

Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks

Tali Moreshet*, R. Iris Bahar* and Maurice Herlihy†

*Brown University, Division of Engineering, Providence, RI 02912

†Brown University, Department of Computer Science, Providence, RI 02912

{tali,iris}@lems.brown.edu, mph@cs.brown.edu

Abstract—One important way in which multiprocessors differ from uniprocessors is in the need to provide programmers the ability to synchronize concurrent access to memory. Transactional memory was proposed as a way of improving throughput especially when the rate of synchronization conflict is low. In this paper we explore power implications of transactional memory on standard and synthetic benchmarks. We propose a new “serial execution” mode that lowers energy consumption during high contention periods by reducing transaction throughput. We conclude that transactional models are a promising approach to low-power synchronization, and serial execution strengthens the energy advantage, but that further work is needed to fully understand how transactions compare to locks at high levels of contention.

I. INTRODUCTION

Energy consumption is an increasingly important issue in multiprocessor design. The need for energy-aware systems is obvious for mobile systems, where low energy consumption translates to longer battery life, but it is also important for desktop and server systems, where high energy consumption complicates power supply and cooling.

While energy consumption in uniprocessors has been the focus of a substantial body of research, energy consumption in multiprocessors has received less attention. This issue is becoming increasingly important as multiprocessor architectures migrate from high-end platforms into everyday platforms such as desktops, laptops, and servers. In particular, the increasing availability of multi-threaded and multi-core machines means that we can expect multiprocessors to replace uniprocessors in many low-end systems. Past studies have estimated that on-chip caches are responsible for at least 40% of the overall processor power (e.g., [2]). However, the dominant portion of energy consumption in the memory hierarchy is due to off-chip caches, resulting from their significantly larger size, and the higher capacitance board buses.

The principal way in which multiprocessors differ from uniprocessors is in the need to provide programmers the ability to synchronize concurrent access to memory. When multiple threads access a shared data structure, some kind of *synchronization* is needed to ensure that concurrent operations do not interfere. The conventional way for applications to synchronize is by *locking* [5]: for each shared data structure, a designated bit in shared memory (the *lock*) indicates whether the structure is in use.

Nevertheless, conventional synchronization techniques based on locks have substantial limitations [9]. Coarse-

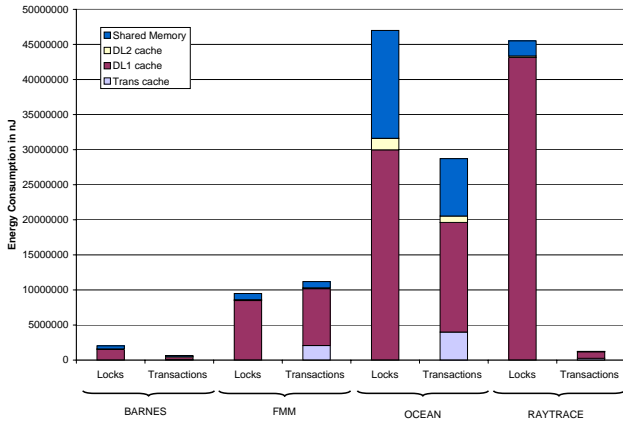
grained locks, which protect relatively large data structures, simply do not scale. Threads block one another even when they do not really interfere, and the lock itself causes memory contention. Fine-grained locks are more scalable, but they are difficult to use. In particular, they introduce substantial software engineering problems, as the conventions associating locks with objects become more complex and error-prone. Locks are also vulnerable to thread failures and delays; if a thread holding a lock is delayed by a cache miss, page fault, or context switch, other running threads may be blocked.

Transactional memory [10] is a synchronization architecture that addresses these limitations. A transaction is a finite sequence of memory reads and writes executed by a single thread. Transactions are *atomic* (each transaction either completes and commits, or aborts) and are serializable. Hardware transactional memory proposals (e.g., [7], [10], [17], [21], [22], [23], [28]) exploit hardware mechanisms such as speculative execution and on-chip caching. Hardware optimistically executes a transaction and locally caches memory locations read or written on behalf of the transaction, marking them transactional. The hardware cache coherence mechanism communicates information regarding read and write operations to other processors. A data conflict occurs if multiple threads access a given memory location via simultaneous transactions and at least one thread’s transaction writes the location. A transaction commits and atomically updates memory if it completes without encountering a synchronization conflict.

Transactional memory was originally proposed as a means of increasing throughput and improving ease of programming relative to locks. Although it seems plausible that transactions may be more energy-efficient than locks, the precise tradeoffs are not clear, especially under high rates of conflict. In this paper, we consider the energy/performance tradeoffs associated with these two approaches to multiprocessor synchronization. We conclude that transactional models are a promising approach to low-power synchronization, but that further work is needed to fully understand how transactions compare to locks when synchronization conflict rates are high.

To compare our hardware transactional memory to a baseline system that uses locks, we started by running the SPLASH-2 benchmark suite [30] since it is the most commonly used benchmark suite for parallel applications. We arbitrarily selected a few benchmarks, ran each benchmark

Fig. 1. Energy consumption of a splash2 benchmarks using locks vs. transactions.



after its initialization stage for 200 locks, and then ran it again with locks replaced by transactions. When replacing the lock with a transaction, the critical section defines the bounds of a transaction.

Figure 1 shows the energy consumption resulting from cache and memory accesses when synchronization was handled with either locks or transactions. We see that in most cases replacing locks with transactions reduced the number of cache and memory accesses, thereby reducing the energy consumption. With this initial analysis, it appears that transactions do indeed have a large benefit over locks in terms of energy consumption. However, we have found that the SPLASH-2 benchmarks do not test our assumptions well when the system is operating under high contention. In Section IV we will discuss these results further, and show that these initial results do not allow for a complete comparison.

Synchronization conflicts cause transactions to abort and restart, causing the system to consume energy doing useless work. Motivated by this tradeoff, in this paper we propose a *serial execution mode* for transactional memory in which transactions are adaptively serialized at the hardware level with the intent of decreasing energy consumption at the possible cost of degraded throughput. We note that in previous work [20] we presented an initial investigation of this topic and showed a single case of using locks vs. transactions as a motivation for the advantage of transactions over locks for energy consumption. Here we extend that work by providing a more detailed analysis into the various tradeoffs.

The rest of the paper is organized as follows: Section II introduces the concept of transactional memory and its variations; Section III describes our experimental setup; Section IV discusses our obtained results; Section V concludes our work.

II. MULTIPROCESSOR SYNCHRONIZATION

As described in the introduction, locks are the most commonly used approach to synchronize concurrent threads. A thread must acquire a lock before accessing a shared object, and release the lock when it is done. Methods to acquire and release locks are typically provided by the operating system,

relying on a read-modify-write instruction such as test-and-set, compare-and-swap, or load-linked/store-conditional. Here we are concerned with an issue that has received relatively little attention: the effect of locking on energy consumption. A lock is physically represented by a field in shared memory. A thread locks an object by testing the field, and if it finds the object is free, setting the field to indicate the object is in use. If the lock is found to be busy, the thread will repeatedly access shared memory until the lock becomes free.

Off-chip memory accesses in multiprocessor systems incur large latencies resulting from the actual memory access as well as interconnect and bus latencies. Since the off-chip memory is shared by multiple processors, contention for the busses and pins may have an additional impact on latency [15]. To maximize throughput, it is therefore beneficial to minimize the number of accesses to off-chip memory.

Transactional Memory has been championed as an alternative to locks both from the software side, in the form of software transactional memory [9], [8] and from the hardware side (e.g., [22], [17], [21], and [25]). Our transactional memory model is loosely based on the original hardware transactional memory proposal [10]. In our model, each processor includes in addition to its main DL1 on-chip cache, a secondary cache called the *transactional cache*. The transactional cache is a smaller, fully-associative cache that is exclusive to the main DL1 cache.

A. Conflict Resolution

Hardware transactional memory requires the ability to roll back and restart transactions when synchronization conflicts are detected. In both lock and lock-free synchronization, it is common to assume that synchronization conflicts are rare. (Barriers are perhaps the only exception to this rule.) Locks are typically optimized for uncontended use, perhaps by making contended use more expensive. In most applications, locks rarely conflict, although there are occasional “hot-spots” that tend to conflict repeatedly. For transactions, short transactions are even less likely to conflict than locks, since contention on the lock itself is eliminated, and only true synchronization conflicts on the data remain. Nevertheless, when transactions have different lengths, longer transactions will tend to encounter more conflicts than short ones.

While some previous work on transactional memory considered conflict rates in general, they did not consider different conflict scenarios, and avoided the aspect of implementing roll-back capabilities [18], [4]. We are only aware of one previous work (presented in [25]) that tried to specifically force high conflict rates, for the purpose of comparing two lock-free schemes in terms of performance. However, energy consumption was not considered in that work.

In the case of conflicting transactions, once it aborts, a transaction needs to roll back its state and restart from a checkpoint. Roll-back and re-execution of transactions is different than re-issuing after a branch misprediction. When a branch is discovered to be mispredicted, all wrong-path instructions are still pending in the pipeline, and none of them are retired.

In contrast, the scope of transactions and their nature is such that when a transaction aborts many instructions that already retired and released their buffer slots and physical registers may need to be re-issued. It is therefore essential that we have some means of checkpointing the state of the system when entering a transaction, so that we can recover from a conflict if required. For this purpose, we could use one of the previously proposed methods of recovering the state of the processor, such as Checkpoint Processing and Recovery (CPR) [1], or SatyNet [27].

B. Energy Consumption in the Memory Hierarchy

A *contention manager* [12] is a software module that determines how to resolve synchronization conflicts. Typically, when one transaction discovers it is about to conflict with another the contention manager decides whether the transaction should proceed, aborting the other transaction, or whether it should wait for a brief duration, giving the other a chance to finish¹. Contention managers have used a variety of strategies, ranging from exponential back-off [10], to timestamps [22], or taking turns [12]. These policies target transaction throughput, not energy consumption. The serial execution mode we introduce is also a form of contention management, but intended to address energy consumption, and implemented at a lower level.

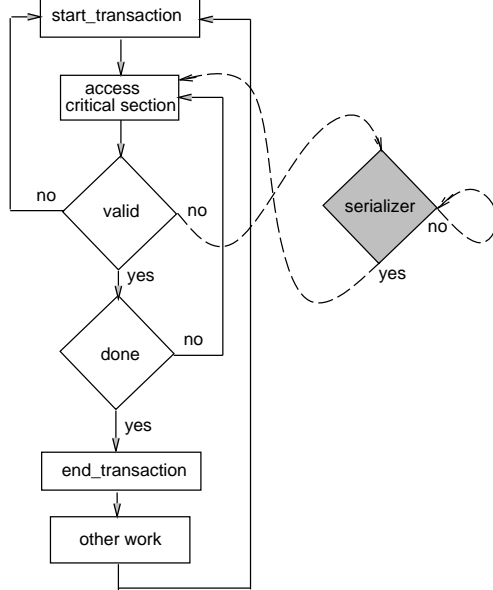
The memory hierarchy is responsible for a significant fraction of the energy consumption of microprocessor systems. Consider a set-associative cache. The main sources of energy in an m -way set associative cache are those resulting from precharging the bit-lines, reading or writing data, assertion of the word-line, and driving external busses. The energy consumption of a single cache is thus a factor of its implementation (SRAM vs. DRAM), its size, the capacitance of the busses it drives (which depend on whether the cache is on or off-chip), as well as the number of read and write accesses. The dominant portion of energy consumption per access in the memory hierarchy is due to off-chip caches, resulting from their significantly larger size, and the higher capacitance board buses [3], [6], [11]. The largest and most remote memory in a multiprocessor system is the shared memory. An important factor in reducing the energy consumption of the memory hierarchy is therefore minimizing the number of shared memory accesses.

Very little past work was dedicated to power reduction in a shared-memory multiprocessor from the system level. The *thrifty barrier* was proposed by Li *et al.* [14] to reduce the energy of spinning on a barrier. In addition, Renau *et al.* [24] showed that thread-level speculation can be energy efficient, provided that tasks needed to be squashed infrequently.

In this work, we target a general multiprocessor system not bound to a specific configuration. A multiprocessor system is generally composed of a series of uniprocessors, each with its own local on-chip cache. Any number of processors may

¹In a hardware implementation, transactions only know of a conflict after the fact, so waiting for the other transaction to finish is not an option.

Fig. 2. Serialized transactions on a high conflict rate.



share a larger cache at the next level. The nature and number of caches in the memory hierarchy of every processor depends on the specific system. However, any large multiprocessor system includes more than a single uniprocessor (or multiprocessor) chip, and thus, in order to allow all the processors in the system to synchronize and share data, they all need to have access to a shared memory. For many of the processors this memory will be located on a remote chip, and we therefore consider it an off-chip memory.

As mentioned, locks require several accesses to the shared memory for acquiring and releasing the lock, regardless of the contention level. When replacing locks with transactions, those additional shared memory accesses are eliminated if no conflicts occur. In the case of conflicts, however, the conflict resolution schemes that involve back off and re-trial target performance but have a negative impact on energy consumption. Assuming we are in a high conflict section of execution, this re-execution is likely to cause recurring conflicts. A transaction that conflicts essentially wastes energy producing unusable results. Our modified version of transactional memory resolves conflicts by resorting to serial execution. That is, when a conflict is detected, the system state is rolled back, as usual. However, instead of attempting the transactions again in parallel (after backing off), we force all conflicting transactions to execute serially.

Figure 2 shows the flow of transactions, with our proposed serial execution approach added in shaded blocks and dotted lines. In this Figure, the validity of the transactional execution is verified while accessing the critical section. If a non-valid execution is detected, the transaction is rolled-back and re-executed. With our serial execution scheme, a conflict will still result in rolling-back the transaction. However, during high contention, instead of attempting to re-execute transactions after a conflict, the serializer will wait for one transaction

to complete before allowing another to re-execute. Those transactions that conflicted will be issued serially, and then the system will return to the standard method of issuing transactions in parallel. The serializer implementation requires a very simple state machine in hardware.

III. SIMULATION MODEL

Our simulations are run using Simics from Virtutech [16]. Simics is a full system simulation platform, capable of simulating high-end target systems running full operating systems and commercial workloads [29]. We use a version of Simics which models Sun Microsystem’s Sun Enterprise 3500-6500 class of servers, which can be configured to run up to 30 UltraSPARC II processors, and 60GB of memory. We use a system running the Linux operating system.

Implementing the transactional memory hardware includes adding some means of rolling back execution and re-issuing transactions in case of a conflict. Although Simics’ limitations do not allow us to implement a realistic roll-back of the processor state after instructions are retired, it does provide a checkpoint mechanism, that allows taking a checkpoint of the entire simulated system state, and then restarting simulation from the checkpoint.

Our initial configuration includes 4 processors, each with its own DL1 cache as well as its own unified data/instruction L2 on-chip cache, and all processors sharing a single off-chip memory. Selecting different configurations would not change the nature of our results, since regardless of the specific memory hierarchy, accesses to shared data will end up in shared memory. Table I summarizes the system configuration.

The energy estimates for the different memory hierarchy components are listed in the third column of Table I. These numbers were extrapolated from Micron SDRAM power calculator [13], from CACTI [26], as well as from a conversation with high-end multiprocessor designers from industry. We assume the processor clock is 1GHz and the SDRAMs is 133MHz. The shared memory energy is the sum of the I/O of the processor Front Side Bus, the SDRAM pins, and the actual SDRAM access. The load/store activity ratio is considered in our approximation. These numbers are meant to illustrate the relative difference between accesses to different levels of the memory hierarchy.

IV. EVALUATION

Our goal is to simulate realistic conflict scenarios that were not tested in the past. For this purpose, we are using a realistic simulation model that models an operating system as well as a relatively detailed processor microarchitecture. This simulation platform makes simulations very slow, forcing us to resort to using small simulation examples and extrapolating from them. As we showed in Section I, running examples from the SPLASH-2 benchmark suite did give us some initial insight into how energy consumption of locking vs. transactional memory scheme compare. However, the evaluation is not complete since it does not test the system under high contention situations.

TABLE I
SYSTEM CONFIGURATION AND ENERGY CONSUMPTION PER
CACHE/MEMORY ACCESS.

Machine Width	4-wide fetch, issue, commit	
L1 DCache	8KB 4-way; 32B line; 3 cycle latency	0.47 nJ
Transactional Cache	64-entry fully associative	0.12 nJ
L2 Cache	128KB 4-way; 32B line; 10 cycle latency	0.9 nJ
Memory	256MB; 200-cycle latency; 64-bit bus	33 nJ

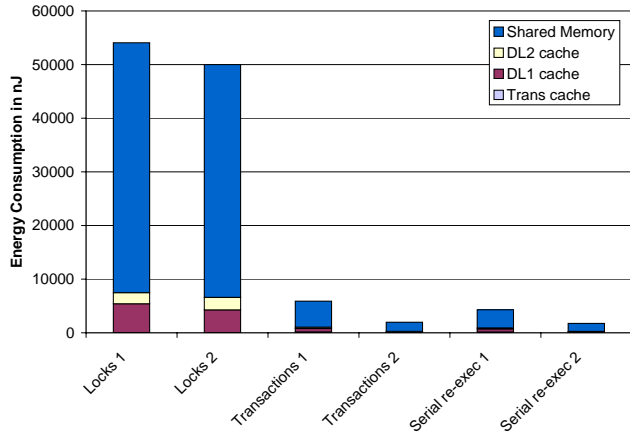
A. Standard Benchmark Evaluation

The locks in the SPLASH-2 benchmarks are PARMACS library locks, which use test-and-test-and-set locks that spin the lock for one loop count and then yield the processor, involving the operating system. As shown in Figure 1, in most cases, replacing locks with transactions reduced the number of cache and memory accesses. In absence of conflicts, replacing the locks with transactions saves unnecessary memory accesses associated with the lock. In addition, when executing a lock, all previous memory operations are allowed to complete, before the lock can be executed. Then, a test-and-set is executed, along with some additional overhead.

The exception is the *finn* benchmark. In this case, transactions and locks have a similar number of cache and memory accesses, and since transactions also incur transactional cache accesses, the overall energy consumption is slightly higher with transactions than with locks. Although this benchmark included a slightly larger number of conflicts than the other SPLASH-2 benchmarks, the conflict rate is still very low. The number of shared memory accesses was very low to begin with, leaving small room for energy reduction when using transactions. Due to the low conflict rate, using serial execution would not make a noticeable difference in this case.

We initially chose SPLASH-2 because it has become a standard to some extent, having been used by other papers to evaluate transactional memory implementations [22], [7]. However, we note two problems we experienced with the SPLASH-2 benchmarks. First, since locks involve operating system interference, as well as waiting for memory accesses and additional overhead, it sometimes causes the processor to execute other code during this idle time. Having useful work done while waiting on a lock is possible in highly parallel sections of the code, and is a feature found in realistic applications. Nevertheless, this feature makes it difficult to compare execution of locks to transactions. Second, although SPLASH-2 is a parallel application suite, and most of its benchmarks include locks, they are designed to avoid conflicts. A low contention rate in SPLASH-2 was also found by Moore *et al.* [19]. We found that the SPLASH-2 benchmarks had very few, if any, conflicting transactions. In fact, very few of the transactions actually ran in parallel. This constraint limits what

Fig. 3. Energy Consumption of microbenchmarks 1 and 2 using locks vs. transactions (labeled 1 and 2 respectively).



we can learn from these benchmarks. To compare the energy consumption of locks and transactional memory at high levels of contention, a different set of benchmarks is needed.

B. Conflict Evaluation

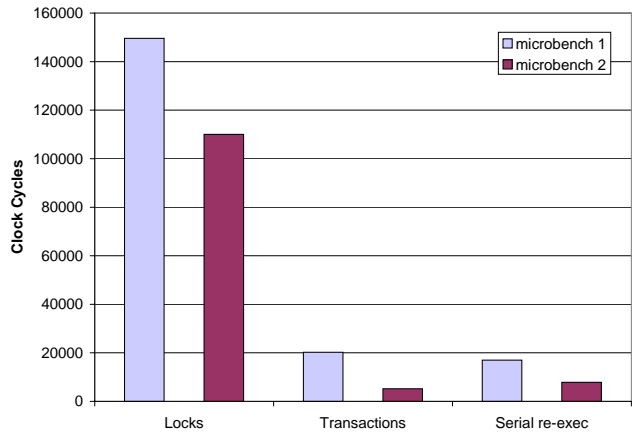
As noted above, the SPLASH-2 benchmarks appear to have been designed to avoid contention. Naturally, this is not to say that contention does not exist in the real world, but rather, that existing standard benchmarks tend to ignore this issue. We expect that many realistic applications such as database lock managers or language runtimes (CLR or Java™ virtual machines) probably do display intervals of high contention. Unfortunately, we found no standard, public-domain benchmarks that are well-suited to test relative energy consumption of locks and transactional memory at high levels of contention, so we had to write our own benchmarks.

Instead of tweaking the SPLASH-2 benchmarks to artificially show more contention, we created our own synthetic microbenchmark. Our microbenchmark has 4 threads, all accessing a shared array. Each array entry is protected by a dedicated lock (which then defines a transaction). We used the pthread library locks. We take a checkpoint before the beginning of each transaction to recover from potential conflicts. The microbenchmark allows us to control the frequency and nature of conflicts, and thus simulate different conflict scenarios with different microbenchmark configurations.

For each microbenchmark configuration, we run the microbenchmark three times. First, it was run using locks, next using transactions, and finally with our modified version of transactional memory, using serial execution.

The bars labeled with ‘1’ in Figure 3 show the energy consumption of each of the benchmark runs in its initial configuration. The first bar shows memory accesses for the microbenchmark using locks, the third bar for the microbenchmark using transactions, and the fifth bar for our serial execution scheme. Shared memory accesses dominate energy consumption, making the locks simulation an unattractive energy-aware solution. Executing transactions in serial execution mode consumes less energy than executing them

Fig. 4. Clock cycles of microbenchmarks 1 and 2 using locks vs. transactions.



without such contention control. The bars labeled *microbenchmark 1* in Figure 4 show the performance of each of the benchmark runs reported above, in terms of clock cycles. A shorter run displays better performance. For microbenchmark 1, transactions exceed locks, in terms of both energy and performance. Switching to serial execution mode as soon as a conflict is detected improves both energy and performance. For this particular microbenchmark, running transactions without serial execution mode generated more re-executions, which eventually resulted in more conflicts. These conflicts, in turn, led to more wasted instruction execution and an increased number of cycles to complete the simulation. On the other hand, if the re-executions had not resulted in conflicts, this would have allowed an earlier commit of transactions and better performance (in terms of total cycles executed).

We note that the simulated transaction executes user-level (that is, microbenchmark) instructions only, while the locking simulation also executes operating system code as part of the standard locking library (once the processor is yielded, following a short spin). We compared transactions to standard locks which require support from the OS. While locks can probably be optimized for energy in various ways (via special quiesce or sleep instructions, or platform-specific optimization), there is no standard way of modeling such optimizations, and no standard optimized locking package currently available to programmers. In this work, we do not claim to analyze the best of all conceivable lock implementations (an unrealistic goal), but only to compare the standard, almost universally-used locking libraries to transactions. Inventing new, energy-aware locking protocols is beyond the scope of this paper.

Depending on the alignment of the transactions, in terms of performance, it may be better to avoid serial execution mode. Assuming repeated conflicts will not occur, transactions may have a chance of committing earlier. This can be seen in the next set of microbenchmark runs. We reconfigured the microbenchmark, by adjusting the length, rate and duration of accesses to the array, in order to simulate a variety of conflict scenarios. The bars labeled with ‘2’ in Figure 3 and those labeled *microbenchmark 2* in Figure 4 show similar results for

the second configuration of the microbenchmark. In this case, the first occurrence of conflicting transactions is not followed by repetitive conflicts. Therefore, the serial execution mode proves to be too conservative, and thus incurs a penalty in performance. However, if lower energy consumption has a higher priority, then following a policy of serial execution of transactions in a situation where a high conflict rate is likely makes for a more appropriate choice. Overall, serializing transactions following a conflict may hurt performance by preventing parallelism, but reducing speculation results in lower energy consumption. The optimal point of switching over from speculative parallelism to serial execution, in order to maximize throughput and minimize energy consumption, is application-dependent. The above results demonstrate that such an advantage exists, and stress the benefit of serial execution of transactions.

To summarize, for locks, the cost of synchronization depends on the number of locks in the application. For transactions, it depends on the conflict rate and the type of checkpointing used. Therefore, transactional energy consumption depends on the specific scenario. Our results illustrate this claim. Also, when it comes to synchronization, cache and memory are the dominant energy components; therefore we focus on them when quantifying energy consumption.

V. CONCLUSION AND FUTURE WORK

We have taken a first step in evaluating the energy costs of two approaches to multiprocessor memory synchronization: transactional memory versus locking. The behavior of these synchronization approaches is highly dependent on the system contention level: When conflicts are rare, transactions have an advantage over locks in terms of performance as well as energy due to fewer accesses to main memory. As conflicts become more common, however, the cost of transaction re-executions becomes a significant drawback in terms of energy.

By simulating several of the SPLASH-2 benchmarks, we were unable to produce high contention levels. To investigate the behavior of high-contention applications, we therefore devised a simple synthetic benchmark in which the level of contention can be easily “tuned” to different levels. We used this benchmark under various configurations to test the high-contention mode for transactional memory.

Our results open a range of further issues. Neither transactional memory nor standard locking code were designed with energy consumption in mind, and the design of energy-aware synchronization mechanisms remains a largely unexplored area. Our results suggest a promising energy-aware approach to handling synchronization in shared-memory multiprocessors: use speculative synchronization via transactions for the majority of low-contention program executions, switching to enforced serialization when contention is high. Forcing serialization may reduce overall system throughput, but is advantageous in terms of energy. We provide a first step in understanding these previously unexplored issues, although further work is needed to obtain a more complete analysis.

REFERENCES

- [1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *36th Intl. Symposium on Microarchitecture*, December 2003.
- [2] D. H. Albonesi. Selective cache ways: On-demand cache resource allocation. In *32nd Intl. Symposium on Microarchitecture*, November 1999.
- [3] R. I. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *Proceedings of the Intl. Symposium on Low Power Electronics and Design*, August 1998.
- [4] J. W. Chung, H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, February 2006.
- [5] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1:115–138, 1971.
- [6] R. Fromm, S. Perissakis, N. Cardwell, C. Kozyrakis, B. McGaughey, D. Patterson, T. Anderson, and K. Yelick. The energy efficiency of IRAM architectures. In *24th Intl. Symposium on Computer Architecture*, June 1997.
- [7] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabh, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *31st Intl. Symposium on Computer Architecture*, June 2004.
- [8] T. Harris and K. Fraser. Language support for lightweight transactions. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2003.
- [9] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Symposium on Principles of Distributed Computing*, July 2003.
- [10] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *23rd Intl. Symposium on Computer Architecture*, May 1993.
- [11] M. Huang, J. Renau, S.-M. Yoo, and J. Torrellas. A framework for dynamic energy efficiency and temperature management. In *33rd Intl. Symposium on Microarchitecture*, December 2000.
- [12] W. N. S. III and M. L. Scott. Contention management in dynamic software transactional memory. In *Workshop on Concurrency and Synchronization in Java Programs*, April 2004.
- [13] J. W. Janzen. *SDRAM Power Calculation Sheet*. Micron, 2001. <http://www.micron.com/products/dram/sdram/partlist.aspx?density=512Mb>.
- [14] J. Li, J. Martinez, and M. Huang. The thrifty barrier: Energy-efficient synchronization in shared-memory multiprocessors. In *10th Intl. Symposium on High-Performance Computer Architecture*, February 2004.
- [15] C. Liu, A. Sivasubramanian, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *10th Intl. Symposium on High-Performance Computer Architecture*, February 2004.
- [16] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, pages 50–58, February 2002.
- [17] J. F. Martinez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.
- [18] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, February 2006.
- [19] K. E. Moore, M. D. Hill, and D. A. Wood. Thread-level transactional memory. Technical Report 1524, Computer Sciences Dept., University of Wisconsin, March 2005.
- [20] T. Moreshet, R. I. Bahar, and M. Herlihy. Energy reduction in multiprocessor systems using transactional memory. In *Intl. Symposium on Low Power Electronics and Design*, August 2005.
- [21] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *Proceedings of the Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

- [22] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.
- [23] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *32nd Intl. Symposium on Computer Architecture*, June 2005.
- [24] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, , and J. Torrellas. Thread-level speculation on a CMP can be energy efficient. In *Intl. Conference on Supercomputing*, June 2005.
- [25] P. Rundberg and P. Stenström. Speculative lock reordering: Optimistic out-of-order execution of critical sections. In *IEEE Intl. Parallel and Distributed Processing Symposium*, April 2003.
- [26] P. Shivakumar and N. P. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical report, Compaq Western Research Laboratory, 2001/2.
- [27] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *29th Intl. Symposium on Computer Architecture*, May 2002.
- [28] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, November 1993.
- [29] Virtutech. *Simics*. <https://www.simics.net>.
- [30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *22nd Intl. Symposium on Computer Architecture*, pages 24–36, June 1995.