Managing Energy-Performance Tradeoffs for Multithreaded Applications on Multiprocessor Architectures

Soyeon Park, Weihang Jiang, Yuanyuan Zhou, and Sarita Adve
Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
{soyeon, wjiang3, yyzhou, sadve}@cs.uiuc.edu

ABSTRACT

In modern computers, non-performance metrics such as energy consumption have become increasingly important, requiring tradeoff with performance. A recent work has proposed performance-guaranteed energy management, but it is designed specifically for sequential applications and cannot be used to a large class of multithreaded applications running on high end computers and data servers.

To address the above problem, this paper makes the first attempt to provide performance-guaranteed energy management for multithreaded applications on multiprocessor architectures. We first conduct a comprehensive study on the effects of energy adaptation on thread synchronizations and show that a multithreaded application suffers from not only local slowdowns due to energy adaptation, but also significant slowdowns propagated from other threads because of synchronization. Based on these findings, we design three Synchronization-Aware (SA) algorithms, LWT (Lock Waiting Time-based), CSL (Critical Section Length-based) and ODP (Operation Delay Propagation-based) algorithms, to estimate the energy adaptation-induced slowdowns on each thread. The local slowdowns are then combined across multiple threads via three aggregation methods (MAX, AVG and SUM) to estimate the overall application slowdown.

We evaluate our methods using a large multithreaded commercial application, **IBM DB2** with industrial-strength online transaction processing (OLTP) workloads, and six SPLASH parallel scientific applications. Our experimental results show that LWT combined with the MAX aggregation method not only controls the performance slowdown within the specified limits but also conserves the most energy.

Categories and Subject Descriptors: B.3 [Memory Structures]:Miscellaneous; C.4 [Performance of System]

General Terms: Algorithms, Performance, Management

Keywords: memory energy management, multithreaded applications, energy and performance tradeoffs, low power design

SIGMETRICS'07 June 12–16, 2007, San Diego, California, USA.

1. INTRODUCTION

1.1 Motivation

Energy consumption has emerged as an important issue in the design of computing systems. Energy conservation is important not only to mobile devices for saving battery life but also to high-end servers, which commonly require a large amount of energy and high cooling costs [13]. The last few years have seen significant research in the use of adaptive architectures to save processor energy [2, 18]. More recently, researchers have recognized that memory is also a key consumer of energy. For example, a study shows that for fully configured IBM server systems [6], memory energy consumption could be as high as 150% of processor energy. To conserve energy, commercial memory chips now support multiple power modes [15], and several control algorithms to adapt between these modes have been proposed [5, 9].

Unfortunately, reducing energy consumption does not come for free. It usually involves trading off performance. For example, energy consumption can be reduced by placing a memory module being infrequently accessed into a low power mode, but a following access to the memory module requires an extra delay to transition it back into a full power mode. The delays caused by energy management can result in significant performance slowdowns, which is unacceptable for high-end data and computation servers. Therefore, a methodology to monitor and control performance slowdowns is required for energy management systems.

To address the above requirement, a recent paper [9] has proposed a performance guaranteeing method that dynamically estimates the performance slowdown introduced by the underlying energy management and keeps it within a specified bound. The performance guarantee is provided at a relative coarse granularity, such as the overall execution time from the beginning, instead of at a small task or operation granularity (our work also follows such a performance guarantee goal). Specifically, this method monitors the performance slowdown to detect when it exceeds a specified limit, and then it disables the underlying energy management by forcing all devices to transition into full power mode. After the cumulative slowdown is safely below the specified limit, the underlying energy management scheme is re-enabled to conserve energy. It was shown that this performance guaranteeing method works well with various static and dynamic energy management schemes for *sequential applications* [9].

While the above technique makes an important step towards in making energy management acceptable to applications that need to honor performance requirements, it is lim-

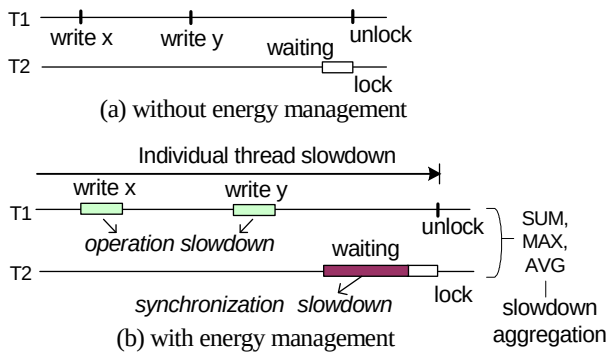


Figure 1: Synchronization slowdown and our synchronization-aware performance guarantee method for energy management.

ited to *sequential applications* and is not applicable to multithreaded scientific and commercial applications on multiprocessor architectures, which are the typical applications and architectures running in host centers. The main reason is that their performance slowdown estimation is designed based on sequential applications and fails to take the thread interactions and synchronization effects into consideration. More specifically, they only count the *local* slowdowns to each device access on a single thread, but do not consider the *slowdown cascading effects* from other threads due to thread synchronization.

The above issue significantly limits the applicability of energy management to the majority of the applications running on host centers. Moreover, this limitation will become increasingly severe and will soon affect many other types of applications due to the prevalent multicore technology. While multithreaded programming on multiprocessor architectures has the advantages of increasing performance via parallelism, it also makes performance estimation and analysis much harder than sequential applications due to synchronization and thread interactions.

To estimate slowdowns for multithreaded applications, the following two issues need to be addressed in addition to those for sequential applications.

The first issue is that the performance slowdown of a thread can affect other threads through inter-thread synchronizations. As shown in Figure 1, the slowdown to regular instructions (i.e. memory accesses) inside critical sections can increase the length of the critical section and thereby make other threads wait for the same lock for a longer time. In this case, the slowdown of the first thread propagates to all other waiting threads in a cascading fashion. If a slowdown estimation scheme does not consider such cascading effects across threads but only count the local effects (e.g. memory access delay due to energy adaptation), it may under-estimate slowdowns and eventually an energy management based on it may fail to meet the specified performance requirement.

The second issue is how to estimate the overall application slowdown from individual thread slowdowns. Within a period of time, energy adaptation can cause different slowdowns to different threads. Then, which thread's slowdown affects the entire application's slowdown? Can the average slowdown approximate the overall application slowdown? The answer to the question relies on each application's synchronization characteristics.

1.2 Our Contributions

To address the above challenge, this paper makes one of the first steps in providing performance guaranteed energy management (conserving energy without violating the user specified slowdown limit) for multithreaded applications on multiprocessor architectures. Our ultimate goal is to make energy management practical to a large class of applications and architectures running in host centers. While most of our ideas are applicable to energy management of different hardware components, as a proof of concept, this paper focuses on memory energy management.

Specifically, we first conducted a comprehensive study on the effects of memory energy management on synchronization characteristics using a large commercial multithreaded application, **IBM DB2** with industrial-strength online transaction processing (OLTP) workloads, and six SPLASH/SPLASH-2 parallel applications. We found that energy management can significantly increase the synchronization waiting time, which indicates that tracking the cascading synchronization slowdown is critically important for the accurate estimation of the overall slowdown in multithreaded systems.

Motivated by the above results, we design several methods to dynamically estimate performance slowdowns due to memory energy management by decoupling the challenge into two sub-problems and provide solutions to both in an orthogonal way as shown on Figure 1:

(1) Individual Thread Slowdown Estimation: To estimate the slowdown for each individual thread, we derive three synchronization-aware (SA) algorithms, LWT (Lock Wait Time-based), CSL (Critical Section Length-based) and ODP (Operation Delay Propagation-based). Their estimations consider not only slowdowns to regular instructions but also the cascading synchronization slowdowns, i.e. the extra synchronization waiting time due to energy management. LWT and CSL conservatively estimate the slowdown upper-bounds and thereby can provide performance guarantees, whereas ODP is a heuristic-based approach and can only reasonably control performance slowdowns.

(2) Slowdown Aggregation: To estimate the overall application slowdown from individual thread slowdowns, we explore three slowdown aggregation methods, SUM, MAX and AVG, in combination with the above three SA individual thread slowdown estimation algorithms and the previous synchronization-oblivious algorithm. The SUM method conservatively takes the sum of slowdowns incurred by all threads and uses it as the application overall delay for monitoring when to disable the underlying energy management scheme. The MAX and AVG methods, respectively, use the maximum and average thread delay as the overall delay.

Our experimental results from a whole system chip multiprocessor (CMP) simulator based on SIMICS, show that the LWT and CSL algorithms with MAX, namely LWT+MAX and CSL+MAX, work well: they not only guarantee the application performance, but also conserve the most energy. The heuristic algorithm, ODP, can control the performance of most applications but fails to guarantee performance in a few cases, especially for applications with many locks. Without considering synchronization slowdowns, MAX or AVG alone cannot guarantee performance, indicating the importance of tracking synchronization slowdowns. Although SUM can control performance degradation within a spec-

ified limit, it is too conservative and thereby consumes up to 43% more energy on average than LWT+MAX.

The remainder of this paper is organized as follows: In Section 2, we introduce the background of memory energy management. We describe our synchronization aware performance guaranteeing algorithms in Sections 3 and 4. Experimental methodology and results are presented in Sections 5 and 6. We discuss the related work in Section 7 and then conclude in Section 8.

2. BACKGROUND

In this section, we present the underlying memory energy management scheme and the previously proposed performance control method [9] as a background for our work. Since the focus of this paper is on how to track and control the performance slowdowns caused by energy management for multithreaded applications, we do not change the underlying energy management scheme.

Memory Power Model: We use a modern memory subsystem that is capable of operating in multiple power modes. In particular, our model follows the specifications for Rambus DRAM (RDRAM) [15]. When not in active use, each RDRAM chip can be placed into a low-power operating mode (*standby*, *nap*, or *powerdown*) to save energy. Accesses to chips in the low-power operating modes incur additional delay and energy for bringing the chip back to active state. The delay time varies from several cycles to several thousand cycles depending on which low power state the chip is in. The lower the power mode, the more time and the more energy it takes to activate it for access.

Energy Control Algorithm: To utilize the lower power modes, the key is to have effective control algorithms to decide when and which power mode to put each chip into. Researchers have recently proposed several memory adaptation algorithms [5, 9] and have shown that a dynamic scheme that transitions a chip into low power modes after a threshold of idle time performs better than a static scheme which places all chips in a fixed power mode except when it is necessary to service a request. Unfortunately, the dynamic scheme requires painstaking threshold tuning for each workload and can significantly degrade performance when the wrong set of thresholds are chosen.

To address this problem, Li et al. [9] proposed a performance directed energy management (PD) that removed the necessity for painstaking parameter tuning and provided performance control by periodically adjusting their decisions based on the available slack and recent workload characteristics. This algorithm outperforms the original dynamic/static algorithm with the best hand-tuned parameter setting. In our work, we use the PD algorithm [9] as our underlying energy control algorithm and integrate it with our synchronization-aware slowdown estimation and aggregation methods.

Providing Performance Guarantee for Sequential Applications: Li et al. [9] also proposed a method that can effectively provide a coarse-grain performance guarantee in memory energy adaptation for sequential applications. This method works with the PD energy management algorithm as well as any other dynamic/static algorithms. The main idea of the performance guaranteeing method is to keep track of the slowdown in execution time introduced by the underlying memory energy adaptation, and when the observed slowdown is greater than the maximum slowdown allowed by users, it forces all chips to active. Specifically, it divides time into epochs (time period). At the beginning of an epoch, it calculates the available slowdown (called "slack") of this epoch with the user specified performance slowdown limit (e.g. 10%) and the estimated slowdowns experienced until that time. During the epoch, it monitors the slowdown. When the slowdown exceeds the slack, it disables the underlying energy adaptation and forces all chips to active until the end of this epoch. It is possible that, if the slowdown exceeds the specified limit in the last few epochs of the program execution, it might be too late to catch up the slowdowns even by transitioning all chips to active. However, such effects would be negligible as long as the program runs for a reasonable long time with respect to the epoch length (e.g. 10,000 CPU cycles).

The key issue to the above performance guaranteeing method is the slowdown estimation. First, it should not be too aggressive: it should not under-estimate the slowdown caused by energy adaptation; otherwise it may violate the specified performance guarantee. Second, it should not be too conservative, i.e. significantly over-estimating the slowdown; otherwise, it would lose opportunity to conserve energy. To improve slowdown-estimation accuracy, this method also considers certain parallelism due to out-of-order execution and instruction pipelining, but not due to multithreading in multiprocessor architectures.

In Table 1, our evaluation results show that, when our tested applications (described in Section 5) are executed on *only one* single thread, the performance control method with the PD algorithm can effectively bound the performance slowdown within the specified limit (20% in these experiments). It indicates that the control method can accurately estimate the energy management-induced slowdown for *sequential* applications.

DB2 OLTP	Cholesky	Barnes	Radiosity	Water	Ocean
17.7%	18.5%	19.2%	18.3%	18.6%	19.3%

Table 1: Performance slowdown of our tested applications with *only ONE* thread using the previous performance control method and PD energy management algorithm (20% slowdown limit)

While we borrow the general idea of performance guaranteeing method from Li et al. [9], including monitoring the slowdown and disabling energy adaptation when the slowdown exceeds the limit, our work focuses on how to estimate slowdown for multithreaded applications on multiprocessor architectures.

3. SLOWDOWN AGGREGATION

The natural first step to extend the previous algorithm [9] for multithreaded applications on multiprocessor architectures is *slowdown aggregation*: estimating the overall application slowdowns from individual thread slowdowns due to energy management. In this section, we discuss three such aggregation methods.

Most importantly, at the end of this section, we show that *simply extending the previous synchronization-oblivious algorithm with slowdown aggregation is not enough to achieve the goal, i.e., minimizing energy consumption without vio-*

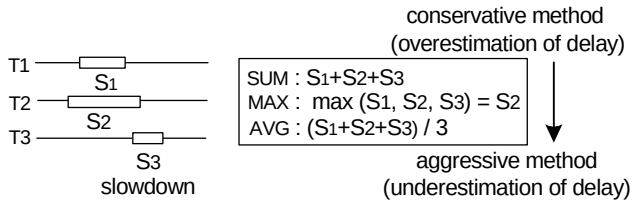


Figure 2: Three slowdown aggregation methods (S1, S2 and S3 are individual thread slowdowns).

lating specified performance slowdown limit. This finding motivates us to explore synchronization-aware slowdown estimation algorithms that not only track local slowdowns due to energy adaptation but can also track the cascading operation slowdowns propagated from other threads through synchronizations.

For simplicity of description, the remainder of this paper refers energy adaptation-induced slowdown to local regular instructions (i.e. memory accesses instructions) as *operation slowdowns*, and the operation slowdown propagated from other threads as *synchronization slowdowns*. The previous synchronization-oblivious performance guaranteeing method proposed by Li et al. [9] considers only operation slowdowns, not synchronization slowdowns.

3.1 Aggregation Methods

Intuitively, as shown on Figure 2, there are three slowdown aggregation methods, including SUM, MAX and AVG, to estimate overall application slowdowns from individual thread slowdowns. The first one, SUM uses the sum of the individual slowdown of each thread as the overall application slowdown. This method conservatively assumes that the slowdown on one thread can propagate to all other threads. Any cascading slowdown effect due to synchronization is already subsumed in this method. Therefore, as long as the individual thread slowdown estimation does not under-estimate the operation slowdowns, SUM can guarantee the slowdown caused by energy adaptation within the specified limit.

The problem with this method, however, is that it is too conservative: it can significantly over-estimate the performance slowdown caused by energy management. As a result, devices are frequently forced to full power mode by the performance guaranteeing mechanism, losing opportunities to conserve energy.

The MAX method uses the maximum of individual thread slowdowns as the overall application slowdown to guarantee performance. This method is based on the intuition that if the performance slowdown of the most delayed thread is within the specified limit, the entire application's performance can also be guaranteed.

This method can work well with some applications, in particular barrier-based applications whose progresses are determined by the last thread reaching a barrier. MAX guarantees that the thread with the largest slowdowns arrives at a barrier within the slowdown limit, and therefore so do other threads, as long as the individual slowdown estimation of each thread is accurate or conservative.

Unfortunately, combining this method with the previous synchronization-oblivious individual thread slowdown estimation cannot provide performance guarantee for lock-based applications, which will be shown in our experimental results in Section 3.2.

The AVG method aggressively uses the average slowdowns of all threads as the overall application slowdown. Obviously for latency-based applications such as SPLASH parallel applications that care about the total execution time, the AVG method cannot provide performance guarantee since in many cases the overall application slowdown can be larger than the average slowdown. However, for throughput-based applications such as DB2/OLTP and Web Servers that run for a long (almost infinite) duration and whose threads handle independent requests or transactions, intuitively AVG may be a suitable heuristic.

3.2 Aggregation Alone is Not Enough

A naive method to estimate energy adaptation-induced slowdown for multithreaded applications is just simply combining the previous synchronization-oblivious slowdown estimation with a slowdown aggregation method described above. Although such method is simple, unfortunately, it is not enough: it is either too aggressive to provide performance guarantee or too conservative to minimize energy consumption.

To evaluate the above naive extension, we combine the previous synchronization-oblivious slowdown estimation scheme [9] with three *slowdown aggregation methods*, namely SUM, MAX, and AVG, on six SPLASH / SPLASH-2 parallel applications. The experimental testbeds and the applications are described in Section 5. The specified performance slowdown limit is 10%.

As shown in Figure 3(a) and 3(b), while SUM with the previous synchronization slowdown estimation method can guarantee the slowdown within the specified limit (i.e. 10%), it consumes substantially more energy than the other two methods. For example, for LU, it consumes almost as twice energy as MAX and AVG.

While MAX and AVG can conserve more energy than SUM, none of them can guarantee performance with the previous synchronization-oblivious method for all applications. Since AVG aggressively uses the average thread slowdown as the overall application slowdown, it significantly under-estimates the slowdown and thereby fails to limit the slowdown to within the specified 10% limit for all six applications. Although MAX is more conservative than AVG, it still fails to provide performance guarantee for two applications, namely Cholesky and Barnes, because the individual thread slowdown estimation is synchronization-oblivious and does not track synchronization slowdowns.

To further understand the above results, we measure the amount of lock contention (the number of dynamic contented lock calls) with and without energy management. As shown on Figure 3(c), with energy management, the number of contented locks significantly increases. This is because, as also explained in Figure 1 in Introduction, operational delays can increase the length of critical sections, and, consequently, increase the chances for other threads to content for the same locks. These results provide a strong evidence for the cascading synchronization slowdowns caused by energy adaptation.

Figure 3(d) further examines the average synchronization slowdowns due to energy adaptation. Both MAX and AVG incur a significant percentage of synchronization slowdowns. Therefore, if the individual thread slowdown estimation does

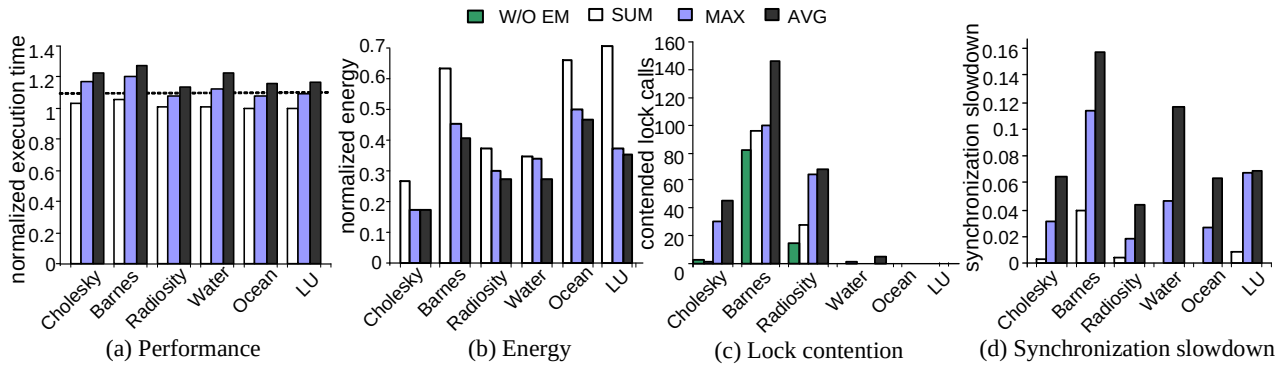


Figure 3: Extending the previous synchronization-oblivious performance guaranteeing scheme with the three slowdown aggregation methods (with 10% slowdown limit).

not consider such slowdowns, it is difficult to provide performance guarantee using the MAX and AVG methods.

In conclusion, the above results indicate that simply extending the previous method with slowdown aggregation is not enough to guarantee performance of multithreaded applications while minimizing energy consumption. To achieve the goal requires a synchronization-aware slowdown estimation method that considers not only operation slowdowns but also synchronization slowdowns propagated from other threads.

4. SYNCHRONIZATION-AWARE SLOWDOWN ESTIMATION

To address the problem discussed in Section 3, we propose three Synchronization-Aware (SA) slowdown estimation algorithms, based on our characteristic study on the effects of energy management on synchronization.

4.1 Effects of Energy Management on Synchronization Characteristics

Intuitively, operation slowdowns on individual threads can affect the synchronization waiting time in both positive and negative ways. *Here, whether an effect is considered positive or negative is viewed only for each isolated synchronization operation, not for the entire execution duration.* On the positive side, an operation slowdown can make a thread arrive at a synchronization later than without energy management and successfully hide all or part of the synchronization waiting time. On the other hand, some local operation slowdowns may also get magnified across multiple threads due to the cascading synchronization waiting effect.

These positive and negative cases are classified in Table 2 according to lock contention situation; Originally non-contended locks (without energy management) can become contended, or contended locks can become more highly contended with energy management (negative effect); conversely, contended locks can get less contended, or become even non-contended (positive effect).

The negative effect seems to be over-dominant the positive effect. In Table 2, negative cases happened almost twice more than positive cases (165 vs. 88). As a result, as shown in Figure 3, energy management significantly increases the number of contented locks and incurs substantial synchronization slowdowns especially with MAX and AVG.

Change in lock contention		Num. of cases	Avg. W.T difference	Effect
non-contended → contended		163	75992	Negative
contended → contended	W.T ⇑	2	9598	
	W.T ⇓	3	16230	Positive
contended → non-contended		85	11771	
non-contended → non-contended		16947	–	–

Table 2: Effect of memory access delay on synchronization slowdown: there are negative and positive effect according to change in lock contention, but negative effect is dominant. (Application : Barnes from SPLASH-2, W.T = lock waiting time)

Additionally, to provide performance guarantee, it is more important to track the negative effect than the positive effect in the slowdown estimation. This is because if the slowdown is under-estimated, it can violate the specified slowdown limit, whereas if the slowdown is over-estimated, it only loses some opportunities to conserve energy but performance is still guaranteed.

From the above reason, our synchronization-methods focus on how to track the negative effect on synchronization slowdowns due to energy management.

4.2 SA Algorithms

While it is infeasible to track synchronization slowdowns 100% accurately, we can guarantee performance under energy management by estimating the **upper-bound of actual slowdowns**. Based on this design principal, we propose three Synchronization-Aware (SA) slowdown estimation algorithms, namely LWT (Lock Waiting Time-based) and CSL (Critical Section Length-based), for performance guarantee, and a heuristic method called ODP (Operation Delay Propagation-based) estimation that can balance performance slowdowns and energy consumption.

(1) LWT (Lock Waiting Time-based) estimation: This algorithm is based on the observation that the synchronization slowdown of a thread at a synchronization cannot exceed the absolute synchronization waiting time with energy management because the latter includes the former plus the original synchronization waiting time, as shown on

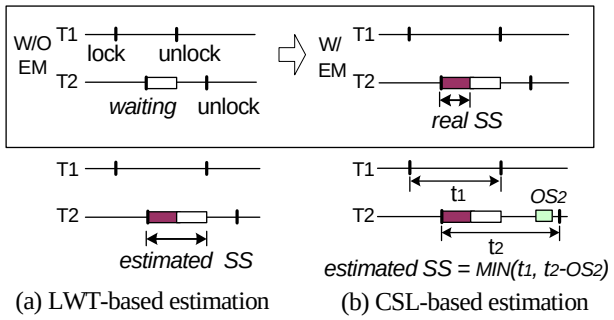


Figure 4: SA algorithms for performance guarantee: SS is synchronization slowdown and OS is operation slowdown. t1 and t2 are critical section length of thread T1 and T2, respectively.

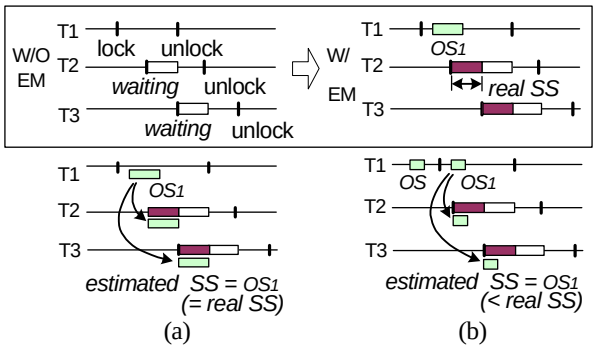


Figure 5: ODP heuristic estimation: SA algorithm for performance control. SS is synchronization slowdown and OS is operation slowdown. (a) shows a case that ODP works, and (b) is an example that ODP under-estimates the synchronization slowdown.

Figure 4.2(a). Specifically, for every lock, it adds the lock acquire time to the slowdown for the current epoch. LWT does not consider barrier waiting time because the barrier slowdown due to energy management is already considered by the MAX slowdown aggregation method.

(2) CSL (Critical Section Length-based) estimation: This algorithm is based on the observation that the synchronization slowdown of a thread $T2$ at a critical section of length L cannot be larger than L subtracting its operation delay, as well as the critical section length of the thread $T1$ that $T2$ is waiting. Formally, as shown on Figure 4.2(b), the synchronization slowdown (SS) for a critical section (CS) can be estimated conservatively as follows:

$$\text{SS(CS)} = MIN(t1(CS), t2(CS) - OS_2)$$

where $t1(CS)$ and $t2(CS)$ are thread T1 and T2's critical section length respectively, and OS_2 is thread T2's operation delay inside the critical section. Note the estimation has already taken the cascading synchronization effect into account: if thread $T2$ waited for another thread $T3$, $T2$'s extra waiting time is already counted in $t_2(CS)$. In most cases CSL's estimation is more conservative than LWT except some corner cases where the lock acquire time of thread $T2$ is dominated by the lock acquire operation but not the lock waiting time and $T1$'s critical section is very short. In addition, CSL also has different implementation implication than LWT, as discussed in more detail in Section 4.3.

(3) ODP (Operation Delay Propagation-based) estimation: The slowdown cascading effect is one of the particular factors to determine synchronization slowdowns. If a critical section grows longer due to memory accesses under energy management, the operation slowdowns inside a critical section can lengthen the lock acquire time of other threads waiting for the same lock. From this intuition, ODP propagates the operation slowdowns inside a critical section from one thread to all other waiting threads as their synchronization slowdowns. For example, in Figure 4.2(a), the estimated synchronization slowdowns on thread T3 is:

$$\text{SS(CS)} = \sum_{i=0}^{N-1} OS_i = OS_1$$

where N is the number of threads that have acquired the same lock while T3 is waiting for the lock, and OS_i means operation slowdown of thread i inside critical section CS.

Different from LWT and CSL that estimate the upper-bound of synchronization slowdown and thereby can provide performance guarantee, ODP is a heuristic method. Therefore, in some cases it may fail to guarantee performance. For example, in Figure 4.2(b), thread T2 and T3 suffer from more synchronization slowdowns than those estimated by ODP. This is because T1 has operation slowdowns both outside and inside the critical sections, all of which contribute to the extra waiting time in T2 and T3. However, ODP propagates only the operation slowdown inside the critical section from T1 to T2 and T3, and, as a result, it under-estimates the synchronization slowdowns for T2 and T3.

4.3 Implementation Issues

Similar to the previous method [9], the above three SA slowdown estimation algorithms can be implemented in the memory controller. The basic performance guaranteeing mechanism for keeping tracking of operation delays, comparing against available slack and forcing all chips to active when the slowdown exceeds the slack are also similar to the implementation of the previous method [9]. The only extension to basic mechanism is that it needs to know the processor ID (and hence the thread ID) of each memory access so it can charge the delay to the corresponding thread.

The implementation of synchronization slowdown estimation may also require some assistance from the processor because it requires synchronization related information. For example, LWT needs to catch each lock acquire attempt and the time when a lock acquire succeeds in order to find the lock waiting time. Since it only needs each thread's own lock waiting time, LWT does not require processor communication. CSL requires two kinds of information: how long a previous lock owner has held the lock if there is lock contention, and how long the critical section is locally executed. First, to measure the duration of a critical section requires identifying a lock acquire and a release point. Second, it needs to know which thread is waiting for a lock just being released. To implement this, a lock variable can be extended to record this information. ODP also requires the identification of lock operations and the transmission of synchronization information across multiple threads. Therefore, with regards to implementation complexity, ODP is similar to CSL.

Processor frequency	2GHz
Fetch, issue, commit width	4 instructions
L1 D-cache, I-cache	64KB 4-way
Unified L2 cache	2MB 4-way
Rambus DRAM base latency	80ns
number of processors	8
number of SCSI disks	7

(a) Target system parameters

Transition (→)		Time
Active	Standby	1 memory cycle
	Nap	8 memory cycles
	Powerdown	8 memory cycles
Standby	Active	5 memory cycles
Nap		48 memory cycles
Powerdown		4800 memory cycles

(b) Power-mode transition time

Table 3: System configuration

5. EVALUATION METHODOLOGY

5.1 Experimental Platform

All our experiments are conducted on a full-system simulator based on SIMICS [10], which models a CMP architecture with eight SPARC V9 chip processors and runs Solaris 9 as the operating system. We use GEMS [11] to model the detailed timing of out-of-order processors and cache-coherence protocol, and integrate a Rambus memory module simulation. Table 3(a) summarizes the system simulated in the study and (b) shows resynchronization time correspondent to power-mode transitions. The values have been obtained from the RDRAM specifications [15].

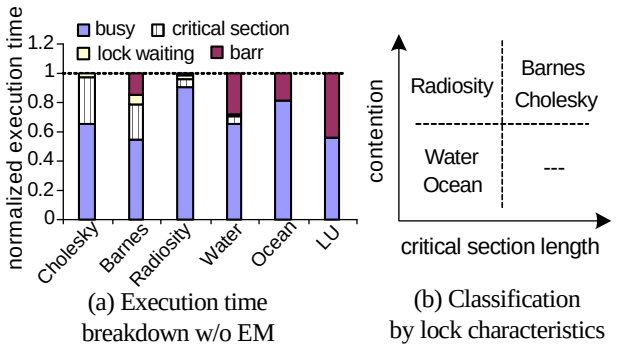
5.2 Tested Applications

In our experiments, we use a large multithreaded commercial database server, IBM DB2, and six SPLASH/SPLASH-2 parallel applications.

IBM DB2 running On-line Transaction Processing (OLTP, TPC-C-like): For experiments with IBM DB2, the whole system simulator is configured with 7 SCSI disks, on which the database data is stripped and can be directly accessed through raw I/O without involving the file system. The database size contains 100 warehouses, representing a balanced workload as reported in previous work [3]. The number of clients is set to 8 so that there are enough threads to fully utilize the 8-processor CMP.

The DB2 OLTP workload is an industrial strength TPC-C benchmark for IBM DB2. It involves a lot of database lock operations, which are implemented inside the IBM DB2. Although the OLTP workload spends a significant amount of time in kernel (30%), kernel locks are rarely contended, mainly because of highly optimized kernel code. Therefore, we focus on DB2's lock operations.

SPLASH/SPLASH-2 parallel applications: In our study, we use 6 representative parallel applications from SPLASH and SPLASH-2 benchmark suite based on their synchronization characteristics. Cholesky and Water comes from SPLASH, and Barnes, Radiosity, LU, and Ocean are SPLASH-2 applications. For these applications, we use spinning-based implementations for both lock and barriers.



(a) Execution time breakdown w/o EM

(b) Classification by lock characteristics

Figure 6: Application synchronization characteristics

The applications cover a wide spectrum of different synchronization characteristics as shown on Table 4. In addition, Figure 6(a) shows application-inherent synchronization delay for each application. LU and Ocean mostly use barriers for synchronizations, whereas the other four applications mostly use locks for synchronization with only a few barrier operations, but each with different levels of lock contention and critical section lengths.

Based on their synchronization characteristics, we classify the applications into four categories as shown in Figure 6(b). First, Cholesky and Barnes have highly contended locks and relatively long critical sections: Cholesky uses locks for the atomic updates of matrix columns and Barnes builds up a tree structure with locks. In addition, Barnes also has barrier synchronization overheads as well. Second, in Radiosity, locks are contended but critical sections are shorter than Cholesky and Barnes. There are multiple task queues that are frequently accessed with lock operations. Finally, in Water and Ocean, critical sections are small and locks are also not highly contended. Water has lock overhead, but it spends much more time for barrier synchronization at the end of each execution phase. Ocean and LU are barrier-based applications, and LU has no lock operations.

Application	Input	Lock calls	Barrier calls
Cholesky	tk14	9118 (13)	16
Barnes	16K particles	17200 (99)	8
Radiosity	room	12697 (14)	8
Water	216 mols, 3 steps	9470 (0)	8
Ocean	258-by-258 grid	208 (0)	168
LU	256-by-256 matrix	0 (0)	64

Table 4: SPLASH/SPLASH-2 parallel applications: the number of contended lock calls are given in parentheses.

As shown on Table 5, we evaluate various combinations of individual thread slowdown algorithms including the previous synchronization-oblivious algorithm and the three synchronization-aware slowdown estimation algorithms, and the three slowdown aggregation methods including SUM, MAX and AVG. Since SUM already takes the synchronization slowdown into consideration, we do not combine it with the three synchronization-aware algorithms.

Since DB2 is a commercial software, we are not able to estimate the critical section length or calculate operation delay

during the critical section through instrumenting its source code. However, fortunately, we can precisely estimate lock waiting time through instrumenting semaphore system calls, which are used to implement database locks. Therefore, for DB2 workload, we only evaluate LWT combined with three aggregation methods. Based on the discussion in Section 4 and our experimental results, LWT is the most promising synchronization-aware algorithm.

Individual Slowdown	Slowdown Aggregation		
Basic	SUM	MAX	AVG
LWT	–	LWT+MAX	LWT+AVG
CSL	–	CSL+MAX	CSL+AVG
ODP	–	ODP+MAX	ODP+AVG

Table 5: Different combinations evaluated in our experiments. Basic refers to the previous synchronization-oblivious slowdown estimation method.

.

6. EXPERIMENTAL RESULTS

6.1 Results for Performance Guarantee

Figure 7 shows the normalized execution time (normalized to the corresponding execution time without energy adaptation) with the nine combinations of individual thread slowdown estimation and slowdown aggregation under 10% and 20% specified slowdown limits.

For all applications, the two SA slowdown estimation algorithms LWT and CSL with MAX aggregation can control the slowdown within the specified limit (for both 10% and 20% cases). This is because of two reasons. First, LWT and CSL use upper-bounds of synchronization slowdowns to conservatively approximate synchronization slowdowns, so they may over-estimate but can never under-estimate synchronization slowdowns. Second, as discussed in Section 3, if the individual thread slowdown estimation does not overestimate, MAX can provide performance guarantee since it uses the maximum individual thread slowdown of each epoch as the overall application slowdown. Essentially, LWT and CSL track the cascading effects of lock-based synchronizations, while MAX takes care of barrier-based synchronizations. As a result, these combinations provide performance guarantee.

ODP+MAX can control the slowdown within the specified limit for almost all applications except a few cases (Water-10%, Cholesky-20%, Barnes-20%, Radiosity-20%, Water-20%), in which cases ODP+MAX slightly exceed the specified limit. This is because ODP is heuristic-based approach. As explained in Section 4, in some cases like the example shown in Figure 4.2(b), ODP may under-estimate the synchronization slowdown.

While MAX works well with the two synchronization-aware algorithms LWT and CSL, MAX with the previous synchronization-oblivious algorithm cannot guarantee performance in all cases, especially for lock-based applications such as Cholesky, Barnes, Radiosity, Water with 20% slowdown limit. As discussed in detail in Section 3.2, the main reason is that the synchronization-oblivious algorithm does not track the cascading synchronization slowdowns propagated from other threads.

The AVG method in general does not work well in guaranteeing performance. As shown in Figure 7(b), no matter which individual thread slowdown estimation algorithm is used, AVG fails to bound the slowdown within the specified limit. For example, in Cholesky with 20% slowdown limit, AVG significantly violates the performance guarantee, degrading the performance by up to 120%.

In contrast, SUM method is very conservative and under 10% slowdown limit it can always guarantee the performance degradation within 3%, but as we will show in the next subsection, it does not effectively save energy consumption.

We also notice that DB2 OLTP and scientific parallel applications show different preferences to different methods. Since DB2 OLTP is a throughput-based workload, it is less sensitive to the method used. As a result, MAX works well in terms of controlling the performance slowdown, regardless which individual thread slowdown estimation algorithm is used. AVG also performs reasonably well except when combined with the previous synchronization-oblivious algorithm. In contrast, SPLASH applications are much more sensitive to the method because they are latency-based workload.

6.2 Energy Saving Results

Figure 8 shows the normalized energy consumption (normalized to the corresponding energy consumption without energy adaptation) with the nine combinations of individual thread slowdown estimation and slowdown aggregation under 10% and 20% specified slowdown limits.

In general, LWT (combined with MAX or AVG) provides almost the maximum energy conservation, especially when compared to only those combinations that also provide performance guarantee. For example, in the 20% slowdown cases, CSL+MAX method can save 26% more energy than SUM method in the best case (Water), and similar results can be observed in 10% limit cases. LWT saves more energy than CSL because it provides a tighter upper bound in estimating the synchronization slowdown in practice (except a few theoretical corner cases that rarely happen in practice).

In most cases, AVG consumes slightly less energy than MAX since it is more aggressive in estimating slowdowns. However, the fact that it cannot provide performance guarantee offsets the small benefit in energy conservation. It is interesting to note that, as Figure 8(c) shows, AVG method does not always conserve more energy than SUM method because of the significantly increased execution time.

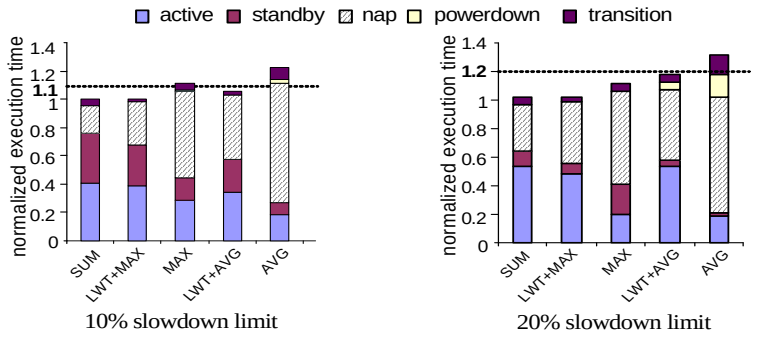
SUM is too conservative in slowdown estimation, so it saves much less energy than other combinations. For example, for LU, SUM almost consumes as twice energy as other schemes.

6.3 Detailed Analysis

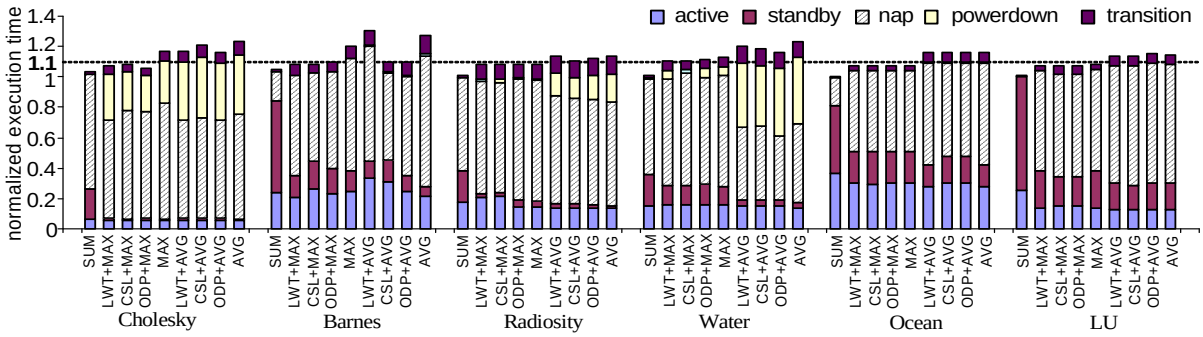
In order to have better understanding on slowdown estimation algorithms, we conduct further analysis on cases with 20% specified limit as examples.

6.3.1 SPLASH Parallel Applications

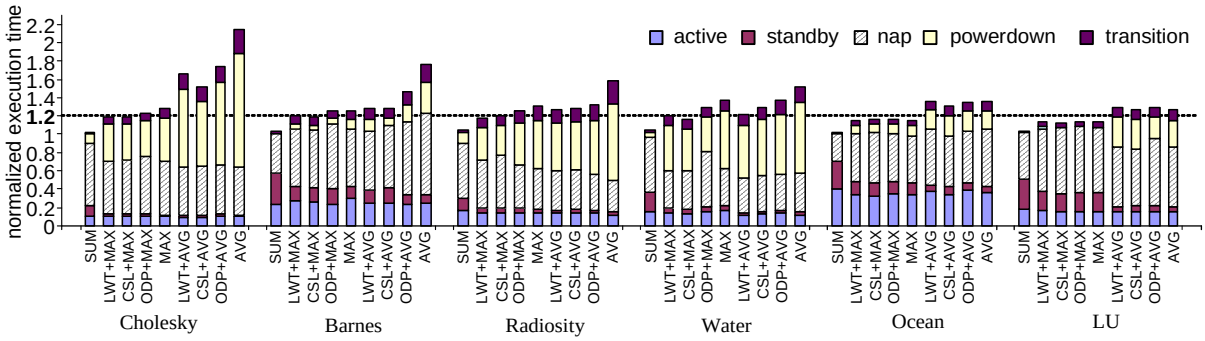
Figure 9(a) shows the synchronization slowdown of SPLASH/SPLASH-2 applications, which is the actual increased synchronization waiting time. It further breaks into the *extra* lock waiting time and barrier time. Figure 9(b) shows the change in lock contention under energy management with each performance guaranteeing scheme.



(a) DB2 OTP



(b) SPLASH/SPLASH-2 applications (10% slowdown limit)



(c) SPLASH/SPLASH-2 applications (20% slowdown limit)

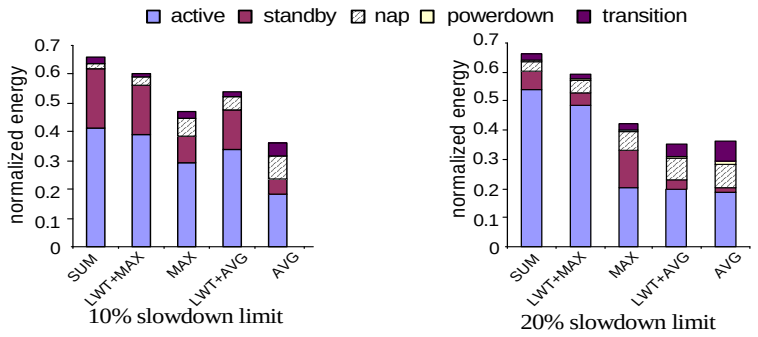
Figure 7: Performance comparisons

Long critical section, high contention: Originally, Cholesky and Barnes have long critical sections: all critical sections together account for 33% and 25% of total execution time, respectively. In the case of Cholesky, because the operation slowdown inside critical section lengthens critical section duration, lock contention significantly increases, especially under aggressive energy management. While the slowdown of LWT+MAX and CSL+MAX can be controlled within the specified limit (20%), none of slowdown estimation algorithms combined with aggressive AVG method can prevent severe performance degradation.

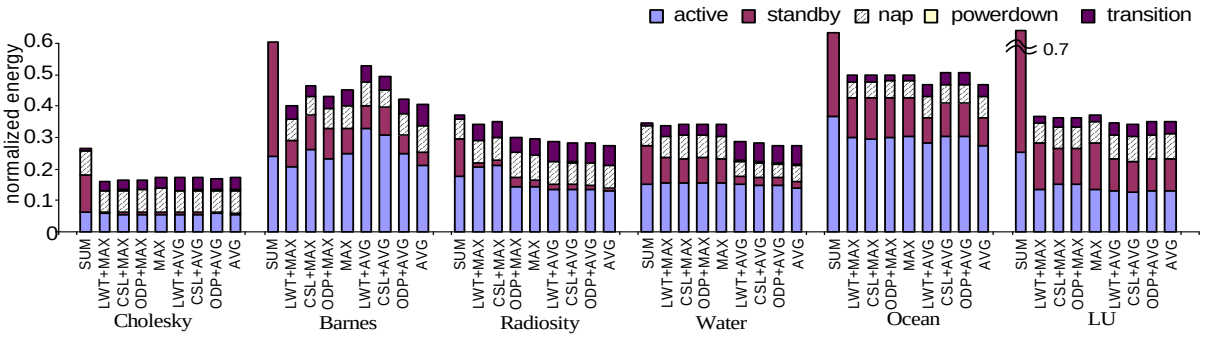
In comparison with Cholesky, Barnes originally has many lock contention and the number does not increase so much with energy adaptation. However, energy adaption further worsens the waiting time for those locks that are already contented (without energy adaptation), resulting in significant synchronization slowdown. Moreover, execution time imbalance among threads can be intensified due to both operation slowdown and synchronization slowdown and thus barrier time increases under energy management. While LWT+MAX and CSL+MAX can guarantee performance, synchronization-oblivious MAX schemes fail to do it.

Short critical section, high contention: Lock contention increases by energy management in Radiosity, but different from Cholesky and Barnes, synchronization slowdown is not magnified due to inaccurate estimation (ODP+MAX and MAX) or aggressive aggregation (AVG method) as much as Cholesky and Barnes. It is because that in Radiosity, critical section length and lock acquire time are inherently very small, as shown in Figure 6(a). Barrier time is not dominant as well. While heuristic ODP+MAX can guarantee performance, synchronization-oblivious MAX slightly exceeds the slowdown limit.

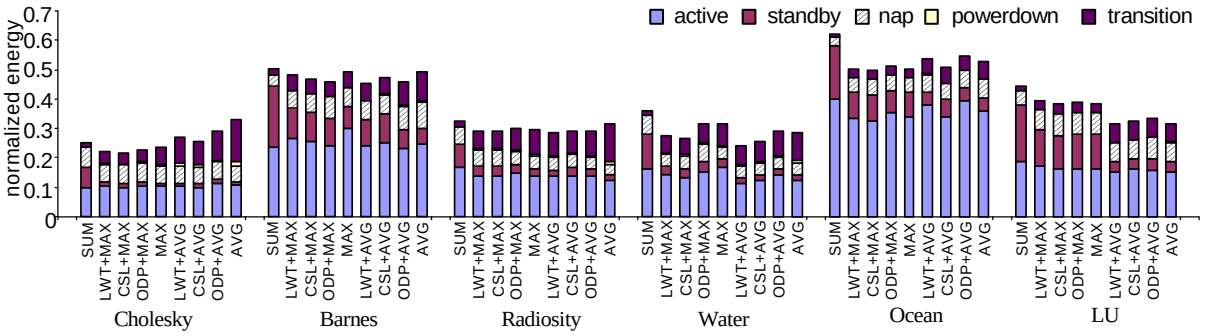
Short critical section, low contention (Barrier-based synchronization): Ocean and LU are barrier-based ap-



(a) DB2 OLTP



(b) SPLASH/SPLASH-2 applications (10% slowdown limit)



(c) SPLASH/SPLASH-2 applications (20% slowdown limit)

Figure 8: Energy comparisons

plications. In Ocean, lock contention is negligible, and LU has no lock synchronizations. Almost all synchronization slowdowns are caused by barrier synchronizations, and thus all estimation algorithms with MAX method can guarantee performance. In contrast, AVG method fails to bound synchronization slowdowns.

Water has a few locks and therefore exhibits different behaviors. Though energy adaptation increases lock waiting time slightly, the distribution of lock waiting time is changed (some threads have longer lock waiting time than the others, though the total waiting time does not change). Since LWT and CSL estimate individual thread slowdown based on the upper-bound of synchronization slowdown, LWT+MAX and CSL+MAX successfully bound the overall degradation. ODP+MAX and MAX fail to bound the degradation because they ignore or under-estimate the increased lock waiting time on individual threads.

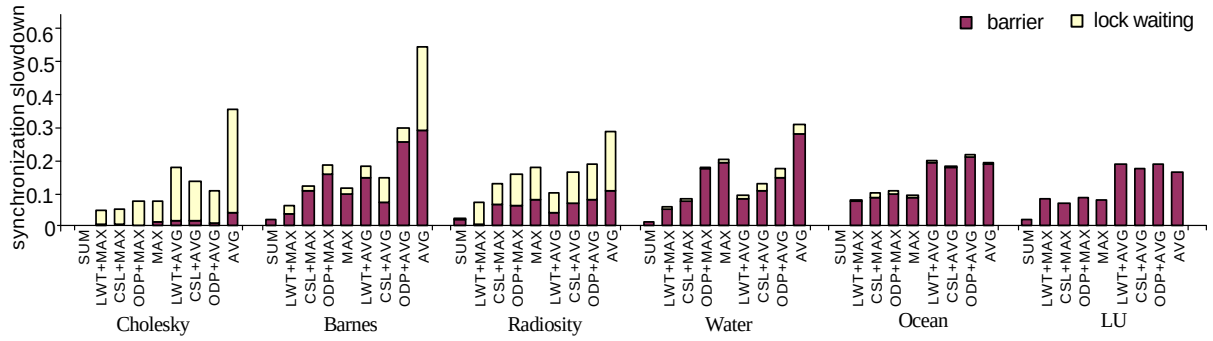
6.3.2 *Database Server Applications: DB2 OLTP*

The slowdown breakdown of DB2 OLTP is showed in Figure 10. LWT+AVG can bound performance degradation more tightly than other methods. Further analysis on DB2 OLTP slowdown reveals that lock contention level remains similar across all slowdown estimation algorithms, but the average lock waiting time differs with different algorithms.

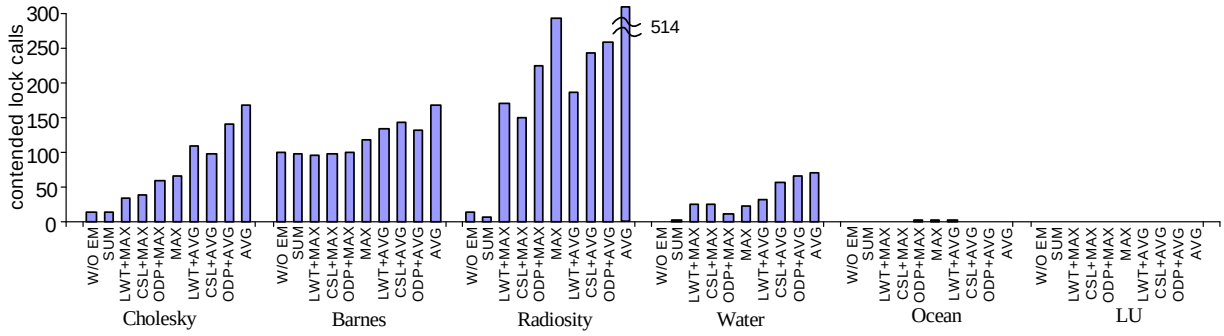
7. RELATED WORK

Since previous work on memory energy management has already been summarized in the Background section, we do not repeat them here.

For multicore system, recently, [4] proposed global power management using per-core power and performance monitoring, but their study focuses on different independent *sequential* applications running on the same multicore system instead of multithreaded applications. Thrifty Barrier [7]



(a) Synchronization slowdown breakdown (normalized to execution time without energy management)



(b) Lock contention

Figure 9: Slowdown analysis of SPLASH/SPLASH-2 applications(20% slowdown limit)

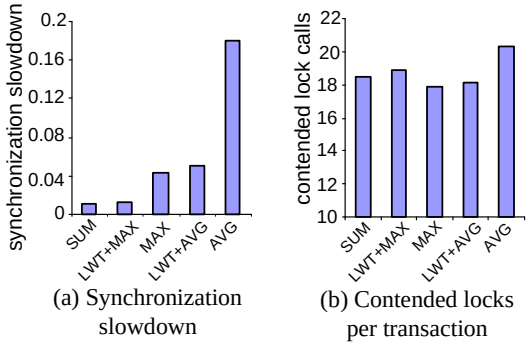


Figure 10: Slowdown analysis of DB2 OLTP workload (20% slowdown limit): synchronization slowdown is normalized to execution time without energy management.

saves energy at barrier synchronizations by predicting barrier stall time and putting threads that arrive early into sleep state. It effective conserves energy without degrading performance much.

Our work compliments the above work by proposing methods to estimate and control performance-energy tradeoffs for multithreaded applications on multiprocessors.

Lock synchronization has already been noticed to be problematic in performance analysis of multithreaded systems. Nayfeh et al. [14] discussed that synchronization can invalidate IPC as a performance metric by inflating the number of synchronization instructions, although in their study, workloads exhibit little synchronization. Based on OLTP workloads, Hankins et al. [3] found that instructions executed per transaction (IPX) can be affected by database lock contention, which varies with the scale of the database. Database locks can be the bottleneck for database workload [12], and kernel synchronizations also heavily affect database workloads, which was observed on a 8-CPU IRIX system [17].

Motivated by the importance of synchronization, Intel introduced Intel Thread Profiler, which can visualize the thread behavior and identify synchronizations [1]. More recently, Li et al. [8] proposed a hardware-based approach to automatically identify synchronizations, though their approach is limited to pure spin locks.

8. CONCLUSIONS AND FUTURE WORK

This paper has made the first attempt in providing performance guaranteed energy management for multithreaded applications on multiprocessor architectures. Specifically, we have conducted a comprehensive study on the effects of memory energy management on synchronization characteristics using one large commercial multithreaded application, **IBM DB2**, and six SPLASH/SPLASH-2 parallel applications and found that energy management can significantly increase the synchronization waiting time, indicating that tracking the cascading synchronization slowdown caused by energy management is critically important to provide performance guaranteed energy conservation for multithreaded applications. Based on our understanding of the complex effects of energy adaptation-induced slowdowns on thread synchronizations, we have proposed three Synchronization-Aware (SA) slowdown estimation algorithms, LWT, CSL

and ODP, to estimate individual thread slowdowns and three aggregation methods, MAX, AVG, and SUM, to estimate overall application slowdowns from individual thread slowdowns.

Our results on a detailed full system simulator that runs a real operating system (Solaris) with IBM DB2/OLTP and six SPLASH/SPLASH-2 applications showed that our SA algorithm LWT can effectively bound performance degradation within a specified limit and also conserves the most energy. Without the SA methods, the two slowdown aggregation methods, MAX and AVG, alone are not be able to provide performance guarantees in many cases. SUM can guarantee performance but saves much less energy than other schemes such as LWT+MAX.

Even though our work focuses only on memory energy management for multithreaded applications on multiprocessor architectures, the insights and the main ideas from this work are also applicable to energy adaptation of other components, such as processors. For example, currently most existing work on processor energy adaptation only considers the slowdown for each individual thread in an isolated way without taking the cascading synchronization slowdowns into account. Our synchronization-aware slowdown estimation algorithms and slowdown aggregation methods can be revised for processor energy management.

Additionally, our main ideas and our performance guaranteeing methods are also useful for other performance-related tradeoffs, such as performance-reliability tradeoff. For example, the selective reliability management scheme [16] induces performance slowdowns while replicating instructions to detect hardware errors. Therefore, our method dynamically tracking slowdowns and guaranteeing performance can definitely benefit such schemes to achieve a balance between performance and target functionalities.

9. ACKNOWLEDGEMENTS

This research has been supported by NSF CAREER Awards CNS-0347854 and CCR-0238182, NSF CCR-03-12286, NSF EIA 02-24453, NSF CCF-0541188 and IBM SUR Grant.

10. REFERENCES

[1] Intel Thread Profiler, http://www.intel.com/software/products/threading/tp/

[2] D. H. Albonesi. Selective Cache Ways: On-demand Cache Resource Allocation. In *Proceedings of the 32nd International Symposium on Microarchitecture*, Nov. 1999

[3] R. Hankins, T. Diep, M. Annavaram, B. Hirano, H. Eri, H. Nueckel, and J. P. Shen. Scaling and Characterizing Database Workloads: Bridging the Gap between Research and Practice. In *Proceedings of the 36th International Symposium on Microarchitecture*, 2003

[4] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proceedings of the 39th International Symposium on Microarchitecture*, 2006

[5] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power Aware Page Allocation. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000

[6] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler and T. W. Keller. Energy Management for Commercial Servers. In *IEEE Computer*, Dec. 2003

[7] J. Li, J. F. Martinez, and M. C. Huang. The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors. In *Proceedings of International Symposium on High Performance Computer Architecture*, Feb. 2004

[8] T. Li, A. R. Lebeck, and D. J. Sorin. Spin Detection Hardware for Improved Management of Multithreaded Systems. In *IEEE Transactions on Parallel and Distributed Systems*, Vol.17, No.6, June 2006

[9] X. Li, Z. Li, P. Zhou, Y. Zhou, S. Adve, and S. Kumar. Performance-Directed Energy Management for Storage Systems. In *IEEE Micro Special Issue on Top Picks from Microarchitecture Conference*, Dec. 2004

[10] 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. In *IEEE Computer*, 2002

[11] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. In *Computer Architecture News*, 2005

[12] D. McWherter, B. Schroeder, A. Ailamaki, and M. Harchol-Balter. Priority Mechanisms for OLTP and Transactional Web Applications. In *Proceedings of the 20th International Conference on Data Engineering*, 2004

[13] F. Moore. More Power Needed. In *Energy User News*, Nov. 2002

[14] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. In *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996

[15] Rambus, http://www.rambus.com

[16] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled Fault Tolerance. In *ACM Transactions on Architecture and Code Optimization*, Vol.2, No.4, Dec. 2005

[17] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995

[18] M. Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for Reduced CPU Energy. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*, Nov. 1994