Lock Behavior Characterization of Commercial Workloads

Jichuan Chang and Xidong Wang

{chang,wxd}@cs.wisc.edu

May 14, 2002

ABSTRACT

Commercial workloads such as databases and web servers are the dominant applications running on shared-memory multiprocessors. These multithreaded programs use synchronization mechanisms (often locks) to ensure serialized access of shared data, leading to a potential performance bottleneck. Speculative Lock Elision (SLE) [2] technique has demonstrated that hardware optimization can remove many lock-induced serializations by speculative execution. As this technique is evaluated only against scientific workloads without the presence of operating system, it's an open question to what extent SLE can improve the performance of commercial workloads whose lock behavior can be different, and whether SLE can be easily implemented when context switches among multiple threads can be complicated and hard-to-recognize in hardware level.

This report characterizes the lock behavior of three classes of commercial workloads, online transaction processing (OLTP), web server (Apache), and Java business middleware (SpecJBB). Traces generated by a full-system simulator (Simics) and a detailed memory system timing-model (Ruby) are used to determine the size and frequency of critical sections, the size and frequency of lock-free sections, and the amount of lock contentions. We observe that lock contentions in commercial workloads are not significant, suggesting little performance improvement can be achieved if not applying more sophisticated hardware optimizations. We also realize that identifying thread switching is not an easy issue but really necessary to implement speculation correctly. The significant amount of kernel locks and the different behavior they demonstrate also indicate that further investigations must not ignore kernel behavior.

1. INTRODUCTION

Commercial workloads such as databases and web servers are currently the dominant applications running on shared-memory multiprocessor servers. These applications are traditionally constructed as multithreaded programs to exploit thread-level parallelism. Synchronization mechanisms (often locks) have to be applied to ensure serialized accesses to shared data structures, leading to a potential performance bottleneck. Memory system characterization of commercial workloads [1] has demonstrated one aspect of commercial workloads that behaves differently from scientific workloads, suggesting further studies to fully understand commercial workloads. As far as we know, little work has been done to characterize the lock behavior of commercial workloads, leaving interesting questions to be answered.

On the other hand, Speculative Lock Elision (SLE for short) [2] has shown that critical section serializations may not be necessary at runtime, partly because the access to a particular piece of shared data takes only a small fraction of the whole critical section, so that hardware optimization can remove many lock-induced thread serializations. However, this technique is evaluated only against scientific workloads without the presence of operating system. It's still an open question whether SLE will help improve the performance of server workloads whose lock implementation could be fine-tuned, and context switches can induce extra misspeculation and complicate the hardware implementation.

In this project, we characterize the lock behavior of commercial workloads using traces generated by a full-system simulator. Our goals are to gain more understanding of commercial workloads in general, and to validate Speculative Lock Elision technique in particular.

The rest of this report is organized as follows. In Section 2 we describe our characterization methods, including simulator and workloads setup and the critical section identification algorithm. The characterization results are presented in Section 3, detailing the size and frequency of critical sections, the size and frequency of lock-free sections, and the amount of lock contentions. Section 3 also contains discussions on timing-models and quantification of lock spinning, to support our interpretation of the observed behavior. Several issues related to context switches, SLE implementation and sophisticated lock implementation are discussed in Section 4. We conclude in Section 5 and describe the directions for future works.

2. METHODS

2.1 Basic Concepts

In this report, "*Critical Section*" is defined as a code segment that only allows one thread to enter at any time. Lock/unlock pairs are one of the most straightforward and commonly used mechanisms to guard a critical section. In this scheme, a lock is acquired by a thread when entering a critical section, and released when it leaves the critical section. Critical sections can be nested in a dynamic execution trace, in which case the outermost critical section marks the region where only one thread can enter. If considering intertwined critical sections, the instructions covered by any of the cross-cutting sections altogether constitute a merged and larger critical section. When acquiring a lock held by another thread, a thread usually *spins* to wait until the lock is free.

As shown in Figure 1, the instructions trace extracted from a certain execution can be broken down into three parts: (1) *critical section*, which includes all the instructions inside any individual critical sections; (2) *lock contention*, which includes instructions spent on spin-waiting when a thread tries to get a busy lock; (3) *lock-free section*, which includes instructions executed outside critical sections and contentions. A large critical section suggests higher probability for one entering thread to encounter another thread in the critical section. A small portion of lock-free section leaves fewer instructions for parallel execution. An execution without lock contention wastes no instruction.

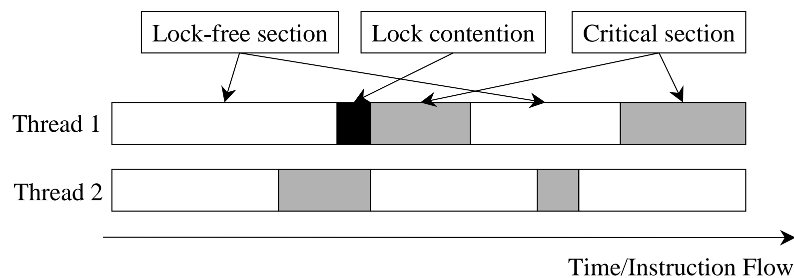

Figure 1: Execution breakdown

2.2 Simulator and Benchmarks

Our benchmarks are provided by the Multifacet research group at University of Wisconsin [4]. The three commercial benchmarks are: (1) OLTP, an IBM DB2 database server running TPCC benchmark; (2) Apache, a HTTP server fetching static web pages; (3) JBB, a SpecJBB implementation to provide middleware functionalities in a three-tier business system. The fourth benchmark Barnes is a scientific benchmark selected from the SPLASH-2 benchmark suite, serving as the scientific workloads representative to compare with the other benchmarks.

Simics [5] is the full-system simulator we use to generate traces for a 16-node system similar to Sun E10000. Simics itself uses a simple timing-model, assuming each instruction takes 1 cycle. A detailed memory system timing-model is provided by Ruby, a simulator developed by Multifacet group, which models the bandwidth consumption and latency for cache and memory accesses. Adding Ruby makes the simulation several times slower, but gives different and more realistic traces.

We run these benchmarks each for 100 transactions (Barnes can only run for 2 transactions before it crashes), and average the results from 3 runs if using Ruby (each with a different random number seed). Number of instructions is used as our measurement unit (instead of cycles), to simply the comparison between results produced by the two timing-models. Simulator Translation Caches (STC) are disabled for us to observe every memory access, `CPU_switch_time` is set to 1 to avoid heterogeneous processor progress caused by Simics' scheduling policy.

2.3 Lock Identification Algorithm

OLTP	Values
ldstub [%o0 + %g0], %o4 brnz,pn %o4, <0x10034b98> stbar	**0x0->0xff**
… …	… …
stb %g0, [%o0 + 12]	**0xff->0x0**

JBB	Values
casa [%l2] 128,%g4,%g3	**0x1->0x8410f8bc**
… … …	… … …
casa [%l2] 128,%l0,%g4	**0x8410f8bc->0x1**

Figure 2: Critical section examples in SPARC V9 assembly code

Our lock identification algorithm is based on the following assumptions, as also described in [2]: (1) Lock acquisition must use at least one atomic instruction to change the value of the lock; (2) As a pair, the stores in lock acquisition and release operations are silent (meaning if critical section memory operations occur atomically, the two stores should appear as if they never happened). The target ISA SPARC V9 has three atomic read-modify-write instructions: (1) *ldstub* (load-store-unsigned-byte), which reads a byte from memory into a register, and writes 0xFF into memory; (2) *swap* which exchanges the values in a register and a memory location; (3) *casa* and *casxa* (compare-and-swap) which does a *swap* between the memory and the second register if the value in the first register equals to the value in memory. Figure 2 shows two critical section examples extracted from the OLTP and JBB benchmarks: OLTP uses *ldstub* to acquire the lock (changing the lock's value from 0xo to 0xFF), and release the lock using a normal store instruction (changing the value back to 0x0); JBB uses a pair of *casa* instructions to guard the critical section, using value 0x1 to indicate that the lock is free.

The algorithm works as follows: for every atomic instruction, if it writes to the memory a value different from the original one, it is recognized as a successful lock acquisition. If the value written back is the same, it means a failed lock acquisition (and marks the beginning of a lock contention). It then examines every store made to the memory address, until a normal store (which is not part of an atomic instruction) by the same CPU completes the silent store pair. Since the two stores indicate the beginning and end of a critical section, we call this pattern a *resolved* critical section. As in the examples, the completing store instruction can be an atomic instruction (say *casa*). The reasons that we limit the completing store to normal stores are (1) except for JBB, over 80% atomic instructions can be recognized as locks to protect critical sections using this pattern; (2) this pattern does realize a critical section, while a silent store pair constituted by two atomic instructions may indicate non-critical sections (for example, a pair of join and departure events in a barrier, or a pair of lock release and acquisition events indicating the beginning and end of a *lock-free section*).

It is worth noting that not all critical sections can be recognized using the silent-store-pair pattern. For example, reader-writer locks are used to increase the concurrency of critical section accesses, but instead of one large sleep/wakeup-protected critical section, two small lock-guarded critical sections (when entering and leaving the real critical section) will be identified by our scheme. To fully understand the synchronization behavior of such applications, more sophisticated critical section identification schemes are required. Since databases use lots of reader-writer locks, we will discuss this issue in Section 4.2.

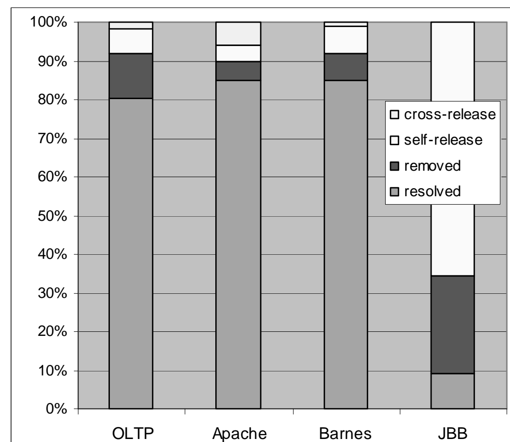

Figure 3: Critical section completion patterns

Three other critical section completion patterns are also observed: (1) *cross-release*, the lock value is changed by an atomic instruction executed on a different CPU; (2) *self-release*, the lock value is changed by an atomic instruction executed on the same CPU, for example, most of the critical sections in JBB are completed in a self-release fashion; (3) *removed*, no lock release is observed

within the limited window (16K instructions) used to contain a critical section. Figure 3 gives the component breakdown of different critical section completion patterns, the number of self-release component is halved since two atomic instructions are used to protect one critical section. By merging JBB's *self-release* and *resolved* components, we can see that using our identification algorithm, over 80% of all atomic instruction occurrences in our benchmarks are associated with critical sections., the majority of atomic instructions are used to protect critical sections. The rest of the report will be based on our observations of these 80% atomic instructions, and we wish to examine the remaining 20% as part of our future work.

3. CHARACTERIZATION RESULTS

In this section we present the result of commercial workloads lock behavior characterization, firstly on lock frequencies, then the size and frequencies of critical sections and lock-free sections. After discussing the difference between results using different timing-models and its implications, the amount of lock contention is given, based on the distinction between lock spinning and waiting.

3.1 Lock Frequency and Execution Breakdown

Figure 4 illustrates the overall critical section (including both kernel and user mode locks) frequencies per 10 thousands instructions. Among the four benchmarks we use, Barnes has the least frequent critical section occurrences, while Apache and JBB have the most frequent ones. Surprisingly, OLTP on average has only 5 critical sections per 10000 instructions, since we know that every query should be inside a critical section to maintain data consistency, the number of instructions within a query (not necessarily inside a critical section that our algorithm can identify) must be large in order to make critical sections less frequent.

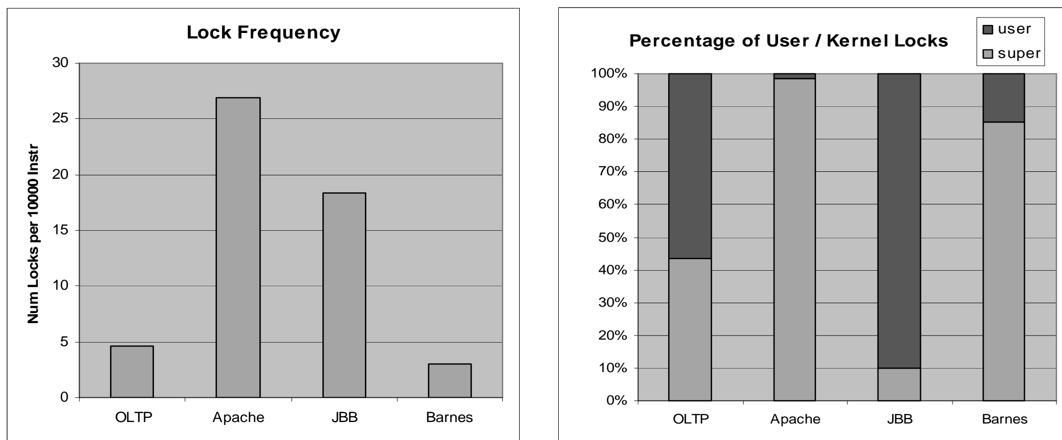

Figure 4: Lock Frequency and components breakdown

The second diagram in Figure 4 shows the fractions of user and kernel locks. The majority of locks in Apache and Barnes are in kernel mode, emphasizing the importance of observing kernel lock behavior for both scientific and commercial workloads. About 90% percent of the locks in JBB are user locks, which can be explained by frequent calls made to the user-level lock implementations provided by Java Runtime Environment. OLTP has significant amount of user locks because the DBMS itself must control the concurrent accesses to shared index and tuples made by different query threads.

Figure 5 outlines the results going to be presented in the later sections, fractions of instructions executed in lock-spinning, critical section, and lock-free section are reported. The breakdown data for both simple and Ruby timing-models are grouped together for each benchmark. Because the majority of JBB user locks can't be recognized by our algorithm, we will not include its data from now on, except for its critical section size distribution.

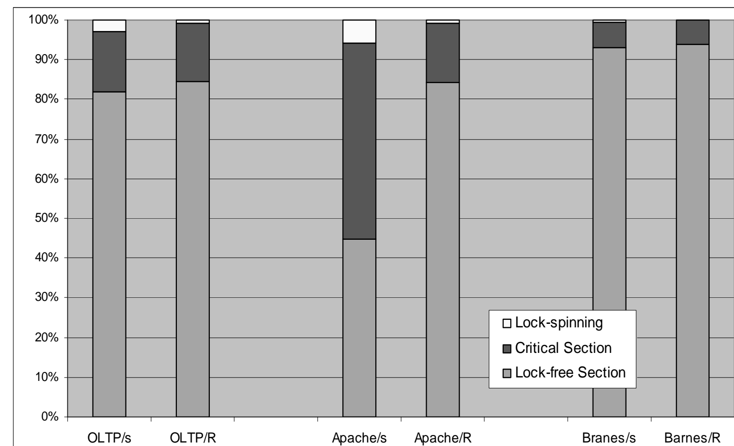

Figure 5: Component breakdown of instruction executed

Figure 5 tells us the fraction of instructions spent on lock-spinning is small, usually within 1-5% of all the instructions executed. These numbers shrinks when using Ruby timing-model, posing a performance improvement limit lower than 1%. Except for Apache running under the simple timing-model (the large fraction of critical section is contributed mostly by very few but large critical sections), about 6% to 16% instructions are executed within critical sections. The remaining 80+% instructions are not related to locks or critical sections.

3.2 Critical Section Sizes

Figure 6 gives the detail data on critical section sizes. The data for each benchmark are grouped in the same column. Only data collected from the simple timing-mode are reported here. The results for both timing-models are shown in the Appendix, demonstrating very little difference.

The first row shows the distributions of user mode critical section sizes, locks implemented with different atomic instructions and the sum of all locks are plotted using different curves. The x-axis represents the buckets containing critical sections whose sizes are not larger than 2 to the *x*th power instructions. The y-axis represents the number of critical section occurrences in each bucket. We only count critical sections whose sizes are smaller than 16K instructions. The second row shows the kernel mode critical section sizes, with the same x-axis and y-axis meaning. The third row shows the cumulative probability functions of user, kernel and total critical section sizes.

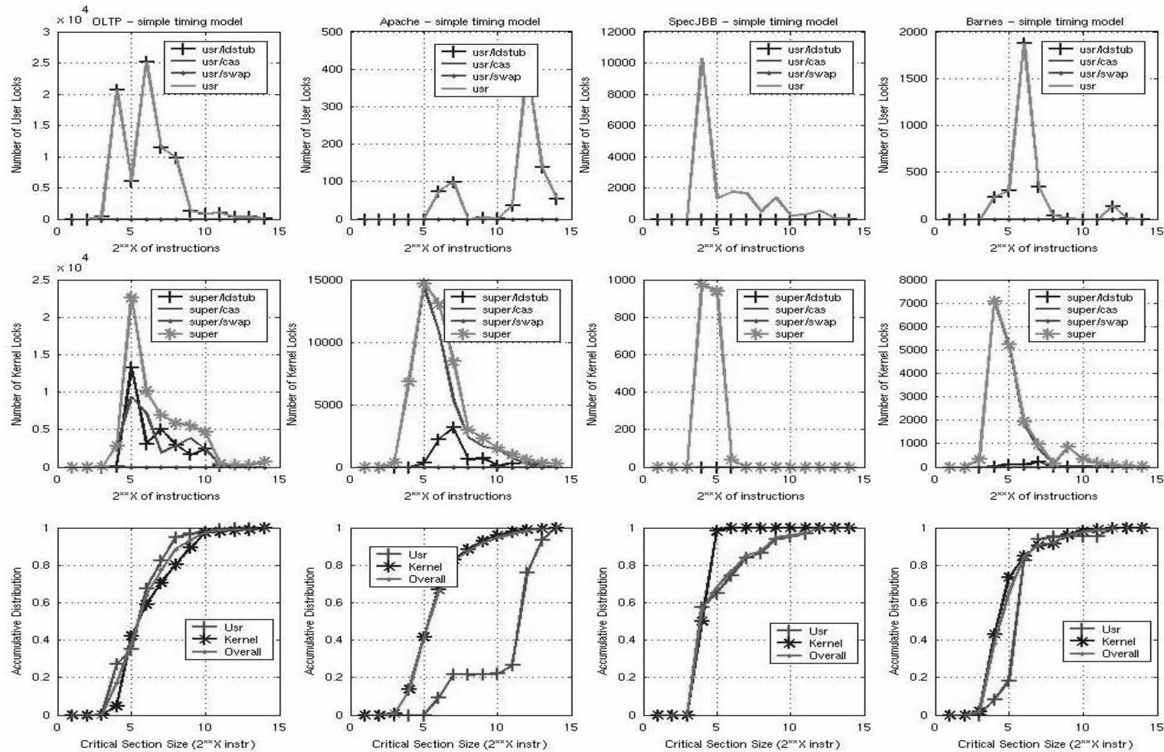

Figure 6: Critical section size distribution (simple timing-model)

Several observations can be made from Figure 6: (1) Inside the Solaris kernel, about 40% critical sections have no more than 32 instructions, over 90% critical sections are not larger than 512 instructions. (2) Except for Apache, commercial workloads have small critical sections, most of the OLTP critical sections are ranged from 16 to 512 instructions, 60% of the JBB critical sections are no more than 32 instructions in size. (3) Apache has some large user mode critical sections, ranging from 1K to 8K instructions. (4) Barnes-Huts also have small critical sections (from 16 to 256 instructions). (4) For this version of Solaris, kernel locks implemented in *ldstub* and *casa* instructions are commonly used. (5) For the benchmarks we use, except for JBB (which uses lots of *casa* locks), user locks implemented in *ldstub* instruction are frequently used.

3.3 Lock-Free Section Sizes

Figure 7 shows the sizes of lock-free sections and outermost critical sections, which also reflects the interleaving between these two types of sections. The traces generated are post-processed to produce these data. The data for each benchmark are grouped in the same row, with results from simple timing-model on the left-hand side, and results from Ruby on the right-hand side. The x and y axes have the same meaning as those in the first row of Figure 5.

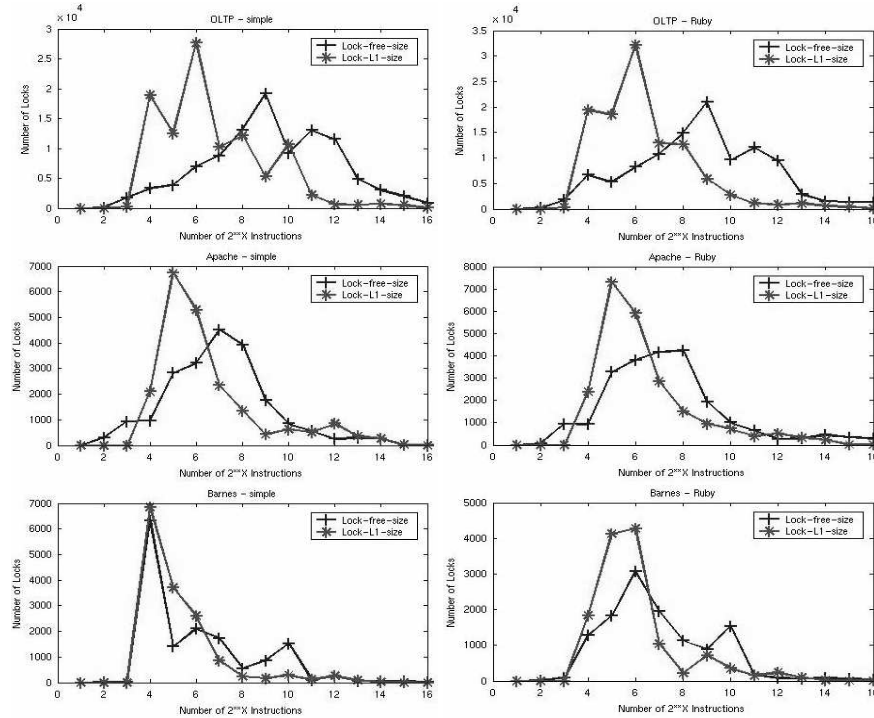

Figure 7: Distribution of lock-free section and outermost nested critical section sizes

Figure 7 show that the size distribution of outermost critical sections coincide with the size distribution of critical sections in all (although 1/3 to 2/3 critical sections are not counted because of nesting). The sizes of lock-free sections are generally 8 times larger than outermost critical sections, except in the case of Barnes where many critical sections occur in a “clustered” fashion, generating many small lock-free sections in between. Interestingly, for OLTP and Apache, the distribution shapes are similar even using different timing-models. The significant difference demonstrated by Barnes might be an artifact of counting small number of lock-free sections occurred in short runs.

3.4 Timing-models

In this section, we discuss the effect of using more realistic timing model and provide our explanation on why Ruby can reduce the amount of lock contentions. Conceptually, using a different timing-model can (1) change the dynamic ordering of events within the same execution

paths taken by multiple threads, (2) expose the non-determinism of multithreaded program, leading to different execution paths [4], both contributing to a different amount of lock contention. Since we are interested in lock contention determined by the ordering of dynamic lock acquisition and release events, a more realistic timing-model will provide more realistic result. However, the problems with realistic timing-models are (1) they slow down the simulation by at least several times, (2) they are harder to implement, modify and integrate with our tracing module. Although we are not clear about how and in which way Ruby is going to change our result, ideally we want to observe little difference between results using the simple and Ruby timing model, to save the effort of adding even more realistic timing details. If difference exists, we hope this can give us insights on the effect of timing-models, to help predict what lock behavior a real execution will demonstrate.

In the previous sections, we notice that using Ruby timing-model doesn't change the critical section and lock-free section size distributions (with one exception of Barnes). It give us confidence in the characterization results for the two parts, making us to expect seeing similar behavior when adding modern processor timing-models. However, we also notice that Ruby surprisingly reduces the amount of lock-spinning (as shown in Figure 5). We see two possible reasons for less contentions, and present some preliminary data to support the "shrinking critical section effect" explanation:

1. "cooperative critical section entrance", in which case multiple threads are arranged to enter the critical section in a cooperative fashion, thus avoiding lock contentions;
2. "shrinking critical section effect", in which case the relative size of critical sections are shrunk so that statistically multiple threads are less likely to enter the same critical section at the same time.

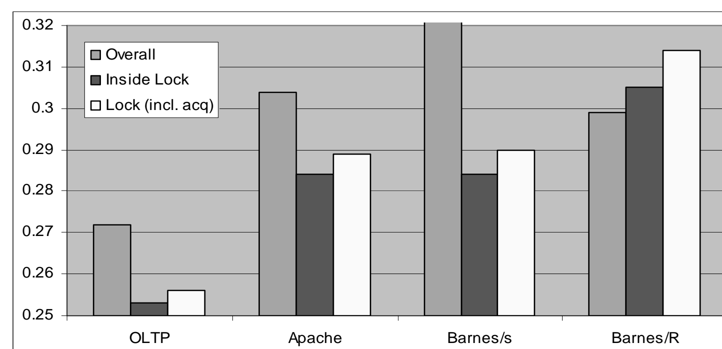

Figure 8: Frequencies of memory access operations inside and outside critical sections

The only difference between the two timing-models is that Ruby stresses cache and memory access latencies. Ideally we want to observe that inside critical sections, smaller fraction of execution time are spent on memory access, which requires both cache miss rate and memory access frequency

information inside and outside critical sections. In practice, we failed to get cache miss information from Ruby, but we observed that for our benchmarks, memory access operations happen less frequently inside critical sections. Figure 8 shows that for OLTP and Apache, instructions inside critical sections have 10% less memory accesses compared with the overall instruction trace[1]. By assuming a uniform cache miss rate within the same execution[2], we see less execution time spent on memory access inside critical sections, which can partly explain why Ruby causes less lock-spinning. Figure 9 illustrates how Ruby changes the relative size of critical sections based on the above observations. Given a detailed Out-of-Order superscalar processor timing-model, we expect to see even less contentions because the memory access latency will be further emphasized.

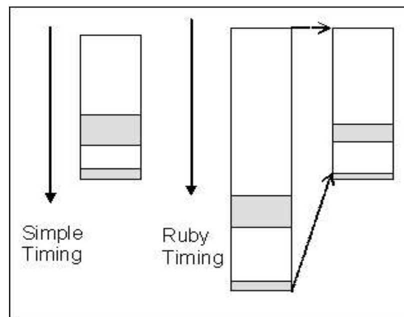

Figure 9: Critical section shrinking

(caused by less memory accesses and longer latency, assuming uniform miss rates)

3.5 Amount of Lock Contention

Lock Waiting Time and Lock Spinning Time

Lock contention is caused when one thread tries to enter the critical section held by another thread. Lock contention wastes computing resources since the acquiring thread has to hold its logical execution until the lock is released by another thread. *Lock waiting time* is the presumptuous idea about how many instructions the thread wastes on acquiring a lock. It is the interval starting from the first try to acquire lock and ending with the successful acquisition. However, system doesn't always waste execution during this interval. After spinning for several times, the thread may choose to be switched off to sleep, giving the CPU to kernel or another thread. Later it will be waken up to resume its lock acquisition. Obviously in this case the other thread or the kernel may perform

---

[1] As shown in Figure 8, the memory access frequency differences using the two timing models are inconsistent for benchmark Barnes-Hut. This undermines the strength of the "shrinking effect" hypothesis, further investigations are required on this issue.

[2] We can collect miss rate information by moving the tracer function into Ruby, to observe both lock/unlocks and cache accesses. This can be part of our future work.

meaningful work within this interval, therefore lock waiting time is not a precise metric to describe how much time is wasted because of lock contentions.

If thread switch could be identified, it is easier to attribute wasted instructions to lock acquisition. However, thread switch identification is not easy, which is explained in section 4.2.

We propose to use *lock spinning* as a more precise metric to describe the actual instructions wasted on lock acquisition. If a lock waiting time is less than 4K instructions, then it is treated as a lock spinning, otherwise the lock waiting time is assumed to include some long waiting event such as thread/process switch. The 4K instruction threshold is based on observations in the simple timing model, in which over 90 percent of lock contentions involve lock waiting time less than 4K instructions. Figure 10 shows that why it is necessary to distinguish between lock waiting and lock spinning. In some cases like OLTP/Ruby, the lock waiting time is even larger than the total execution time since it counts meaningful execution of other threads. Lock spinning uses 4K threshold to filter out those long waiting events which are susceptible to include thread/process switch.

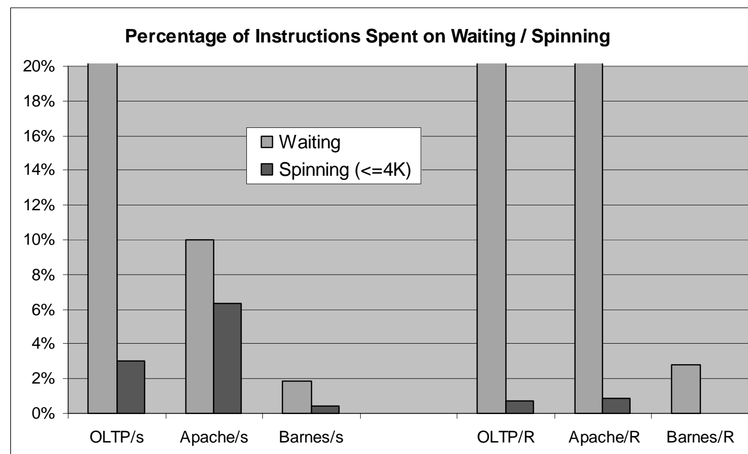

Figure 10: "lock waiting time and lock spinning time"

Lock Contention Characterization

Features of lock contention are discussed here, trying to reveal potential opportunities for performance improvement. The first phenomenon we notice is that, a small fraction of static lock instructions account for a large part of lock contentions during dynamic execution. For kernel lock contention, 5% of the most contented locks account for 90+% dynamic lock contentions, while for user lock, 23% locks are identified as highly contented. The converged lock contention makes the impression that they are ideal candidates to apply performance optimizations.

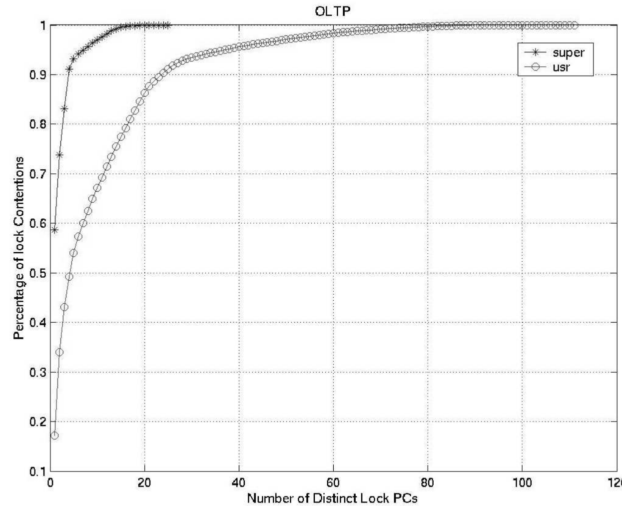

Figure 11: “lock waiting time and lock spinning time”

However, that is not the case. Only a small percentage of wasted instructions is caused by those highly contented locks. The 5% most highly contented locks only account for 7% wasted instructions, although they account for over 98% dynamic contentions. It means that even if lock optimization techniques are applied to those highly contented locks, they won’t help much.

Still, it is very encouraging to observe that a different and small fraction of static locks waste a large percentage of instructions. In Figure 12, 15% static locks (20-25%, 30-35% and 50-55%) account for 60% percent of wasted instructions. Therefore it is still possible to apply optimizations on these hot spots for better performance. Further study is needed on whether those locks are predictable, and compiler or profiling techniques might be needed to locate them.

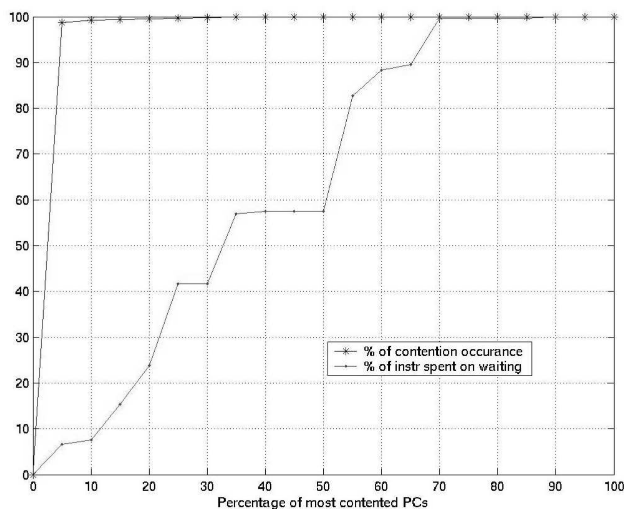

Figure 12: “lock contention and wasted instruction”

4. RELATED ISSUES

SLE implementation is discussed in this section with characterization data collected from commercial workload, its possible performance gain is also estimated. After that, some important issues such as context switch and synchronization are described focusing on their implication on SLE implementation.

4.1 Speculative Lock Elision Implementation Issues

Size of speculative buffer

SLE idea requires that speculative execution within critical section must hold uncommitted state in a speculative buffer. Small critical sections in commercial workloads determine that the buffer used to hold uncommitted states don't have to be large. Assuming 1/3 memory access are store operations and 30% of the instructions are memory access instructions, a 64-entry (512 * 30% * 1/3 = 64) buffer should suffice to hold the output for a critical section with 512 instructions.

How to organize those speculative buffers in the multithreading/multiprocessor case is discussed in detail in section 4.2 (context switch). Ravi Rajwar, the author of SLE[2], suggests that it is better to have a single shared buffer among multiple threads running on the same CPU.

Possible performance gain

If only considering the spinning instructions for which SLE can remove, there are only 1-5% performance improvement on commercial workloads. There are two major reasons to explain this unexpected observation. The small critical section size means overlapping execution within it doesn't achieve much benefit compared with sequential execution. Secondly, infrequent lock contention is observed, which means there are not too many opportunities left for SLE to exploit.

Lock spinning latency can be larger if Ruby is used to model the latency of read/write memory used within the lock-spinning phase. Therefore SLE could have more wasted time to save. However, less lock contentions are also observed in Ruby case, decreasing the amount of improvement available for SLE. The combined effect of these two factors is not clear yet.

Hot spot phenomenon of lock contentions makes it attractive to treat the locks causing hot spot and the rest differently. SLE involves some overhead since it needs to rollback in case of misspeculation. Based on observation in lock contention characterization, SLE idea could achieve maximum performance improvement if it only focuses on the small fraction of static locks which cause a large part of wasted instructions.

4.2 Context Switches and Synchronization

Need to identify process/thread switch

In section 3.4, it is mentioned that thread/process switch makes difference between lock waiting time and lock spinning time. If thread/process switch could be identified by hardware, then the lock waiting time interval could be divided correctly into lock spinning time belonging to the acquiring thread and meaningful execution of other threads. Therefore lock spinning time could be precisely quantified to show how much computing resource are wasted on lock acquisition, replacing that 4K threshold solution.

Correct implementation of SLE idea is another reason for identifying process/thread switch. SLE doesn't address exception/trap occurring within critical section execution. Some exceptions/traps, such as data/instruction TLB miss exceptions, resume the interrupt execution, therefore, occurrence of those exceptions/traps doesn't hurt system integrity. However, there are other exceptions/traps which will switch execution into kernel and may invoke process/thread scheduling. Some commercial workloads have large enough critical section to permit occurrence of those exceptions/traps (Apache have some user level critical section sized from 1K to 8K instructions). This thread/process switch within speculative execution of critical section code can change the atomicity semantics of critical section. The next thread/process could observe the uncommitted state of the previous interrupted execution. Therefore it is required either to rollback speculative execution or to hold uncommitted states in thread/process specific speculative buffer.

Rollback option is a simple and effective solution. When hardware detects that a process/thread switch occurs within speculative execution of critical section code, it will respond in the same way as in case of misspeculation. Execution will be rollback to the lock acquisition code, and all states in speculative buffer be flushed. The machine state will eventually be the same as the process/thread switch happens before lock acquisition.

Thread/process specific speculative buffer might not be a good idea. Supposes a thread is speculatively executed within critical section and thread scheduling happens, the thread has to hold its uncommitted state in its specific buffer. The next thread may also speculatively execute the same critical section. A conflict detection mechanism like cache coherence protocol is needed to monitor speculative executions of different threads. It is non-trivial task and the non cost-effective implementation expense makes it unattractive.

In short, no matter which option is chosen, process/thread switch within critical section execution should be identified and proper measures should be taken to guarantee correct execution of SLE.

Process Switch Identification

When process context switch happens, system will deactivate some entries in TLB corresponding to the current context. This operation is implemented by a kernel method sfmmu_tlb_demap() in Solaris. By observing occurrence of this method during dynamic execution, process context switch could be identified.

Our analysis of trace shows that process switch happens very infrequently. For apache, the average execution interval between process switch is 210K instructions, with maximum interval of 360K instructions and minimal interval of 160K instructions. Considering each apache transaction executes 118K instructions on average, process context switch is a very rare case.

Thread Switch Identification

Thread switch identification is really a hard issue. Up to now, we could only conclude that it is an unresolved issue. There are a couple of reasons leading to this conclusion.

There is no special operation to uniquely indicate that a thread switch occurs. Thread switch involves a set of interleaved method invocations (resume, _resume_from_idle, disp, swtch..), and there are no regular pattern how these methods are interleaved and called. Also it is impossible to identify kernel thread switch by only observing register window swap since it also happen in user thread switch. Another primary obstacle is that there are no feedbacks from OS to validate our assumption. Up to now we could only have some brute observations, like resume() may indicate a kernel thread switch and user_rtt() may indicate a user level thread switch.

Synchronization Issues

It is hard to identify the complex synchronization like barrier, reader-writer lock by only observing instructions at the hardware level. Their implementations don't expose regular behavior as simple locks do.

Another phenomenon we observe is mutual exclusion behavior in source code level is not directly mapped to hardware level. Sometime it is composed of several locks. For example, mutex exclusion mechanism in pthread library, pthread_mutex_lock(&lock) acquires three locks. In source code level some complex synchronization mechanism is used to guard one critical section. However, it is observed as a couple of critical sections in hardware level. For example, reader-writer lock maintains some data structures protected by the critical sections within procedure write_enter() and write_exit(), and those data structures will determine current execution can enter the program level critical section or just wait in write_enter(). By understanding this, it is clear that there is a long

way ahead to improve performance of source code level behavior by applying optimization techniques in hardware level.

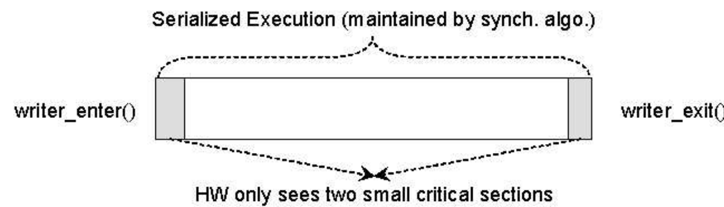

Figure 13: “reader-writer lock example”

5. CONCLUSION AND FUTURE WORK

In the paper, we present the methods and results of lock behavior characterization for commercial workloads, as well as the issues related to hardware optimization to improve concurrency. We observe that small critical sections dominate commercial workloads, and lock contention is infrequent and insignificant in terms of instructions spent on spinning. We also notice that user and kernel code have rather different characteristics, emphasizing the importance of charactering both user and kernel locks. Based on the above observations, SLE will at most improve performance by 1-5% (in terms of instruction count), probably not as much as it does for the scientific benchmarks evaluated in [2]. Furthermore, switches among multiple processes and threads can complicate the correct implementation of speculative execution.

Future work can be done in both (1) completing the unfinished characterization and (2) identifying possible micro-architectural innovations to improve critical section performance. For the first task, we can characterize the synchronization mechanism implemented using the remaining 20% atomic instruction occurrences, measure the cache miss rates inside and outside critical sections and further validate our results against a detailed out-of-order processor timing model. For the second task, we should focus on identifying more parallelism currently not recognizable by hardware, either via compiler analysis or software annotation, or by providing powerful synchronization primitives using sophisticated hardware support.

REFERENCES

[1] Luiz Andre Barroso, Kourosh Gharachorloo, and Edward Bugnion, Memory System Characterization of Commercial Workloads, Proc. International Symposium on Computer Architecture, June 1998

[2] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. 34th International Symposium on Microarchitecture, 2001.

[3] J. Martinez and J. Torrellas. Speculative Locks for Concurrent Execution of Critical Sections in Shared-Memory Multiprocessors. Workshop on Memory Performance Issues, June 2001.

[4] Alaa R. Alameldeen, Carl J. Mauer, Min Xu, Pacia J. Harper, Milo M.K. Martin, Daniel J. Sorin, Mark D. Hill and David A. Wood. Evaluating Non-deterministic Multi-threaded Commercial Workloads. Computer Architecture Evaluation using Commercial Workloads (CAECW-02), February 2002.

[5] Virtutech AB. Simics Full System Simulator. http://www.simics.com/.

[6] David L. Weaver and Tom Germond. The SPARC Architecture Manual, Version 9. SPARC International, Inc., Prentice Hall, Englewood Cliffs, NJ 07632, 1994.

ACKNOWLEDGEMENTS

We thank Professor Mark Hill for suggesting the topic for our course project and his constant guidance and support. We also thank Min Xu our TA for helpful discussions on lock identification and comments on how to use the simulators correctly, Carl Mauer for providing us helpful information on various topics (interactions between Simics, Ruby and Opal, context and thread switching inside Solaris, as well as kernel disassembly code), Ravi Rajwar for many discussions on Speculative Lock Elision and its implementation issues.

APPENDIX

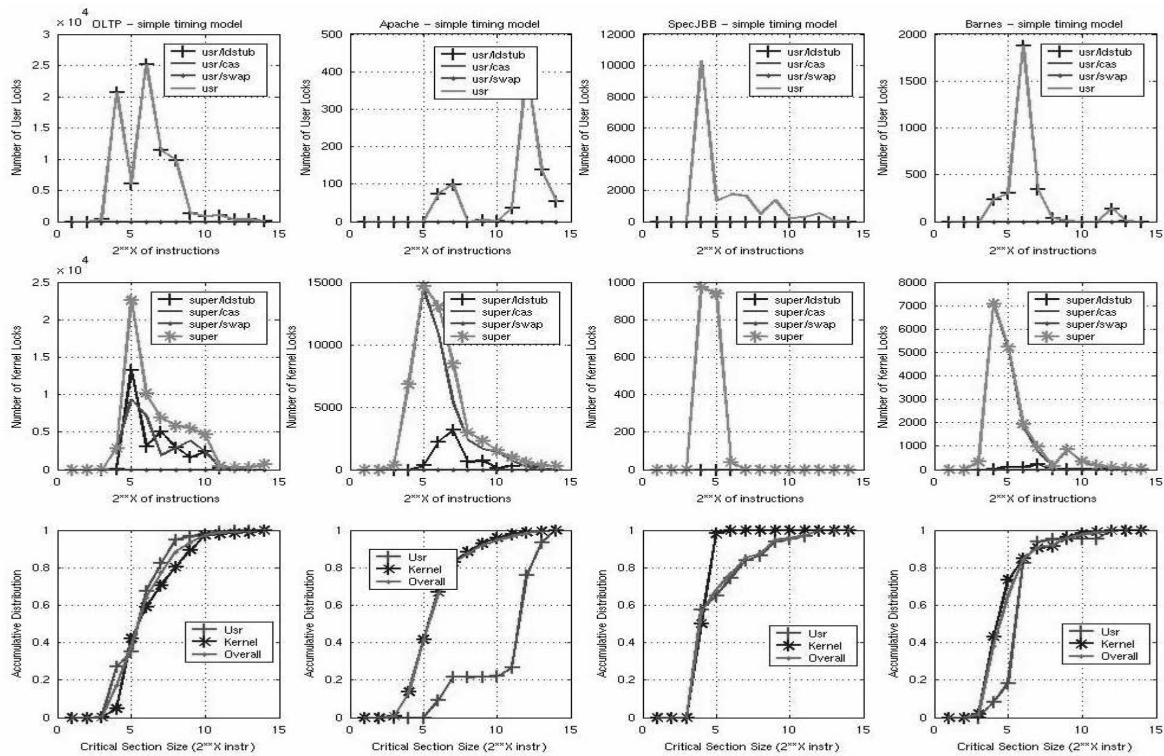

Figure A1: Critical section size distribution using the simple timing-model

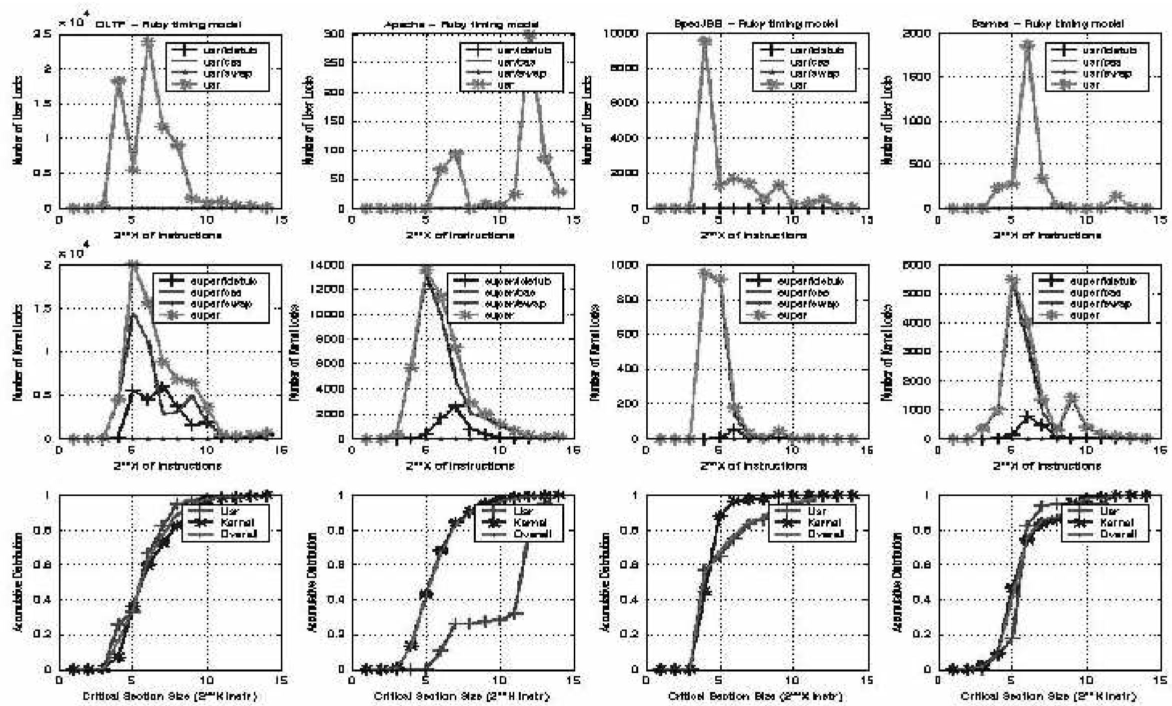

Figure A2: Critical section size distribution using Ruby timing-model