Investigating CMP Synchronization Mechanisms

Koushik Chakraborty
kchak@cs.wisc.edu

Anu Vaidyanathan
vaidyana@cs.wisc.edu

Philip Wells
pwells@cs.wisc.edu

CS 838
Dec 19, 2003

1 Introduction

Synchronization is an important aspect of multi-processor programs. Though necessary to communicate data or control information across different threads of a parallel program, excessive synchronization can *serialize* execution to the point where little benefit is seen from creating a parallel program.[1] Synchronization in the operating systems can also have a large impact on performance when coordinating shared resources across different processors such as the scheduling queue.

Much research across many decades has resulted in algorithms and hardware solutions that attempt to understand the numerous trade-offs in terms of latency, bandwidth, implementation complexity, etc. Nonetheless, synchronization remains a significant concern for parallel applications and architectures. Especially now that the computer industry is at a point where multiple processors can be placed onto a single chip, the trade-offs in the design of synchronization need to be revisited.

Chip Multiprocessors (CMPs) and their workloads have a number of interesting properties that affect the costs of synchronization. In the next section, we make a few basic assumptions about these properties, and form hypotheses about CMP synchronization. We then characterize the amount and cost of synchronization occurring in a particular CMP design, limiting our preliminary study on mutual exclusion (mutex) locks. Finally, we propose an on-chip lock-arbiter to dynamically handle *spin-locks* in a queue-based manner.

1.1 Synchronization in a CMP

To motivate why the synchronization scene changes for a CMP, we make three basic assumptions about a the properties of a single-chip CMP system and its workloads: fast on-chip data transfers, relatively large amounts of on-chip bandwidth, and relatively infrequent synchronization. We thus consider the following hypothesis: *Spin-locks are an attractive mutex mechanism for CMPs*.

More interesting trade-offs arise when considering synchronization both within and across CMP chips because of the much slower off-chip communication and limited off-chip bandwidth. We expect to find for these systems that results agree with previous work [2] indicating that queue-based locks are a much better alternative. In addition, CMPs may enable other workloads which perform much more frequent synchronization, and are thus more affected by the cost of that synchronization. We expect that spin-locks are less appropriate in these situations. Finally, CMPs have a large number of transistors that may find better use in novel ways other than aggressive cores and bigger caches, possibly aiding synchronization.

2 Related Work

Previous work includes QOSB [5] that introduced the idea of queuing requests for locks in hardware. This was followed by the MCS proposal [4] that implemented the idea in software. QOLB [2] followed the MCS work, with extra hardware support and extended coherence protocols. I-QOLB [7] introduced speculation on what *might* be the execution of a critical section to guide the cache-coherence protocol.

[1] *Thread* is used as a generic term to denote concurrent parts of a program, and not specifi cally kernel threads and the like.

Rajwar and Goodman proposed the concept of speculative lock elision to dynamically remove lock induced serialization [6]. This is possible, and beneficial, when critical sections are not executed concurrently, or when operations performed in a critical section do not have inter-thread dependencies. The work cited here is that their evaluations have primarily used scientific workloads, whereas we investigate commercial workloads as well.

3 Identifying and Tracking Locks

We used the Simics full-system simulation [3] infrastructure from the UW Multifacet group, which runs commercial and scientific workloads on an unmodified Solaris 9 operating system. Modifications were made to the out-of-order processor model (*opal*), the memory model (*ruby*), and the cache coherence protocol definition for *slicc*.

The workloads used in this study are Barnes-Hut (512 bodies), OLTP (10 transactions), Apache and Zeus (100 transactions each), and SpecJBB (1000 transactions). Larger workloads were not run due to the length of simulation required. High variability in these relatively short runs is mitigated as described in [1] and taking a sample of 10 runs per data point.

3.1 Hardware Primitives

The SPARC V9 ISA provides the application/system writer with atomic read/write primitives that are used to implement synchronization. These primitives are *test&set* (`ldstub`), *compare&swap* (`casa`), and *swap-always* (`swap`).

These primitives can be used to construct a wide variety of synchronization mechanisms. For the purpose of this study, we focus on synchronization that enforces mutual exclusion (*mutex*) by using using these primitives to manage a lock. For mutex locks, only one thread is allowed to hold the lock at a particular point in time. In the case of contention — other threads trying to acquire the same lock — these other threads must wait until it is released. We also focus only on *spin-locks*, which continue to emit acquire attempts until one succeeds.

3.2 Identifying Lock Operations

Because SPARC does not provide obvious *acquire* and *release* instructions, we must infer these events by examining the dynamic instruction stream. To accomplish this, a global *Lock Address Table* is maintained in *opal*, which tracks all memory operation on addresses that have been previously accessed by at least one atomic instruction. Every time an atomic instruction is seen, the physical address is looked up in this table. If the address is not present, we assume this instruction is accessing a lock, and add an entry. All subsequent memory accesses to this address (atomic or not) are tracked via the table entry. Each table entry tracks the status of the lock assumed to reside at the entry's address.

Acquire and release events are inferred based on the lock's status in the table, the sequencer performing the operation, and the values read and written by the memory instructions to these addresses as follows:

- An atomic operation writing a non-zero value and reading a zero that was previously stored is considered a successful acquire.
- An atomic operation that reads a non-zero value while storing a non-zero is considered a failed acquire attempt (contention).
- A store instruction writing zero is considered a release operation.
- A load instruction is considered a precursor to an acquire attempt (e.g, first *test* operation of *test and test&set*).

These inference rules are fairly restrictive. Non-mutex synchronization will not be observed, and mutex locks that do not contain a value of zero when unset will not be observed. Due to time restrictions, we were

```
#define TSB_LOCK_ENTRY(tsb8k, tmp1, tmp2, label)                              \
        ld  [tsb8k], tmp1                                                    ;\
label:                                                                       ;\
        sethi   %hi(TSBTAG_LOCKED), tmp2                                     ;\
        cmp     tmp1, tmp2                                                   ;\
        be,a,pn %icc, label/**/b        /* if locked spin */                 ;\
            ld      [tsb8k], tmp1       /* delay slot */                     ;\
        casa    [tsb8k] ASI_N , tmp1, tmp2                                   ;\
        cmp     tmp1, tmp2                                                   ;\
        bne,a,pn %icc, label/**/b       /* didn't lock so try again */       ;\
            ld    [tsb8k], tmp1         /* delay slot */                     ;\
        /* tsbe lock acquired */                                             ;\
        membar #StoreStore
```

Figure 1: *Test and Test&Set* Code Sequence from Solaris 8 MMU

unable to account for other classes of synchronization in this study, and our results thus present an underestimate of synchronization activity. Nonetheless, a significant number of locks matching these semantics *are* observed for nearly all our workloads.

To validate our assumptions, we combed through the source code of the Solaris 8 kernel to examine locking mechanisms. This was done by simply looking at the places within the kernel that used the atomic operations provided by the SPARC ISA. Solaris synchronizes using four higher-level primitives: mutex locks, condition variables, reader-writer locks and semaphores. We limited our code search to mutex locks because that was the focus of our study. While the `ldstub` (load-store unsigned byte) instruction directly implements *test&set* by storing a 0xff to the memory location, and returning its previous contents, we have observed other atomics such as `casa` (compare&swap) being used for mutex locks as well, and included them in our source code search. Because `casa` can set the value to other numerical values, it can consequently be used to set things like thread IDs to provide more information about who is actually holding the lock.

We found many examples of spin-locks in the kernel, one of which is shown in Figure 1. This code sequence, taken from sfmmu_asm.s, defines a macro that tries to grab the lock for a translation store buffer (MMU) entry. This set of instructions is clearly executing a *test and test&set* type of primitive even though there is a casa instead of an ldstub. The comments (except for "delay slot") are copied directly from the code.

3.3 Tracking Lock Latencies

Once mutex-related instructions are identified, various statistics are recorded including the latency of the acquire and release operations. As illustrated by Figure 2, we divide the latency involved in these lock operations into three categories: *acquire latency*, *release latency* and *transfer latency*. For an uncontended lock, the *acquire latency* is simply the latency of the instruction performing the acquire operation (often involving a cache miss). However, for a contented lock, a thread may execute many failing acquire (or test) instructions before actually acquiring the lock. We consider the *acquire latency* to be the time from the beginning of the first acquire (or test) attempt until the end of the successful acquire. *Release latency* is simply the latency of release operation — in most cases a store instruction, which might also involve a cache miss. For a contended lock, there is also a latency involved in transferring the lock. Basically, this is the delay between the start of the release operation and the end of the subsequent acquire operation from one of the contending threads. In the case of an uncontended lock, which sees no acquire attempt before the release, transfer latency is considered to be zero.

Context switches introduce a potential complication in measuring acquire latency. Since a thread can be de-scheduled while trying to acquire a lock,[2] calculating acquire latency by taking the difference between the first attempt and successful acquire will result in an overestimate of contention (because the thread wasn't

[2]This happens frequently, as some of the Solaris locks sleep the thread after some number of failing acquires

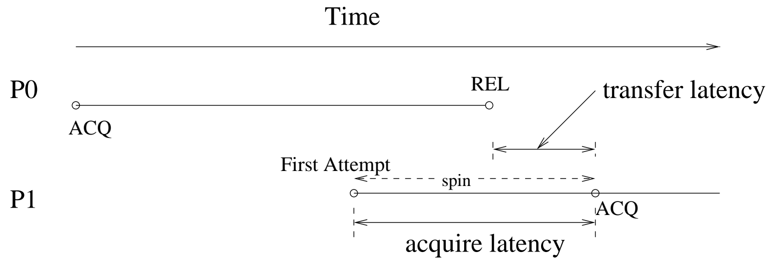

Figure 2: Example: acquire and transfer latency

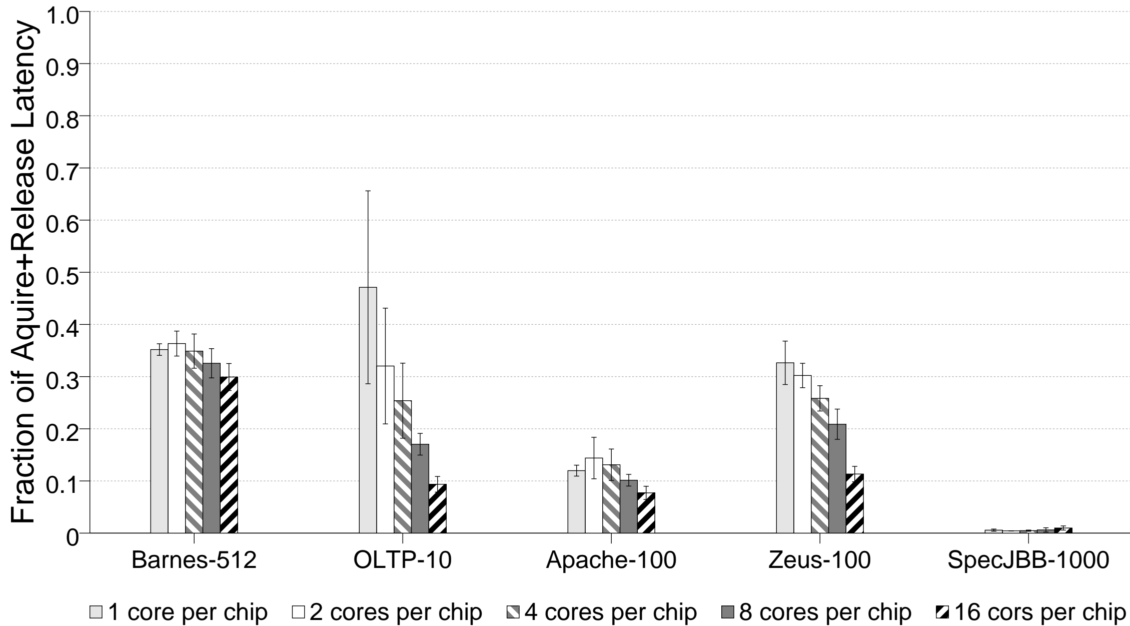

Figure 3: *Fraction of Cycles Spent on Mutex Operations* Leftmost bar is a single processor core per chip (16 chips), and rightmost bar is 16 processors per chip with one chip.

contending for the lock while it wasn't running). Normally, acquire latency is accumulated over all acquire attempts for a particular lock from a particular sequencer. If we observe a context switch between any two successive attempts, we do not accumulate the latency between these attempts.

3.4 Tracking Memory Data Sources

The *slicc* coherence protocol description was augmented to track the data source for all memory operations (e.g. local-L1, on-chip-L1, L2, main memory, etc.), and the interface between *ruby* and *opal* was the modified to pass this source back to *opal*. *opal* then records the latency and source for each completing instruction that initiates a memory request (i.e. does *not* hit in an MSHR). Because memory operations are initiated by squashed instructions as well, there is not a direct correspondence between instructions for which *opal* records information and memory operations which *ruby* observes. We turned off the prefetching of store addresses before the store data is available to accurately record sources of these operations as well.

4 Results

All experiments were run for a 16-processor CMP system. Configurations vary the number of processors per chip from 1 to 16. Each processor had private, split L1 I/D caches, and each chip has a shared L2 cache. The

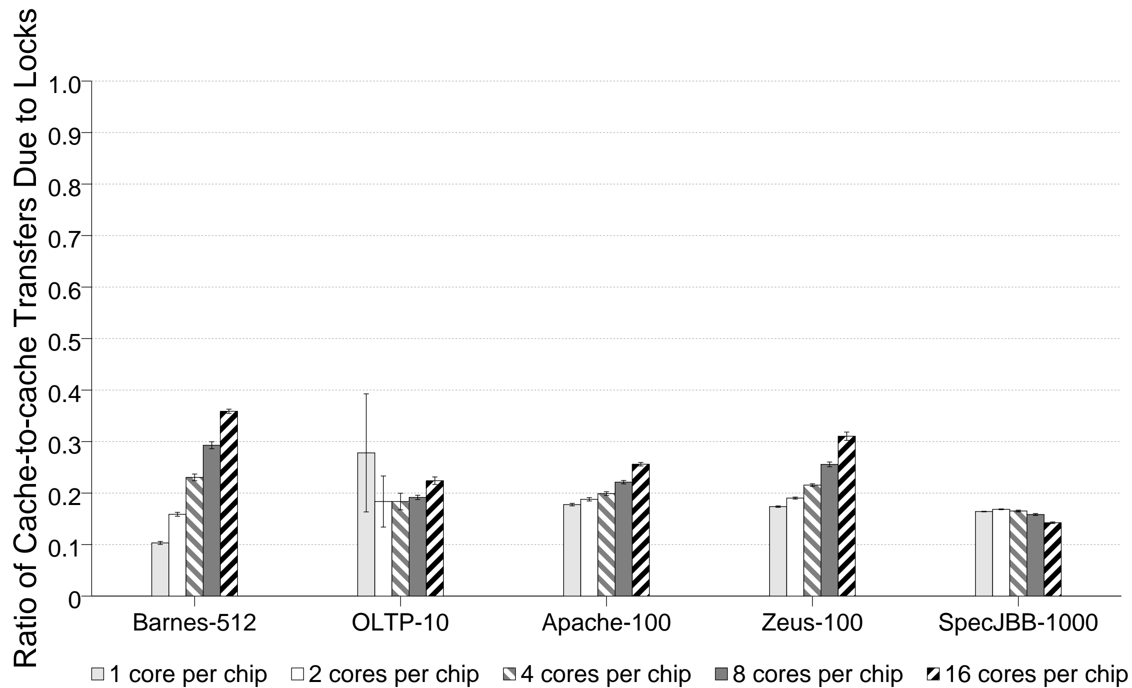

Figure 4: *Ratio of Cache-to-cache Transfers Resulting from Locks* Leftmost bar is a single processor core per chip (16 chips), and rightmost bar is 16 processors per chip with one chip.

size of the L2 cache per chip remains constant for all configurations, thus the configuration with 1 processor per chip (and hence 16 chips) has 16 times the aggregate L2 capacity of the configuration with 16 processors per chip.

Each data point in our results is the average of ten runs, with the 95% confidence interval shown using error bars.

4.1 The Cost of Mutual Exclusion

The cost of maintaining mutual exclusion comes both in terms of the latency to contend, acquire and release a lock, and in terms of the bandwidth consumed while performing these three functions.

Figure 3 shown the first of these costs: the number of cycles all processors spend waiting for contention, acquire, and release, as a ratio of the total cycles executed by all processors. As evident from the graph, most workloads spend a significant fraction of their time performing mutex synchronization, which for some configurations is over 30%. One anamoly is SpecJBB, which for all configurations, spends less than 1% on mutex operations. This appears to be a result of most of the synchronization in SpecJBB not conforming exactly to our assumptions described in Section 3.2, and not a result of JBB performing almost no synchronization. Further study is needed to make a more rigorous claim.

An interesting trend that is apparent in Figure 3 is that the fraction of mutex cycles diminishes as the number of cores on a chip increases. As shown later in Figure 6, a high frequency of mutex operations result in cache-to-cache transfers, which become much faster on-chip. These fast transfers aid mutex operations more on average than other operations.

Figure 4 presents the total number of cache-to-cache transfers that are triggered by locks within the workload runs. These are measured as a ratio of the total cache-to-cache transfers that are observed for that benchmark. We see that the number of cache-to-cache transfers that occur due to locks are anywhere between 10% to 35%, which can be though of as a very rough estimate of the cost of mutual exclusion in terms of inter-processor bandwidth.

Figure 5 presents the source of data on an L1 miss while varying the number of processing cores. We measure five quantities in this graph and the next one. These are the cases when the source of the L1

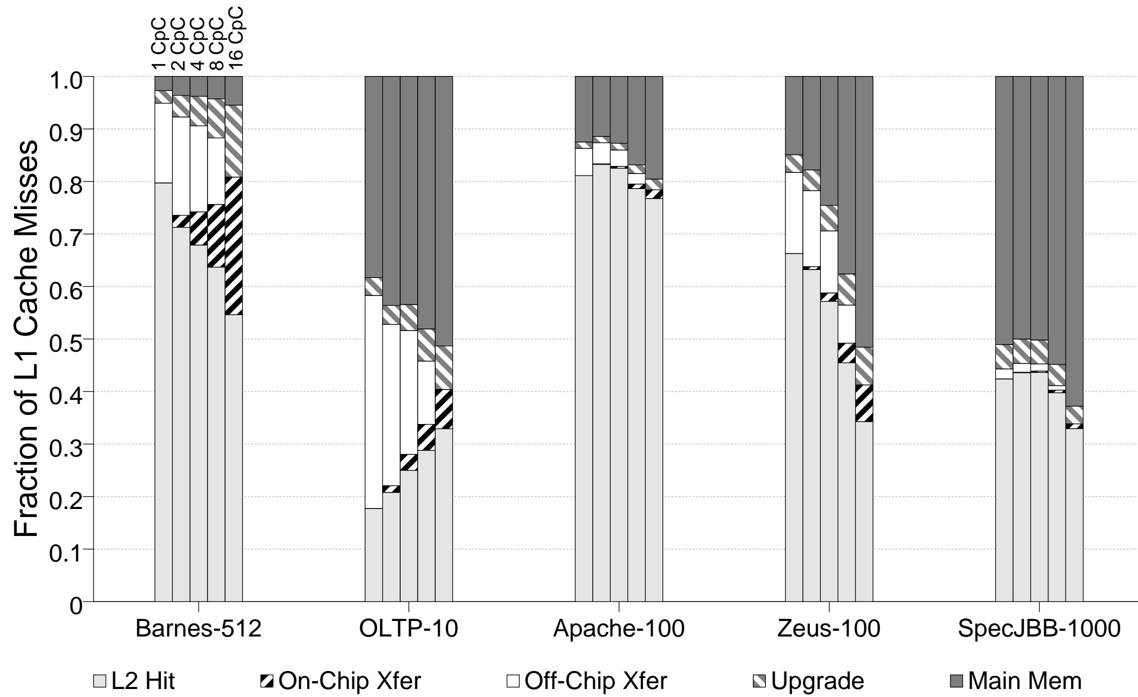

Figure 5: *Histogram of data sources for L1 misses for all Operations* Leftmost bar is a single processor core per chip (16 chips), and rightmost bar is 16 processors per chip with one chip.

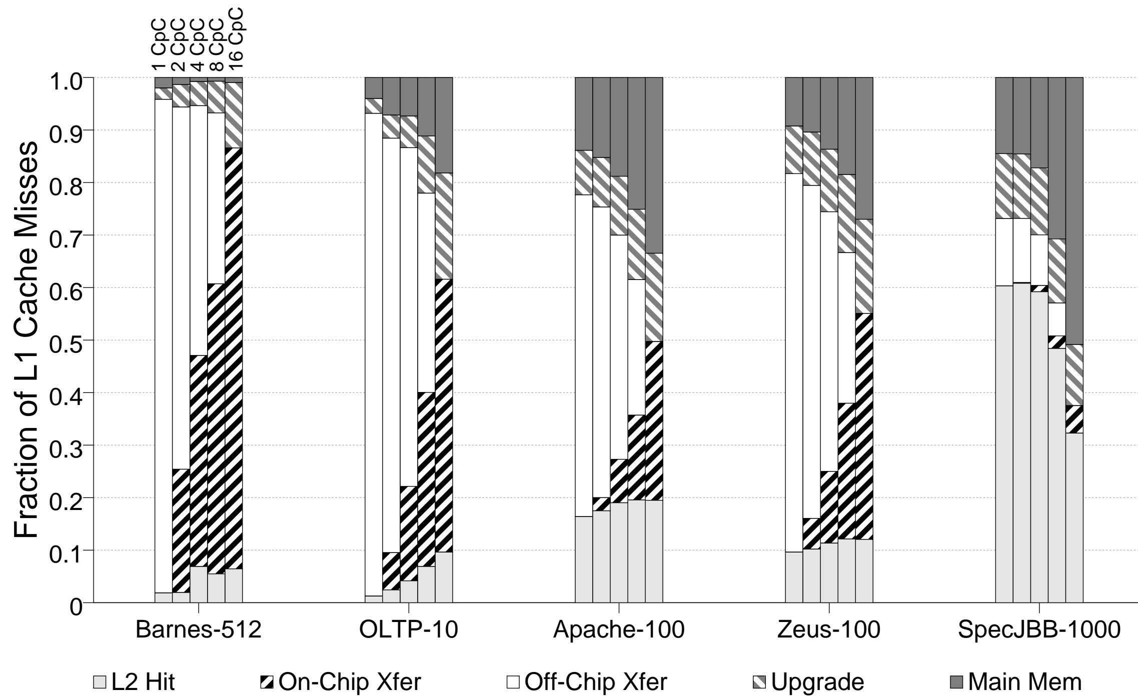

Figure 6: *Histogram of Data Sources for L1 Misses for Lock Operations* Leftmost bar is a single processor core per chip (16 chips), and rightmost bar is 16 processors per chip with one chip.

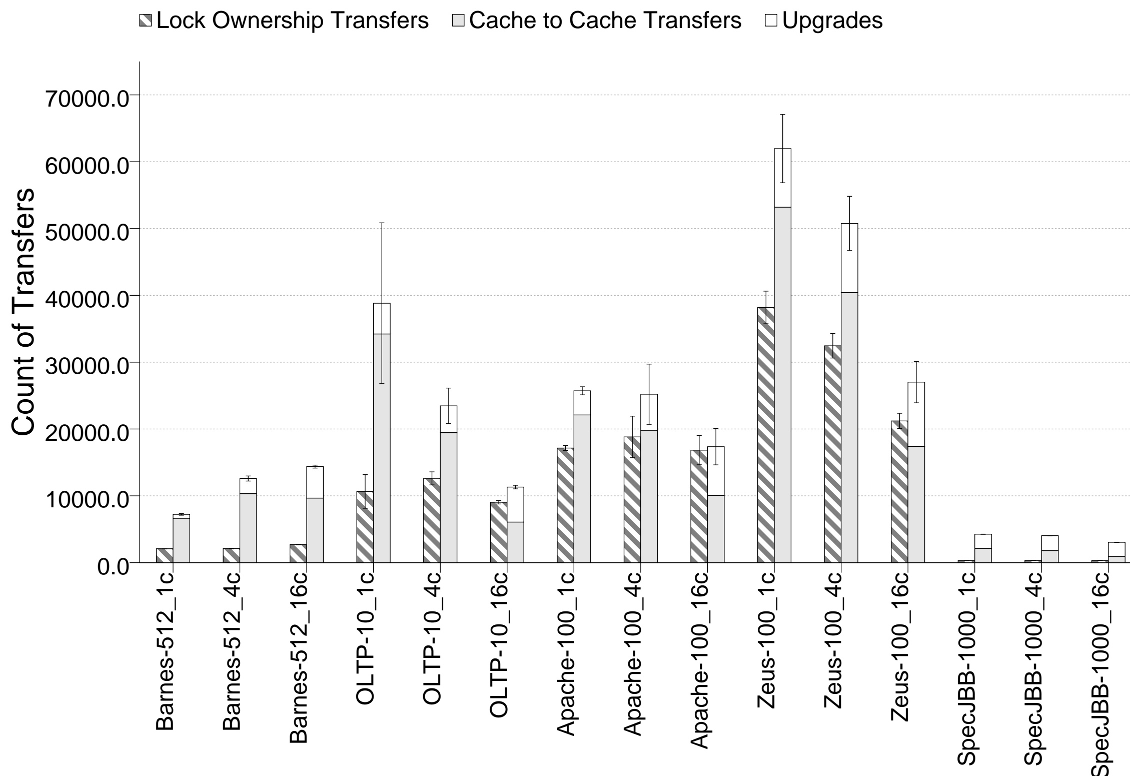

Figure 7: Number of Cache-to-cache Transfers Resulting from Locks Compared to Number of Lock Ownership Transfers

cache miss is: located in the L2 cache, causes an off chip data-transfer, causes an on-chip data transfer, causes a remote upgrade and causes the request to go to main memory. Because of the decreasing L2 size per processor for the multi-core-per-chip configurations, we observe a decreased number of requests being satisfied in the L2, and a corresponding increase in requests satisfied by main memory. The only benchmark that does not conform to this trend is OLTP, which shows an *increased* L2 hit rate as more processing cores share an L2. We also note that the total cache-to-cache transfers (measured as a sum of the off-chip and on-chip transfers) decreases as we add processing cores to a chip. This could simply be a capacity issue at the L1, i.e. the small L1s evict the shared data before it needs to be transfered. This phenomenon is most pronounced in OLTP, which is likely the cause of the increase in L2 hits.

Figure 5, however, is significant for our work when compared to Figure 6, which presents the source of data of an L1 miss for lock operations only. If the lock request misses in the L1 cache, it translates to a large amount of cache-to-cache transfers — much larger than the number of cache-to-cache transfers seen from all memory operations.

5 On-Chip Lock Arbiter

Due to the relatively coarse granularity of these workloads, and the fast nature of on-chip cache-to-cache data transfers, only a small percentage of total execution time is spent waiting on mutex locks for the CMP configuration with all 16 cores on a single chip. However, as the number of chips increases, the amount of time spent acquiring and releasing mutex locks increases substantially. Thus it seems that simple spin-locks are sufficient for single-chip multiprocessors, but are less adequate for multiple chip configurations.

Figure 7 presents the total number of cache-to-cache transfers resulting from lock operations vs. the number of lock ownership transfers. In the best case, only one cache-to-cache transfer is necessary per ownership transfer. However, even in the presence of moderate contention, the number of cache-to-cache transfers per ownership transfer increases quickly for spin-locks. In most cases, the number of cache-to-

cache transfers is significantly larger than the number of ownership transfers, especially for the one core-per-chip configurations. Though upgrades transaction due to lock operations do not involve a data transfer, they do involve numerous invalidate and acknowledgment messages, are not strictly necessary to transfer ownership of a lock, and are included on this graph as well.

This result indicates the amount of bandwidth spent in grabbing ownership of the lock resulting from the spin-lock mechanism. In some cases, like OLTP, the number of cache-to-cache transfers is four times that of lock ownership transfers. Though this may be acceptable for fast on-chip communications, these excess cache-to-cache transfers are more costly for off-chip communication, indicating that queue based locks may be more appropriate.

One possible way to improve the performance of spin-locks in a multi-chip configuration is to limit the number of times the lock ping-pongs between chips. This could be accomplished by dynamically converting spin-locks into queue-based locks, and assigning a queue priority based on the locality of the requesting processor to the processor holding the lock.

We propose using an *on-chip lock arbiter* to perform this functionality. At the high level, the arbiter observes the L1 miss stream and intercepts requests for addresses which it predicts contain locks. Lock addresses, as well as test, acquire and release operations, are inferred much the same way as in Section 3.2. These addresses are simply under the control of a separate coherence mechanism, which must ensure the same invariants as the regular coherence protocol, but can change the timing of these operations.

Acquire operations for a lock which the arbiter believes is held are simply delayed until a release operation is observed for that lock. At this point, the first queued acquire request is allowed to complete. This causes the processors to block on contention, rather than continuously spin. The arbiter must also intercept off-chip requests for the addresses it cares about, but can prioritize on-chip and off-chip requests. One possible policy might be to allow all on-chip processors to acquire the lock once (if they wish) before the remote request is processed. Careful thought must be performed to ensure that consistency model violations will not occur, and safety mechanisms such as a time-out are necessary as well.

We expect to see little performance improvement when using a lock arbiter with a single-chip configuration for these workloads because the amount of time processors spend contending for locks in these workloads is fairly small. A reduction in on-chip bandwidth use may indirectly affect performance, but it remains unclear whether this improvement would justify the complexity of the arbiter. However, by using a priority policy, we expect that the number of off-chip cache-to-cache transfers for lock addresses would be significantly reduced which would likely reduce the overall acquire latency and inter-chip bandwidth.

Due to the complexity of modifying the simulator to perform this functionality, we decided against implementing the lock arbiter for this project. In addition, the many complex interactions between the high-level software, locking libraries, and operating system must be considered as well to do justice to an evaluation, and we lacked the time to do that.

6 Conclusions and Future Work

Though we limited our study to a subset of synchronization mechanisms, we still observe a surprisingly high amount of cycles spent waiting for lock acquisition: 8-30% for single-chip CMPs, and 10-40% for multi-chip CMP configurations (excluding SpecJBB). Had we considered all classes of synchronization, this amount would only have risen.

We came to several conclusions as part of this study. First, for most workloads with a single-chip CMP configurations (with many processing cores on a single chip), spin locks are sufficient. Spin locks are easy and an attractive alternative to something more complex, like queue-based locks. In the case of CMP configurations with one processing core per chip (much like olden-day SMPs where individual chips, usually with one processing core, plugged into a backplane bus), queue-based locks make more sense. This is because off-chip latency is much longer and bandwidth is more precious. The second part of this claim (with old-style SMP configurations) agrees with earlier QOLB [2] work. We also believe that CMPs enable finer granularities for parallelization, but this remains to be evaluated.

There are several avenues for future work on this topic. First, we propose to study the synchronization primitives in greater detail, above and beyond mutex locks. The second path for future work could include

implementing the on-chip arbiter to see whether intelligently passing locks between chips actually gains us anything. Looking at the temporal locality of lock requests within one chip of a CMP might provide intuition into sizing the arbiter. We are also interested in writing synthetic benchmarks with very fine grain parallelism to see what their performance is like on CMPs.

References

[1] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th IEEE Symposium on High-Performance Computer Architecture*, February 2003.

[2] Alain Kagi, Doug Burger, and James R. Goodman. Efficient synchronization: Let them eat QOLB. In *ISCA*, pages 170–180, 1997.

[3] Peter Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[4] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[5] Efficient Synchronization Primitives. Appeared in asplos-iii, april 1989, pp. 64-75.

[6] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multi-threaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[7] Ravi Rajwar, Alain Kagi, and James R. Goodman. Improving the throughput of synchronization by insertion of delays. In *HPCA*, pages 168–, 2000.