Determining the Minimum Energy Consumption using Dynamic Voltage and Frequency Scaling [*]

Min Yeol Lim   Vincent W. Freeh
{mlim, vwfreeh}@ncsu.edu
Department of Computer Science
North Carolina State University
Raleigh, North Carolina, USA

Abstract

While improving raw performance is of primary interest to most users of high-performance computers, energy consumption also is a critical concern. Some microprocessors allow voltage and frequency scaling, which enables a system to reduce CPU power and performance when the CPU is not on the critical path. When properly directed, such dynamic voltage and frequency scaling can produce significant energy savings with little performance penalty. Various DVFS scaling algorithms have been proposed. However, the benefit is application-dependent. We can not see if they achieve the energy consumption as minimum as possible. So, it is important to establish the baseline of the DVFS scheduling for any application.

This paper determines minimum energy consumption in voltage and frequency scaling systems for a given time delay. We assume we have a set of fixed points where scaling can occur. A brute-force solution is intractable even for a moderately sized set (although all programs presented in this paper can be solved with the brute-force). Our algorithm efficiently chooses the exact optimal schedule satisfying the given time constraint by estimation. We evaluate our time and energy estimates in NPB serial benchmark suite. The results show that the running time can be reduced significantly with our algorithm. Besides, our time and energy estimations from the optimal schedule have reasonable accuracy with 1.48% of differences at maximum.

1 Introduction

Recent advances in computing systems tend to push performance at all costs. Unfortunately, the "last drop" of performance tends to be the most expensive. One reason is power consumption, because power is proportional to the product of the frequency and the square of the voltage. As an example of the problem that is faced, several years ago it was observed that on their current trend, the power density of a microprocessor will reach that of a nuclear reactor by the year 2010 [4].

To balance the concerns of power and performance, new architectures have aggressive power controls. New microprocessors allow frequency and voltage scaling on the fly, which allows the application or operating system to provide *dynamic voltage and frequency scaling* (DVFS). DVFS is a promising technique, performed in adjusting a power-performance tradeoff because CPU is the major power consumer in a single node, consuming 35-50% of the nodes' total power [6]. We denote each possible combination of voltage and frequency a processor state, or *p-state*.[1] While changing p-states has broad utility, including extending battery life in small devices, the primary benefit of DVFS occurs when the p-state is reduced in code regions where the CPU is not on the critical path. In such a case, power consumption will be reduced with little or no decrease in end-user performance.

Once we scale the CPU voltage and frequency, a cubic drop in power usage occurs. However, the performance decrease is at most linear with frequency reduction, which (usually) increases overall execution time. So, the overall energy savings can be realized only if cost is less than benefit. The relationship between cost and benefit depends on characteristics of applications. If the CPU is on a critical path, then its execution time will be increased in proportion to frequency decrease. On the other hand, if the bottleneck is caused not by CPU but by other factors, such as I/O or network communication, then execution time will not

[*]This research was supported in part by IBM UPP and SUR awards.

[1]The p-state is a processor performance state which defines possible combinations of voltage and frequency. The CPU scaling represents the transition of p-state. In this paper, lower p-state indicates higher CPU frequency, meaning faster speed.

increase as much. Therefore, applying CPU scaling judiciously can result in energy savings with little or no time delay.

DVFS is a popular and effective technique for increasing energy efficiency. However, its effectiveness is highly dependent on applications. This paper attempts to establish an optimal baseline for DVFS scaling for any application. In this paper, we determine the minimum energy consumption for a given time delay. Given the baseline, one can evaluate a specific DVFS technique.

The problem we address is determining the minimum overall energy consumption using DVFS for a given delay. We assume that we are given a finite number of program locations where shifting may occur. The locations define a set of regions in programs. Then, we assume we know the time and energy of each region for each p-state. Further, we suppose that we have the time and energy to transition between any two p-states. A schedule is a list of p-states—one for each region. Our algorithm finds the schedule with the minimum energy from those under the given time limit. There are a potentially large number of schedules: $\mathcal{O}(F^R)$ where F is the number of available p-states and R is the number of regions.

In our approach, we build time and energy models and use empirically obtained data. Then, we design a scheduling algorithm in which the optimal schedule with minimum energy consumption within given time delay is determined with precise estimation. For evaluation of our approach, we use programs from the NAS serial benchmark suite. Our results verify that our time and energy estimations are highly accurate. In addition, we show that our algorithm is efficient—obtaining huge reduction in running time over the brute-force method.

In section 2, we describe related work in power-aware computing research. Section 3 and Section 4 describe our approach and implementation respectively. The section 5 provides the evaluation of our approach and the verification of our scheduling estimates. Then, we close with the summary, point out limitations of our approach.

2 Related work

For decades, performance improvement for both systems and applications has been the only major consideration in HPC research areas. Consequently, it resulted in the advent of large scale power consuming clusters, which causes substantial energy cost. Hence, much research for power awareness in HPC community has been introduced recently. The paper [13] introduces a dynamic compilation technique of HPC applications to reduce power consumption. This work focuses only on I/O optimization in order to save both time and energy. The paper [3] shows the energy savings impact of DVS scheduling strategies in power aware HPC clusters.

There is a body of work to propose DVFS scheduling algorithms for significant energy savings while causing little or no harm to overall execution time. Prior work [1] shows the potential time/energy tradeoff of applying the DVS mechanism in the NAS benchmark suite. In the paper [2], we also proposed a DVFS algorithm, which uses multiple p-state per iteration. The algorithm allows us to choose a highly promising p-state in terms of attaining maximum profit in energy/delay tradeoff. Another recent work [6] exploited inter-node slack time to reduce power consumption. The algorithm automatically identifies idle time at the end of each iteration and then steers the p-state for energy savings. This is an effective approach, especially in a load unbalanced job execution, such as N-Body simulation. More recently, we found an adaptive way to apply DVFS in communication phases of MPI programs [9] because CPU is not usually on a critical path, and we have more chances to achieve an energy savings effect. Our previous work intended to minimize the energy delay product (EDP)

We divide programs into one or more computational regions. Some work has been done in partitioning regions either statically [7, 11] or dynamically [5, 12]. Note that the decision for finding regions, is out of scope in our work.

2.1 Finding optimal schedule

Our problem is equivalent to the problem of finding a path minimizing cost from the root node to the leaf node in a tree. Each node has F (available p-states) children. There exist R (number of regions) levels in the tree. Given this tree, finding the optimal path satisfying problem requirements requires huge computational complexity ($\mathcal{O}(F^R)$). Branch and bound [8] is a general algorithmic method for finding exact optimal solutions for various optimization problems. There are several branching ways, such as depth-first-search and breadth-first-search. The bounding is a fast way of finding bounds (upper or lower) for the optimal solution within a feasible subregion. The efficiency of the method depends critically on the effectiveness of the branching and bounding algorithms used. The paper [14] proposed a scheduling algorithm using it, which is evaluated on architectural simulators. However, we use a branch and bound method with three branching orders after determining the scaling regions on real benchmark applications. We develop our algorithm with a unique branching technique and bounding function to reduce the processing time significantly.

3 Approach

The objective of this paper is to choose a DVFS schedule that minimizes energy consumption while satisfying the allowable time delay in uni-processor programs. The idea is to explore all possible schedules for p-state transitions in program execution. In this work, we assume that we are given a maximum allowable time, T_{Limit} ($T_{Limit} \geq T_0$, T_0 is the execution time at the highest frequency). We next present our time and energy model. Then, we describe our scheduling algorithm to find the optimal schedule.

3.1 Modeling

In this section, we present our model of execution time and energy consumption. We denote $\mathcal{F}$ is a set of available p-states and $\mathcal{R}$ is a set of regions. Moreover, *F* is $|\mathcal{F}|$ and *R* is $|\mathcal{R}|$. First, we are given the time and energy for each region for each p-state, i.e.,

$$T_f^r \text{ and } E_f^r, \ \forall\, r \in \mathcal{R} \text{ and } \forall\, f \in \mathcal{F}$$

Second, we are given the time and energy to transition between p-states, i.e.,

$$T_T(f,\ g) \text{ and } E_T(f,\ g), \ \forall\, f,\ g \in \mathcal{F}$$

where $T_T(f,g)$ and $E_T(f,g)$ are the time and energy to transition from p-state *f* to p-state *g*, respectively (Section 4 explains how we collect the above data.). Then, the problem is to find a schedule (S) which is a list of p-states:

$$S = (\ f_1,\ f_2,\ f_3,\ ...,\ f_R\ )$$

that satisfies the following conditions for a given delay, T_{Limit}:

$$\hat{T}(S) \ = \ \sum_{i=1}^{R} T_{f_i}^i \ + \ \sum_{i=2}^{R} T_T(f_{i-1},\ f_i) \ \leq \ T_{Limit}, \text{ and}$$

$$\hat{E}(S) \ = \ \sum_{i=1}^{R} E_{f_i}^i \ + \ \sum_{i=2}^{R} E_T(f_{i-1},\ f_i) \text{ is minimized.}$$

$\hat{T}(S)$ and $\hat{E}(S)$ are estimated values computed from the given time and energy data and the optimal schedule. In Section 5, we execute the program with the optimal schedule and measure the actual execution time, T(S), and energy consumption, E(S).

3.2 Scheduling algorithm

In brute-force approach, F^R schedules must be evaluated to determine the schedule with minimum energy consumption and at most a given time delay. In this section, we first describe our algorithm in detail. Then, we introduce p-state search ordering to improve the efficiency of our scheduling algorithm.

3.2.1 Finding the optimal schedule

Our algorithm requires the following inputs: an allowable time delay (T_{Limit}), time and energy per region per p-state (T_f^r, E_f^r), and time and energy for all possible pairs of p-state transitions (T_T, E_T). The scheduling algorithm needs to explore a state space tree to determine the optimal schedule. The state space tree is a tree with $\sum_{r=0}^{R} F^r$ nodes to express F^R schedules in leaf nodes. Each internal node has F children. An exhaustive scheduling algorithm will visit all the nodes in the tree. Therefore, finding the optimal schedule through the exhaustive algorithm demands a huge computational complexity.

In our approach, we have designed a scheduling algorithm that is based on the branch and bound algorithm [8]. Thus, our algorithm finds the exact optimal schedule with the same worst case complexity as the exhaustive method. In order to achieve the significant reduction in running time, however, we define a branching way and two bounding functions. The branching way is a best-first-search method [10] in which we first look at the p-state with a lower index in the p-state search list. That reduces the chances of visiting unnecessary nodes because it is highly probable that the remaining nodes will be pruned by the bounding functions later. Our bounding functions are specified from observations in time and energy estimations, respectively.

First, we estimate the time at the *k*-th level in the tree indicated by the partial schedule, $S_k = (f_1, f_2, ..., f_k)$:

$$\hat{T}(S_k) \ = \ \sum_{i=1}^{k} T_{f_i}^i \ + \ \sum_{i=2}^{k} T_T(f_{i-1}, f_i)$$

Then, we check if this partial schedule is valid in time as defined below:

$$\text{If } \hat{T}(S_k) \ + \ MIN_T(k) \ \leq \ T_{Limit} \text{ then } \textbf{valid}$$

$$\text{where } MIN_T(k) \ = \ \sum_{i=k+1}^{R} T_0^i$$

T_{Limit} is a given maximum execution time and $MIN_T(k)$ is the minimum execution time for the $R-k$ remaining regions. This time bounding function implies we do not need to follow the current path in the tree because the minimal value of our estimation already exceeds the maximum time limit.

Next, we estimate the energy for the same schedule:

$$\hat{E}(S_k) \ = \ \sum_{i=1}^{k} E_{f_i} \ + \ \sum_{i=2}^{k} E_T(f_{i-1}, f_i)$$

Then, we check if this partial schedule is valid in energy as defined below:

$$\text{If } \hat{E}(S_k) \ + \ MIN_E(k) \ \leq \ E_{Best} \text{ then } \textbf{valid}$$

Region	Q1	Q2	Q3	Q4	P-State Search Order with 5% Time Delay
R1	[]	[4, 5]	[0, 1, 2, 3]	[6]	[5, 4, 3, 2, 1, 0, 6]
R2	[]	[3, 4, 5]	[0, 1, 2]	[6]	[4, 5, 2, 3, 1, 0, 6]
R3	[]	[2, 3, 4, 5]	[0, 1]	[6]	[4, 5, 2, 3, 1, 0, 6]
R4	[]	[5]	[0, 1, 2, 3, 4]	[6]	[5, 4, 3, 2, 1, 0, 6]
R5	[]	[2, 3, 4]	[0, 1]	[5, 6]	[2, 3, 4, 1, 0, 5, 6]

Table 1. Analysis per region in LU benchmark

$$\text{where } MIN_E(k) = \sum_{i=k+1}^{R} \mathbf{min}(E^i_{f_j} \mid_{f_j \in \mathcal{F}})$$

E_{Best} is a minimum energy consumption found until this point in the algorithm, and $MIN_E(k)$ is the lowest possible energy consumption for the $R-k$ remaining regions. This energy bounding function implies that there is no need to look at the path in case that our estimated value is greater than the current minimum energy.

3.2.2 P-state search ordering

At each node there are F children to be evaluated. The order in which those children are evaluated can have a significant effect on the running time. Therefore, we identify a p-state search order per region to reduce its searching space so that we reach the optimal schedule as fast as we can. The p-state search order contains all available p-states in the order where most likely selected p-state in the optimal schedule is considered first.

There are several ways to specify the ordering. In this paper, we use three ways to order p-states: ordering by time, energy, and quadrant. By time, the p-state list is sorted from faster to slower execution time. By energy, the p-state with lower energy consumption is taken first. The ordering by quadrant divides the p-state list into 4 groups by the baselines of allowable time delay and energy. The groups are ranked by probability in the following order: the group with less time and less energy (Q1), the group with more time but less energy (Q2), the group with less time but more energy (Q3), the group with more time and more energy (Q4). The p-states in each group is also ordered by EDP because the p-state with lower EDP has high probability to be chosen in the optimal schedule. Table 1 gives the p-state search order per region after ordering by quadrant in the NAS LU benchmark.

4 Implementation

For any application, we first collect the time and energy data empirically. Next, we find the optimal schedule by applying our algorithm. Then, we verify our estimation to validate the schedule in the actual algorithm. In this section, we describe several technical issues in the procedure.

4.1 Time and energy data collection

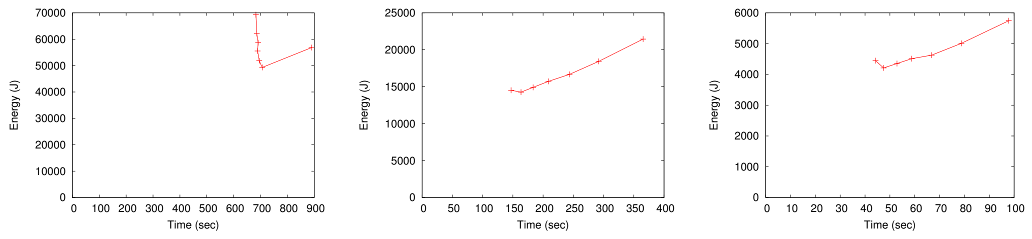
We need to collect data for each program. To do so, the target program is repeatedly executed for all p-states without transitions. On each run, we measure time (T^r_f) and energy (E^r_f) for each region *r* and each p-state *f*.

We also need the time (T_T) and energy (E_T) during p-state transitions. Our research was carried out on AMD Athlon64 processor that is capable of DVFS. In the AMD, the p-state is defined by a frequency identifier (FID) and voltage identifier (VID) pair. The p-state is changed by storing the appropriate FID-VID to the *model specific register* (MSR). The p-state transition has three phases. In the first phase, the processor voltage is transitioned to the level required to support frequency transitions. Next, the processor frequency is transitioned to the frequency associated with OS-requested p-state. In the last phase, the processor voltage is transitioned to the voltage associated with the OS-requested p-state. The overhead of changing the p-state is dominated by the time to scale—not the overhead of the system call. For the processor used in our implementation, the time overhead on a p-state transition (including system call overhead) is in the range between 280 microseconds and 5842 microseconds. It is noted that time increases as the gap between p-states increases. This is because the maximum voltage step is prespecified, and we have more steps to the requested voltage level. We calculate the energy consumption by multiplying the time and the idle power of target p-state

The time and energy per region are context-insensitive. These data are dependent on the current p-state for a region exclusively, not the p-state before or after the region. Therefore, we can compose p-state transition schedule.

4.2 Power measurement

In order to measure the energy consumption of the whole system, we use a WattsUp Pro power meter. The meter is directly attached between the wall power and the system's power connector. serial interface is used to allow the system to monitor its own power consumption. Our meter takes the average of 4,000 samples per second (once per 250 microseconds) at maximum electrically. However, the meter



(a) CG (b) EP (c) Synthetic

Figure 1. Execution time vs. energy consumption from single p-state execution

only reports its measurements at one second intervals. It causes a precision problem of power meter because the current power usage is always emitted after 1 second at maximum. This makes direct energy measurements of regions infeasible. Therefore, we store timestamps of all regions during program executions and also generate the log for power usages separately. From the power logs, we know the average of power usage every a second. For each timestamp, we can take the latest power usage before the time by comparing with the logs. Then, we estimate energy consumption per region by multiplying the elapsed time by the power matched in the log.

4.3 Finding regions

In our implementation, the regions in target programs are specified statically by hand for each application. To do this, we edit the source code of target programs. We recognize this could be improved by automatic insertion. Furthermore, it would be selected from strong analysis for ideal region findings.

We define a data collecting operation as an entry point of region. The region is specified as the code section between two consecutive operations. The operations have two execution modes:*profiling* and *scaling*. In the *profiling* mode, it emits the current timestamps (by *gettimeofday* call) and its program location. The *location identifier* is defined as a pair of hash values of program counters in call history and its counting value to uniquely identify an execution point. In the *scaling* mode, it executes the assignment of schedule by shifting appropriate p-state.

After inserting operations, regions are defined statically in program context. Grouping several operations into one region may be possible depending on the size of regions. Because of the transition time between p-states, we do not consider the region with the time less than 200 milliseconds. NAS serial benchmarks we evaluated have very small code sequence. This limits the number of static regions. For example, only three regions are found in CG benchmark because there is a nested loop and most of time is spent in inner loop.

We also collapse all regions in iterative sections into one region. We can do this for the NAS benchmark because the time and energy for a region do not change over time. However, this is not true for all applications. But our data is sufficient to tell when collapsing is appropriate.

The effectiveness of our approach relies on correct partitioning of regions. We regard the general phase partitioning as orthogonal and complementary research to our work. We have already mentioned several related papers to this area in Section 2.

5 Results

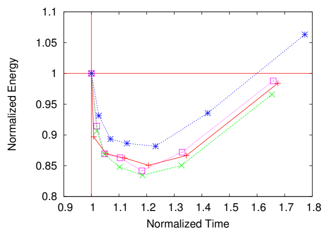
For all experiments, we used a single-node AMD Athlon-64 system with 1 GB of main memory. The Athlon-64 CPU supports 7 p-states (from 2000 to 800 Mhz). The system runs the Fedora Core 3 OS and Linux kernel 2.6.8. All applications were compiled with *gcc* compiler 4.0.2 or the Intel Fortran compiler 8.1 using the O2 optimization.

We use applications from the NAS serial benchmark suite for evaluation purpose. We test class B of these benchmarks. We do not use FT because it takes more than 10 hours for each execution.

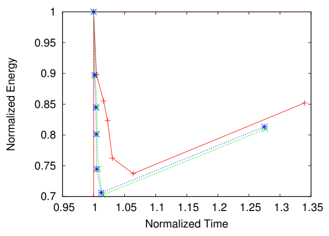
5.1 Detailed analysis

First, we describe the energy-time tradeoff for six NAS benchmarks using a single p-state. Figure 1 shows the results of executing them on a single Athlon-64 processor. For each figure, the total energy consumption at each p-state is plotted on the y-axis and the total execution time is plotted on the x-axis. There are 7 points. Each of them indicates the p-state used. The higher of two points uses more energy, and the further right of two points takes more time. Therefore, a near-vertical slope indicates an large energy savings with little time delay between adjacent gears, whereas a horizontal slope indicates a time penalty and no energy savings.

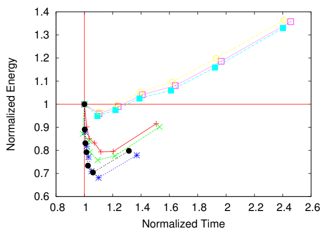
EP is a CPU bound benchmark, so there is little benefit through p-state transitions. On the other hand, CG is a memory bound benchmark which results in large energy



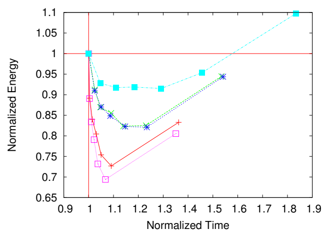
(a) MG (4 regions)



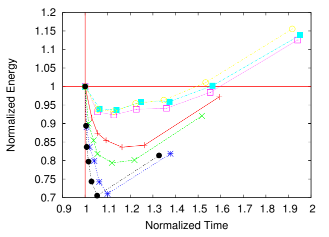
(b) CG (3 regions)



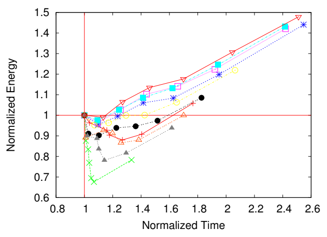
(c) BT (7 regions)



(d) LU (5 regions)

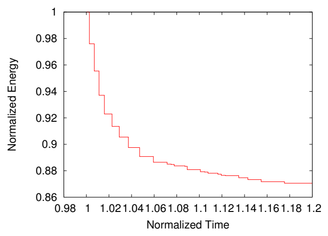


(e) SP (7 regions)

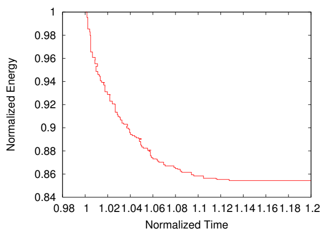


(f) Synthetic (10 regions)

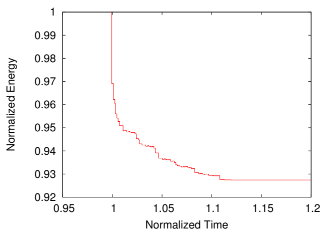
Figure 2. Execution time vs. energy consumption per region



(a) MG



(b) SP



(c) Synthetic

Figure 3. Minimum energy over time

savings and little time delay. All the others are more typical applications which are in the middle of the two extreme benchmarks. EP is not considered further.

In the Figure 2, we plot the execution time and the energy consumption per region by normalizing the time and energy values. For our evaluation, we figured out 1-10 regions by the region finding method described in Section 4.3. We use the average of time and energy values per region in case that there is one region shown repeatedly over iteration. The time and energy values per region have 0.68% and 2.78% of standard deviations on average.

The figures show that time and energy tradeoffs per region can be different each other. In other words, we get different energy savings and time penalty depending on the characteristics of code regions. In our experiments, we use the NAS benchmarks which are iterative with several timesteps during execution. So, we see that many regions have different patterns of tradeoffs because of the static partitioning in our implementation identifies different code sections as a separate region.

After figuring out the time and energy per region, we applied the scheduling algorithm to each benchmark. Table 2 presents overall statistics of the algorithms on each benchmarks to reach the optimal solution. Each benchmark has three values. The first row indicates the number of evaluated schedules which is equal to the number of leaf nodes we visited in the state space tree. The second row corresponds to the number of internal nodes visited while running the algorithm. The last row shows the running time to get the optimal schedule. There is a strong correlation between nodes and running time across all benchmarks. We compared the workloads of the scheduling with three different p-state search ordering by varying the given time delay from 5% to 15%. The results reported our scheduling algorithm can reduce the scheduling workload significantly, compared with the number of possible schedules (F^R). Furthermore, ordering the p-state list provides more opportunities to get rid of node visits in the tree in most cases.

We verified our estimations of time and energy from the scheduling algorithm. The results are shown in Table 3. We first show the allowable time delay varying from 5% to 15%. Next, the estimated execution time and energy con-

		Brute-force	Ordering by time			Ordering by energy			Ordering by quadrant		
		(F^R)	**5%**	**10%**	**15%**	**5%**	**10%**	**15%**	**5%**	**10%**	**15%**
MG	Schedules	$2.401 * 10^3$	23	37	5	5	3	3	5	4	5
(R=4)	Nodes	2801	71	110	119	30	40	23	26	41	25
	Time (sec)	0.059	0.005	0.008	0.008	0.002	0.003	0.002	0.002	0.003	0.002
CG	Schedules	$3.430 * 10^2$	40	40	40	1	1	1	1	1	1
(R=3)	Nodes	400	58	58	58	4	4	4	4	4	4
	Time (sec)	0.005	0.003	0.003	0.003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003
BT	Schedules	$8.235 * 10^5$	94	129	132	3	2	1	3	2	1
(R=7)	Nodes	960800	782	274	273	343	19	8	343	10	8
	Time (sec)	14.398	0.093	0.027	0.027	0.102	0.002	0.001	0.091	0.002	0.001
LU	Schedules	$1.680 * 10^4$	64	120	166	4	4	1	3	3	5
(R=5)	Nodes	19608	183	319	296	65	121	11	64	120	19
	Time (sec)	0.256	0.015	0.026	0.022	0.007	0.020	0.001	0.006	0.035	0.001
SP	Schedules	$8.235 * 10^5$	96	161	167	7	3	1	4	3	1
(R=7)	Nodes	960800	510	520	512	223	24	8	165	24	8
	Time (sec)	14.828	0.068	0.061	0.058	0.071	0.003	0.001	0.068	0.003	0.001
Synthetic	Schedules	$2.825 * 10^8$	73	54	55	12	7	1	4	7	2
(R=10)	Nodes	$3.296 * 10^8$	1485	290	254	632	169	11	269	95	14
	Time (sec)	6007.005	0.224	0.038	0.033	0.095	0.024	0.001	0.04	0.013	0.001

Table 2. The workload analysis of finding the best schedule

sumption are presented. These are generated by the optimal schedule of our scheduling algorithm. Then, we measured the real execution time and energy consumption by using the optimal schedule. In the last two columns, we calculated the differences between our estimation and real measurements in time and energy. For the accuracy of real measurements, we take the median values of time and energy after running the executions 5 times. In cases of IS and CG, the schedules are same for all given time delay because the minimum energy is achieved with less than 5% delay (IS is not shown in this paper due to having only one region). Our results show that the time and energy estimation from our scheduling algorithm are accurate with 1.48% error at maximum in overall benchmarks. In our experiments, we populated the time and energy values per region by only one program execution before performing the scheduling.

In Figure 3, we finally show the estimated minimum energy over time for all the benchmarks. We get these plots by applying our scheduling algorithm repeatedly with a halved time delay until the same schedule is generated. The results show that the minimum bound of energy consumption we can achieve, given regions.

5.2 Discussion

In our approach, the region partitioning is a critical concern to achieve near-optimal scheduling. This is not straightforward but we have guidelines to figure out it effectively. First of all, we know the lower bound of execution time in each region. The time should be much greater than the maximum value of p-state transition overheads. Otherwise, our time estimation will be dominated by the transition overheads, which prevents our scheduling algorithm from estimating minimal energy consumption. Second, we also have the upper bound in the number of regions. In reality, we do not need to define large number of regions for the scheduling. Again, note that more regions may cause more p-state transition overheads.

In this paper, we assume that target programs are single processor applications for now. As future work, we will address this problem on multiple processor programs with mutual communications, such as MPI programs. Extending our model and estimation to deal with parallel programs requires more complicated analysis because the programs in parallel systems often use collective operations to synchronize themselves at some points. The p-state transition in one node may cause collateral delay in other nodes, which do not guarantee to finish the job within given allowable time delay. Thus, the state space tree has much more nodes to be considered.

6 Conclusion

In this paper, we proposed a way to achieve the minimum energy consumption by applying voltage and frequency scaling on the fly. The basic idea is to build the time and energy estimation models, given regions in programs. Then, we designed an efficient scheduling algorithm to compose the best p-state transitions for all regions while satisfying a given time delay. Results on the NAS serial benchmark suite showed that the running time can be reduced significantly with our algorithm. Our time and energy estimations are highly accurate with 1.48% of differences at maximum, compared by real measurements.

Our future plans include designing an automatic region finding method so that we determine the minimum energy consumption efficiently. Additionally, we will extend this research on parallel programs. In this case, we have much

		Time Delay (sec)	Estimated Time (sec)	Estimated Energy (J)	Measured Time (sec)	Measured Energy (J)	Time Error (%)	Energy Error (%)
	5%	34.97	34.75	3218.56	34.60	3257	-0.43	1.19
MG	10%	36.43	36.11	3168.91	35.81	3158	-0.83	-0.34
	15%	37.88	37.63	3142.34	37.20	3189	-1.14	1.48
	5%	715.98						
CG	10%	749.16	697.16	49409.77	697.01	49165	-0.00	-0.50
	15%	782.34						
	5%	566.68	566.20	53203.87	565.87	53521	-0.06	0.60
BT	10%	593.37	592.78	52132.53	594.42	52388	0.28	0.49
	15%	620.07	594.38	52070.00	594.24	52180	-0.02	0.21
	5%	571.60	571.34	49035.22	572.06	49094	0.13	0.12
LU	10%	598.50	598.33	47807.41	598.25	47891	-0.01	0.17
	15%	625.40	618.70	46755.73	618.52	46782	-0.00	+0.00
	5%	489.24	489.07	42807.32	489.12	42878	+0.00	0.17
SP	10%	512.33	510.99	41519.96	513.80	41994	0.55	1.14
	15%	535.43	525.59	41326.46	523.21	41158	-0.45	-0.41
	5%	46.43	46.41	4163.49	46.11	4141	-0.65	-0.54
Synthetic	10%	48.64	48.50	4130.29	48.18	4082	-0.66	-1.17
	15%	50.85	49.18	4121.60	49.70	4114	1.06	-0.18

Table 3. Estimation verification

more complexity not only because of the number of nodes, but also because of the communications each other. The ultimate goal is to design and implement a system which efficiently figure out the energy saving potential by DVFS for all programs.

References

[1] V. W. Freeh, D. K. Lowenthal, F. Pan, N. Kappiah, and R. Springer. Exploring the energy-time tradeoff in mpi programs on a power-scalable cluster. In *International Parallel and Distributed Processing Symposium*, 2005.

[2] V. W. Freeh, F. Pan, N. Kappiah, and D. K. Lowenthal. Using multiple energy gears in mpi programs on a power-scalable cluster. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005.

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

[4] R. Goering. Current physical design tools come up short. *EE Times*, April 14 2000.

[5] M. Huang, J. Renau, and J. Torellas. Positional adaptation of processors: Application to energy reduction. In *International Symposium on Computer Architecture*, June 2003.

[6] N. Kappiah, V. W. Freeh, and D. K. Lowenthal. Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs. In *ACM Supercomputing conference*, 2005.

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

[8] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems, 1960.

[9] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, transparent frequency and voltage scaling of communication phases in mpi programs. In *ACM Supercomputing conference*, 2006.

[10] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[11] U. Rencuzogullari and S. Dwarkadas. Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations. In *Principles and Practice of Parallel Programming*, June 2001.

[12] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[13] S. W. Son, G. Chen, M. Kndemir, and A. Choudhary. Dynamic compilation for reducing energy consumption of i/o-intensive applications. In *International workshop on Language and Compilers for Parallel Computing*, 2005.

[14] F. Xie, M. Martonosi, and S. Malik. Bounds on power savings using runtime dynamic voltage scaling: An exact algorithm and a linear-time heuristic approximation. In *International Symposium on Low Power Electronics and Design*, Aug. 2005.