Bounding Energy Consumption in Large-Scale MPI Programs

Barry Rountree
University of Georgia
Athens, GA
rountree@cs.uga.edu

David K. Lowenthal
University of Georgia
Athens, GA
dkl@cs.uga.edu

Shelby Funk
University of Georgia
Athens, GA
shelby@cs.uga.edu

Vincent W. Freeh
North Carolina
State University
Raleigh, NC
vin@csc.ncsu.edu

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

Martin Schulz
Lawrence Livermore
National Laboratory
Livermore, CA
schulzm@llnl.gov

ABSTRACT

Power is now a first-order design constraint in large-scale parallel computing. Used carefully, dynamic voltage scaling can execute parts of a program at a slower CPU speed to achieve energy savings with a relatively small (possibly zero) time delay. However, the problem of when to change frequencies in order to optimize energy savings is NP-complete, which has led to many heuristic energy-saving algorithms.

To determine how closely these algorithms approach optimal savings, we developed a system that determines a bound on the energy savings for an application. Our system uses a linear programming solver that takes as inputs the application communication trace and the cluster power characteristics and then outputs a schedule that realizes this bound. We apply our system to three scientific programs, two of which exhibit load imbalance—particle simulation and UMT2K. Results from our bounding technique show particle simulation is more amenable to energy savings than UMT2K.

1. INTRODUCTION

Power is now a first order constraint in high-performance computing. Hardware designers have responded to this by adding dynamic frequency and voltage scaling (DVS) to newer chips. Lowering the frequency typically involves an energy/time tradeoff. Several systems [1, 9, 11] utilize DVS to produce a better overall time/energy combination for MPI programs, usually measured by the product of energy-delay or energy-delay-squared [7].

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

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

We apply our system to three programs: Jacobi iteration, particle simulation, and *UMT2K* [13]. All programs have at least ten thousand MPI communication events. Given zero allowable delay, a likely goal for HPC applications, our LP-based bounding technique correctly predicts that there is no significant energy to be saved for Jacobi, and shows that while the particle code can save up to 15% energy, *UMT2K* can save only 3.3%.

We emphasize that the significance of this work does not lie in these individual values: a 3.3% savings for an 8K-node cluster is far more interesting than the same savings on a 8-node cluster. Rather, our contribution is the ablity to determine the percentages. If a run-time algorithm does not save as much energy as expected for a particular application, we can now determine if the algorithm is suboptimal or significant energy savings are not possible.

The rest of this paper is organized as follows. We discuss our execution model in Section 2 and the translation of our execution model to a linear programming formulation in Section 3. Next, Section 4 presents the implementation of our LP system. Section 5 discusses the measured results on a real power-scalable cluster. Finally, Section 6 describes related work, and Section 7 summarizes and describes future work.

2. EXECUTION MODEL

Our execution model is a distributed-memory multicomputer on

Copyright 2005 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
SC07 November 10-16, 2007, Reno, Nevada, USA
Copyright 2007 ACM 978-1-59593-764-3/07/0011 ...$5.00.

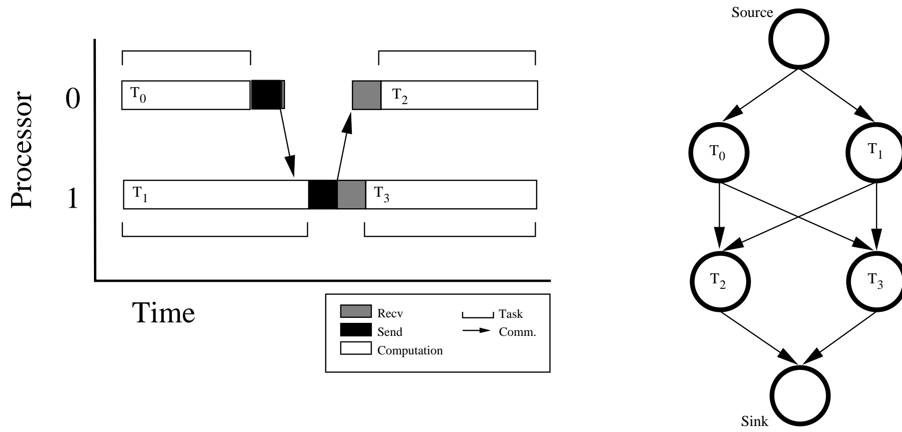


Figure 1: On the left, a sample program shown pictorially. On the right, the resulting task graph (with the small tasks in between the send and receive omitted for brevity).

which each processor (node) executes a unique, predetermined set of tasks. We define a *task* as program execution bounded by communication. Each task runs to completion. The tasks are totally ordered on each processor, and MPI communication events can create intertask dependencies across processors. Any remote communication between tasks (e.g., *send* or *receive*) occurs either before or after the task. Figure 1 (left) shows pictorially an example program in our framework. The task boundaries, which occur at the *send* and *receive* commands, are shown.

The pictorial version of the program on the left side of Figure 1 translates directly to the program *task graph* on its right side. This graph represents tasks by vertices and (communication) dependencies by edges. It has a single source vertex and a single sink vertex.

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

Given a communication graph, the goal is to find both the total time and total energy. Table 1 summarizes the key variables in the execution model. We denote the start time of task i as S_i and the execution time of task i as T_i. All tasks that immediately precede task i in the task graph are its predecessors, denoted $Pred_i$. For task j in $Pred_i$, M_i^j is the time for the message to travel from j to i. We assume M_i^j is zero if i and j reside on the same processor. A processor may commence execution of task i when the predecessors of i in the task graph have completed and it has received its input data from all (remote) predecessors. Thus,

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

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

To minimize energy, we must determine two additional factors. First, we need to determine the execution time of each task if it were to run only using a single available frequency. For $f \in \mathcal{F}$,

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

Table 1: Key variables used in execution model.

we denote C_i^f as the execution time for task i if it were to run only in frequency f (we discuss how we determine C_i^f in Section 4). The fraction of task i completed at frequency f is δ_i^f. We estimate execution time as:[1]

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

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

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

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

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

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

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

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

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

$$E_I = W_I I.$$

Finally, the total system energy is

$$E = E_C + E_I.$$

Our work focuses on minimizing E.

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

3. LINEAR PROGRAMMING FORMULATION

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

We minimize the objective function

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

subject to the following constraints:

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

$$S_j + T_j - S_i \leq 0$$

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

$$S_i + T_i - \omega \leq 0$$

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

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

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

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

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

The most efficient execution will use at most two adjacent frequencies for a given deadline and set of fixed frequencies if the relation between CPU frequency and voltage is at least quadratic [10]. While this relation holds for CPU energy, it may not necessarily hold for total system energy. Other system components' (e.g., fans) activity can result in a system power graph that is not strictly quadratic. In this case, we have observed that the LP will still choose at most two frequencies, but that they may not be adjacent.

4. IMPLEMENTATION

This section describes the implementation of our LP-based infrastructure that bounds energy savings. It consists of a trace collection mechanism, an LP solver, and a run-time mechanism to leverage slack that the LP solver cannot remove.

Broadly speaking, our system determines a near-optimal energy savings using the following procedure:

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

4.1 Trace collection

First, we collect multiple traces by executing the application at the each frequency available on the machine. At the fastest available frequency, we capture the local communication information: the timestamp, source, and destination of each MPI call. This is accomplished with a custom PMPI library that intercepts selected MPI calls. This provides us with communication slack information. We run the program at the other frequencies in order to obtain the execution time of each task at each frequency (which gives us memory slack information) and the average power that an application consumes at each frequency (power usage varies by application). An example of power consumption per frequency for *UMT2K* is shown in Table 2.

We note that this does require $|\mathcal{F}|$ runs of the program. An alternative way to measure slowdown in just a single execution is to compute the *memory pressure* of each task in a program—in other words, how much memory traffic it generates—and map this to task slowdown. While that is more efficient, our approach is more accurate for bounding the gains available for on-line heuristic techniques. Alternatively, we could use an average power per frequency based on microbenchmarks. However, this method is inherently inaccurate.

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

Table 2: Measured power per frequency for *UMT2K*.

4.2 Solution via linear programming

Next, our system combines the data into a single, system-wide task graph, and the graph is transformed into an LP matrix. The matrix, the total execution time limit, and the time for each task at each frequency are passed into the LP solver `glpk`, the GNU Linear Programming Kit [15]. The solution to the linear program is converted into a schedule, listing how long each task should run in each frequency so that system energy is minimized.

4.3 Task binding

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

4.4 Validation

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

4.5 Limitations

Our current prototype has a few limitations; we note that even with these limitations, our system produces schedules that result in a relatively tight bound. First, we do not instrument the MPI library itself, so we do not model most computation within the library. Thus, we generally treat time spent in the MPI library as communication. Second, in the case of the asynchronous communication operations we currently do not consider scaling `MPI_Wait`. Scaling `MPI_Wait` operations that match `MPI_Isend` operations could cause an increase in execution time if the MPI implementation blocks until the matching receive is posted—turning asynchronous communication into synchronous communication. As our large-scale benchmark, *UMT2K*, uses these operations to send multi-megabyte messages, such serialization via MPI does occur. Finally, we do not instrument every MPI operation, but focus on just the key communication operations.

5. EXPERIMENTAL RESULTS

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

5.1 Experimental Methodology

For all experiments, we used a cluster of eight machines, each containing two AMD Opteron 265 dual-core processors. We used only one processor on each node, so we use the terms interchangeably in this section. We chose OpenMPI [28] as our MPI implementation. The machines are connected by gigabit ethernet and have 2GB of RAM. The Opteron 265 supports CPU frequencies 1800 MHz through 1000 MHz in steps of 200 MHz. All frequency shifting was done through the `sysfs` interface made available by a modified Fedora Core 2 OS running the 2.6.16 kernel. All applications were compiled with `gcc` using the `-O2` optimization flag, except for *UMT2K*, which was compiled with `icc` and `ifort`. There were no other processes running on the system except for the usual daemons.

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

For each application, we compute a bound on energy savings using our linear-programming technique described in the previous sections (denoted *LPS* here). We then present the results of using several typical mechanisms to save energy with the application.

The first, called *Comm*, reduces the frequency while the processor is blocked at a communication point. The second, *Slack*, collects the *total* slack time over all nodes and assigns a single (fixed) frequency to each node based on that slack. The node with the least slack runs at the fastest frequency, and the other nodes are assigned a frequency intended to save as much energy as possible without a significant time delay. Third, $Freq_n$ assigns each node a fixed frequency (e.g., $Freq_1$ for the second-fastest; Frequency 0 is fastest and is used as the baseline for normalization). Finally, *Theoretical Bound* is an upper bound (but *not* in general an achievable one). We determine this bound with the LP solver, using two optimistic assumptions: switching cost is zero and the MPI implementation does not block on any send operation. Whereas *LPS* is conservative on operations such as `MPI_Wait`, *Theoretical Bound* is free to aggressively lower the frequency.

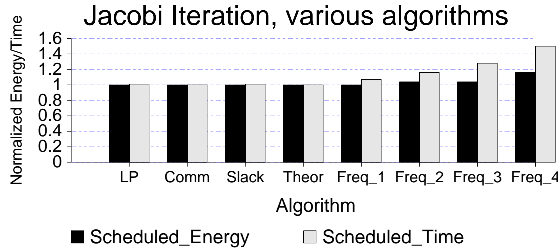
We normalize all execution time and energy results to the unscheduled execution. The unscheduled execution uses the stock OpenMPI library with no communication calls intercepted. For the energy results, a number below one means the scheduled version saves energy.

5.2 Applications

This section describes our three test applications. The first is *Jacobi*, which is a PDE solver and is a canonical benchmark. The second, denoted *Particle*, is a particle simulation based on MP3D from the Splash suite [24]. The third is *UMT2K*, which is from the ASC Purple suite [12]; it is a photon transport code that operates on unstructured meshes. *Jacobi* is load-balanced and is intended to verify that essentially no energy savings are available, while the latter two programs exhibit some degree of load imbalance and hence are amendable to saving energy.

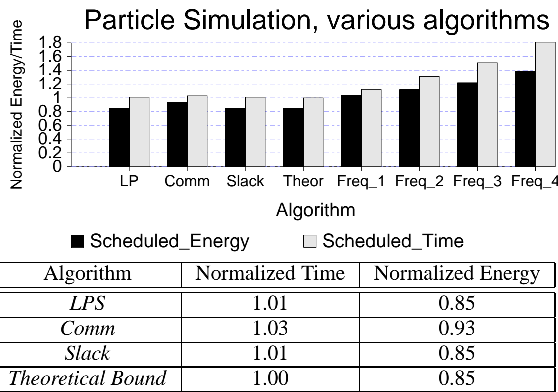
We first consider *Jacobi*; Figure 2 presents the results. Clearly, no energy savings are available due to the balanced load, large computation to communication ratio, and modest demand on memory. Our bounding technique, *LPS*, clearly shows this as expected.

Next, Figure 3 shows that *Particle* is amenable to saving energy. Specifically, our bounding technique reports a 15% potential en-



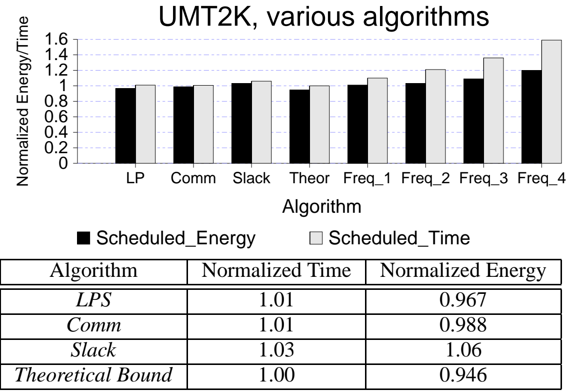
Algorithm	Normalized Time	Normalized Energy
LPS	1.01	1.00
Comm	1.00	1.00
Slack	1.01	1.00
Theoretical Bound	1.00	1.00

Figure 2: `Jacobi` with various algorithms. We normalize all results to the unscheduled execution. The table shows the normalized values for all algorithms other than $Freq_i$.



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

Figure 3: `Particle` with various algorithms. We normalize all results to the unscheduled execution. The table shows the normalized values for all algorithms other than $Freq_i$.



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

Figure 4: *UMT2K* with various algorithms. We normalize all results to the unscheduled execution. The table shows the normalized values for all algorithms other than $Freq_i$.

ergy savings without any time increase. Inspection of the schedule generated revealed that the load imbalance was significant enough to allow early-finishing nodes to execute in the slowest frequency (1000 MHz). For the *Particle* program, the *Slack* technique performs very well; in fact, it saves the maximum amount of energy (15%), also without any significant time increase. This optimal result is because (by chance) the amount of slack available on each node allows scaling such that execution is nearly identical to the schedule generated by *LPS*. In general, we would expect *Slack* to perform well but not quite optimally. Finally, *Comm* saves about half as much energy as *Slack* and also has a 3% overhead to do so. Clearly, *Particle* is a program where energy saving is available and relatively straightforward to achieve.

Finally, we studied a large-scale parallel program, *UMT2K*. This program has been categorized as load-imbalanced [29] and uses both MPI and OpenMP (we disabled OpenMP for our tests). To cut down on machine usage, we reduced the number of iterations that the standard *UMT2K* input file uses.

Figure 4 presents the results. *LPS* computes a bound of 3.3% energy savings, while *Theoretical Bound* for this application is 5.4%. *LPS* incurs a time delay of just under 1%, which we believe is due to a combination of frequency switching time (not modeled by the LP solver) along with the overhead to execute the schedule. Again, we emphasize that *Theoretical Bound* is an upper bound on energy savings that is not actually achievable, i.e., the optimal energy savings is *smaller* than *Theoretical Bound*. Nevertheless, the results show that *LPS* results in a reasonable bound—it generates a schedule that is close to *Theoretical Bound* in terms of energy savings.

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

on entire program iterations and so is too coarse-grained to handle programs such as *UMT2K*. Finally, using a fixed lower frequency takes more time, as expected. More importantly, the energy consumption increases no matter which frequency is chosen.

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

Next, we investigate the effects of memory slack. To measure the improvement gained by considering memory slack, we implemented an alternate version of our LP solver, denoted *Worst Case*. This version instruments the program in only the fastest frequency and then assumes a worst-case slowdown for all other frequencies. For example, at 1600 MHz, *Worst Case* assumes that every task slows down by a factor of 1.125 (1800/1600).

The schedules for each version are shown in Table 3. For reference, the table also shows the schedule for *Theoretical Bound*. In *UMT2K*, the execution time of some tasks slows down by less than the ratio of the frequencies, which results in the *Worst Case* LP solver generating a schedule that executes more in 1600 MHz and less in 1400 MHz. The schedule when using *Worst Case* results in an increase in predicted energy consumption of about 0.5%. We ran *UMT2K* using this schedule and found the actual energy increase to be slightly greater than 1%.

6. RELATED WORK

Several people have developed techniques and systems to save energy without greatly increasing execution time. Cameron et al. [1] and Hsu et al. [9] developed run-time systems to save energy in a performance constrained manner. Kappiah [11] developed a heuristic to reduce frequency in an adaptive manner to save energy in load imbalanced programs. Additionally, Springer et al. [25] and Ge et al. [4] developed analytic models to predict or to understand energy consumption in the context of scalability.

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

Many have addressed finding optimal energy savings without a time increase in the real-time community. Wu provides a taxonomy of dynamic voltage and frequency scaling: online vs. offline, and formal vs. ad hoc [31]. Our work falls into the offline and formal categories. Two of three examples cited by Wu are intended for serial codes [8, 14]. The third example is introduced by Xie, and is closest to our work. Xie's work [32, 34, 33] uses an ILP model to determine maximum energy savings on a single processor, develops a heuristic that approximates the ILP very well and runs much faster, and proposes an analytic model that allows DVS to be applied across separate programs. Our goals and techniques are similar in that we both use mathematical models to bound the maximum energy savings. Our work, however, uses linear programming in the distributed computing domain, which introduces significant additional complexity for the LP model.

Another axis of classification is serial vs. parallel vs. distributed. Wu's work covers serial and Core Multiprocessor [31]. The latter assumes that any processor core can be assigned any task that is ready to run. Our domain is slightly different in that all tasks are assigned to specific processors at the start of the program.

Other researchers [10, 26, 27, 22] have used Mixed Integer Linear Programming to solve the DVS scheduling problem. All have been for single processors. Zhang et al. used an LP approximation of an ILP solution for the parallel real-time domain [35]. This work was continued by Mochocki et al. [17, 16] with an emphasis on accounting for frequency transition overhead costs. Non-optimal distributed real-time energy scheduling has been investigated by Zhu (slack reclamation), and Moncusi (hard real time end-to-end deadlines) [36, 18].

Dramatic energy savings usually require the user to accept longer execution times. Whether or not this delay is significant is determined in part by how much energy is saved, but also by the priorities of the user. For this reason, no metric of "good" performance has been widely-accepted [7]. We have assumed that the user will tolerate as little delay as possible so our schedules last no longer than the run time at the fastest available clock frequency.

Ishihara first investigated the implications of DVS [10]. DVS has been used to reduce energy in environments as diverse as web server farms [2, 23] and mobile devices [3, 19].

DVS scheduling algorithms have been explored in great detail (e.g., [5, 30, 19]). For example, Weiser et al. used past CPU utilization history to determine a frequency for future time periods [30]. Our work is similar only in that we use DVS—we are looking at specific applications, and we must honor a specific time bound.

Instead of finding the most cost-effective way to speed up a schedule, (aka "crashing"[21]), we find the most resource efficient way to slow it down. We assume the schedule modifications are deterministic, much like CPM. Also, PERT networks are able to model stochastic activity times, and recent work shows promise in being able to crash these types of networks [6].

Modeling a parallel program by a task graph is not a new idea. It is common especially in handling programs that exhibit both task and data parallelism—the vertices are tasks, which can be assigned one or more processor, and the edges are dependencies between tasks. The CPR method [20] gave an efficient algorithm, similar to crashing, to find an effective allocation of processors to tasks. Our work focuses on saving energy in tasks, not determining how many processors to assign tasks.

There are several examples of using program traces to analyze performance. These range from the low-level MetaSim and DIMEMAS tools that allow measurements on one architecture to be used to predict performance on another, to the KOJACK toolset that gives a visual representation of performance based on the traces taken. Noeth et al. are an interesting compliment to our work, as the communication aspects of program execution are carefully modelled at the expense of computational fidelity.

7. SUMMARY AND FUTURE WORK

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

Our near-term goals include incorporating other MPI operations, such as `MPI_Startall`, into our system. In addition, we will study a range of allowable delay values to investigate how much ad-

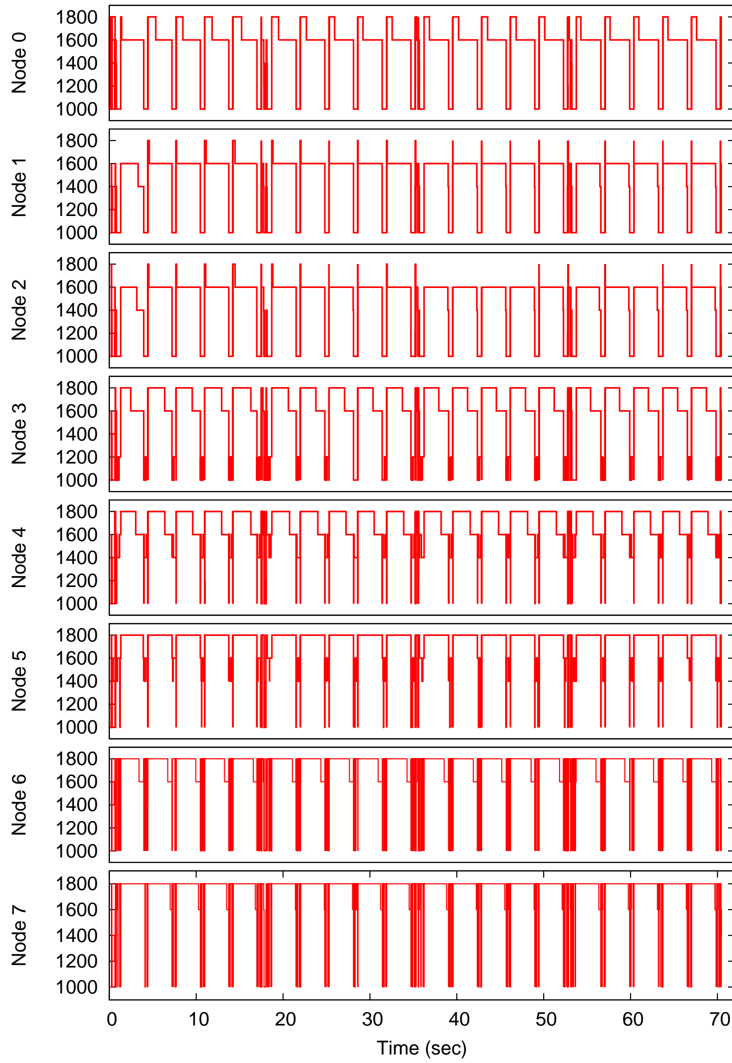


Figure 5: *LPS* schedule used by each node for *UMT2K*, MHz on y-axis.

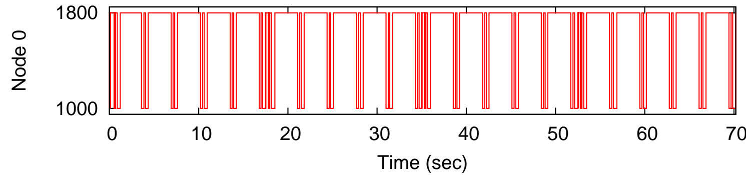


Figure 6: *Comm* schedule on node 0 for *UMT2K*, MHz on y-axis.

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

Table 3: Time in seconds in each frequency according to the schedule generated by three versions of the LP solver. (Blocked time is executed in 1000 MHz.) *Standard* refers to the LP solver used in this section, where the program is executed in each frequency. *Worst Case* refers to an LP solver that assumes worst-case slowdown per task. *Theoretical Bound* refers to the best that can be done under a set of assumptions (see text for details).

ditional energy can be saved. We also plan to model one-way (asynchronous) versus two-way (synchronous) communication more precisely, which will require a greater understanding of the particular MPI implementation.

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

8. ACKNOWLEDGEMENTS

We thank Dr. Franklin Lowenthal for timely assistance with scheduling theory and Dr. Robin Snyder for continuting help with linear programming. We also thank the anonymous reviewers for their detailed comments.

9. REFERENCES

[1] K.W. Cameron, X. Feng, and R. Ge. Performance-constrained, distributed DVS scheduling for scientific applications on power-aware clusters. In *Supercomputing*, November 2005.

[2] E. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Workshop on Power-Aware Computing Systems*, February 2002.

[3] 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*, July 2001.

[4] R. Ge and K.W. Cameron. Power-aware speedup. In *In Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 07)*, March 2007.

[5] D. Grunwald, P. Levis, K. Farkas, C. Morrey, and M. Neufeld. Policies for dynamic clock scheduling. In *Operating System Design and Implementation*, October 2000.

[6] Wayne A. Haga and Tim O'Keefe. Crashing pert networks: A simulation approach. In *4th International Conference of the Academy of Business and Administrative Sciences Conference*, July 2001.

[7] C. Hsu, W. Feng, and J. S. Archuleta. Towards efficient supercomputing: A quest for the right metric. In *Workshop on High-Performance, Power-Aware Computing*, April 2005.

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

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

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

[11] 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. In *Supercomputing*, November 2005.

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

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

[14] J. Lorch and A. J. Smith. Improving dynamic voltage scaling algorithms with pace. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 50–61, June 2001.

[15] Andrew Makhorin. *GNU Linear Programming Kit*. http://www.gnu.org/software/glpk/glpk.html, January 2005.

[16] Bren Mochocki, Xiaobo Sharon Hu, and Gang Quan. A realistic variable voltage scheduling model for real-time applications. In *Proceedings of ICCAD*, 2002.

[17] Bren Mochocki, Xiaobo Sharon Hu, and Gang Quan. Practical on-line DVS scheduling for fixed-priority real-time systems. In *Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, 2005.

[18] 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.

[19] B. D. Noble, M. Satyanarayanan, D. Narayanan, J. E. Tilton, J. Flinn, and K. R. Walker. Application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems and Principles*, pages 276–287, October 1997.

[20] Andrei Radulescu, Cristina Nicolescu, Arjan J.C. van Gemund, and Pieter P. Jonker. CPR: Mixed task and data parallel scheduling for distributed systems. In *The 15th International Parallel and Distributed Processing Symposium*, April 2001.

[21] Barry Render and Ralph M. Stair Jr. *Quantitative Analysis for Management*. Prentice–Hall, seventh edition, 2000.

[22] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J.S. Hu, C-H. Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In *LCTES/SCOPES*, June 2002.

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

[24] J. P. Singh, W. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. Technical report, Stanford University, 1991.

[25] Rob Springer, David K. Lowenthal, Barry Rountree, and Vincent W. Freeh. Minimizing execution time in MPI programs on an energy-constrained, power-scalable cluster. In *11th ACM Symposium on Principles and Practice of Parallel Programming*, March 2006.

[26] Vishnu Swaminathan and Krishnendu Chakrabarty. Investigating the effect of voltage-switching on low-energy task scheduling in hard real-time systems. In *Proc. Asia South Pacific Design Automation Conference*, pages 251–254, January 2001.

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

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

[29] Jeffrey S. Vetter and Andy Yoo. An empirical performance evaluation of scalable scientific applications. In *Proc. SC*,

November 2002.

[30] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Operating Systems Design and Implementation*, November 1994.

[31] Q. Wu, P. Juang, M. Martonosi, L-S Peh, and D. W. Clark. Formal control techniques for power-performance management. *IEEE Micro Special Issue on Energy-Efficient Design*, 25(5):52–63, September 2005.

[32] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-Time Dynamic Voltage Scaling Settings: Opportunities and Limits. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages, Design, and Implementation*, June 2003.

[33] Fen Xie, Margaret Martonosi, and Sharad Malik. Intraprogram dynamic voltage scaling: Bounding opportunities with analytic modeling. *ACM Transactions on Architecture and Code Optimization*, 1(3):1–45, September 2004.

[34] Fen Xie, Margaret Martonosi, and Sharad Malik. Bounds on power savings using runtime dynamic voltage scaling: An exact algorithm and a linear-time heuristic approximation. In *ISLPED*, August 2005.

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

[36] 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.