Appears in Intl. Symp. on High-Performance Computer Architecture (HPCA), Madrid, Spain, Feb. 2004

The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors

Jian Li and José F. Martínez
Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA
{li,martinez}@csl.cornell.edu

Michael C. Huang
Electrical and Computer Engineering
University of Rochester
Rochester, NY 14627 USA
michael.huang@rochester.edu

ABSTRACT

Much research has been devoted to making microprocessors energy-efficient. However, little attention has been paid to multiprocessor environments where, due to the co-operative nature of the computation, the most energy-efficient execution in each processor may not translate into the most energy-efficient overall execution.

We present the *thrifty barrier*, a hardware-software approach to saving energy in parallel applications that exhibit barrier synchronization imbalance. Threads that arrive early to a thrifty barrier pick among existing low-power processor sleep states based on predicted barrier stall time and other factors. We leverage the coherence protocol and propose small hardware extensions to achieve timely wake-up of these dormant threads, maximizing energy savings while minimizing the impact on performance.

1 INTRODUCTION

Reducing energy consumption has become an important design goal for high-performance microprocessors and computer systems based on them. High energy consumption not only limits battery life in portable devices, but also complicates heat dissipation support and electricity supply in large-scale computing facilities. Past research on power-aware computer systems mostly focuses on uniprocessor systems. While many energy-efficient techniques have been developed for uniprocessors, there are issues unique to multiprocessors that warrant investigation. In a parallel workload, the overall performance depends on all the threads; however, at any point in time, the critical path may depend on only a few threads. In that case, slowing down threads not on the critical path to save energy may not affect the overall performance at all. Conversely, slowing down threads in the critical path will negatively impact performance, and the local energy savings may be easily negated by the extra energy waste on other processors due to longer execution time.

In this paper, we look at a source of energy waste that is unique to parallel systems: the energy waste in barrier synchronization. In barrier-synchronized parallel codes, threads often spin-wait at barriers for all other threads before moving to a different phase of the computation. The time a thread spends spinning at the barrier is, to a large extent, determined by the speed of *another* thread—the last one to arrive at the barrier. Traditional low-power techniques for uniprocessors consider the energy-performance trade-offs of the processor in isolation, and are therefore unfit for this multithreaded scenario.

We propose the *thrifty barrier*, which reduces the energy waste in barrier spinloops by estimating the wait time and forcing the processor into an appropriate low-power sleep state. Many commercial processors offer various such sleep states, each providing different levels of energy savings and requiring correspondingly different transition times. The thrifty barrier predicts the stall time based on past history, and depending on the predicted slack, the processor may transition into one of these low-power sleep states. In anticipation of the barrier release, the thrifty barrier also strives to wake up the processor just in time to avoid potential performance degradation. The goal is to provide maximum energy savings while maintaining the same level of performance.

One key challenge in the design of the thrifty barrier, and a primary contribution of this paper, is to accurately estimate barrier stall time for each barrier instance. An accurate estimation allows judicious selection of the right low-power state to maximize energy savings, and minimizes performance degradation by striving to bring the processor back up in a timely fashion. The thrifty barrier achieves high accuracy by predicting the barrier stall time indirectly through the interval time between consecutive barrier invocations, which is much more predictable.

The thrifty barrier is implemented using a combination of modest software-hardware support, and it can be made transparent to the application by encapsulating the code in a typical barrier macro or directive. The thrifty barrier respects the original barrier semantics, and thus thrifty and conventional barriers may co-exist in the same binary.

We present the thrifty barrier in the context of shared-memory multiprocessors. Nevertheless, the idea is conceptually viable in other environments such as message-passing machines. Overall, our design of the thrifty barrier is simple and effective. For the applications and low-power sleep states studied, significant energy savings can be achieved at the expense of only a small performance degradation.

The rest of the paper is organized as follows: Section 2 presents the overview of the thrifty barrier; Section 3 discusses design details; Section 4 and Section 5 present the experimental setup and evaluation; Section 6 discusses related work.

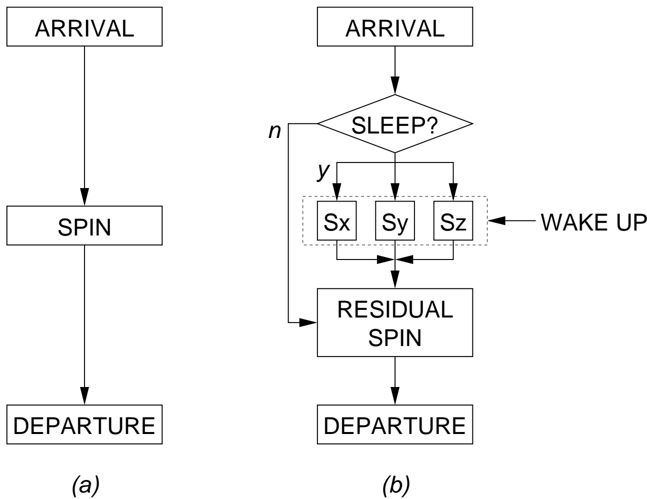

Figure 1: Simplified diagrams of actions taken by a thread that arrives early at a conventional *(a)* and a thrifty *(b)* barrier. As an example, S_x-S_z represent three possible low-power processor sleep states.

2 OVERVIEW

Figure 1*(a)* is a simplified diagram of the actions taken by a thread that arrives early at a conventional barrier. Early threads stop at the barrier and wait for all slower threads to arrive. To do so, they typically spin-wait on a barrier flag until the last thread to arrive flips its value. Once flipped, all threads may continue execution past the barrier.

This barrier spinloop is highly inefficient by definition: of all the iterations that are executed by the CPU, only the last one is productive, when the thread learns that the flag has been flipped. As a consequence, nearly all the spin energy is wasted in unproductive computation. This energy waste is largely proportional to the barrier imbalance.

Many modern processors feature one or more low-power sleep states, often following the *Advanced Configuration and Power Interface (ACPI)* specification [4]. This is true not only for mobile and embedded processors, but also for high-end server products, such as the Intel Xeon [15]. Low-power states offer different power savings and correspondingly different transition times in and out of them. The CPU does not execute code while in these low-power states. In this paper, we propose to leverage this support to improve the energy efficiency of imbalanced parallel codes.

In a thrifty barrier (Figure 1*(b)*), an early arriving thread tries to bring its CPU into one among several possible low-power states instead of spinning. The hope is to save the energy that would be wasted in the spinloop. Once the last thread arrives, all dormant CPUs are woken up, and all threads are allowed to proceed past the barrier.

Because transitioning to and from a low-power sleep state requires a non-negligible amount of time (on the order of tens of microseconds [14, 16]), it is important that this transition does not add to the execution time significantly. For example, if an application has many barriers that present little stall time, driving the CPU repetitively into a sleep state whose transition time exceeds the barrier stall time could add up to a significant slowdown. For this reason, an early arriving thread must first estimate whether enough stall time lies ahead to warrant transition into a low-power sleep state. If several sleep states are available, the estimation is also used to select the best fit. As an example, Figure 1*(b)* shows three sleep states S_x-S_z. Section 3.1 discusses the details.

Even if the transition time is small compared to the barrier imbalance, the dormant CPUs must wake up in a timely manner for optimum results. If the CPU wakes up too early (i.e., not all threads have arrived to the barrier), the corresponding thread ends up spinning for the remainder of the barrier (*Residual Spin* in Figure 1*(b)*), wasting energy unnecessarily. Conversely, a thread that wakes up late (i.e., the barrier has long been completed) may affect the execution time adversely if, for example, the thread is or becomes the last thread to arrive at the next barrier. In general, the issue of early vs. late wake-up presents an important energy-performance trade-off. Section 3.3 covers this issue in detail.

The thrifty barrier is supported by a modest combination of software and hardware. We augment the barrier with simple prediction code to decide whether to bring the CPU into a low-power sleep state, and if so, which among the possible sleep states yields maximum energy savings with little or no performance impact. Also, we minimally extend the on-chip cache controller and leverage the existing cache coherence protocol to provide a timely wake-up mechanism.

For the sake of simplicity, we describe our mechanism in the context of a dedicated multiprocessor environment with one CPU per thread. Section 3.4 discusses other scenarios.

3 DESIGN

In this section we describe our technique and the required support in detail. To facilitate understanding of the trade-offs involved, we describe sleep and wake-up mechanisms in increasing complexity.

Figure 2 is an example of a conventional barrier with sense reversal [10]. A thread arriving to such a barrier first checks in by incrementing the thread count (statement S1). If the count remains lower than the total number of threads, the thread knows it has arrived early. Then, it proceeds to spin (i.e., continuously read) on a shared flag, waiting for the other threads to catch up (statement S2). When the last thread arrives and checks in, the count equals the total number of threads. At that point, this last thread toggles the flag value, signaling the release of the barrier, and proceeds past the barrier. As the spinning threads detect the value change, they also proceed past the barrier.

3.1 Sleep

In general, a thread arriving to a thrifty barrier first updates the thread count as before. If early, the thread executes code to bring the CPU to a low-power sleep state. We comment on several possible design choices.

In its simplest form, the CPU can support a single low-power sleep state, and make threads go to sleep every time they arrive early to a barrier. In many existing CPUs, simply executing a Halt instruction brings the CPU to a low-power sleep state. Therefore, upon reaching a barrier, the thread increases the thread count by one and, if early, it puts the CPU to sleep.

```
local_f = !local_f;
lock(c);
count++; // increment count (S1)
if(count==total) { // last one
  count = 0; // reset count
  f = local_f; // toggle
  unlock(c);
}
else { // not last one
  unlock(c);
  while(f != local_f); // spin (S2)
}
```

Figure 2: Example of conventional barrier code with sense reversal.

Conditional Sleep

As hinted in Section 2, one problem with the above approach is that transitioning to and from a low-power sleep state involves some latency. This latency is typically in the order of tens of microseconds, largely attributable to the period of stabilization in the PLL and in other parts required by frequency and/or voltage changes. As a result, in the context of the thrifty barrier, it is possible that this latency exceeds the stall time that the early-arriving thread faces. If this occurs, departure from the barrier may be delayed for that thread. This may in turn affect the execution time of the application as a whole, especially if the thread is or becomes the critical thread for the next barrier—the last one to arrive.

To address this issue, we propose that the thread put the CPU to sleep only when enough stall time lies ahead. Naturally, this stall time is not known a priori. We propose to utilize history-based prediction of the barrier stall time by each thread; we address the specifics of this support later in Section 3.2.

Therefore, if conditional sleep is supported, an early arriving thread makes a prediction of the barrier stall time ahead. If enough time lies ahead, the thread brings its CPU into the low-power state as before. If not, the thread starts spinning on the barrier flag—the traditional way. We envision this decision to be encapsulated in a *sleep( )* library call that is linked at run-time, and abstracts away the specific timing characteristics of the low-power sleep state from the application. The call either returns immediately (too little stall time) or it enters the low-power sleep state (enough stall time).

Multiple States

Once there is support for barrier stall time prediction, we need not restrict our design to a single low-power sleep state. Often times, processors feature multiple sleep states to choose from. The energy savings typically depend on what parts of the processor are disabled (typically by clock gating), and whether the supply voltage is lowered. In general, a *deeper* sleep state saves more energy, but it takes more time to transition in and out of it. Some deeper sleep states may also gate the processor caches in order to save energy. Even though data are preserved (the supply voltage is not interrupted), the processor may need to flush the dirty shared data if the cache will not respond to protocol interventions. Note that the cache controller can immediately acknowledge invalidations to clean data, and delay internal action until the cache is accessible. If the cache controller ever runs out of buffer entries, the processor can always be woken up.

The support for multiple sleep states can be encapsulated in the *sleep( )* library call mentioned. This time, the function accesses a table to determine the deepest sleep state that can be used within the estimated stall time. This table can be filled at the time the application starts, or by the OS at startup, or can even be hardcoded in the library itself. In any case, the selection is transparent to the application. The library procedure scans the table for a best fit, and brings the CPU to that low-power sleep state, or returns immediately if not enough sleep time lies ahead (as before).

3.2 Barrier Stall Time Estimation

In order to choose the optimum sleep state, as explained in Section 3.1, early-arriving threads must make an estimation of the barrier stall time ahead of them. This information is also useful later at wake-up (Section 3.3). In this section we propose a history-based prediction mechanism to estimate this barrier stall time.

We define *barrier stall time ($BST_{t,b}$)* as the time that a given thread t spends waiting at a given barrier instance b. In a conventional barrier, this is the time the thread invests spinning at the barrier. For each thread t, the ratio of cumulative barrier stall time $\text{BST}_t = \sum_{b=1}^{B} \text{BST}_{t,b}$ to overall execution time is a rough indicator of the potential energy savings resulting from putting that CPU to sleep during such periods.

In a parallel application, the variability of barrier stall time can be considerable across barrier invocations in the code. However, we empirically observed that, in the applications used in our study (Section 4), there was significantly less variability across instances of the same barrier invocation in the code. This can be intuitively justified by the fact that the "computation phases" surrounding a particular barrier tend to perform the same type of computation every time they are executed. Very often parallel programs are written in SPMD style (single program, multiple data). In these, the computation phases can be easily identified using the program counter (PC) of the barrier at their end. Thus, we can easily achieve indexed prediction by using the PC of the barrier at each point. In the more general case, it would be necessary to identify such computation phases by other means. This could be accomplished by allocating separate barrier structures in memory for each phase, and using their memory address to index the predictor. To simplify our discussion, we assume a SPMD programming style from now on.

Still, a nontrivial variability in PC-indexed barrier stall time remains in many cases. Even if the computation that precedes the barrier repeats itself, it is not uncommon for computation and data access costs to shift among threads across instances of the code. Because the barrier stall time is not only barrier- but also thread-dependent, this behavior makes direct estimation hard. In our effort to estimate barrier stall time accurately, we would like our prediction to be more insensitive to such changes.

We find this predictability in the *barrier interval time (BIT)*. We define the barrier interval time BIT_b as the time between the release of two consecutive barrier instances $b-1$ and b (zero denotes the beginning of the program). The key to the predictability of barrier interval time is that it is thread-

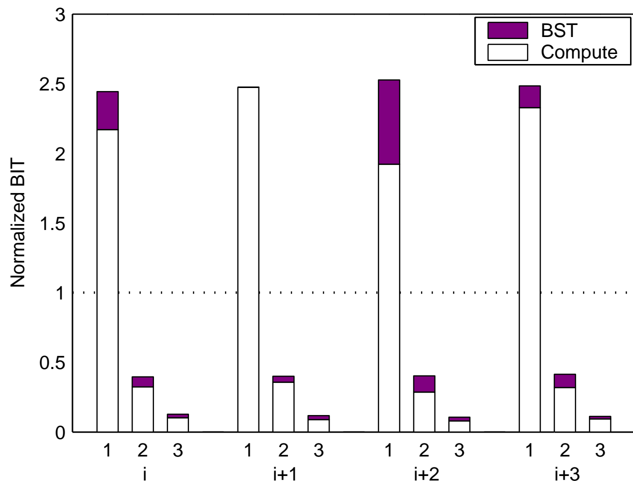

Figure 3: Variability of BIT and BST for three important barriers (labeled 1-3) that execute in the main loop of SPLASH-2's FMM application [28], as observed by a randomly picked thread (the same one in all twelve barrier instances). Four consecutive iterations are shown. Values are normalized to the average BIT across all barrier instances.

independent. Moreover, it is observable by any participating thread as the sum of its compute time plus barrier stall time, measured from the release of the last barrier instance. For the applications studied, we found that PC-indexed barrier interval time is by far the most stable of all the discussed metrics.

One possible concern regarding barrier interval time estimation is that it does not allow us to directly distinguish barrier stall time from compute time. Fortunately, our sleep mechanism is invoked at the time the thread is about to stall at the barrier, and thus the compute time can be determined (see Section 3.2.1). Thus, we can estimate barrier stall time indirectly by subtracting the compute time from the estimated barrier interval time in each case.

Figure 3 is a representative example of the above discussion. The plot shows barrier interval times (*BIT*) for three important barriers (labeled 1-3) that execute in the main loop of SPLASH-2's FMM application [28]. The results were obtained in a simulation using the shared-memory multiprocessor model described in Section 4.1. We show four consecutive iterations of the loop, and group bars accordingly. Each bar is broken down into compute time (*Compute*) and barrier stall time (*BST*), as observed by a randomly picked thread (the same one in all twelve barrier instances). All values are normalized to the average BIT across all barrier instances.

We see that both BIT and BST vary rather significantly across barriers. Much less variability is observed across invocations of the same barrier, which suggests the use of PC indexing for prediction. Nevertheless, important differences across BST values still remain. It is in BIT, a thread-independent metric, that we obtain a significantly more predictable behavior. Thus, by using BIT prediction, and subtracting the thread's compute time from it (known at the time it hits the barrier), the thread's BST can be derived.

We experimented with different types of predictors, and were pleased to find that, for most of the applications that we studied, simple last-value prediction of PC-indexed barrier interval time was very accurate. Therefore, in our proposal, we predict the barrier interval time using the value measured in the last occurrence of the same barrier. Later, once the actual BIT is known, we update the prediction by storing the new value in the appropriate entry. In the next section we discuss how this information is determined and maintained.

3.2.1 Managing Timing Information

As indicated before, threads that arrive early at a barrier must estimate the stall time ahead of them by predicting the barrier interval time, and subtracting from it their compute time for that interval. Because the barrier is released by the last thread to arrive, the exact release and interval times are not directly available to other threads. This is especially true if threads overpredict their stall time and sleep beyond the barrier release (see late wake-up in Section 3.3). One could think of communicating the barrier release time to all threads; however, this would imply the existence of a global clock, which we do not assume here.

To address this issue, we use a combination of global knowledge of past barrier interval times, and local knowledge of past barrier release times. We do reasonably assume that (1) all processors operate at the same nominal clock frequency (i.e., base cycle counts are meaningful system-wide), and (2) the time period between the flipping of the barrier flag and the realization of this fact by spinning threads is much smaller than the barrier interval time itself. Naturally, we also require that the local clock be such that processors can measure time intervals accurately, even if the CPU is put to sleep between readings.

We describe the mechanism inductively; Figure 4 supports this discussion. In the general case, assume that threads are advancing toward barrier instance b. We assume that we have (1) a shared location to (eventually) store the barrier interval time leading to barrier instance b, BIT_b, and (2) a local timestamp of the release of barrier instance $b-1$ for each thread t, $\mathrm{BRTS}_{t,b-1}$. In PC-indexed last-value prediction, (1) is simply the table entry for this barrier.

When thread t arrives early at barrier instance b, it obtains a prediction of BIT_b, and adds this value to $\mathrm{BRTS}_{t,b-1}$. (In PC-indexed last-value prediction, the BIT previously stored in the table entry for this barrier is used as the prediction.) The result is an estimation of the upcoming wake-up time. Furthermore, by subtracting the current (local) time from the estimated wake-up time, the thread can derive an estimation of the stall time ahead of the thread, $\mathrm{BST}_{t,b}$. Using this information, the thread may decide to force its CPU into a sleep state.

When the last thread t' arrives at barrier instance b, it calculates the actual BIT_b by subtracting $\mathrm{BRTS}_{t',b-1}$ (local timestamp) from the current (local) time. Then, the thread updates the shared BIT variable (and the predictor) and releases the barrier. Finally, each thread t adds (once awake) the newly available BIT_b to its $\mathrm{BRTS}_{t,b-1}$.[1] The result represents the (local) release timestamp of barrier b, $\mathrm{BRTS}_{t,b}$. At this point we know (1) BIT_b, stored in the shared BIT variable, and (2) local timestamps for the release of barrier instance b in each

[1] In a relaxed memory consistency implementation, availability of BIT_b to all threads is ensured by placing a write fence between the BIT update and the flipping of the barrier flag.

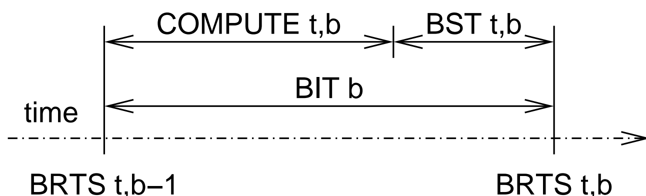

Figure 4: Illustration of timing information for two consecutive barrier instances. BIT, BRTS, and BST stand for barrier interval time, barrier release timestamp, and barrier stall time, respectively.

thread t, $\text{BRTS}_{t,b}$. Thus, induction holds for the general case.

In order for the mechanism to work from the start, interval time and release timestamp variables are all initially set to zero, and the first barrier instance is handled conventionally (non-thrifty) as warm-up. Early arriving threads spin on the barrier flag and, upon detecting the flipping by the last thread, all threads record their local release timestamp.

Overall, we see that threads can estimate barrier intervals based on available past values and derive stall times, all without the need for a global clock.

3.3 Wake-up

Once the CPU is put to sleep, an external wake-up signal must be produced in a timely manner so that the thread can resume execution as the barrier is released. In this section we address different ways this signal can be generated, and what the consequences are on the timeliness of the wake-up, and on the overall energy and performance benefits.

3.3.1 External Wake-up

One simple way of generating the wake-up signal is to leverage the coherence actions that take place when the last thread flips the barrier flag. In a conventional barrier, an early arriving thread that spins on the barrier flag typically brings a (shared) copy to its cache in the first iteration (cache miss), and then hits in the cache in every subsequent read. As the last thread tries to flip (i.e., write to) the barrier flag, the coherence protocol sends invalidations to all sharers. Threads spinning on that flag then miss in their caches and retrieve a fresh copy of the flag, which now reads that access has been granted. In the thrifty barrier, we can use this *external* invalidation message as our wake-up signal.

Though conceptually easy, a careful design of this mechanism is in order for a number of reasons. An early thread whose CPU is put to sleep must be notified when the barrier flag is flipped by the last thread to arrive. However, some low-power sleep states may not respond to coherence actions on cached data, in particular the barrier flag. We propose to add some small control logic in the on-chip cache controller. The added logic is programmable from within our *sleep( )* library call, and therefore transparent to the application. When supplied with the flag address and the target value, it reads in the flag and *(a)* prevents the CPU from sleeping if the flag has already been flipped, or *(b)* allows the CPU to sleep if the flag is not flipped, but promptly issues a wake-up signal if an invalidation to the flag is later received. Similar extensions have been proposed elsewhere for different purposes [21, 23]. Of course, we assume that the cache controller is not disabled, even if the cache itself may be.

Finally, a *residual* spinloop must be present after the sleep call to verify that the flag has indeed been flipped, lest we fall prey of some unfortunate (but correct) type of exclusive prefetch by another thread. Should this *false wake-up* take place, the thread is left spinning on the flag for the duration of the barrier. The proposed solution to this exceedingly rare problem is suboptimal but correct, and it is much simpler than the alternative of re-assessing whether to put the CPU back to sleep.

One potential problem with external wake-up is that, with the possible exception of a false wake-up, it *guarantees* a late wake-up behavior. Indeed, quiescent CPUs are sent the wake-up signal *as the barrier is released*. Consequently, the transition latency to exit the low-power sleep state comes fully into the critical path of the thread. This happens whenever a thread puts its CPU to sleep, and is thus likely to affect the overall execution time. The impact depends largely on whether the stall time is large enough as to amortize this transition latency.

3.3.2 Hybrid Wake-up

In order to improve wake-up behavior, we can further leverage the prediction support described in Section 3.1. Once an early thread has estimated the stall time ahead and is about to go to sleep, it is possible to program a timer to generate the wake-up signal at expiration. We place this timer in the on-chip cache controller—where the external wake-up support also resides. (Recall that we assume the cache controller is never disabled, even when the CPU enters a sleep state.) Upon invoking the *sleep( )* library routine, and once determined that the CPU is to be put to sleep, the timer begins a countdown starting with the predicted stall time. Once the count reaches zero, the wake-up signal is generated.

Notice that, unlike the case of external wake-up, this mechanism is independent of external actions by other threads. Thus, we call this mechanism *internal* wake-up. Also unlike the case of external wake-up, dormant CPUs in this case are subject to early, as well as late wake-up, depending on the accuracy of the prediction. Recall that an early wake-up will produce some energy waste due to residual spinning, and that a late wake-up may cause performance degradation. Depending on what is more critical to the application, the prediction could be adjusted accordingly.

A potentially important drawback of pure internal wake-up is that, in principle, there is no bound to how late threads may wake up as a result of overprediction. We conducted a number of preliminary experiments using internal-only wake-up, and concluded that the performance of some applications may be penalized significantly by even a few (severe) late wake-ups. Among other effects, severely late wake-ups may create a ripple effect in that they may affect subsequent barrier intervals. Although more sophisticated prediction may alleviate this problem, we prefer to keep our mechanism conceptually simple.

Fortunately, external and internal wake-up mechanisms can be combined into a single *hybrid* wake-up. Not only are the two mechanisms largely independent of each other, they complement each other quite well. Indeed, we can use the external mechanism to bound a possible performance penalty

due to late wake-up, and rely on the internal mechanism to anticipate the wake-up point and initiate the transition out of the low-power sleep state *before* the barrier is released (at the risk of incurring early wake-up). Because both mechanisms are integrated in the same on-chip cache controller, there is no risk of duplication: the first mechanism to trigger the wake-up signal will automatically cancel the other.

3.3.3 Overprediction Threshold

In general, overprediction of the internal wake-up mechanism in large barriers is not a concern, since the external wake-up mechanism forces the CPU out of its sleep state soon enough as to not cause a noticeable performance degradation. Smaller barriers, however, depend more critically on the accuracy of the internal wake-up mechanism. Inevitably, overprediction will occur in some of these barriers, for which the penalty associated with an external wake-up is relatively large. If this happens frequently, the aggregate penalty may amount to a noticeable application slowdown.

We resolve this situation by imposing a very simple *overprediction threshold* mechanism. We require that every thread t that wakes up after sleeping over barrier b annotates its wake-up timestamp in a temporary variable. Once barrier b is released and thread t is awake (this can happen in any order), thread t subtracts its local barrier release timestamp $\text{BRTS}_{t,b}$ (see Section 3.2.1 on how to derive this number) from its recorded wake-up time. A positive value indicates that the thread overpredicted the release time; the actual amount is the penalty incurred. If the penalty incurred is beyond a certain threshold, relative to the barrier interval time BIT_b (Section 3.2.1), future prediction for this thread on this barrier is disabled (a bit is set for this thread on the corresponding predictor entry). The hope is to cut potential future performance losses, even at the expense of suboptimal energy savings. This threshold may be tuned for different environments; empirically, a 10% threshold worked well in our study (Section 5).

More complex solutions with sophisticated predictors and/or confidence estimators are possible, however once more we strive for minimizing complexity. The use of more elaborate prediction and wake-up techniques is the subject of future work.

3.4 Other Design Considerations

3.4.1 Time-Sharing Techniques

In some multiprogrammed environments, threads may yield their CPU to other processes after they spin in the barrier for a while without success. The goal is to improve the CPU utilization. Similarly, in an *overthreaded* application (more threads than available CPUs), threads may hand over the CPU to others at the time they reach a synchronization point. Both techniques may result in reduced energy waste due to spinning, which makes them an interesting option to the problem that the thrifty barrier is trying to solve. However, unless scheduling is carefully planned, time-sharing may hurt performance significantly. For example, in the case of multiprogramming, the barrier may be released but some threads may not be able to resume execution because they lack a CPU. In the case of overthreading, the added computation time of the threads that time-share the CPU may come into the critical path of the application. In contrast, the thrifty barrier tries to achieve lower energy consumption while at the same time striving for maintaining the same level of performance.

Processor	1GHz, 6-issue dynamic
ALU	6 integer, 4 FP
Ld/St	2 units, 32Ld+32St
Branch Pred.	2k-entry 2b sat. counter
Branch Penalty	15 cycles
Memory	CC-NUMA
L1 Cache	1GHz, 16kB, 64B lines, 2-way
L2 Cache	500MHz, 64kB, 64B lines, 8-way
Memory Bus	250MHz, split trans., 16B wide
Main Memory	Interleaved, 60ns row miss
Cache RT: L1, L2	2ns, 12ns
Network	Hypercube, wormhole
Router	250MHz, pipelined
Pin-to-pin Latency	16ns
Endpoint (un)Marshaling	16ns
System size	64 nodes

Table 1: Architecture modeled in the simulations. In the table, RT stands for minimum round-trip latency from the processor.

3.4.2 Interaction with Context Switching and I/O

Context switches in a multiprogrammed system can make some threads unavailable for extended periods of time. Even in a dedicated environment, the operating system may preempt a thread, for example, to handle an I/O request or a page fault. Consequently, barrier intervals in which context switching or I/O takes place may be very unpredictable. Fortunately, this kind of activity can be trivially detected by the last thread to arrive at the barrier, by observing an inordinate increase in the barrier interval time. In this case, the barrier interval time is not updated in the prediction table. Notice that some threads may have forced their CPUs into sleep mode, however this is unlikely to affect performance, since the actual interval time will be significantly longer than the one predicted by the threads. The next time around, threads will once again use the older, shorter barrier interval time as their prediction. This *underprediction threshold* can be tuned to the particular system or application, and need not—and probably should not—be equal to the overprediction threshold discussed in Section 3.3.3.

As for the preempted thread, if it was already asleep at the barrier, the operating system will cancel the flag monitoring at the cache controller. Notice that preemption does not affect the BIT prediction update (which is conducted by the last thread to arrive at the barrier), nor does it hamper the update of the barrier release timestamp for the preempted thread (since it is computed indirectly using the BIT—Section 3.2.1). It also does not compromise correct synchronization, since preempted threads will still spin on the barrier flag once they resume execution (residual spin, Section 3.3.1).

4 EXPERIMENTAL SETUP

4.1 Architecture

We conduct detailed execution-driven simulations of a CC-NUMA multiprocessor model that features release consistency and a coherence protocol along the lines of DASH

Application	*Problem Size*	*B. Imbalance*
Volrend	head	48.20%
Radix	1M integers, radix 1,024	19.50%
FMM	16k particles, 8 time steps	16.56%
Barnes	16k particles, 8 time steps	15.93%
Water-Nsq	512 molecules, 12 time steps	12.90%
Water-Sp	512 molecules, 12 time steps	9.79%
Ocean	514 by 514 ocean	7.60%
FFT	64k points	3.82%
Cholesky	tk15	1.64%
Radiosity	room -ae 5000.0 -en 0.05 -bf 0.1	1.04%

Table 2: Applications from the SPLASH-2 suite used in this study. Applications are sorted in decreasing barrier imbalance.

[20]. Each node has one processor and two levels of caches. The processor modeled is a six-issue out-of-order CPU. The caches are relatively small to capture the behavior that real-sized input data would exhibit on an actual machine with larger caches, as suggested in [28]. Shared data pages are distributed in a round-robin fashion among the nodes, and private data pages are allocated locally. Table 1 summarizes the parameters of the architecture modeled.

4.2 Applications

To evaluate our proposal, we use the applications of the SPLASH-2 suite [28]. Table 2 lists the applications studied, in descending order of barrier imbalance, as measured in simulations on our multiprocessor model. Two applications from the suite, Raytrace and LU, are not included in this study: Raytrace does not synchronize using barriers, and a more efficient version of LU that uses flags instead of barriers is readily available [23]. While we report results for all ten applications, our technique is naturally aimed at reducing energy waste for applications with a nontrivial barrier imbalance. Half of the applications listed exhibit a barrier imbalance of 10% or higher. These we call out *target* applications.

The problem sizes that we use are at least as large as the suggested sizes in [28]. We do not reduce the input size for any application, as that may exacerbate the barrier imbalance. The parallel efficiency (speedup over number of processors) for the studied applications ranges from 40.3% to 82.0%, with an average of 58.3% for 64 processors. This range is considered acceptable for these applications and system sizes [19]. For n-body problems, we increase the number of time steps, in order to extend the execution as to observe more barrier instances. This is compliant with the nature of these applications in a more realistic setting [27].

4.3 Energy Model

We focus our study on the overall energy consumption of the CPUs (core plus on-chip caches). Processors are by far the most power-hungry components of a node [9]. To model the energy consumption in active processors, we integrate Wattch [3] into our simulation infrastructure. To model the energy consumption in dormant CPUs, we resort to published datasheets for modern power-aware microprocessors [14]. We notice that the power consumption in the low-power sleep states varies noticeably across processor models, even within the same family. However, the relative ratios are very simi-

State	*P. Savings*	*Tr. Latency*	*Snoop?*	*V. Reduction?*
Sleep1 (Halt)	70.2%	$10\mu s$	Yes	No
Sleep2	79.2%	$15\mu s$	No	No
Sleep3	97.8%	$35\mu s$	No	Yes

Table 3: Low-power sleep states used in our study. Power savings are relative to TDP_{max}.

lar [14, 17]. Also, while Wattch is reasonably accurate in relative terms (energy breakdowns, comparisons), accurate modeling of absolute energy is not in its design goal [3]. Thus, even if we carefully model after a particular commercial processor, we cannot directly plug in absolute energy numbers reported in product datasheets. Therefore, in order to derive meaningful low-power sleep states, we proceed as follows: We conduct some microbenchmarking to obtain an estimate of the maximum thermal design power (TDP_{max}) in our processor model. This represents the total power dissipation of the processor while executing a worst-case instruction mix at nominal voltages and normal operating conditions. Then, we compute the ratios between published TDP_{max} and low-power sleep states, and apply those ratios to our TDP_{max}. This way, we derive the power consumption in those low-power sleep states in our processor model. Then, when simulating, we use those power values whenever we put a CPU to sleep, and the simulated power consumption during active computation.

We simulate a processor with three low-power sleep states *Sleep1-3*, inspired by actual low-power states of the Intel Pentium family[14, 16]. Sleep1, which we refer to as Halt, is a light low-power state that results from executing a Halt instruction. Sleep2 and Sleep3 are deeper sleep states that preserve the context in processor and caches, but cannot service external protocol requests. Sleep3 differs from Sleep2 in that it lowers the supply voltage, which results in reduced leakage. The power savings (relative to TDP_{max}) and transition latencies are summarized in Table 3. During the transitions in and out of these sleep states, we assume that power consumption changes linearly along the transition latency. We also directly simulate the power consumption at the spinloop. On average across the applications, the power consumption of executing the spinloop is about 85% of that of regular computation.

5 EVALUATION

In this section we evaluate the energy savings and performance implications of applying our proposed thrifty barrier mechanism in place of conventional barriers. We assess the effectiveness of the thrifty barrier in a 64-node CC-NUMA multiprocessor (Section 4), and analyze the different contributing factors.

5.1 Overall Results

We explore several configurations as follows: a baseline system with conventional barriers (*Baseline*); a thrifty configuration with Halt as its only low-power sleep state (Thrifty-Halt); an oracle version of the same system, in which BIT prediction is perfect (Oracle-Halt); a thrifty configuration with all three low-power sleep states available (Thrifty); and an ideal thrifty configuration with perfect BIT prediction and no flushing overhead for any low-power sleep state (Ideal).

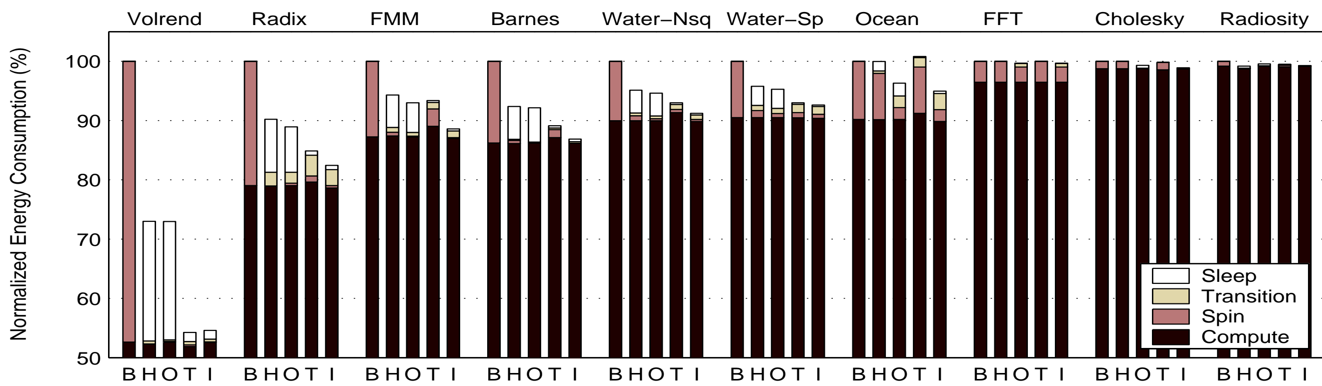

Figure 5: Normalized energy consumption for the SPLASH-2 applications under study on a 64-processor system. B, H, O, T, and I stand for *Baseline*, *Thrifty-Halt*, *Oracle-Halt*, *Thrifty*, and *Ideal*, respectively.

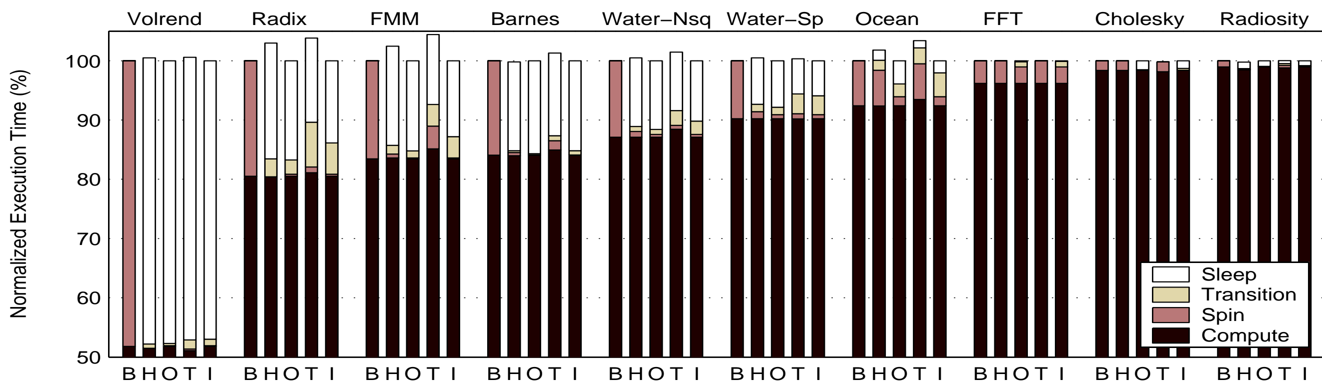

Figure 6: Normalized execution time for the SPLASH-2 applications under study on a 64-processor system. B, H, O, T, and I stand for *Baseline*, *Thrifty-Halt*, *Oracle-Halt*, *Thrifty*, and *Ideal*, respectively.

Figures 5 and 6 show the energy consumption and execution times, respectively, for the SPLASH-2 applications described in Section 4.2. Applications appear in descending order of their barrier imbalance in Baseline, as listed in Table 2 Five bars are shown for each application, corresponding to the five different configurations under study. In each group, all bars are normalized to that of Baseline. Figure 5 shows the average energy consumption; a configuration is better the lower its energy bar is. On the other hand, in Figure 6, which plots execution time, the goal is to keep performance as close to Baseline's as possible.

The results are very encouraging. Thrifty is effective at saving energy, particularly among the five target applications (10% or higher imbalance), for which it reduces energy consumption by about 17% on average. At the same time, performance degradation is well bounded, amounting to about 2% on average for the target applications. For applications with smaller barrier imbalance, Thrifty naturally does not show as much energy reduction, but at the same time the performance loss is virtually zero in all cases except Ocean. Thrifty in fact expends a little more energy and time in Ocean than Baseline; we comment on this case a bit later. In the case of FFT and Cholesky, Thrifty (and Thrifty-Halt) behaves just like Baseline. This is because these applications only have a handful of non-repeating barriers, which leaves Thrifty's PC-indexed predictor unused.

When comparing Thrifty against Thrifty-Halt and Oracle-Halt, we can see that exploiting multiple sleep states is indeed beneficial. When only the Halt sleep state is available, Thrifty-Halt is unable to accrue energy savings beyond 11% for the five applications of interest (itself an encouraging figure), vs. 17% for Thrifty. (If we substitute Volrend in the averages with the next application in line, Water-Sp, energy savings are 6.5% vs. 10.5%, respectively.) Performance losses are definitely below Thrifty's 2% for these applications, although losses do occur. Oracle-Halt does not fare much better in terms of energy savings. More conventional low-power synchronization techniques, like executing Halt after spinning unsuccessfully for a while, or using a Pause instruction (originally conceived for different reasons) in a spinloop [13], would likely find a lower bound in Oracle-Halt, itself inferior to Thrifty.

5.2 Contributing Factors

To further analyze these results, we break down each bar into up to four segments, each corresponding to the energy (time for Figure 6) spent in one of the following states: *Compute*, which represents the energy (time) spent not at a barrier (note that other types of stalls, such as memory or locks, fall into

this category as well); *Spin*, which is the energy (time) used up by spinning on the barrier flag; *Transition*, which comprises the energy (time) spent by the CPUs transitioning in and out of low-power sleep states; and *Sleep*, which represents the energy (time) used in some low-power sleep state.

As expected, Thrifty is the only configuration for which Compute energy/time increases for many applications, mainly due to cache flush overheads associated with deep sleep states. This is most noticeable in FMM, Water-Nsq, and Ocean. In Volrend, however, Compute actually decreases a bit, not only in Thrifty but also in Thrifty-Halt. We believe this is because the occasional thread delays caused by overprediction happen to have a beneficial effect in lock contention (threads in Volrend sift through tens of thousands of locks). In any case, this effect is small.

Spin energy/time in Thrifty is generally quite small compared to Baseline. This means that our mechanism, when faced with a barrier, often finds a low-power sleep state to move into. Exceptions to this (other than FFT and Cholesky) are FMM and, above all, Ocean. Ocean has a number of barriers that are invoked often and whose interval times can swing significantly across instances. In general, the simple last-value prediction used in Thrifty and in Thrifty-Halt does not work well for this pattern. In the case of Ocean, the barrier interval times of these barriers often drop to the point that Thrifty overkills in selecting a sleep state, and external wake-up kicks in. This not only exposes the transition time out of the sleep state in both Thrifty and Thrifty-Halt, but also the flushing of dirty data (and subsequent compulsory misses) required by the overkill sleep state in the case of Thrifty. Without a prediction cut-off mechanism (Section 3.3.3) in place, experiments showed that Ocean could degrade in performance by as much as 12% over Baseline. Fortunately, our cut-off provision is very effective here, containing losses in Thrifty within 3.5% of Baseline. As a logical side effect, Ocean ends up spinning quite a bit at these barriers, as Thrifty and Thrifty-Halt show. Notice that the theoretical lower bounds Oracle-Halt and Ideal, which never mispredict, would actually save energy without incurring any performance penalty. The still noticeable Spin in these is further evidence that such short barriers do occur.

Other than Spin in the cases mentioned above, barrier energy/time is dominated by Transition+Sleep in all configurations except Baseline (which sees neither). It is here that Thrifty shines over Thrifty-Halt and even Oracle-Halt, in selecting deeper sleep states that shrink energy consumption at little performance overhead. By far, the application that benefits the most from deeper sleep states is Volrend, whose large barrier interval times and imbalance create an ideal scenario for Thrifty, which matches the savings of Ideal.

Overall, we see that Thrifty is effective in exploiting the available sleep states to save energy, while at the same time containing performance degradation.

6 RELATED WORK

Low-power computing has long been an important design objective for mobile, battery-operated devices. However, reducing energy consumption for high-performance systems became important relatively recently. In a uniprocessor environment, various components are reconfigured during non-critical time to save energy consumption with little performance impact [1, 2, 8]. Sometimes, multiple components are adapted simultaneously to increase the effectiveness [5, 11, 12, 18]. These techniques are orthogonal to our approach in that they try to reduce energy consumption while the processor is actively executing useful instructions. Instead, our approach reduces energy waste in useless spinning, when waiting at a barrier. In a web server cluster environment, studies have been done to evaluate different policies to control the number of active servers (and thus their performance level) to preserve power while maintaining acceptable quality of service [6, 7, 24, 25]. This body of work is different from ours in that the target system is a loosely-coupled cluster running independent workloads, while we focus on shared-memory multiprocessor systems running parallel applications.

To our best knowledge, the only other works that address the energy issue in shared-memory multiprocessors look at energy savings in cache coherence management [22, 26]. Inspired by the observation that a large fraction of snoops do not find copies in many of the other caches in a snooping bus-based SMP, Moshovos, et al [22] propose *Jetty* to reduce the energy consumed by snoop requests. Saldanha and Lipasti [26] examine the effects of reducing speculation in a scalable snoop-based design, and observe significant potential of energy savings by using serial snooping for load misses. In the thrifty barrier, we instead work on the CPU energy savings potential stemming from barrier imbalance in parallel applications. These techniques are not exclusive of each other and could be combined for improved overall results.

Kumar et al. [19] study synchronization issues on shared-memory multiprocessors. They find that load imbalance contributes the most to the synchronization in parallel applications; in contrast, the overhead of synchronization operations does not dominate, even in a balanced situation where it is expected to be substantial. This suggests that introducing lightweight control algorithms in synchronization constructs, such as the one used in the thrifty barrier, should present little performance impact.

ACPI [4] allows OS-directed power management with definition of hardware registers and BIOS interfaces. It defines industry standards for configuration and thermal management. It considers all system states as *power states*. The proposed thrifty barrier is designed to leverage the support of such power states to improve the energy efficiency of imbalanced parallel applications.

7 CONCLUSIONS

We have presented the *thrifty barrier*, a novel synchronization solution that reduces energy waste incurred by imbalanced applications in shared-memory multiprocessors. The thrifty barrier predicts the stall time based on past history and drives the processor into one of multiple CPU low-power sleep states of different characteristics. In anticipation of the barrier release, the thrifty barrier also strives to wake up the processor just in time to avoid potential performance degradation.

One key design issue for the thrifty barrier is to accurately predict the stall time for each thread at a barrier. We found

that barrier stall time can be accurately estimated in many cases, indirectly by using simple last-value prediction of the interval time between two consecutive barriers. By combining this prediction mechanism with simple hardware extensions, the thrifty barrier is able to wake up dormant threads as the barrier is released, effectively achieving important energy savings without incurring significant performance degradation.

We are looking at ways to extend this work, including extending this concept to other parallel computing environments, such as message-passing systems, and to other synchronization constructs, such as locks.

ACKNOWLEDGMENTS

We thank Evan Speight and the anonymous reviewers for useful feedback. This work was supported in part by gifts from Intel. Jian Li's support was provided in part by NSF grants ACI-0103723 and ACI-012140.

REFERENCES

[1] D. Albonesi. Selective cache ways: On-demand cache resource allocation. *Journal of Instruction-Level Parallelism*, 2(1–6), May 2000.

[2] R. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *International Symposium on Computer Architecture*, pages 218–229, Gothenburg, Sweden, June–July 2001.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *International Symposium on Computer Architecture*, pages 83–94, Vancouver, Canada, June 2000.

[4] Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd. and Toshiba Corporation. *Advanced Configuration and Power Interface (ACPI) Specification, Revision 2.0c*, Aug. 2003.

[5] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. Scott. Integrating adaptive on-chip storage structures for reduced dynamic power. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 141–152, Charlottesville, VA, Sept. 2002.

[6] E. N. Elnozahy, M. Kistler, and R. Rajamony. Energy-efficient server clusters. In *Workshop on Power Aware Computing Systems*, pages 179–196, Cambridge, MA, Feb. 2002.

[7] E. N. Elnozahy, M. Kistler, and R. Rajamony. Energy conservation policies for web servers. In *USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, Mar. 2003.

[8] D. Folegnani and A. González. Energy-efficient issue logic. In *International Symposium on Computer Architecture*, pages 230–239, Gothenburg, Sweden, June–July 2001.

[9] R. Graybill and R. Melhem, editors. *Power Aware Computing*. Kluwer Academic/Plenum Publishers, 2002.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier Science Pte Ltd., third edition, 2003.

[11] M. Huang, J. Renau, and J. Torrellas. Positional adaptation for processors: Application to energy reduction. In *International Symposium on Computer Architecture*, pages 157–168, San Diego, CA, June 2003.

[12] C. J. Hughes, J. Srinivasan, and S. V. Adve. Saving energy with architectural and frequency adaptations for multimedia applications. In *International Symposium on Microarchitecture*, pages 250–261, Austin, TX, Dec. 2001.

[13] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2003.

[14] Intel Corporation. *Intel Pentium M Processor Datasheet*, Mar. 2003.

[15] Intel Corporation. *Intel Xeon Processor with 533 MHz Front Side Bus at 2GHz to 3.06GHz Datasheet*, Mar. 2003.

[16] Intel Corporation. *Mobile Intel Pentium III Processor-M Datasheet*, Jan. 2003.

[17] International Business Machines Corporation. *IBM PowerPC 750FX RISC Microprocessor Datasheet, Version: 2.0*, June 2003.

[18] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Design, Automation and Test in Europe*, pages 190–196, Munich, Germany, Mar. 2001.

[19] S. Kumar, D. Jiang, R. Chandra, and J. P. Singh. Evaluating synchronization on shared address space multiprocessors: Methodology and performance. In *International Conference on Measurement and Modeling of Computer Systems*, pages 23–34, Atlanta, GA, May 1999.

[20] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *International Symposium on Computer Architecture*, pages 148–159, Seattle, WA, June 1990.

[21] J. F. Martínez and J. Torrellas. Speculative Synchronization: Applying thread-level speculation to parallel applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, San Jose, CA, Oct. 2002.

[22] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: Filtering snoops for reduced energy consumption in SMP servers. In *International Symposium on High-Performance Computer Architecture*, pages 85–96, Nuevo Leone, Mexico, Jan. 2001.

[23] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An evaluation of memory consistency models for shared-memory systems with ILP processors. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, Cambridge, MA, Oct. 1996.

[24] E. Pinheiro, R. Bianchini, E. Carrera, and T. Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *International Workshop on Compilers and Operating Systems for Low Power*, Barcelona, Spain, Sept. 2001.

[25] K. Rajamani and C. Lefurgy. On evaluating request-distribution schemes for saving energy in server clusters. In *International Symposium on Performance Analysis of Systems and Software*, pages 111–122, Austin, TX, Mar. 2003.

[26] C. Saldanha and M. Lipasti. Power efficient cache coherence. In *Workshop on Memory Performance Issues*, Gothenburg, Sweden, June 2001.

[27] J. P. Singh, J. L. Hennessy, and A. Gupta. Implications of hierarchical n-body methods for multiprocessor architectures. *ACM Transactions on Computer Systems*, 13(2):141–202, May 1995.

[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.