Exploiting Barriers to Optimize Power Consumption of CMPs

Chun Liu

Anand Sivasubramaniam

Mahmut Kandemir

Mary Jane Irwin

Dept. of Computer Science and Eng.,

The Pennsylvania State University,

University Park, PA 16802.

{chliu,anand,kandemir,mji}@cse.psu.edu

Abstract

Power consumption is an important concern for future billion transistor designs. This paper proposes a novel technique for optimizing the power consumption of chip-multiprocessors (CMPs) using an integrated hardware-software mechanism. By using a high level synchronization construct, called the barrier, our technique tracks the idle times spent by a processor waiting for other processors to get to the same point in the program. Using this knowledge, the frequency of the processors can be modulated to reduce/eliminate these idle times, thus providing power savings without compromising on performance. Using real applications from the SpecOMP suite, and a complete system CMP simulator, we demonstrate that this approach can provide as much as 40% power savings (and 32% on the average across five applications) with little impact on performance.

1. Introduction

There are several compelling reasons that make chip-multiprocessors (CMPs) an attractive option for future high performance designs. Designing a single high performance core to exploit the denser integration capabilities of future billion transistor designs can make this core very complex to design and verify. CMPs consisting of several simpler processor cores can offer a more cost-effective and simpler way to exploit these denser levels of integration. CMPs also offer a larger granularity (thread/process level) at which parallelism in programs can be exploited by compiler/runtime support, rather than leaving it to the hardware to extract the parallelism at the instruction level on a single (larger) multiple-issue core. Further, as a recent study [19] comparing CMP and SMT designs shows, the CMP can be a more energy efficient way of utilizing the available silicon space. There is clear evidence of the trend towards CMP designs in several commercial offerings and research projects [17, 11, 15, 16, 2].

At the same time, the denser levels of circuit integration make power dissipation a serious concern in chip design.

Slowing down the clock to accommodate lower supply voltage levels to reduce power consumption can have ramifications on the performance of applications. Consequently, it is imperative to design techniques that can provide considerable power savings without having a significant impact on performance. A symbiotic interaction between the underlying hardware and the high level software running on it can go a long way towards meeting this goal. While the underlying hardware can provide several mechanisms for controlling power consumption, the software can use high-level workload information towards modulating the hardware mechanisms in an effective manner to maximize the rewards.

In this paper, we present one such novel symbiotic relationship between a hardware power¹ saving mechanism (namely dynamic voltage-frequency scaling (DVS) [3]) and the application software, wherein the latter exploits high-level workload information regarding work imbalance between the parallel threads to control operating frequencies/voltages. Applications running on CMPs can be expected to take advantage of the multiple processor cores by creating activities (either light-weight threads or heavy-weight processes) to run in parallel. Without loss of generality, we assume that there is one such activity per processor and consequently use the terms process/thread/processor interchangeably. High-level programming models, such as the OpenMP [6] standard, typically allow application programs to create these processes, specify the work to be performed by each, and employ mechanisms for synchronizing them when needed. Synchronization is needed to ensure that data and control dependencies are correctly enforced before a processor performs the next chunk of work assigned to it. At such synchronization points, a processor may need to wait for one or more processors to get to a specific part of the code before all can proceed. As a result, it is possible for a processor to stall/wait for a considerable amount of time. We believe that one can exploit this high level information of processors idling at synchronization points in order to scale the frequencies/voltages appropriately such that the amount of idling is minimized, i.e. without impacting the overall execution time, the frequency/voltage is lowered to provide power savings.

One such synchronization construct that is popularly used in parallel programs is a *barrier*. When a processor

¹ In this paper, even though we are mainly focusing on saving energy, we use the terms power and energy interchangeably.

comes to a barrier that is pre-set to a specified count x , it is stalled until $x-1$ other processors also come to that barrier. All processors, except the last one(s) to get to the barrier, are thus wasting time which is not devoted to any useful work. Elimination of this idleness will not contribute to lengthening the program execution. However, it can help in power optimization in two ways. One way of exploiting this idleness is to transition the processor to a low power state (from a fully active state) when arriving at the barrier, and wake it back up when the last processor gets there. This is the option that is explored in [13]. An alternate strategy, which is explored in this paper, is to lower the frequency/voltage of processors that would have reached the barrier ahead of time, when they are executing useful work, so as to make sure they reach the barrier at the same time as the last processor. By applying DVS, we can reduce the power consumption even during the periods of useful work being performed, without affecting the execution time (as long as we are able to predict the idleness at the barrier accurately).

The effectiveness of this approach depends largely on (i) the imbalance among the processors in the work assigned before getting to the barrier, and (ii) the ability to predict this imbalance (i.e. the idleness of a processor at the barrier) accurately. Consequently, it is important to examine the execution characteristics of real applications with a detailed architectural model to evaluate the pros and cons of this approach. In this paper, we investigate this issue using applications from the SpecOMP[1] suite. These are parallel applications written using OpenMP [6] directives, that are meant to be representative of realistic parallel workloads. We execute these applications on a detailed CMP platform that models multiple processor cores, their caches and an on-chip interconnect between the cores, using the Simics [14] complete system simulator that models the operating system code as well.

From our simulations, we observe that there is considerable work imbalance between the processors, suggesting that there is significant scope for power savings using our technique. We also demonstrate that simple predictors based on prior history (in fact, the immediately previous value suffices in many cases) can provide accurate estimate of processor idleness at the barriers. Based on these observations, our barrier-based frequency/voltage scaling strategy provides 32% energy savings on the average across the benchmarks, with only a 3.7% degradation in performance on the average.

The rest of this paper is organized as follows. The next section discusses related work; Section 3 and 4 elaborate on our method; Section 5 gives our experimental setup and section 6 presents the performance and power results. Finally, Section 7 concludes the paper.

2. Related Work

2.1. Multi-Core Processors

In [12], the authors discuss the possibility of using single-ISA heterogeneous cores to attack the power consumption problem. Since the previous generation cores are much smaller and consume less power, dynamically switching between cores can bring better energy efficiency. Due to the fact that only one core can be active at any time, such an architecture is targeting serial applications.

ACPI (Advanced Configuration and Power Interface)-type power management is exploited on barrier constructs in [13] to investigate the power efficiency of parallel applications. Basically, the authors argue that a core can be put into sleep mode when it reaches the barrier early. However, on-off control between sleep and full operation modes can be very costly. Further, scaling of voltage even during the execution before getting to the barrier can possibly give much higher energy savings.

An earlier study [19] have pointed out that the CMP can be an energy-efficient alternative to exploiting future billion transistor designs, and has also mentioned that voltage scaling can further complement this architecture. They show around 9% to 15% power savings in multimedia applications that use independent threads. Our work is motivated by this observation, and demonstrates a novel way of applying voltage scaling to show energy savings in the context of SpecOMP parallel applications.

3. DVS in Chip Multi-Processors

(a) Synchronous

(b) Asynchronous

Figure 1. Two dynamical voltage and frequency scaling approaches for CMPs. VC: Voltage Controller, PLL: Phase Locked Loop

At least two alternatives are available for implementing DVS in CMPs. In the first one, depicted in 1(a), the scaling is applied to all processors uniformly. In this case only one supply voltage regulator and programmable PLL is needed. Since the processors are all running at the same clock rate and supply voltage, the interconnect between them can be synchronous and can run at the current clock rate.

In the second alternative, depicted in 1(b), each processor has its own supply voltage regulator and programmable PLL. Thus, each processor can be set to its "ideal" clock rate with its supply voltage determined by its computational load. However, since the processors are no longer running in lock-step, an asynchronous design (multiple clock domain [20] or globally asynchronous locally synchronous (GALS) design [8]) will be required, incurring the additional overhead of request/acknowledge lines, circuit and timing overhead. Additionally, buffers may be required at the sender and/or receiver to accommodate processor speed mismatches.

The PLL and main driver accounts for about 10% of the clock energy [7], and this percentage increases as the technology scales. If we employ the second approach, the single PLL and main clock driver are replaced by local PLLs, and the main driver can be eliminated. Since the PLL itself does not consume much energy, we conservatively assume it uses 1% of the clock energy. Therefore, we could expect about 6% savings in clock system energy by employing asynchronous design for a four-core processor.

In [20], the frequency and voltage of different regions of a single core processor, namely front-end, integer unit, floating-point unit and load-store unit, are adjusted independently and dynamically. The study shows that 16.7% energy reduction could be achieved while incurring a 3.2% performance loss. Similar results are shown in [9]. Considering that the parallelized applications tend to have partitioned work to achieve good speedup, especially with the MIMD (or SPMD) programming style, the interconnect between the cores do not need to be as closely coupled as the different units of a single core processor. Consequently, it could be more rewarding to bring multiple clock domains to different cores of a CMP than to the different regions of a single core processor.

Since our methodology exploits work imbalances across the cores, we would like to employ separate scaling for each core, and thus use the second configuration shown in Figure 1.

4. Our Approach

In this section, we give a quick overview of the barrier construct that is in wide use in parallel programs, following which we present our proposed optimizations.

4.1. Barriers

The parallel activities (processes/threads) of a parallel program often need to exchange information during their execution. Synchronization is used to avoid race conditions, and to ensure the validity of the data values that a processor uses in its computation. One commonly used synchronization construct is the *barrier*. A barrier construct is typically initialized with a count field, say x . A processor making a barrier call, is stalled/blocked until x processors in all

have come to that barrier. This is pictorially depicted in Figure 2 with four processors, where P3 gets to the barrier first, and needs to wait for time s_3 , before the last processor, P0, also gets to the same barrier (whose stall time is 0). The other two processors incur stall times between 0 and s_3 .

This barrier construct is in wide use in parallel programs, since it is an easy way of demarcating phases of a program, or a way of ensuring each processor is done with one or more iterations of a loop, before proceeding. Consequently, it is extensively used at loop iteration boundaries (as in Figure 2) to ensure that the data values which every processor reads in later iterations are not stale. Many shared memory programming libraries support this barrier interface, including OpenMP [6] that is becoming a standard for shared memory parallel programs. Our experimental evaluations use parallel programs written using OpenMP, though the optimizations that we propose below are general enough to be useful across a wide class of parallel programs. Note that we are not proposing any new barrier construct itself. Rather, we use the existing barrier construct (implemented as busy-wait in OpenMP) and simply insert code around this construct as will be detailed shortly.

Figure 2. Active and stall times using a barrier that is being invoked iteratively.

4.2. Proposed Optimizations

Reducing/eliminating the stall times (s_1 through s_3 in Figure 2) is a promising way to optimize the power consumption problem without affecting performance, as long as we do not affect the time taken by the last processor to get to the barrier. There are two ways of tackling this issue: (i) transiting the processor core to a low power mode over the duration of the stalls (as explored in [13]), or (ii) applying DVS during the work execution periods of P1, P2 and P3 (i.e., during times a_1 , a_2 , a_3), such that we push s_1 through s_3 close to zero. We believe that the latter approach provides larger scope for power savings (since the work execution periods - referred to as active times - are typically much longer than the wait times, and a lower voltage/frequency during those periods can provide significant power savings), and

the transitioning costs between voltage/frequency levels are also much lower (typically a few cycles as in [4]) than powering up/down the core completely. As we will show later in our experimental results, our technique does provide higher power savings.

Our study mainly focuses on barrier constructs that are used in loops to avoid data hazards across different iterations. It is to be noted that loops constitute the bulk of the execution time of most applications, and any optimizations within loops can contribute to substantial overall savings. Consider the following program fragment where the application uses an OpenMP compiler directive (`#pragma omp parallel`) to parallelize a loop construct that is itself inside an outer loop:

```
for (i = 1 to niter) {
  #pragma omp parallel
  for (j = 1 to N) {
    ....
  }
}
```

The compiler translates this code fragment to the following form that is executed by each of the parallel threads (whose creation is not explicitly shown here), where `mystart` and `myend` delimit the iteration space for each thread.

```
for (i = 1 to niter) {
  for (j = mystart to myend) {
    ....
  }
  --- (A) Arrive at Barrier
  barrier(); --- Inserted by compiler
  --- (B) Leave Barrier
}
```

The barrier ensures that each processor is done with the inner loop before they all start the next iteration of the outer loop. The compiler analysis to figure out when and what kinds of synchronization to use is quite involved. In this study, we simply use an off-the-shelf OpenMP compiler that inserts these barrier calls, and additionally insert code at points (A) and (B) to perform certain functionalities as explained below. The overheads for these are captured in our experimental results.

Setting the frequencies/voltages to reduce the idleness at the barrier in the above example requires estimating the stall times of a processor at the barrier, and the time it spends in executing useful work itself (the inner loop) before it gets there. Let us say that we estimate the time to be spent in useful work for the next iteration of the outer loop to be a at the maximum clock frequency f_{max} , and the idle time at the barrier to be s . Then we can set the frequency (the compiler can generate code for doing this at point (B) in the program) for the next outer loop iteration as $f = \frac{a}{a+s} \times f_{max}$. We assume that the hardware supports an instruction (as in [4, 18]) to effect such frequency changes. Since the frequency levels that are offered by the hardware are discrete, once we calculate the target frequency using the above approach and find that it falls in-between two levels, we set it to the higher of the two levels in order to not penalize performance.

There are different techniques that can be used for predicting the active and idle times, and sophisticated predictors can be developed for doing so. For most of our experiments, we use a simple predictor – *last-value predictor* – wherein the current observation (for both a and s) is used as the estimate for the next iteration. We show that in most cases such a simple predictor suffices. In one application alone, we find a Markov predictor (as in [10]) can provide better estimates than the last value predictor. The last value, or the history of previous values (if a more sophisticated scheme is needed), of active and idle times can be obtained by reading the processor clock at points (A) and (B) in the program, and the compiler emits code for doing so.

5. Experimental Setup

We implemented our approach using Simics [14] and performed extensive experiments with several parallel applications from the SpecOMP suite [1]. Simics is a platform for full system simulation that can run actual application code and completely unmodified kernel and driver code. We modified the simulator by enhancing it with accurate cache timing models and then captured the time differences between processors in arriving at barrier (synchronization) points as explained in the previous section.

The chip multiprocessor under consideration in this paper is of the shared multiprocessor kind, where a certain number of CPUs share the memory address space. We assume that each CPU has its own private L1 instruction and data caches that it can access without going across a shared interconnect. Several proposed CMP designs from industry and academia already use such private L1-based configurations [2]. We keep the subsequent discussion simple by using a shared bus as the interconnect (though one could use higher bandwidth interconnects as well). We also use the MOESI [5] protocol (the choice is orthogonal to the focus of this paper) to keep the caches coherent across the CPUs. We also assume that there is a shared L2 data cache. Finally, we assume the existence of hardware mechanisms for implementing DVS for each core independently.

The default simulation parameters used in our experiments are shown in Table 1. Unless stated otherwise, all experiments use these parameters. We performed experiments with two, four, and eight discrete voltage/frequency levels and the values used are listed in Table 2. When we make experiments with two frequencies/voltages, we use 1000/1.30 and 533/0.95; and, when we make experiments with four voltage levels, we use 1000/1.30, 800/1.15, 533/0.95, and 300/0.80. These voltage/frequency values are similar to those employed by Transmeta’s Crusoe [4]. The important characteristics of our benchmarks are given in Table 3. The second column gives a brief description of each benchmark and the third column gives the number of parallel loop nests in each benchmark. The next column lists the important loop nests, i.e., the loop nests that take bulk of execution time. Each loop is named after its host function and the line number of source code. For example, `applu.rhs.34` refers to the loop located in function `rhs` at line 34 of application `applu`. Finally, the last column gives the number of instructions (in millions) simulated for each benchmark.

In this section, unless stated otherwise, the term “energy” is used for energy consumption in the data-paths of the processors. Voltage/frequency scaling is applied only to the data-path of the processors to save energy; the caches

Parameter	Default Value
Number of Processors	4 and 8
L1 Size	8KB
L1 Line Size	32 bytes
L1 Associativity	4-way
L1 Latency	1 cycle
L2 Size (Total Capacity)	2MB
L2 Associativity	4-way
L2 Line Size	64 bytes
L2 Latency	10 cycles
Memory Access Latency	120 cycles
Bus Arbitration Delay	1 cycles
Replacement Policy	Strict LRU

Table 1. Base simulation parameters used in our experiments.

and system interconnect components operate with the highest available voltage/frequency. This is to ensure that we do not affect the snoop performance of the caches and interconnects, even when the data-path is operating at a lower frequency. The term “execution cycles” is used to denote the total number of cycles taken by the application when the number of instructions shown in the last column of Table 3 is simulated. In our results, we focus on two metrics: (i) *normalized energy consumption*, and (ii) *percentage increase in execution time*, and quantify these for the proposed optimization with respect to a system without these optimizations.

Freq (MHz)	1000	900	800	677	533	433	300	150
Voltage (V)	1.30	1.25	1.15	1.05	0.95	0.875	0.80	0.75

Table 2. Frequencies and voltages used in the evaluation.

6. Results

We present our experimental analysis in several parts. First, we document results that illustrate the time variance between processors in getting to the barrier points. Following this, we discuss whether it is possible to predict the idle time a processor waits at a given barrier point. After that, we present our energy saving and performance results to give a picture of how our approach behaves. To explain our findings, we also give data showing how our approach utilizes available frequency levels. Finally, we present a comparison of our approach to the one proposed in [13].

6.1. Disparity at Barrier Arrival Times

Figure 3 gives CDF (cumulative distribution function) graphs that show the time difference (in terms of cycles) between the processors arriving at barrier points for the four processor case. Each graph corresponds to a single loop (with some representative loops of each application being shown) and plots four curves (one per processor). For each

graph, the curve marked CPU0 corresponds to the processor which arrives last at the barrier point (i.e. its idle time is zero). We first describe its curve as it is interpreted differently from the remaining three curves. An (x,y) point in the CPU0 curve indicates that fraction y of the time (i.e., fraction y of total arrivals to this loop) the CPU0 spends x cycles or less executing the loop (the active time is the same as the total loop execution time in this case since there is no idling) before coming to the barrier. The remaining curves, on the other hand, reflect the time duration the corresponding processor waits at the barrier point. More specifically, a point (x,y) says that, for a fraction y of the time, the processor in question waits x cycles or less at the barrier point. Therefore, if the curves for CPU1 through CPU3 are close to the CPU0 curve, this indicates larger idle times for CPU1 through CPU3 (thus, more opportunity for our approach to save energy). Looking at these figures, we see that different loops (even those that belong to the same application code) exhibit different behaviors. For example, while in `mgrid.zero3.15`, there is not much opportunity to save power, `applu.rhs.34` indicates that we can save substantially through DVS. In general, we find that there are several opportunities brought by barrier arrival time disparities that we can benefit from.

Still, even a large disparity between processor arrival times may not necessarily indicate that we can predict the idleness. In order to exploit this idleness through DVS, we need to be able to predict it. The next subsection discusses this point in detail.

6.2. Idle Time Predictability

As noted earlier, we can track the time that a processor arrives at the barrier and the time it leaves the barrier. The idleness at the barrier can be calculated using the difference between these two (discounting costs for the implementation of the barrier itself), and the active times can be calculated by taking the difference between the current arrival at the barrier and the previous departure from the barrier. In order to scale the voltage, we need to predict the idle and active times for the next iteration (to set the frequency to $\frac{a}{a+s} \times f_{max}$) upon leaving the barrier. Our default predictor is a history-based one, which predicts that the next idle and active times (for the next iteration) will be the same as that for the current iteration. In the rest of this paper, this predictor is referred to as the *last-value predictor*. The curves in Figure 4 show, for each benchmark code, the accuracy of the distribution of predictions of a , summed over all processors and all loops in the benchmark. The x-axis shows the difference between the predicted loop execution a_{pred} and the real a_{actual} in terms of $\frac{a_{pred}-a_{actual}}{a_{actual}}$.

Consequently, a negative value corresponds to underestimation, whereas a positive value implies overestimation. The y-axis gives the percentage of time a particular accuracy (on the x-axis) is achieved. Obviously, a large concentration around 0 on the x-axis indicates good predictability. We see from these graphs that, except for `mgrid`, the predictability of the remaining four benchmarks is quite good. In `mgrid`, we mostly underestimate the active times, and as will be explained shortly this forces our approach to operate with lower frequencies/voltages that can hurt performance. As for the other benchmark codes, we can expect energy savings through the last-value predictor-based DVS

Figure 3. Stall times and execution times for some important parallel loops in our applications. An (x,y) point in the CPU0 curve indicates that fraction y of the time the last processor to arrive at the barrier spends x cycles or less executing the loop before coming to the barrier. For other CPUs, a point (x,y) says that for a fraction y of the time, the processor in question waits x cycles or less at the barrier point.

Benchmark	Description	Important Loops	Number of Parallel Loops	Simulated Inst. (in millions)
applu	solution of five coupled nonlinear PDE's	rhs.34, rhs.56, rhs.178	15	1,770,699
apsi	Lake environment model operates on a 112x112x112 area array of data and iterates over 70 time steps	all except advc.1330, advt.1461, advu.1675, advv.1847	21	93,630
galgel	numerical analysis of oscillatory instability of convection	dswap.4222, dgemm.435, dger.5067, dtrsm.8220	21	111,324
mgrid	demonstrates the capabilities of a simple multi-grid solver in computing a three dimensional potential field.	zero3.15, comm3.176	3	125,887
swim	weather prediction operates on a 1335x1335 area array of data and iterates over 512 time steps	shalow.116, calc1.276, calc2.331, calc3z.381	8	39,272

Table 3. The benchmark codes used in this study.

strategy without impacting performance significantly.

Figure 4. Predictability of active times. The y-axis histograms the percentage occurrence of the normalized estimation error (i.e. x-axis is $\frac{a_{pred} - a_{actual}}{a_{actual}}$).

6.3. Energy and Performance Results

We present energy and performance data for two idleness predictors. The first one is the last-value predictor described above. The second one is an *oracle predictor*, which performs perfect prediction for each processor at each loop arrival. Clearly, the oracle is not implementable; the reason that we give its results here is to check how much inaccuracy the imperfect prediction brings and what its consequences are from both performance and energy angles.

For each application in Figure 5, the bars marked 2Levels, 4Levels and 8Levels correspond to two, four, and eight voltage/frequency levels using the oracle predictor with four and eight processors. One can see from these results that, except swim, our benchmarks take advantage of increased

number of voltage/frequency levels. The reason that swim does not generate good results (savings) is because it has little opportunity for scaling voltage/frequency, as illustrated for some of its loops in Figure 3. When we look at the eight processor results with the oracle predictor, we see similar trends. In fact, in most cases, the results with 8 processors show better energy savings than those with 4 processors. The main reason is that the work imbalance increases with a larger number of processors, generally resulting in longer idleness as can be seen from the average idle times for the 4 and 8 processor configurations for several loops in Table 4. Consequently, the importance of exploiting idleness grows with increasing number of processors.

Figure 5. Overall energy consumption with Oracle Predictor.

We now focus on the last-value predictor and present its energy results in Figure 6 for the four and eight processor cases. We observe similar results as for the oracle predictor suggesting that our predictor is doing a good job. We note that the last-value predictor results show better savings for mgrid with 4 processors. The main reason for this because we underestimate the active time in this application (see Figure 4), causing more operation at lower frequencies than with the oracle predictor.

To further gain insight on these results, we give the frequency distribution for the last value predictor and oracle predictors for two of our benchmarks under 8 voltage levels: mgrid and galgel in Figure 7. A bar in these graphs

Loop Name	4 Procs	8 Procs
applu.rhs.34	31.4%	29.7%
applu.rhs.56	25.1%	29.0%
applu.rhs.178	21.5%	37.2%
galgel.dswap.4222	0.551%	47.1%
galgel.dger.5067	59.3%	64.5%
galgel.dtrsm.8220	2.11%	8.04%
mgrid.zero3.15	33.2%	40.4%
mgrid.comm3.176	33.2%	40.4%
swim.shallow.116	1.21%	0.96%
swim.calc1.276	1.47%	2.13%
swim.calc2.331	2.27%	1.33%
swim.calc3z.381	2.61%	2.67%

Table 4. Idle times at barrier averaged over all processors and iterations for selected loops from our benchmarks.

Figure 6. Overall energy consumption with the Last Value Predictor.

gives the percentage of time the corresponding frequency level (and the associated voltage) is exercised. We see that the frequency distributions of the last-value predictor and the oracle predictor in galgel are very similar to each other, which explains the similar energy trends as well. In mgrid, the last-value predictor (which is not very accurate) puts the processors at lower frequencies, which gives power savings at the expense of performance loss.

It must be noted that the oracle predictor does not have any performance penalty since it selects the most suitable voltage/frequency for the idleness in question. However, the last-value predictor can lead to performance degradation when it underestimates the active times. The reason is that underestimation of a leads to selection of lower voltage/frequency than the optimum one, and this can increase the execution time of the parallel loop under consideration. The percentage increases in execution time of our benchmarks are given in Figure 8. We see that the performance degradation in most benchmarks is less than 10% for both four processor and eight processor cases. In particular, the performance penalty is really low in galgel since our predictor is very accurate (Figure 4). The reason for the negligible

Figure 7. Percentage of the execution time spent in a particular frequency (8 processors).

degradation in swim is that we do not have much opportunity in this benchmark for voltage/frequency scaling at all. The largest degradation occurs with mgrid as a result of underestimates of a as explained earlier, and we look to address this problem in the next subsection.

6.4. Using a More Powerful Predictor

We further studied the behavior of mgrid and found that the behavior exhibited by its idleness pattern cannot be captured adequately by the last-value predictor. However, when we examined the time series patterns of active and idle times, we found a pattern that repeats itself, suggesting that a Markov predictor could be very useful in this case. We use a simple Markov predictor (described in [10]) that uses a small table indexed by the last value to give the next estimate. The normalized energy consumptions for mgrid with the Markov predictor are given in Figure 9. For ease of comparison, we also reproduce the energy results with the last-value predictor. We see that, for both 4 and 8 processors, the Markov predictor leads to less energy savings than the last value predictor. However, when we look at the performance overhead results (shown in Figure 10), we see that the performance penalty is considerably lower. Note that this simple Markov predictor also subsumes a last-value prediction mechanism, and can be a more generic way of implementing active/idle time prediction. When we compare the running frequencies of mgrid in Figure 7 and in Figure 11, it is clear that the Markov predictor is able to reduce the under-

Figure 8. Percentage Increase in execution time with Last-Value Predictor.

estimates of active times, thus putting the processors more frequently at the higher frequencies.

Figure 9. Normalized energy consumption for mgrid: Last-Value Predictor vs Markov Predictor. (xp means x processors are used.)

6.5. Comparison with the Thrifty Barrier

As observed earlier, there are two ways of exploiting the idleness at barriers for power savings. While our approach uses voltage scaling to reduce the power during periods of active execution, a related study [13] has used the idleness to transition the processor to a low power mode (called the *thrifty barrier* approach). In order to compare our savings with those of [13], we show in Table 5 the energy savings of our approach (using the oracle and our prediction mechanisms) along the savings obtained with the thrifty barrier. In the interest of fairness, we use the same oracle and the same prediction mechanisms to estimate the idleness in both the approaches.

As can be observed, whether it be perfect knowledge, or with the predicted idleness, our approach provides better energy savings than the thrifty barrier in nearly all cases

Figure 10. Percentage increase in execution time for mgrid: Last-Value Predictor vs. Markov Predictor.

Figure 11. Percentage of the execution time spent in a particular frequency for mgrid with the Markov Predictor (8 processors).

(except swim where there is really no scope for power savings anyway). The main reason for those results is that the idleness durations may not be large enough to accommodate the cost of turning down the processor and bringing it back up fully for the thrifty barrier. On the other hand, our mechanism can still scale down the frequency/voltage even while idleness is not very significant. Our mechanism is able to provide these energy savings, while being comparable in performance to the thrifty barrier.

7. Concluding Remarks

In this paper, we have proposed a novel approach to optimizing the power consumption of data-paths in a chip-multiprocessor by using high-level software information for scaling frequencies/voltages. The barrier construct, which is widely used in parallel programming to synchronize processors, provides a convenient mechanism for tracking work imbalances among the processors and their resulting idleness. By predicting this idleness, it is possible to scale the frequencies for the duration of the work to be done before getting to the barrier the next time, so that we obtain

Benchmark		4P					8P				
		applu	apsi	galgel	mgrid	swim	applu	apsi	galgel	mgrid	swim
Energy Consumption	TB-Oracle	80.7%	84.2%	99.7%	94.2%	98.3%	66.4%	73.4%	70.2%	71.1%	98.4%
	Ours-Oracle	70.5%	74.6%	76.1%	77.9%	99.4%	55.5%	63.9%	66.7%	68.4%	99.1%
	TB-LVP	73.7%	80.8%	99.3%	79.0%	98.2%	67.8%	72.2%	68.7%	61.0%	98.5%
	Ours-LVP	67.5%	74.1%	77.1%	57.5%	99.1%	57.0%	62.6%	64.7%	56.8%	99.4%
Performance	TB-Time Increase	8.31%	6.38%	0.07%	9.8%	1.40%	7.22%	3.97%	0.31%	9.07%	1.16%
	Ours-Time Increase	10.48%	6.95%	3.04%	1.6%	0.36%	7.73%	4.61%	4.53%	1.5%	0.18%

Table 5. Normalized energy consumption and increase in execution time of our mechanism vs. Thrifty Barrier (TB) using the same Oracle and Last-Value Predictors (LVP) for both.

power savings and reduce/eliminate the idleness. We show that simple predictors can do an effective job of estimating this idleness for different SpecOMP applications. Using detailed complete system (including OS and OpenMP library) simulation of application code, we demonstrate that our approach can provide considerable power savings for these applications. We also show that this strategy provides higher power savings than a previously proposed one that simply switches the executing core to a low power mode during the idleness at the barrier.

References

- [1] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *Proceedings of the International Workshop on OpenMP Applications and Tools*, pages 1–10, 2001.
- [2] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzky, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *Proceedings of ISCA'00*, pages 282–293, 2000.
- [3] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-Power CMOS Digital Design. *JSSC*, 27(4):473–484, 1992.
- [4] *Crusoe Processor Model TM5700/TM5900 Data Book*, 2004. http://www.transmeta.com/crusoe_docs/tm5900_databook_040204.pdf.
- [5] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.
- [6] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.
- [7] D. E. Duarte, N. Vijaykrishnan, and M. J. Irwin. A clock power model to evaluate impact of architectural and technology optimizations. *IEEE Transactions on VLSI Systems*, 10(6):676–686, 2002.
- [8] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Oberg, P. Ellervee, and D. Lundqvist. Lowering power consumption in clock by using globally asynchronous locally synchronous design style. In *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 873–878. ACM Press, 1999.
- [9] A. Iyer and D. Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 379–386. ACM Press, 2002.
- [10] D. Joseph and D. Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.
- [11] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.*, 48(9):866–880, 1999.
- [12] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, page 81. IEEE Computer Society, 2003.
- [13] J. Li, J. Martinez, and M. Huang. The thrifty barrier: Energy-efficient synchronization in shared-memory multiprocessors. In *Proceedings of HPCA'04*, pages 14–23, 2004.
- [14] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.
- [15] MAJC5200. <http://www.sun.com/microelectronics/MAJC/5200wp.html>.
- [16] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *Proceedings of ISCA'96*, pages 67–77, 1996.
- [17] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *Proceedings of ASPLOS'96*, pages 2–11, 1996.
- [18] *Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor*, 2004. order #:301170-001.
- [19] R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The energy efficiency of cmp vs. smt for multimedia workloads. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 196–206. ACM Press, 2004.
- [20] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 356–367. IEEE Computer Society Press, 2002.