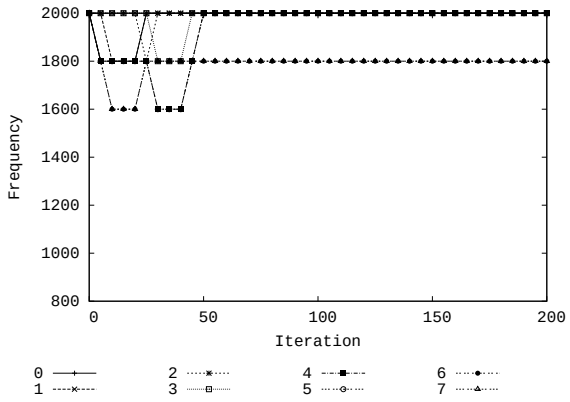

Figure 2: Gears for each node for each iteration in Sweep3d.

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 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 Section 3. 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. Note 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 cycle it becomes harder to reduce due to the increasing downshift factor d_g (see Section 3). 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 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 3 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.

Figure 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.

4.3 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

	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 3: Sweep3d results

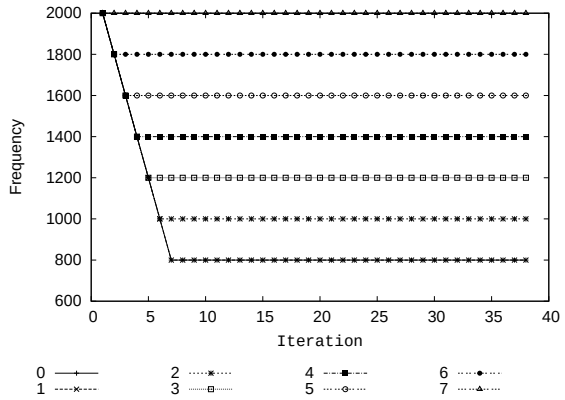

Figure 3: Gears for each node for each iteration in synthetic benchmark with stable, non-uniform load.

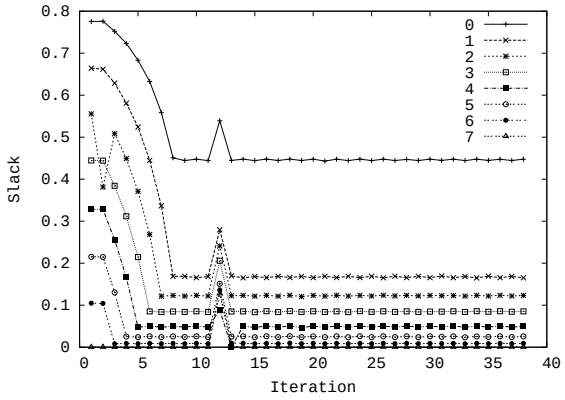

Figure 4: Slack for each node and each iteration in synthetic benchmark with stable, non-uniform load.

	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 4: Synthetic benchmark results

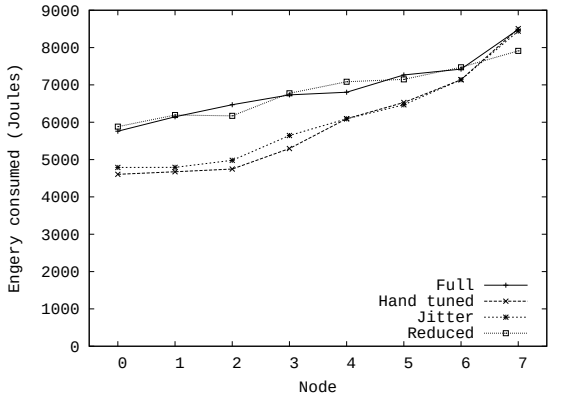

Figure 5: Energy consumed for each node in synthetic benchmark with stable, non-uniform load.

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 executes a barrier.

Figure 3 (on page 6) 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 4 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 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 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 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 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 11.1% less energy.

4.4 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 5). For these programs, the hand-tuned version is on average 0.6% slower

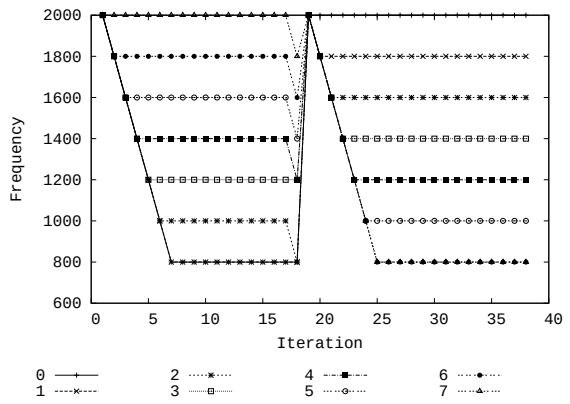

Figure 6: Gears for each node for each iteration in synthetic benchmark with unstable, non-uniform load.

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 [16]. 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 [15].

5. CONCLUSION

In this paper we have designed and implemented a system we call 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.

6. REFERENCES

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%)	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%)	40.0 (0%)
	Reduced	56.8 (2.1%)	36.6 (-8.5%)

Table 5: 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.

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

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

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

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

[5] 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.

[6] Pat Bohrer, Elmootazbellah Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. The case of power management in web servers. In Robert Graybill and Rami Melham, editors, *Power Aware Computing*. Kluwer/Plenum, 2002.

[7] 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.

[8] 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.

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

[10] 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.

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

[12] Elmootazbellah Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy conservation policies for web servers. In *USITS '03*, 2003.

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

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

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

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

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

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] M. Warren, E. Weigle, and W. Feng. High-density computing: A 240-node beowulf in one cubic meter. In *Supercomputing 2002*, November 2002.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.