Adagio: Making DVS Practical for Complex HPC Applications

Barry Rountree
Dept. of Computer Science
The University of Arizona
barry.rountree@arizona.edu

David K. Lowenthal
Dept. of Computer Science
The University of Arizona
dkl@cs.arizona.edu

Bronis R. de Supinski
Lawrence Livermore
National Laboratory
bronis@llnl.gov

Martin Schulz
Lawrence Livermore
National Laboratory
schulzm@llnl.gov

Vincent W. Freeh
Dept. of Computer Science
NC State University
vin@cs.ncsu.edu

Tyler Bletsch
Dept. of Computer Science
NC State University
tkbletsc@ncsu.edu

ABSTRACT

Power and energy are first-order design constraints in high performance computing. Current research using dynamic voltage scaling (DVS) relies on trading increased execution time for energy savings, which is unacceptable for most high performance computing applications. We present *Adagio*, a novel runtime system that makes DVS practical for complex, real-world scientific applications by incurring only negligible delay while achieving significant energy savings. *Adagio* improves and extends previous state-of-the-art algorithms by combining the lessons learned from static energy-reducing CPU scheduling with a novel runtime mechanism for slack prediction. We present results using *Adagio* for two real-world programs, *UMT2K* and *ParaDiS*, along with the NAS Parallel Benchmark suite. While requiring no modification to the application source code, *Adagio* provides total system energy savings of 8% and 20% for *UMT2K* and *ParaDiS*, respectively, with less than 1% increase in execution time.

1. INTRODUCTION

Excessive power consumption continues to be an important problem in high performance computing (HPC). Dynamic voltage scaling (*DVS*) technology addresses this issue by allowing the CPU clock frequency to be changed dynamically. Lower frequencies require less power, and the resulting reduction in energy leads directly to reduced heat and indirectly to longer component mean time between failure [28], less energy required for cooling, and the possibility of greater component density.

Our goal is to develop a runtime system that uses DVS to save energy in scientific applications with *only negligible execution delay*. Other research in runtime systems has achieved impressive energy savings, but at the risk of increasing execution time.

Unlike existing runtime systems, previous work in offline scheduling using linear programming [22] demonstrated CPU frequency schedules resulting in near-optimal energy saving with negligible delay. This approach relies on scheduling changes at MPI communication calls, identifying the critical path of execution to ensure it is never slowed, and approximating ideal frequencies by splitting execution time over available discrete frequencies. However, offline scheduling requires a *complete* program trace at *each* discrete frequency. Further, the use of a linear programming solver to generate the schedule is far too costly to be done at runtime.

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

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

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

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

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

The above *Adagio* features are distinct from the cited offline scheduling approach, where all of these problems are solved using an execution trace from *all* available discrete frequencies.

Adagio is a unique run-time approach to HPC energy savings in that execution delay is negligible *and* the application source code need not be modified. Previous HPC energy-saving algorithms are fundamentally different. Approaches that use a fine-grain history mechanism to predict future behavior [7, 8] will save less energy than *Adagio* when load imbalance can be leveraged to reduce the frequency during computation bursts. Another approach uses per-iteration delay to determine per-processor frequencies [10], but does not detect the critical path correctly in the general case and therefore risks significant program slowdown. *Adagio* differs from these approaches in that it slows only that computation known to be off of the critical path in order to realize energy savings.

We show the effectiveness of *Adagio* for real-world programs as well as standard benchmarks. This includes *UMT2K* [12] and *ParaDiS* [1], two complex, real-world programs. While incurring less than 1% delay, *Adagio* reduces total system energy consumption up to 8% for *UMT2K* and 20% for *ParaDiS*. These are significant savings because the power difference between the fastest and slowest

ICS'09, June 8–12, 2009, Yorktown Heights, New York, USA.

frequencies on our experimental platform is only 39%. We include comparisons of *Adagio* to existing energy-saving algorithms.

The paper is organized as follows. We present definitions, assumptions and a taxonomy of existing runtime algorithms in Section 2. Next, Section 3 details *Adagio*, our new runtime algorithm. In Section 4 we compare the effectiveness of *Adagio* to similar algorithms using real-world applications and standard benchmarks. Finally, we discuss related and future work in Sections 5 and 6.

2. OVERVIEW

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

2.1 Definitions and Basic Assumptions

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

Figure 1 illustrates our execution model. A *task* is the basic unit of scheduling, comprising total communication and computation that takes place on a single processor between the completion of two successive MPI communication calls. The computation portion of a task is measured by an instruction count and an observed per-frequency instruction execution rate. We measure communication by recording the time spent within the MPI library.

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

We define time spent blocked in an MPI communication call as *slack*. By definition, while a processor executes on the critical path, it does not block on MPI communication calls: any process blocked waiting on remote communication can be slowed in order to complete exactly when the remote communication completes without affecting overall execution time. Thus, any blocked process cannot be on the critical path.

The *ideal frequency* is the slowest CPU frequency at which a given task can be run without incurring any slack, that is, the frequency necessary to finish "just in time". The ideal frequency exists in the continuous domain: while the ideal frequency uses the minimum amount of energy [9], it is usually not one of the discrete frequencies available on the processor. If the ideal frequency occurs between the fastest and slowest frequency, we can approximate the ideal frequency by executing part of the task in the higher neighboring frequency and the remaining portion in the lower neighboring frequency. We use the slowest available frequency when it is faster than the ideal frequency.

2.2 Taxonomy

Ideally, runtime HPC *DVS* algorithms satisfy three simultaneous goals: save as much energy as possible, increase execution time as little as possible, and support both simple and complex applications. No existing approach meets all of these goals. Table 1 summarizes the current state of the art in three classes of runtime algorithms along with a near-optimal offline scheduler and compares them to *Adagio*. We discuss existing approaches from each class in more detail in Section 5.

2.2.1 Offline Scheduling

We first briefly review the offline scheduler [22] that uses a complete execution trace for *every* available CPU frequency as input. Given this input, linear programming determines a near-optimal schedule based on MPI call granularity, i.e., the critical path could move from one processor to another at any MPI communication call. Thus this granularity allows critical path identification and prevents slowing of any computation along the critical path. While the MPI communication calls indicate *where* frequencies are to be changed, this algorithm is near-optimal because it lowers the frequency of the computation surrounded by these calls, thus (so far as is possible) eliminating slack. Often, there is no available single frequency that is low enough to remove all slack but not so low that the slowed computation impinges on the critical path. Judiciously splitting the computation across two neighboring frequencies (as detailed in Section 3.2) allows close approximation of the ideal frequency, saving additional power with no additional delay.

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

2.2.2 Scheduled Communication

The simplest class of runtime algorithms, which we term *Scheduled Communication*, uses *DVS* to reduce energy consumption when program execution blocks on MPI communication [14, 15, 22]. This matches the granularity used in offline scheduling. Because data transfer is not computationally intensive, slowing these transfers generally incurs a negligible increase in overall execution time. Because no computation is slowed, the critical path is not affected, avoiding calculation of the ideal frequency. *Scheduled Communication* algorithms save energy in highly complex, production-quality MPI programs, with no source code modification. However, they leave significant potential energy savings untapped.

2.2.3 Scheduled Iteration

Scheduled Iteration methods [10, 16] compute the total slack per processor per timestep, then schedule a single discrete frequency for each processor for the upcoming *timestep* . We define a timestep intuitively as an iteration of a scientific application's outermost loop. This timestep-level granularity works well for simple applications where the critical path remains on a single processor for the duration of each timestep. However, applications with complex communication patterns may have a critical path that crosses several processors during a timestep. In this case, algorithms of this type will choose not to slow any processor that contains any portion of the critical path, thus forgoing any energy savings that might be had on those processors elsewhere in that timestep. As such, we classify these algorithms as not critical-path-aware and not using ideal frequency, although they can be significantly more effective than *Scheduled Communication* algorithms, at least on simple applications, due to the slowing of computation during the timestep. Also, this class of algorithms may require modification of the application source code to indicate the boundaries of the timestep.

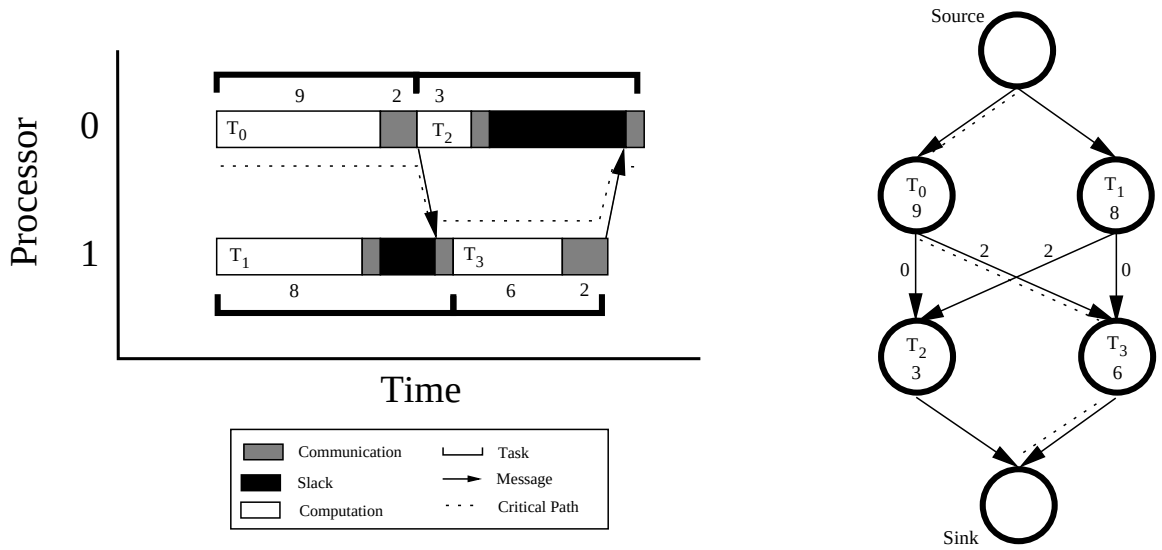

Figure 1: Typical Program Execution in SPMD style (left) and the resulting task graph (right).

Algorithm Class	Online vs. Offline	Granularity	Unmodified App. Source	Critical Path Aware	Slows Comp.	Slows Comm.	Ideal Freq.	References
Offline Scheduling	Offline	**Communication call**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	[22]
Scheduled Communication	**Online**	**Communication call**	**Yes**	**Yes**	No	**Yes**	No	[14, 15, 22]
Scheduled Iteration	**Online**	Timestep	No	No	**Yes**	No	No	[10, 16]
Scheduled Timeslice	**Online**	Timeslice	**Yes**	No	**Yes**	**Yes**	No	[3, 7]
Adagio	**Online**	**Communication call**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	(this paper)

Table 1: Comparison of near-optimal offline scheduling to runtime classes (bold entries show desired characteristics).

2.2.4 Scheduled Timeslice

Scheduled Timeslice methods [3, 7] schedule at fixed time intervals. These algorithms predict the execution characteristics of the upcoming interval (i.e., timeslice) based on recent intervals. They generally select the lowest discrete available frequency for each processor such that predicted slowdown does not exceed a user-specified limit. This timeslice granularity cannot track the critical path, nor do these algorithms use the ideal frequency to match the specified delay. Thus, any delay specification smaller than what would be achieved using the second-highest frequency will result in no computation being slowed. This approach does not require modification to the application source and can save significant energy, but only where significant delay can be tolerated.

2.2.5 Conclusions

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

3. ADAGIO

We begin this section with the *Adagio* implementation. We then provide a discussion of three optimizations: ideal frequency calculation, slack reclamation, and handling large messages.

3.1 Adagio Implementation

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

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

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

Figure 2 shows pseudocode for *Adagio* for the simple case of using a single frequency per task. We detail the optimized split-frequency case in Section 3.2. As runtime algorithms have no prior information about program execution characteristics, *Adagio* schedules execution at the fastest frequency (represented by $\hat{f}$) the first time a task is encountered. If the task reoccurs, *Adagio* schedules it under the assumption of *worst-case slowdown* (execution slowdown proportional to that of the change in frequency). Com-

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

Figure 2: Adagio algorithm with no optimizations.

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

Table 2: Variables used within *Adagio* and their purpose.

putation will not be slowed by more than the ratio of the change in frequencies, *e.g.*, running a task at 1.6GHz instead of 1.8GHz will cause no more than a 12.5% delay. The communication and memory-boundedness of a task may lower this delay substantially. Thus, *Adagio* records the observed slowdown when a task is scheduled and executed in a particular frequency. *Adagio* then uses this refined estimate for subsequent scheduling.

Table 2 summarizes the variables that we use to describe *Adagio*'s algorithm. Throughout this discussion, we will use $\hat{f}$, and $\bar{f}$ to denote the fastest and slowest operating frequencies (i.e., MHz) and will use them to index into tables $FreqSchedule$ and $Rates$ (instead of standard array indices).

Adagio records the number of instructions I and the instructions per second R for the current frequency and task when it completes. Recording R allows *Adagio* to estimate how fast a task would run if it ran at the fastest frequency—a significant contribution beyond previous work [6, 24], which lacked an algorithm to determine execution time as a function of frequency. Further, recording I allows *Adagio* to determine when execution characteristics (i.e., computation) have changed between task instances. We measure the number of instructions using PAPI [21]. We also use `gettimeofday` to measure the total execution time of a task, which we need to compute the instructions per second metric. We emphasize that these counters are collected at runtime and only for those frequencies that are actually used. No training runs are necessary.

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

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

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

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

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

3.2 Optimizations

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

3.2.1 Split Frequencies

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

We can approximate the ideal frequency by using its neighboring frequencies [9]. Our optimized schedule still uses the fastest available frequency when the computation lies on the critical path and the slowest available frequency when the ideal frequency is even slower (slack remains in this case). In any other case, we calculate how long to run the epoch in the two frequencies immediately above and below the ideal frequency. Let q be the percentage of time to execute at frequency f, and let frequencies $f > \phi > f'$. We must satisfy

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

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

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

Using a split-frequency schedule can lead to using a particular frequency for a small amount of time. This choice would incur a time penalty for the additional switch and decrease system stability (at least on our hardware). To counter this, we have empirically arrived at a *switching threshold* of $100ms$ for our cluster. We require any frequency switch to remain in the new frequency for at least $100ms$. Thus, a task will not be scheduled for a lower frequency if the scheduled time would be less than the threshold, split frequencies will not be used unless the time spent in both frequencies will be greater than the threshold (the higher frequency will be used for the entire task), and the threshold time must be exceeded before switching to a lower frequency while in the MPI library.

3.2.2 Slack Reclamation

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

3.3 Large Message Handling

The *FT* benchmark is unusual in several respects (Section 4 contains all results). Among these, required communication for a particular `MPI_Alltoall` call is measured in seconds instead of milliseconds. In this case, our threshold value is too short to handle the amount of communication that occurs. There are several methods to deal with this issue. We could construct an *FT*-specific solution or we could attempt to model expected communication time for these calls. We have instead created a simple, general solution. A side effect of an all-to-all call is synchronization. We wish to separate out the communication time spent blocking waiting for other processes to arrive at the call from the communication time spent transferring data. So, at the beginning of large `MPI_Alltoall` calls, our library inserts an `MPI_Barrier`. Any slack present at this barrier can be reclaimed by scheduling the computation immediately before it as usual. The task terminated by the `MPI_Alltoall` has almost no computation, and slack reclamation occurs as outlined in the previous section.

4. RESULTS

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

(in this case, we used a single core per node). We can independently set the frequency of each processor, but this early-model multicore processor cannot scale core frequencies independently. Consequently, the second core on a processor consumes energy while doing no useful computation, reducing the energy savings we can achieve. *Adagio* has been designed in anticipation of processors with per-core *DVS* control. Future work will extend *Adagio* to handle assignments of processes to cores consistent with our energy-saving goals.

We chose OpenMPI [27] as our MPI implementation. The nodes are connected by gigabit ethernet and have 2GB RAM each. The Opteron 265 supports CPU frequencies 1000 MHz through 1800 MHz in steps of 200 MHz. We use the `sysfs` interface made available by a modified Fedora Core 2 OS running the 2.6.16 kernel for frequency shifting. We compiled all applications with `gcc` or `g77` using the `-O2` optimization flag. The system ran no other processes during our experiments other than the usual daemons.

Our application set includes two complete applications, *UMT2K* and *ParaDiS*, as well as the programs in the NAS suite [20]. For each application, we measure execution time (elapsed wall clock time) and energy consumed. We measure the total system power with precision multimeters at the wall outlet and compute energy using $energy = power \times time$ so energy is *total system energy*, not just CPU energy. While time, and thus energy, can vary across many runs of a benchmark, power does not vary much at all. All results are from direct program executions and measurements, not simulations or emulations. Each benchmark was executed using each indicated algorithm a minimum of five times, with the median time and energy values normalized against the median time and energy for benchmark execution with no DVS scheduling. We also provide a *lower bound* to the energy consumption. This was computed using program traces and indicates the maximum amount of energy that can be saved by *Adagio* given perfect knowledge.

When computing in a tight loop, each compute node in the system consumes 180 watts at the fastest available frequency and 117 at the slowest. Blocking at the slowest frequency reduces the 117 watts to 110. Thus, assuming no time increase due to frequency scaling, an overly optimistic upper bound on possible DVS energy savings on these nodes is 39%. Real applications cannot achieve this bound without increasing execution time because at least one processor must be on the critical path and run at the fastest frequency and, generally, not all non-critical-path processors can be run at the slowest frequency.

4.1 Algorithms

In Section 3 we described the design and implementation of *Adagio*. We now describe our comparison algorithms: *Fermata-1800*; *Adagio-Comp*; *Timeslice*; and *Jitter*.

From the class *Scheduled Communication*, we use *Fermata-1800* [22], which uses the same technique as the slack reclamation algorithm in *Adagio*. All computation is executed at 1800 MHz, while communication runs at the slowest frequency if blocking time exceeds a 100ms threshold. *Fermata* slows only communication. We use *Adagio-Comp*, the *Adagio* algorithm with no slack reclamation, to show the effect when we slow only computation.

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

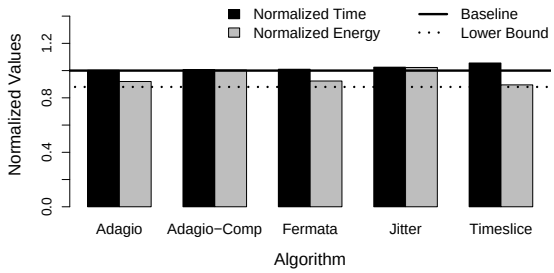

Figure 3: Normalized time, energy, and power for *UMT2K*.

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

From the *Scheduled Iteration* algorithms we use *Jitter* [10]. This algorithm slows the processor for an entire iteration. These iterations are not identified automatically: the user must annotate the source code with a call to `MPI_Pcontrol` and recompile. *Jitter* can detect when an iteration on a particular processor ran longer than expected (when, for example, a portion of the critical path crossed that processor and was inadvertently slowed) and restores the processor to the fastest frequency for subsequent iterations. In the worst case, this can occur on every processor that has some amount of slack, and in this case *Jitter* will eventually schedule all the processors to run at the highest frequency. No energy will be saved, but the only delay will come from the slowdown of the few iterations that were tested.

4.2 UMT2K

The *UMT2K* benchmark [12] is part of the ASC Purple Benchmark Suite [11] assembled by Lawrence Livermore National Laboratory (LLNL). Extensive studies of this benchmark [22] have shown that it is a very challenging test of energy savings for offline, let alone runtime, scheduling.

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

We see this reflected in our results as shown in Figure 3, which indicates the lower bound for energy use as determined by offline scheduling. After finding the median values of runs with no energy scheduling, we recorded the median values of runs for each algorithm and normalized them to the nonscheduled median values. For this application, *Adagio* saved 8% energy with only 0.2% delay.

Fermata, *Adagio-Comp* and *Adagio* ran with less than 1% delay.

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

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

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

UMT2K presents the intriguing possibility that communication could be reordered to yield even greater energy savings. Currently, the application uses a barrier for synchronization after a large computation task. Because the computation is balanced, there is essentially no slack. Then the application performs a sequence of load-imbalanced asynchronous communications that terminate in another barrier or `MPI_Waitall`. Moving to a synchronous communication model could eliminate the need for barriers as well as placing the inevitable slack into the same task as the load-balanced computation. To our knowledge, no research has been done concerning MPI programming techniques that allow for increased energy savings. We plan to revisit this issue in future work.

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

4.3 ParaDiS

ParaDiS [1] is a dislocation dynamics simulation in production use at LLNL. It is a "chaotic" program that converges using a varying number of iterations for the same initial inputs over different runs. Program performance reflects this behavior—total run times in our experiments that do not use energy savings vary up to 4.9%. This nondeterminism makes the program unsuitable for offline scheduling. The power consumed, however, is consistent within an algorithm: the power requirement for each iteration is the same (to the extent we can measure it), and the varying number of iterations are reflected in the varying energy. Thus, while a lower bound on execution time would require multiple traces, a normalized bound on energy savings can be computed from a single trace.

ParaDiS exhibits load imbalance. It is possible to configure *ParaDiS* to perform dynamic load balancing, which reduces (but does not eliminate) slack available for DVS scheduling and makes tasks more difficult to predict. We have successfully integrated *Adagio* into the dynamic load balancer provided by *ParaDiS* and have

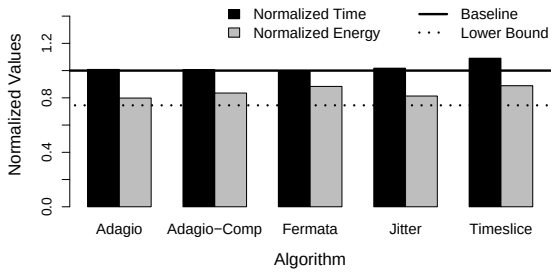

Figure 4: Normalized time, energy, and power for *ParaDiS*.

saved significant energy with less than 1% execution time delay. Space limitations prevent us from detailing these results; this section discusses only *ParaDiS* without dynamic load balancing.

The results are shown in Figure 4. Three of the algorithms had less than 1% delay, and all five achieved significant energy savings. This application is structured so that the critical path tends to stay on the same processor throughout the entire program. The task with the largest computation also has a large amount of blocking communication time (on processors off the critical path). There are also tasks that have short computation combined with long communication. As such, both *Adagio-Comp* and *Fermata* do well in isolation, and the combination into *Adagio* results in a 20.2% energy savings.

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

The challenge with *ParaDiS* does not lie so much in identifying slack, but rather in making sure reclaiming the slack does not slow overall program execution. We must predict the next timestep using only prior information, However, prior information is not necessarily a reliable guide to performance, especially at the beginning of the program.

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

4.4 Summary of UMT2K and ParaDiS results

Broadly speaking, there are two methods for saving energy using *DVS* in MPI programs: slowing communication and slowing computation to reduce slack. The former tends to work well in most programs with significant communication time due to large message sizes, and *UMT2K* is an excellent example of this. However,

there is relatively little slack present in *UMT2K*, and thus very little opportunity to save energy by slowing computation. *ParaDiS* has far greater load imbalance and thus greater slack. This not only allows for the slowing-communication approach to work, but also for slowing of computation.

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

4.5 NAS Parallel Benchmarks

The NAS Parallel Benchmark suite (NPB) [20] is a well-known collection of benchmarks for parallel computing maintained and distributed by the NASA Advanced Supercomputing division These benchmarks generally have a well-balanced computational load, implying no communication slack and thus no obvious opportunity for energy savings. However, *FT* and *CG* proved to be the exceptions. While both are load-balanced benchmarks, the ratio of communication time to computation time in *FT* was high enough that energy could be saved by only slowing communication, and *CG* is sufficiently memory-bound that slowing computation on the critical path resulted in a less than 1% delay. We explore the implications of both of these benchmarks later in this section.

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

We present the results in Table 3. As expected, with the exception of *FT*, no significant energy was saved with *Adagio*; the largest delay was 0.4% (in *SP*). This result is important because *Adagio* is able to recognize when saving energy would incur non-negligible delay. Moreover, the tasks in some of the NAS programs are small enough that if *Adagio* tried to schedule them, scheduling overhead (e.g., frequency switching) would dominate, again leading to too much delay—and possibly even *increased* energy due to this extra delay. These results show that *Adagio* can be used *safely* on applications. When energy savings are possible, as with *UMT2K* and *ParaDiS*, *Adagio* will realize these savings with negligible delay. Where no energy savings are available, *Adagio* does no harm.

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

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

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

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

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

The former characteristic allowed *Adagio* to save 18% energy. Communication is not CPU bound, so executing it in the lowest possible frequency saved energy with negligible delay (the *Fermata* algorithm accomplished the same savings with only 0.4% delay). The latter characteristic caused an unusually high delay of 1.7%. The *Adagio-Comp* algorithm saved essentially no energy while causing a 1.5% delay, and the *Timeslice* algorithm accomplished additional savings by scheduling all of the iterations, which *Adagio* cannot do because of its goal of negligible delay.

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

We have chosen not to implement any of these solutions because we are not persuaded that a real problem exists. The C class version of *FT* ran for a handful of iterations; a more realistic benchmark would have amortized the error over a greater number of iterations. We have observed a similar pattern of behavior in *ParaDiS*; benchmarking runs of a dozen iterations produced suboptimal results, as

[1]The previous reported *Jitter* results for the NAS applications are for 8 processors [10]; this explains some of the discrepancy with our results.

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

Table 3: Normalized Time and Energy for the NAS Parallel Benchmarks.

there is a large amount of variation at startup. But in practice more realistic *ParaDiS* runs show our simple predictor is more than adequate to meet our defined limits.

5. RELATED WORK

Previous work on static scheduling [22] has most heavily influenced the design and implementation of *Adagio*. Specifically, the concepts of MPI-level scheduling granularity came from that work, as did the use of split frequencies (the latter ultimately originating in the real-time work of Ishihara and Yasuura [9]). Several other dynamic voltage scaling runtime algorithms exist. In this section we detail algorithms from the classes *Scheduled Communication*, *Scheduled Iteration*, and *Scheduled Timeslice*, and briefly describe other related work.

5.1 Scheduled Communication

We choose three algorithms to illustrate the class *Scheduled Communication*. We described the first, *Fermata* [22], in Section 4. Li et al. [14], implemented the second, *thrifty barriers*, a similar idea in spirit but aimed at chip multiprocessors. Lim et al. [15] developed the third, a technique to infer communication regions and lower the frequency during those regions. Unlike *Fermata*, this approach lowers the frequency on some computation. However, it is not aware of the critical path and so does not provide time guarantees; instead, it attempts to minimize the energy-delay product.

5.2 Scheduled Iteration

Section 4 described *Jitter* [10], which is the primary *Scheduled Iteration* algorithm of which we are aware for message passing programs. Liu et al. [16] slowed down computation before barriers, a similar idea, for chip multiprocessors. As mentioned earlier, because these approaches make scheduling decisions across the entire timestep, they cannot handle situations where the critical path migrates across processors within a timestep—even if the migration occurs at global synchronization points. Unfortunately, such migration is not unusual; in particular, it occurs in complex applications such as *UMT2K*. As *Adagio* predicts such migration, it provides better results for these kinds of applications.

5.3 Scheduled Timeslice

We select two algorithms to illustrate the class *Scheduled Timeslice*. The first, *CPU-Miser* [7], divides a timestep into many small timeslices, the size of which depends on the current frequency. *CPU-Miser* gathers performance counters for each timeslice and uses past history to select a single frequency for the next timeslice. Another approach uses `cpufreq` [3], a simple command-line interface that makes use of the "userspace" CPU frequency governor in the Linux kernel. Frequency switches occur based on user-specified CPU idle levels and/or CPU temperature.

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

5.4 Other Related Work

Several researchers have developed techniques and systems to save energy with a modest increase in execution time. Cameron et al. [2] and Hsu et al. [8] developed some of the earliest runtime systems to save energy in a performance constrained manner for HPC applications. Additionally, Springer et al. [24] and Ge et al. [7] developed analytic models to predict or to understand energy consumption in the context of scalability. Similarly to Springer et al., Li and Martinez [13] considered both reducing parallelism and frequency scaling, although in the context of chip multiprocessors. Their results showed power savings in almost every situation.

Recent work has explored reducing the amount of concurrency in programs, with one of the benefits of such reduction being lower energy. Ding et al. adapt behavior when cores on a chip multiprocessor are unavailable (which can occur for multiple reasons) [5]. Curtis-Maury et al. fork fewer threads for parallel regions when beneficial [4]. Both papers use linear regression to predict the effect on performance and minimize energy-delay. In contrast, *Adagio* aims at saving energy with negligible delay.

Many researchers have addressed finding optimal energy savings without a time increase in the real-time community. Several have used Mixed Integer Linear Programming to solve the DVS scheduling problem [9, 23, 25, 26] but are limited to a single processor. Zhang et al. used an LP approximation of an ILP solution for the parallel real-time domain [29]. Mochocki et al. [17, 18] continued this work with an emphasis on accounting for frequency transition overhead costs. Zhu (slack reclamation) [30] and Moncusi (hard real time end-to-end deadlines) [19] have investigated non-optimal distributed real-time energy scheduling. Other work [22] used Linear Programming to derive an approximate upper bound on potential energy savings. Unlike *Adagio*, these solutions are all offline.

6. SUMMARY AND FUTURE WORK

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

We are exploring important open issues including the development of techniques that guarantee no added delay when slowing

MPI communication. Porting this work to OpenMP as well as providing a hybrid MPI/OpenMP solution will allow many more applications to save energy. Incorporating processor sleep states may allow savings of even greater amounts of energy.

Many architectural issues also remain. Multicore chips should enable per-core DVS control. Multicore optimization is a very active area of research, and the possibilities of leveraging this work while simultaneously saving energy are intriguing. At a lower level, an architectural description of how DVS affects HPC applications will allow a greater understanding of the design of runtime algorithms. Finally, we are actively working on using this approach in real-time systems, where bounding delay is vitally important.

7. REFERENCES

[1] Vasily Bulatov, Wei Cai, Masato Hiratani, Gregg Hommes, Tim Pierce, Meijie Tang, Moono Rhee, Kim Yates, and Tom Arsenlis. Scalable line dynamics in ParaDiS. In *Supercomputing*, November 2004.

[2] Kirk W. Cameron, Xizhou Feng, and Rong Ge. Performance-constrained, distributed DVS scheduling for scientific applications on power-aware clusters. In *Supercomputing*, November 2005.

[3] Linux kernel cpufreq subsystem.

[4] Matthew Curtis-Maury, James Dzierwa, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *International Conference on Supercomputing*, 2006.

[5] Yang Ding, Mahmut Kandemir, Padma Raghavan, and Mary Jane Irwin. A helper thread based EDP reduction scheme for adapting application execution in CMPs. In *International Parallel and Distributed Processing Symposium*, April 2008.

[6] 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 *International Parallel and Distributed Processing Symposium*, April 2005.

[7] Rong Ge, Xizhou Feng, Wu-chun Feng, and Kirk W. Cameron. CPU MISER: A performance-directed, run-time system for power-aware clusters. In *International Conference on Parallel Processing*, 2007.

[8] Chung-Hsing Hsu and Wu-chun Feng. A power-aware run-time system for high-performance computing. In *Supercomputing*, November 2005.

[9] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *International Symposium on Low Power Electronics and Design*, pages 197–202, August 1998.

[10] Nandani Kappiah, Vincent W. Freeh, and David K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in MPI programs. *Journal of Parallel and Distributed Computing*, 68:1175–1185, 2008.

[11] Lawrence Livermore National Laboratory. *The ASCI Purple Benchmarks*. http://www.llnl.gov/asci/platforms/purple/rfp/benchmarks, 2001.

[12] Lawrence Livermore National Laboratory. *The UMT Benchmark Code*. http://www.llnl.gov/asci/platforms/purple/rfp/benchmarks/limited/umt/, January 2002.

[13] Jian Li and Jose Martinez. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *International Symposium on High-Performance Computer Architecture*, 2006.

[14] Jian Li, Jose F. Martinez, and Michael C. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *International Symposium on High Performance Computer Architecture*, 2004.

[15] Min Yeol Lim, Vincent W. Freeh, and David K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in MPI programs. In *Supercomputing*, November 2006.

[16] Chun Liu, Anand Sivasubramaniam, Mahmut Kandemir, and Mary Jane Irwin. Exploiting barriers to optimize power consumption of CMPs. In *International Parallel and Distributed Processing Symposium*, 2005.

[17] Bren Mochocki, Xiaobo Sharon Hu, and Gang Quan. A realistic variable voltage scheduling model for real-time applications. In *International Conference on Computer-Aided Design*, 2002.

[18] Bren Mochocki, Xiaobo Sharon Hu, and Gang Quan. Practical on-line DVS scheduling for fixed-priority real-time systems. In *Real Time and Embedded Technology and Applications Symposium*, 2005.

[19] M. Angels Moncusí, Alex Arenas, and Jesus Labarta. Energy aware EDF scheduling in distributed hard real time systems. In *Real-Time Systems Symposium*, December 2003.

[20] NAS Parallel Benchmark Suite v3.3.

[21] PAPI: Performance application programming interface.

[22] Barry Rountree, David K. Lowenthal, Shelby Funk, Vincent W. Freeh, Bronis R. de Supinski, and Martin Schulz. Bounding energy consumption in large-scale MPI programs. In *Supercomputing*, November 2007.

[23] Hendra Saputra, Mahmut Kandemir, Narayanan Vijaykrishnan, Mary Jane Irwin, Jie S. Hu, Chung-Hsing Hsu, and Ulrich Kremer. Energy-conscious compilation based on voltage scaling. In *LCTES/SCOPES*, June 2002.

[24] Rob Springer, David K. Lowenthal, Barry Rountree, and Vincent W. Freeh. Minimizing execution time in MPI programs on an energy-constrained, power-scalable cluster. In *Symposium on Principles and Practice of Parallel Programming*, March 2006.

[25] Vishnu Swaminathan and Krishnendu Chakrabarty. Investigating the effect of voltage-switching on low-energy task scheduling in hard real-time systems. In *Asia South Pacific Design Automation Conference*, January 2001.

[26] Vishnu Swaminathan and Krshnendu Chakrabarty. Real-time task scheduling for energy-aware embedded systems. In *IEEE Real-Time Systems Symposium*, November 2000.

[27] OpenMPI Development Team. OpenMPI. http://www.open-mpi.org, 2006.

[28] Ram Viswanath, Vijay Wakharkar, Abhay Watwe, and Vassou Lebonheur. Thermal performance challenges from silicon to systems. *Intel Technology Journal*, Q3 2000.

[29] Yumin Zhang, Xiaobo Sharon Hu, and Danny Z. Chen. Task scheduling and voltage selection for energy minimization. In *Design Automation Conference*, June 2002.

[30] Dakai Zhu, Rami Melhem, and Bruce Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. *IEEE Trans. on Parallel and Distributed Systems*, 14(7):686–700, 2003.