Energy Reduction in Multiprocessor Systems Using Transactional Memory*

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

The emphasis in microprocessor design has shifted from high performance, to a combination of high performance and low power. Until recently, this trend was mostly true for uniprocessors. In this work we focus on new energy consumption issues unique to multiprocessor systems: synchronization of accesses to shared memory. We investigate and compare different means of providing atomic access to shared memory, including locks and lock-free synchronization (i.e., transactional memory), with respect to energy as well as performance. We show that transactional memory has an advantage in terms of energy consumption over locks, but that this advantage largely depends on the system architecture, the contention level, and the policy of conflict resolution.

Categories and Subject Descriptors: C.1.2[Computer Systems Organization] Processor Architectures: Multiprocessors

General Terms: Design

Keywords: Multiprocessor, power, transactional memory

1. INTRODUCTION

Reducing energy consumption is a pressing issue for many high-performance microprocessor systems. However, while this issue has been studied extensively for uniprocessors, little research has been targeted at reducing energy consumption in large multiprocessor systems. Multiprocessors are becoming increasingly common, both in the form of chip multithreading and chip multiprocessors. Moreover, multiprocessors are increasingly used not only for servers, but also for desktops and smaller devices.

Although many energy-consumption issues that apply to uniprocessors apply to multiprocessors as well, multiprocessors involve new energy issues. In this work we investigate the energy consumption resulting in one of the principal ways in which multiprocessors differ from uniprocessors: synchronizing accesses to shared memory. Memory accesses compose a large fraction of the overall energy consumption both in uniprocessor and multiprocessor systems. For example, consider a simple bus-based multiprocessor system consisting of several processors, each with its own on-chip L1 and L2 caches, and all processors sharing a single off-chip main memory. If a processor misses in its L1 and L2 caches, it obtains the data from main memory or from another processor's cache through an invalidate operation. The number of concurrent accesses to a single shared memory block determines the system contention level. In addition, synchronization of shared memory accesses increases the number of memory accesses in a multiprocessor system, regardless of the contention level.

In this paper, we investigate different means of providing atomic access to shared memory, and their benefits and drawbacks in terms of energy consumption. We focus in particular on *transactional memory*, an alternative to locks that provides lock-free synchronization and has become a popular research area. Transactional memory was originally proposed as a better solution than locks to shared-memory synchronization, since it provides ease of programming and avoids the problems of serialization and deadlocks involved with locks [7]. It was shown to also have a performance advantage over locks. In this work we are also interested in the energy aspects of transactional memory and analyze whether it still has an advantage over locks when energy, performance and ease of programming need to be considered together.

The rest of the paper is organized as follows: Section 2 gives an overview of energy consumption related issues in recent work; Section 3 introduces the concept of transactional memory and its variations; Section 4 describes our design, experimental setup and evaluation. Section 5 concludes our work.

2. OVERVIEW AND RELATED WORK

Energy consumption is a pressing issue for processing systems in general. While energy consumption is a well-known issue for mobile systems, where lower energy consumption enables longer battery life, it is also an issue for stationary systems since they incur problems of power delivery, high power densities and cooling.

The memory hierarchy is one of the main sources of power dissipation in superscalar uniprocessors. Although power dissipation in uniprocessor systems has been well examined, very little work has been dedicated to power reduction in a shared-memory multiprocessor from the system level. Of the work related to multiprocessors, Ekman *et al.* [4] studied the power and performance variation between chip multiprocessors that tradeoff a single wide issue superscalar processor for many narrow-issue cores. While a moderate number of cores with moderate issue widths was found to be optimal in terms of performance of multi-threaded applications, the power dissipation was found to be constant across the different systems. Wide-issue cores had higher core power and systems with many cores had higher power dissipation in the memory hierarchy.

Of particular interest to us is the work of Li *et al.* [10], energy reduction in the synchronization of shared memory multiprocessors by using a variation of barrier synchronization. A barrier is used to synchronize parallel threads such that no thread may execute past the barrier before all other threads reached the barrier. Threads gen-

erally spin-wait at the barrier for all other threads. They propose the *thrifty barrier* to reduce the energy of spinning on a barrier. When an early arriving thread reaches the thrifty barrier, a software predictor is used to decide in which low power mode to place the CPU. Then the modified hardware of the cache controller wakes up the thread in time to resume execution. Thus, the energy wasted in the barrier spin loops is reduced. To the best of our knowledge, this is the only work that addresses energy concerns related to memory contention. While this approach appears quite promising, it proposes improvements to the way a processor responds to the spin loop produced by a barrier, whereas we are proposing alternatives to the spin loop itself.

3. CONTENTION IN SHARED MEMORY

In a shared memory multiprocessor system, locks are traditionally used to provide atomicity and mutual exclusion. Before a thread can enter a critical section (i.e., obtain exclusive access to a shared variable), it must acquire a lock. After the thread leaves the critical section it must release the lock, allowing other threads to acquire it. Locks are implemented using test-and-set, compare-and-swap, or load-linked/store-conditional instructions, and are relatively easy to implement. However, since a lock is physically represented by a field stored in shared memory every attempt to acquire a lock requires a shared memory access to read the lock, followed by another memory access to write the lock, assuming that it was found to be free. If the lock read from memory is found to be busy, repetitive memory accesses are required to read the lock (test it, if using test-and-set, or comparing it, if using compare-and-swap), until it is eventually found to be free. This increased memory traffic results in high memory contention, thereby reducing program throughput and performance and increasing energy consumption. In addition to numerous memory accesses, locks also introduce software engineering problems. In order to maximize parallelism, fine-grained locks are used, which minimize the scope of the lock. Fine-grained locks are difficult to use, since the programmer must ensure mutual exclusion as well as deadlock freedom. Finally, locks are conservative, forcing a thread to acquire a lock even if contention is very low and a conflict is unlikely.

As an alternative to locks, *Transactional Memory* was first proposed several years ago by Herlihy *et al.* [7], and concurrently by Stone *et al.* [18]. Transactional memory has subsequently been studied both from the software side, in the form of software transactional memory [6], [5] and from the hardware side [14], [12], [13], and [15].

A transaction is defined by the scope of a lock. Each transaction is executed speculatively by a single thread without acquiring a lock. The execution of the transaction is optimistic, and if it completes without conflicts, it will *commit* and no further action is required. Otherwise, if conflicts were detected during the execution of the transaction, the transaction will *abort* and its effects will be discarded. If a transaction aborts, it is required to be rolled back and re-issued. The roll-back mechanism is required to guarantee forward progress, and thus this process will eventually result in an atomic commit of the transaction. Until a transaction successfully commits, its effect will not be apparent to other threads.

Hardware transactional memory keeps memory entries which were modified within a transaction in local cache. These entries are not visible to other threads until the transaction commits, at which point they may be written back to the shared memory. Any attempt of other threads to access a transactional memory entry in local cache will cause the transaction to abort. When using hardware transactional memory, the scope of a transaction is limited by the size of the local cache.

Software transactional memory, on the other hand, uses software support to manage the modified entries within a transaction. In [6], the transactional memory is implemented by a Java package, allowing transactional objects to be created dynamically. The package also includes a contention manager that is responsible for conflict resolution. The main advantage of the software transactional memory implementation is that it is independent of hardware resources, and allows transactions to be almost unlimited in size. Its main disadvantage, however, is the performance overhead involved with using software support for the implementation.

While hardware transactional memory shows superior performance to that of the software transactional memory, it is limited by the availability of hardware, which is more costly. Very recently Ananian *et al.* [2] introduced *unbounded transactional memory*, which allows transactions that do not fit in the cache to overflow into a data structure kept in main memory. This model relies largely on operating system support to enable transactions with size limited only by the size of the virtual memory. All the above approaches proposed for handling contention in shared-memory multiprocessor systems focus either on ease of programming or performance (or perhaps a combination of the two). While both these aspects are important, they ignore the impact on energy consumption in employing these different approaches.

4. DESIGN

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], [8]. The largest and most remote memory in a multiprocessor system is the shared memory. Therefore, an important factor in reducing the energy consumption of the memory hierarchy is minimizing the number of shared memory accesses.

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 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 most of the processors this memory will be located on a remote chip, and we therefore consider it an off-chip memory.

4.1 From Locks to Transactions

In order to run an application that uses locks on a transactional memory environment, locks need to be replaced with transactions. This can be achieved by replacing *lock(x)* instructions with *start_transaction*, and *unlock(x)* instructions with *end_transaction*. No nested transactions are allowed. Since nested locks will now belong to an existing transaction, they will be treated as part of the atomic region defined by the outer transaction. Replacing lock instructions with start and end transaction instructions can be done as a preprocessing step on the code, or alternatively the translation may be done at run time by the decode stage hardware.

When a thread is running within the bounds of a transaction (i.e.,

executing in *transactional mode*), all memory modifications must be kept in local cache, and not be visible to other threads. In addition, we may want to save the old value of these memory blocks in local cache, in order to make re-execution of a transaction in case of a conflict faster and less costly, avoiding shared memory accesses when rolling-back to the pre-transaction state.

Our transactional memory model is loosely based on the original hardware transactional memory proposal of [7]. In our model, each processor includes, in addition to its main DL1 on-chip cache, a secondary cache called *transactional cache*. The transactional cache is a smaller, fully-associative cache that is exclusive to the main cache. Cache lines in the transactional cache have an additional *transaction tag* which can hold one of the following values: *Empty*, *Normal*, *Xcommit* and *Xabort*. The first two values mark lines with invalid and non-transactional mode data, respectively. A line marked as *Xcommit* indicates that the line contains data that was valid before entering transactional mode, and may be used to recover from an aborted transaction. An *Xabort* tag indicates that the line contains data that was modified within the transaction, it is not visible to other threads, and any attempt to access it by another thread will result in a conflict and cause the transaction to abort.

4.2 Rollback and Checkpointing

An important aspect of the hardware transactional memory architecture is the ability to rollback on a transaction abort, and return to the state of the system before the transaction started. Previous work on hardware transactional memory largely ignored the aspect of implementing rollback capabilities. Rollback 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.

Several works propose using some form of checkpointing as a method for recovering the state of a uniprocessor or multiprocessor. Akkary *et. al* [1] introduced Checkpoint Processing and Recovery (CPR), a new microarchitecture that replaces a reorder buffer-based design. Sorin *et. al* [17] introduced SafetyNet, a hardware mechanism to recover the state of a multiprocessor system, used to enable recovery from deadlocks and transient or permanent hardware faults by periodically creating a system-wide logical checkpoint of the system. The system-wide checkpoint includes the state of the processor registers, memory values, and coherence permissions.

For our purposes of recovering from a conflicting transaction, when starting a transaction we can take a checkpoint that saves only the register state of the processor running the transaction. Checkpointing the memory state is not necessary, since the memory state will be saved in the transactional cache. As an alternative, we can use a system-wide checkpointing mechanism, if it already exists in the system for other types of recoveries (such as fault recovery). In this case, the checkpoint is taken periodically, and on a conflict we can rollback to the most recent checkpoint. The only limitation on the checkpointing mechanism is that a checkpoint can be taken only when no transactions are in progress. The length of checkpoint intervals will determine the rollback penalty.

Another aspect of checkpointing is the cost of taking a checkpoint. Although we adapt the assumption of SafetyNet, that checkpoint creation is a lightweight process, the energy overhead of this process must be taken into account when analyzing the overall energy consumption of transactional memory. On the other hand, if we have a system that already takes periodical checkpoints, then unless we have contention, this additional energy will be independent of whether we use locks or transactions.

4.3 Simulation Model

After testing various simulation models, we found Virtutech's Simics [11] to be the most appropriate for our needs. Simics is a full system simulation platform, capable of simulating high-end target systems running full operating systems and commercial workloads [19]. 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.

The limitations of Simics include limited flexibility in manipulating the processor microarchitecture. It is a functional simulator, which assumes that each instruction takes one cycle to execute. However, Simics provides an interface to a detailed memory hierarchy simulation, as well as a detailed operation system simulation, both of which are essential for our purposes.

In Simics, an atomic instruction that implements a lock will be blocked if it may conflict with an earlier load that has not been executed or an earlier store that has not been retired. To avoid a possible deadlock situation, an atomic instruction will be stalled until there are no possible conflicts. The load part of the atomic instruction will always be sent to memory, even if its data is found in the load-store queue.

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 rollback of the processor state after instructions are retired, it does provide a checkpoint mechanism which 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 and unified data/instruction L2 on-chip cache. All processors share a single off-chip memory. Table 1 summarizes the system configuration.

4.4 Evaluation

The energy estimates for the different memory hierarchy components are listed in the third column of Table 1. These numbers were extrapolated from Micron SDRAM power calculator [9] and CACTI [16], as well as from a conversation with high-end multiprocessor designers from industry. We assume the processor clock is 1GHz and the SDRAMs' clock is 133MHz. The shared memory energy is the sum of the I/O of the processor Front Side Bus, the I/O of 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.

In order to simulate a series of transactions with some conflicts on a multiprocessor system, we use our own microbenchmark with 4 threads, all accessing a shared array. Each array entry is protected by a dedicated lock (which then defines a transaction). We force each thread to run on a different processor, and make sure the threads run concurrently and conflict. We are using global check-

Table 1: 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	0.47 nJ
Trans. Cache	64-entry fully associative	0.12 nJ
L2 Cache	128KB 4-way; 32B line; 10 cycle	0.9 nJ
Shared Memory	256MB; 200-cycle latency; 64-bit bus	33 nJ

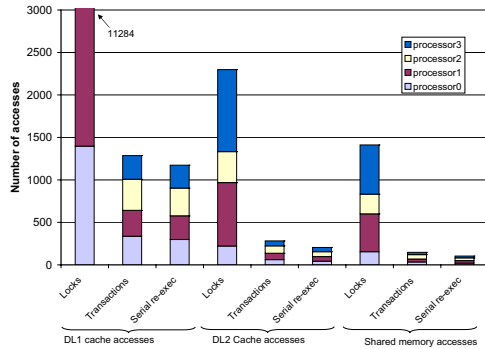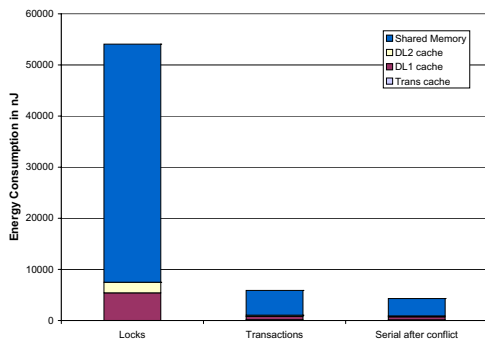**Figure 1: Memory overhead of locks vs. transactions.**

Figure 2: Energy consumption of locks vs. transactions.

points of the system to recover from conflicts. Also, since we are interested in an energy-aware approach to using transactional memory, we consider two different schemes for handling a conflict:

1. the standard approach of re-issuing the transactions in parallel (after a random backoff), and
2. allowing all transactions that were pending during the conflict to execute serially.

Figure 1 compares the number of cache and memory accesses for each of the microbenchmark runs. The first set of bars shows DL1 cache accesses, the second set of bars DL2 cache accesses, and the third set of bars shared memory accesses. In each set of bars, the first bar shows accesses for the microbenchmark using locks, the second bar for the microbenchmark using standard transactions, and the third bar for our serial re-execution scheme. Note that the version using locks incurs the most memory accesses (this is discussed below), and that the serial re-execution run has fewer memory accesses than the standard transactions. In this case, the standard transaction version had more re-executions that resulted in conflicts, and therefore in wasted instruction execution and cycles. If, however, the re-executions had not resulted in conflicts, this would have allowed an earlier commit of transactions and better performance.

For the simulations using transactions, only user instructions, i.e., the instructions from the microbenchmark, were executed. For the simulation using locks, the supervisor (operating system) executed a large number of additional instructions, regardless of the conflict rate. The locks we used are standard library locks and are not optimized; however, even with an optimized lock, it is unclear what the extent of the benefit will be. We leave this lock optimization for future work.

Figure 2 shows the energy consumption of each microbenchmark run. The shared memory accesses dominate the energy consumption, making the use of locks an unlikely energy-aware solution. The serial re-execution of transactions consumes less energy than

the standard transactions version, which follows as a direct result of the reduced number of accesses reported in Figure 1.

The performance results, not shown here, vary between different conflict scenarios. However, with serial re-execution of transactions we are assuming that more conflicts will occur, and as a result, we are preventing possible parallelism in case this assumption was overly pessimistic. Therefore, in terms of performance, it is overall better to use standard transactions. On the other hand, if lower energy consumption has a higher priority, then resorting to serial execution of transactions in a situation where a high conflict rate is likely makes for a more appropriate choice.

5. CONCLUSION

This is a first step in trying to evaluate the energy cost of managing memory contention in a multiprocessor environment, focusing specifically on conflict scenarios within transactions. When conflicts are rare, transactions have an advantage over locks in terms of performance as well as energy, due to fewer accesses to shared memory. As conflicts become more common, however, the cost of rolling back and restoring checkpoints justifies an alternative conflict resolution scheme. We propose to serialize transactions following a conflict (assuming we are entering a high-conflict region of execution). Forcing synchronization may not seem beneficial for performance since it hurts parallelism, but being conservative is advantageous in terms of energy.

Acknowledgments

6. REFERENCES

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO*, December 2003.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, Feb. 2005.

[3] R. I. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *ISLPED*, August 1998.

[4] M. Ekman and P. Stenström. Performance and power impact of issue-width in chip-multiprocessor cores. In *ICPP*, October 2003.

[5] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA*, October 2003.

[6] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, July 2003.

[7] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993 May.

[8] M. Huang, J. Renau, S. Yoo, and J. Torrellas. A framework for dynamic energy efficiency and temperature management. In *MICRO*, December 2000.

[9] J. W. Janzen. *SDRAM Power Calculation Sheet*. Micron, 2001.

[10] J. Li, J. Martinez, and M. Huang. The thrifty barrier: Energy-efficient synchronization in shared-memory multiprocessors. In *HPCA*, February 2004.

[11] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hllberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, pages 50–58, February 2002.

[12] J. F. Martinez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *ASPLOS*, October 2002.

[13] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *ASPLOS*, October 2002.

[14] R. Rajwar and J. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS*, October 2002.

[15] P. Rundberg and P. Stenström. Speculative lock reordering: Optimistic out-of-order execution of critical sections. In *IPDPS*, April 2003.

[16] 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.

[17] 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 *ISCA*, May 2002.

[18] 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, Nov. 1993.

[19] Virtutech. *Simics*. https://www.simics.net.