Synchronization and Chip Multiprocessing (CMP)

Hermoso, Sheryl M.
CS 252 Midterm Paper
August 21, 2006

I. Introduction

Processor chips are becoming smaller and smaller. Personal computing needs are ever increasing while the silicon's physical and thermal limitations are bound to be reached. With today's transistor technology that limits the ability of single processors, manufacturers tend to switch on multiprocessors as a solution.

This new revolution in computer architecture technology, which we now call Chip Multiprocessing (CMP) or simply multicore, is a combination of two or more independent processors into a single integrated circuit (IC) package. It allows the device to exhibit some sort of parallelism - thread-level parallelism (TLP) and/or instruction level parallelism (ILP) - while enjoying fewer components, lower cost, and less interconnection overheads.

Multicore has its pros and cons. Although promising and advantageous, having multiple cores in a single chip presents several problems, cache coherencies and memory synchronization are among them. However, these concerns were similar to those addressed by multiprocessors and parallel processors since the late 1980s.

Multiprocessing is not a new technology. Multiple processors communicate by way of sharing data structures. A standard problem in a system running multiple processes is how to control accesses to the shared data to ensure correct behavior and data consistency. Such can be addressed with synchronization, the coordination of simultaneous threads and processes accessing the same memory address space. Synchronization mechanisms, although mostly rely on user-level software algorithms, also require support by hardware primitives.

II. Classification of Synchronization

A general classification of synchronization is either lock-based or lock-free.

Lock-based algorithm is the traditional way to synchronize multiple processes or threads by using locks. Locks, barriers, and mutual exclusion came to be tagged as lock-based since they need to "lock" a shared variable to exclusively access and modify the data. Another process that wishes to use the variable stays in busy-wait or spinning mode, repeatedly checking to see if the lock has become available, and competes for lock ownership with other processes once the variable becomes free. The implementation of this technique is through hardware primitives such as mutexes or semaphores to ensure that no two processes are concurrently accessed, thus preventing the corruption of the shared memory. Some downsides to this method include deadlocks, starvation, and priority inversion. Deadlock occurs when several processes may acquire the semaphore causing both to wait forever for the other semaphore to be released. Starvation is insufficiency in resources to complete a process, and priority inversion occurs when a higher priority thread waits for the lower-priority thread. Of course, the process in busy-wait is rendered useless because it only wastes clock cycles in idle state.

Lock-free algorithms aim to solve this problem. Lock-free and wait-free algorithms are designed to allow multiple threads to read and write shared data concurrently without corrupting it, thus the term "lock-free." Wait-free" refers to the fact that a thread can complete any operation in a finite number of steps, regardless of the actions of other threads [2].

Hardware synchronization mechanisms such as atomic instructions should be supported by the CPU architecture. Lock-based Test-and-Set, Fetch-and-Increment, Test-and-Test-and-Set are some of the popular ones. Lock-free primitives include Compare-and-Swap and Load-Linked/Store-Conditional instructions. Compare-and-Swap (CAS) CPU instructions compare the contents of a memory location to a given value then modify the contents of that location if they are the same.

III. Papers

Several papers have proposed solutions to synchronization for shared memory multiprocessor systems. Crumney and Scott [1] in 1991 collected some of these existing algorithms and compared them with a new technique. The analysis implemented two popular busy-wait synchronization constructs, namely spin locks and barriers. Spin locks provide a means for achieving mutual exclusion ensuring that only one processor can access a particular shared data structure at a time. Barriers provide a means of ensuring that no processes advance beyond a particular point in a computation until all have arrived at that point.

The paper provided an analytical comparison of five spin lock implementations, simple test-and-set lock, ticket lock, two array-based queuing lock, against the list-based queuing lock and evaluated them based on scalability and induced network load, one processor latency, space requirements, fairness sensitivity to preemption, and implementability with given atomic operations. Five different barriers were also compared: four previous papers, the centralized barrier, software combining tree barrier, dissemination barrier, tournament barrier, and the author's

own design called tree-based barrier. Evaluation criteria include length of critical path, total number of network transactions, space requirements, and implementability with given atomic operations. All of the implementations were tested using the BBN Butterfly, a distributed shared memory multiprocessor, and the Sequent Symmetry Model B, a cache coherent, shared-bus multiprocessor. The network latency increase for busy-wait and barrier schemes in the Butterfly with 60 processors are shown in Table 1 and Table 2.

The algorithms does not require special-purpose mechanisms. Commonly-available hardware atomic primitives were enough to support synchronization. Among these general purpose primitives, shared memory machines would benefit most with fetch and increment operations, including compare-and-swap. Based on architecture, the paper recommended a compatible algorithm. The results are found in Table 3.

Busy Wait Lock	**Increase in Network Latency Measured From**	
	Lock Node	**Idle Node**
Test and set	1420%	96%
Test and set w/ Linear Backoff	882%	67%
Test and set w/ Exp. Backoff	32%	4%
Ticket	992%	97%
Ticket w/ prop. backoff	53%	8%
Anderson	75%	67%
MCS	4%	2%

Table 1 Increase in Network Latency on the Butterfly caused by 60 Processors Competing for a Busy-Wait Lock [1]

Barrier	**Local Polling**	**Network Polling**
Tree	10%	124%
Dissemination	18%	117%

Table 2 Network Latency Increase using Local and Network Polling Strategies [1]

Lock or Barrier Algorithm	**Recommended for...**
MCS Lock	Hardware provides Fetch and Store
Ticket Lock w/ Backoff	No fetch and store
Simple Lock w/ Exponential Backoff	Process will be preempted while spinning
Centralized Counter barrier	Broadcast-based cache-coherent multiprocesses
Dissemination barrier or tree-based barrier	Multiprocessor without cache coherence

Table 3 Recommended Lock and Barrier for Different Architectures [1]

However, a breakthrough in synchronizing techniques was the introduction of lock-free and wait-free algorithms by Maurice Herlihy [2]. He published a series of papers focusing on lock-free objects. He defined lock-free as a process which does not require mutual exclusion.

His paper [2] on wait-free synchronization in 1993 presents a proof that atomic registers have few applications in wait-free implementations. The contribution of this paper is the statement that it is impossible to construct wait free implementation of (1) common data types such as set, queues, stacks, priority queues, or lists, (2) most classical synchronization primitives, such as test and set, compare and swap, and fetch and add. It presents a model of computation for simple universal objects from which a wait-free implementation can be constructed for any sequential object. It suggests the use of compare and swap primitives for universal construction over classical synchronization primitives (test and set, fetch and add), which they commented are as weak as message-passing primitives.

Transactional memory [3] shows its advantages over locks to address energy consumption issues. However this advantage is dependent on the architecture, contention level and the conflict policy being implemented in certain architectures. Transactional memory was proposed by Herlihy in an earlier paper. There have been rich studies on both software and hardware implementations since then. 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 is optimistic, and if it completes without conflict, it will commit and no further action is required. Otherwise, if conflicts were detected during the execution, the transaction will abort, its effects will be discarded and will be required to roll back and re-issued. Hardware transactional memory proved to be superior over software transactional memory.

Reference [3] used the SPLASH-2 benchmark suite to compare the hardware transactional memory and a synthetic microbenchmark for high contention operations. The results in Fig. 1 shows a reduction in energy consumption at high contention, which is attributed to the reduced number of cache and memory accesses.

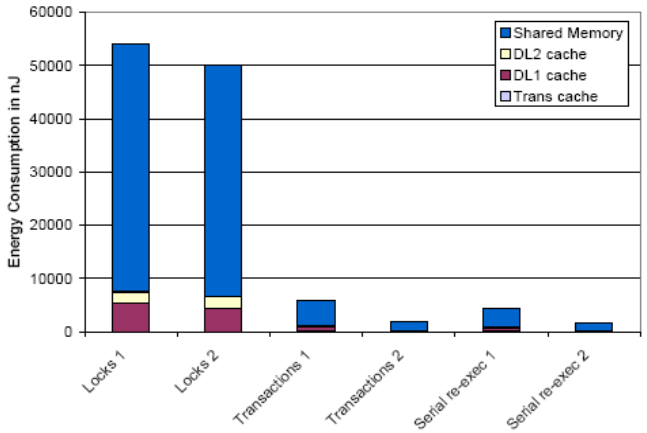

Figure 1 Energy Consumption of Microbenchmarks Using Locks vs. Transactions [3]

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

Table 4 Energy Consumption per Cache/Memory Access [3]

Energy estimates for different memory hierarchy are shown in Table 4. The shared memory energy was computed as the sum of the I/O processor of the Front Side bus, the I/O of the SDRAM pins, and the actual SDRAM access. The paper shows that a transaction is advantageous over locks both in performance and energy for rare conflicts. However, as conflicts rise, transactions may not be beneficial due to the cost of rolling back. They proposed to serialize transactions following a high-conflict region of execution.[3]

The advantages of transactional memory over mutual exclusion have been investigated for different architectures. Specifically interesting are its effects on large-scale shared-memory multiprocessor units and the hardware support best suited for each system.

Another paper [4] which focused on hardware synchronization solutions studied different atomic primitives for distributed shared memory multiprocessors (DSM). It considered three general purpose hardware primitives, fetch-and-increment, compare-and-swap, load-linked/store-conditional on directory-based cache-coherent DSM multiprocessors. These primitives are largely used for lock-free software algorithms such as introduced by Herlihy. Having seen that generic primitives provide greater concurrency, efficiency and fault-tolerance, the purpose of the paper was to find the best atomic primitive to use for future DSM multiprocessors. Three fetch-and-increment, five CAS, and three LL/SC, were categorized according to coherence policy as INValidate, UPDate, and UNCached. The result achieved is highly in favor of compare-and-swap instructions, which the researchers recommended for cache controllers with write-invalidate coherence policy. Fetch-and-add was found to be efficient for lock-free counters, and is recommended for uncached memory.

Several more papers that followed focused on both software and hardware synchronization techniques. Topics range from distributed data structure that implements concurrent, lock-free, low-contention read-modify-write registers to nonblocking queues using FIFO buffers that can work both on symmetric and unsymmetrical processors.

IV. Industry and Academic Trends

Emerging architectures such as Simultaneous Multithreading (SMT) and Chip Multiprocessing (CMP) are bringing multiprocessing to the desktop. Multiple cores communicate directly by way of shared hardware cache to increase concurrency. With SMT and CMP at play, shared-memory parallel processing has once again become an important topic of investigation both for the industry and the academe. The problem with synchronization one decade ago is shifting to the level of multicores. There is in fact a growing community that study hardware architectures employing these techniques. Its major effects on programming languages, compilers and operating systems, present problems and thus opportunities for more intensive researches in the software side.

Chip manufacturers have started shipping multicore processors. Industry efforts to synchronize data focuses on locks and semaphores. IBM Power 4 [6], the first of its kind, included two noncacheable units to handle cache and synchronization operations. It used lock and unlock functions to execute synchronization instructions, such as *sync* and *lwsynch*, provided in the PowerPC Instruction Set Architecture. In software, *Win32* and *POSIX* API provide both blocking and non-blocking thread synchronization primitives. Reference [7] however pointed out that although lock-free approaches are promising, this area is still in theoretical and experimental stages, and are left in the realm of university studies.

Academic efforts to address the synchronization issues include an abundance of talks on transactional memory and its applications on CMPs. One example is the keynote speech by Maurice Herlihy in the recent ACM PODC 06 held July 22-26, 2006 [8].

Wells [5] hypothesized that spin-lock is an attractive mutex mechanism for CMPs. Using the SPARC v9 ISA, they then proved this as sufficient for single-chip CMPs as compared to queue-based locks, being simpler to implement. An improved cache-to-cache transfer is observed by increasing the number of cores per chip using different benchmarks (see Fig. 2). To further improve performance of spin-locks, the paper also proposed an on-chip lock arbiter that will convert spin-locks to queue-based locks and assign priority based on the locality of requesting processor.

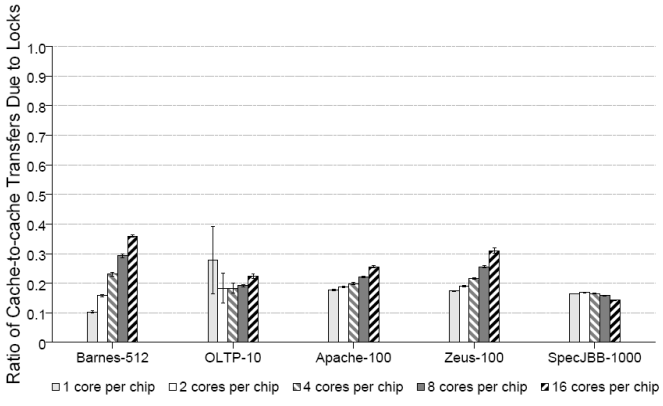

Figure 2 Cache to Cache Transfers Resulting from Locks [5]

V. INSIGHTS AND CONTRIBUTION

Although little is known on the advances of lock-free algorithms in chip multiprocessing, efforts on the academic side are being put up to fill in the gap. Transactional memory and lock-free objects have been found beneficial for shared memory multiprocessors in the past. It would perhaps provide a parallel impact with CMPs today, especially when it is backed up with the multithreading technology.

REFERENCES

[1] J. M. Mellor-Crummey and M. L. Scott, *Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors*, ACM Trans. on Computer Systems, 9(1), February 1991.

[2] M.P. Herlihy. Wait-free synchronization. ACM Transactions on Programming Languages and Systems, 13(1):124--149, January 1991. (Preliminary version PODC'88.)

[3] Tali Moreshet, R. Iris Bahar, and Maurice Herlihy. *Energy-Aware Microprocessor Synchronization: Transactional Memory vs. Locks*. http://www.lems.brown.edu/~tali/publications/ moreshet_wmpi06.pdf

[4] Maged M. Michael and Michael L. Scott. *Implementation of atomic primitives on distributed shared memory multiprocessors*. In Proceedings of the First International Symposium on High-Performance Computer Architecture, pages 222--231, January 1995.

[5] Philip Wells. *Investigating CMP Synchronization Mechanisms*. http://www.cs.wisc.edu/~david/courses/cs838/projects/

[6] Bill Hay and Gary Hook. *Power 4 and Shared Memory Synchronisation*. http://www128.ibm.com/developerworks/eserver/articles/power4_mem.html

[7] Walsh, George. *Multiple Approaches to Multithreaded Applications.* http://www.intel.com/cd/ids/developer/asmo-na/eng/dc/threading/implementation/151201.htm?page=1

[8] Herlihy, Maurice. *The Art of Multiprocessor Programming.* Annual ACM Symposium on Principles of Distributed Computing, http://delivery.acm.org/10.1145/1150000/1146382/p1-herlihy.pdf?key1=1146382&key2=4898216511&coll=ACM&dl=ACM&CFID=15151515&CFTOKEN=6184618#search=%22chip%20multiprocessing%20herlihy%22