Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures

M. Aater Suleman
University of Texas at Austin
suleman@hps.utexas.edu

Onur Mutlu
Carnegie Mellon University
onur@cmu.edu

Moinuddin K. Qureshi
IBM Research
mkquresh@us.ibm.com

Yale N. Patt
University of Texas at Austin
patt@ece.utexas.edu

Abstract

To improve the performance of a single application on Chip Multiprocessors (CMPs), the application must be split into *threads* which execute concurrently on multiple cores. In multi-threaded applications, critical sections are used to ensure that only one thread accesses shared data at any given time. Critical sections can serialize the execution of threads, which significantly reduces performance and scalability.

This paper proposes *Accelerated Critical Sections (ACS)*, a technique that leverages the high-performance core(s) of an Asymmetric Chip Multiprocessor (ACMP) to accelerate the execution of critical sections. In ACS, selected critical sections are executed by a high-performance core, which can execute the critical section faster than the other, smaller cores. Consequently, ACS reduces serialization: it lowers the likelihood of threads waiting for a critical section to finish. Our evaluation on a set of 12 critical-section-intensive workloads shows that ACS reduces the average execution time by 34% compared to an equal-area 32-core symmetric CMP and by 23% compared to an equal-area ACMP. Moreover, for 7 of the 12 workloads, ACS also increases scalability (i.e. the number of threads at which performance saturates).

Categories and Subject Descriptors C.0 [*General*]: System architectures

General Terms Design, Performance

Keywords CMP, Critical Sections, Heterogeneous Cores, Multi-core, Parallel Programming, Locks

1. Introduction

It has become difficult to build large monolithic processors because of their excessive design complexity and high power consumption. Consequently, industry has shifted to Chip-Multiprocessors (CMP) [22, 47, 44] that provide multiple processing cores on a single chip. To extract high performance from such architectures, an application must be divided into multiple entities called *threads*. In such multi-threaded applications, threads operate on different portions of the same problem and communicate via shared memory. To ensure correctness, multiple threads are not allowed to update shared data concurrently, known as the *mutual exclusion* principle [25]. Instead, accesses to shared data are encapsulated in regions of code guarded by synchronization primitives (e.g. locks). Such guarded regions of code are called *critical sections*.

The semantics of a critical section dictate that only one thread can execute it at a given time. Any other thread that requires access to shared data must wait for the current thread to complete the critical section. Thus, when there is contention for shared data, execution of threads gets serialized, which reduces performance. As the number of threads increases, the contention for critical sections also increases. Therefore, in applications that have significant data synchronization (e.g. Mozilla Firefox, MySQL [1], and operating system kernels [36]), critical sections limit both performance (at a given number of threads) and scalability (the number of threads at which performance saturates). Techniques to accelerate the execution of critical sections can reduce serialization, thereby improving performance and scalability.

Previous research [24, 15, 30, 41] proposed the *Asymmetric Chip Multiprocessor (ACMP)* architecture to efficiently execute program portions that are not parallelized (i.e., Amdahl's "serial bottleneck" [6]). An ACMP consists of at least one large, high-performance core and several small, low-performance cores. Serial program portions execute on a large core to reduce the performance impact of the serial bottleneck. The parallelized portions execute on the small cores to obtain high throughput.

We propose the *Accelerated Critical Sections (ACS)* mechanism, in which selected critical sections execute on the large core[1] of an ACMP. In traditional CMPs, when a core encounters a critical section, it acquires the lock associated with the critical section, executes the critical section, and releases the lock. In ACS, when a core encounters a critical section, it requests the large core to execute that critical section. The large core acquires the lock, executes the critical section, and notifies the requesting small core when the critical section is complete.

To execute critical sections, the large core may require some *private data* from the small core e.g. the input parameters on the stack. Such data is transferred on demand from the cache of the small core via the regular cache coherence mechanism. These transfers may increase cache misses. However, executing the critical sections exclusively on the large core has the advantage that the *lock* and *shared data* always stays in the cache hierarchy of the large core rather than constantly moving between the caches of different cores. This improves locality of lock and shared data, which can offset the additional misses incurred due to the transfer of private data. We show, in Section 6, that critical sections often access more shared data than private data. For example, a critical section that inserts a single node of private data in a sorted linked list (shared data) accesses several nodes of the shared list. For the 12 workloads used in our evaluation, we

[1] For simplicity, we describe the proposed technique assuming an implementation that contains one large core. However, our proposal is general enough to work with multiple large cores. Section 3 briefly describes our proposal for such a system.

find that, on average, ACS reduces the number of L2 cache misses inside the critical sections by 20%.[2]

On the other hand, executing critical sections exclusively on a large core of an ACMP can have a negative effect. Multi-threaded applications often try to improve concurrency by using data synchronization at a fine granularity: having multiple critical sections, each guarding a disjoint set of the shared data (e.g., a separate lock for each element of an array). In such cases, executing all critical sections on the large core can lead to "false serialization" of different, disjoint critical sections that could otherwise have been executed in parallel. To reduce the impact of false serialization, ACS includes a dynamic mechanism that decides whether or not a critical section should be executed on a small core or a large core. If too many disjoint critical sections are contending for execution on the large core (and another large core is not available), this mechanism selects which critical section(s) should be executed on the large core(s).

Contributions: This paper makes two contributions:

1. It proposes an asymmetric multi-core architecture, ACS, to accelerate critical sections, thereby reducing thread serialization. We comprehensively describe the instruction set architecture (ISA), compiler/library, hardware, and the operating system support needed to implement ACS.
2. We analyze the performance trade-offs of ACS and evaluate design options to further improve performance. We find that ACS reduces the average execution time by 34% over an equal-area 32-core symmetric CMP (SCMP) and by 23% over an equal-area ACMP.

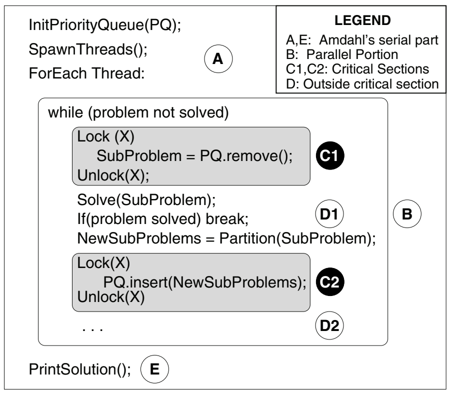
2. Background and Motivation

2.1 Amdahl's Law and Critical Sections

A multi-threaded application consists of two parts: the serial part and the parallel part. The serial part is the classical Amdahl's bottleneck [6] where only one thread exists. The parallel part is where multiple threads execute concurrently. When multiple threads execute, accesses to shared data are encapsulated inside critical sections. Only one thread can execute a particular critical section at any given time. Critical sections are different from Amdahl's serial bottleneck: during the execution of a critical section, other threads that do not need to execute the same critical section can make progress. In contrast, no other thread exists in Amdahl's serial bottleneck. We use a simple example to show the performance impact of critical sections.

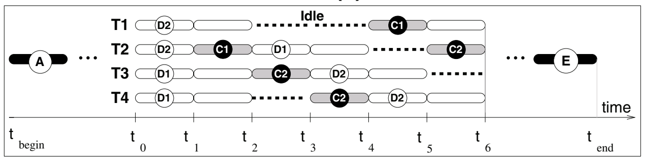
Figure 1(a) shows the code for a multi-threaded kernel where each thread dequeues a work quantum from the priority queue (PQ) and attempts to solve it. If the thread cannot solve the problem, it divides the problem into sub-problems and inserts them into the priority queue. This is a very common parallel implementation of many branch-and-bound algorithms [27]. In our benchmarks, this kernel is used to solve the popular 15-puzzle problem [50]. The kernel consists of three parts. The initial part A and the final part E are the serial parts of the program. They comprise Amdahl's serial bottleneck since only one thread exists in those sections. Part B is the parallel part, executed by multiple threads. It consists of code that is both inside the critical section (C1 and C2, both protected by lock X) and outside the critical section (D1 and D2). Only one thread can execute the critical section at a given time, which can cause serialization of the parallel part and reduce overall performance.

[2] We simulated a CMP with private L1 and L2 caches and a shared L3 cache. Section 5 describes our experimental methodology.

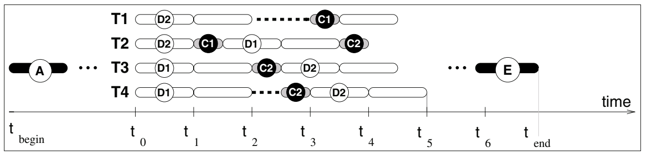


```
InitPriorityQueue(PQ);
SpawnThreads();
ForEach Thread:
    while (problem not solved)
        Lock (X)
            SubProblem = PQ.remove();
        Unlock(X);
        Solve(SubProblem);
        If(problem solved) break;
        NewSubProblems = Partition(SubProblem);
        Lock(X)
            PQ.insert(NewSubProblems);
        Unlock(X)
        . . .
PrintSolution();
```

(a)



(b)



(c)

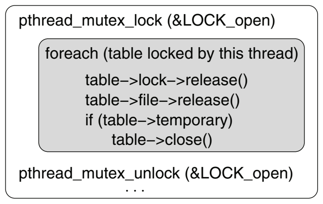
Figure 1. Serial part, parallel part, and critical section in a multi-threaded 15-puzzle kernel (a) Code example, and execution timeline on (b) the baseline CMP (c) accelerated critical sections

2.2 Serialization due to Critical Sections

Figure 1(b) shows the execution timeline of the kernel shown in Figure 1(a) on a 4-core CMP. After the serial part A, four threads (T1, T2, T3, and T4) are spawned, one on each core. Once part B is complete, the serial part E is executed on a single core. We analyze the serialization caused by the critical section in steady state of part B. Between time t_0 and t_1, all threads execute in parallel. At time t_1, T2 starts executing the critical section while T1, T3, and T4 continue to execute the code independent of the critical section. At time t_2, T2 finishes the critical section and three threads (T1, T3, and T4) contend for the critical section – T3 wins and enters the critical section. Between time t_2 and t_3, T3 executes the critical section while T1 and T4 remain idle, waiting for T3 to finish. Between time t_3 and t_4, T4 executes the critical section while T1 continues to wait. T1 finally gets to execute the critical section between time t_4 and t_5.

This example shows that the time taken to execute a critical section significantly affects not only the thread that executes it but also the threads that are waiting to enter the critical section. For example, between t_2 and t_3 there are two threads (T1 and T4) waiting for T3 to exit the critical section, without performing any useful work. Therefore, accelerating the execution of the critical section not only improves the performance of T3 but also reduces the useless waiting time of T1 and T4. Figure 1(c) shows the execution of the same kernel assuming that critical sections take half as long to execute. Halving the time taken to execute critical sections reduces thread serialization, which significantly reduces the time spent in the parallel portion. Thus, accelerating critical sections can provide significant performance improvement. On average, the critical section shown in Figure 1(a) executes 1.5K instructions. During an insert, the critical section accesses multiple nodes of the priority queue (implemented as a heap) to find a suitable place for insertion. Due to its lengthy execution, this critical section incurs high contention. When the workload is executed with 8 threads, on average 4 threads wait for this critical section at a given time. The average number of waiting threads increases to 16 when the workload is executed with 32 threads. In contrast, when this critical section is accelerated using ACS, the average number of waiting threads reduces to 2 and 3, for 8 and 32-threaded execution respectively.

We find that similar behavior exists in commonly-used large-scale workloads. Figure 2 shows a section of code from the database application MySQL [1]. The lock, `LOCK_open`, protects the data structure `open_cache`, which tracks all tables opened by all transactions. The code example shown in Figure 2 executes at the end of every transaction and closes the tables opened by that transaction. A similar function (not shown), also protected by `LOCK_open`, executes are the start of every transaction and opens the tables for that transaction. On average, this critical section executes 670 instructions. The average length of each transaction (for the `oltp-simple` input set) is 40K instructions. Since critical sections account for 3% of the total instructions, contention is high. The serialization due to the `LOCK_open` critical section is a well-known problem in the MySQL developer community [2]. On average, 5 threads wait for this critical section when the workload is executed with 32 threads. When ACS is used to accelerate this critical section, the average number of waiting threads reduces to 1.4.



```
pthread_mutex_lock (&LOCK_open)
    foreach (table locked by this thread)
        table->lock->release()
        table->file->release()
        if (table->temporary)
            table->close()
pthread_mutex_unlock (&LOCK_open)
...
```

Figure 2. Critical section at the end of MySQL transactions

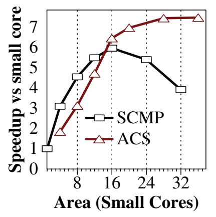


Figure 3. Scalability of MySQL

2.3 Poor Application Scalability due to Critical Sections

As the number of threads increases, contention for critical sections also increases. This contention can become so high that every thread might need to wait for several other threads before it can enter the critical section. In such a case, adding more threads to the program does not improve (and in fact can degrade) performance. For example, Figure 3 shows the speedup when MySQL is executed on multiple cores of a symmetric CMP (SCMP). As the number of cores increase, more threads can execute concurrently, which increases contention for critical sections and causes performance to saturate at 16 threads. Figure 3 also shows the speedup of an equal-area ACS, which we will describe in Section 3. Performance of ACS continues to increase until 32 threads. This shows that accelerating the critical sections can improve not only the performance of an application for a given number of threads but also the scalability of the application.

3. Accelerated Critical Sections

The goal of this paper is to devise a practical mechanism that overcomes the performance bottlenecks of critical sections to improve multi-threaded application performance and scalability. To this end, we propose *Accelerated Critical Sections (ACS)*. ACS is based on the ACMP architecture [30, 24, 15, 41], which was proposed to handle Amdahl's serial bottleneck. ACS consists of at least one large core and several small cores. The critical sections and the serial part of the program execute on a large core, whereas the remaining parallel parts execute on the small cores. Executing the critical sections on a large core reduces the execution latency of the critical section, thereby improving performance and scalability.

3.1 Architecture: A high level overview

The ACS mechanism is implemented on a homogeneous-ISA, heterogeneous-core CMP that provides hardware support for cache coherence. ACS leverages one or more large cores to accelerate the execution of critical sections and executes the parallel threads on the remaining small cores. For simplicity of illustration, we first describe how ACS can be implemented on a CMP with a single large core and multiple small cores. In Section 3.9, we discuss ACS for a CMP with multiple large cores.

Figure 4 shows an example ACS architecture implemented on an ACMP consisting of one large core (P0) and 12 small cores (P1-P12). Similarly to previous ACMP proposals [24, 15, 30, 41], ACS executes Amdahl's serial bottleneck on the large core. In addition, ACS accelerates the execution of critical sections using the large core. ACS executes the parallel part of the program on the small cores P1-P12. When a small core encounters a critical section it sends a "critical section execution" request to P0. P0 buffers this request in a hardware structure called the *Critical Section Request Buffer (CSRB)*. When P0 completes the execution of the requested critical section, it sends a "done" signal to the requesting core. To support such accelerated execution of critical sections, ACS requires support from the ISA (i.e., new instructions), from the compiler, and from the on-chip interconnect. We describe these extensions in detail next.

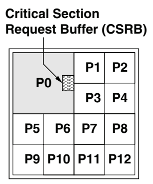
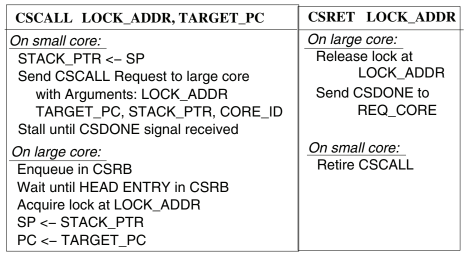


Figure 4. ACS



CSCALL LOCK_ADDR, TARGET_PC	CSRET LOCK_ADDR
On small core: STACK_PTR <- SP Send CSCALL Request to large core with Arguments: LOCK_ADDR TARGET_PC, STACK_PTR, CORE_ID Stall until CSDONE signal received *On large core:* Enqueue in CSRB Wait until HEAD ENTRY in CSRB Acquire lock at LOCK_ADDR SP <- STACK_PTR PC <- TARGET_PC	*On large core:* Release lock at LOCK_ADDR Send CSDONE to REQ_CORE *On small core:* Retire CSCALL

Figure 5. Format and operation semantics of new ACS instructions

3.2 ISA Support

ACS requires two new instructions: *CSCALL* and *CSRET*. CSCALL is similar to a traditional CALL instruction, except it is used to execute critical section code on a remote, large processor. When a small core executes a CSCALL instruction, it sends a request for the execution of critical section to P0 and waits until it receives a response. CSRET is similar to a traditional RET instruction, except that it is used to return from a critical section executed on a remote processor. When P0 executes CSRET, it sends a CSDONE signal to the small core so that it can resume execution. Figure 5 shows the semantics of CSCALL and CSRET. CSCALL takes two arguments: LOCK_ADDR and TARGET_PC. LOCK_ADDR is the memory address of the lock protecting the critical section and TARGET_PC is the address of the first instruction in the critical section. CSRET takes one argument, LOCK_ADDR corresponding to the CSCALL.

3.3 Compiler/Library Support

The CSCALL and CSRET instructions encapsulate a critical section. CSCALL is inserted before the "lock acquire" and CSRET is inserted after the "lock release." The compiler/library inserts these instructions automatically without requiring any modification to the source code. The compiler must also remove any register dependencies between the code inside and outside the critical section. This avoids transferring register values from the small core to the large core and vice versa before and after the execution of the critical section. To do so, the compiler performs *function outlining* [52] for every critical section by encapsulating it in a separate function and ensuring that all input and output parameters are communicated via the stack. Several OpenMP compilers already do function outlining for critical sections [28, 37, 9]. Therefore, compiler modifications are mainly limited to the insertion of CSCALL and CSRET instructions. Figure 6 shows the code of a critical section executed on the baseline (a) and the modified code executed on ACS (b).

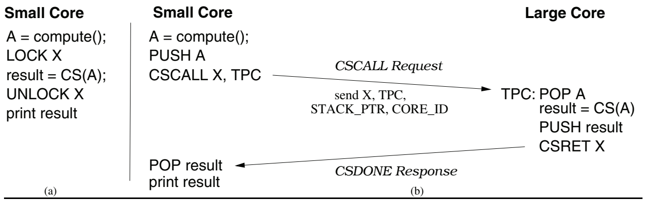


Figure 6. Source code and its execution: (a) baseline (b) ACS

3.4 Hardware Support

3.4.1 Modifications to the small cores

When a CSCALL is executed, the small core sends a CSCALL request along with the stack pointer (STACK_PTR) and its core ID (CORE_ID) to the large core and stalls, waiting for the CSDONE response. The CSCALL instruction is retired when a CSDONE response is received. Such support for executing certain instructions remotely already exists in current architectures: for example, all 8 cores in Sun Niagara-1 [22] execute floating point (FP) operations on a common remote FP unit.

3.4.2 Critical Section Request Buffer

The Critical Section Request Buffer (CSRB), located at the large core, buffers the pending CSCALL requests sent by the small cores. Figure 7 shows the structure of the CSRB. Each entry in the CSRB contains a valid bit, the ID of the requesting core (REQ_CORE), the parameters of the CSCALL instruction, LOCK_ADDR and TARGET_PC, and the stack pointer (STACK_PTR) of the requesting core. When the large core is idle, the CSRB supplies the oldest CSCALL request in the buffer to the core. The large core notifies the CSRB when it completes the critical section. At this point, the CSRB dequeues the corresponding entry and sends a CSDONE signal to the requesting core. The number of entries in the CSRB is equal to the maximum possible number of concurrent CSCALL instructions. Because each small core can execute at most one CSCALL instruction at any time, the number of entries required is equal to the number of small cores in the system (Note that the large core does not send CSCALL requests to itself). For a system with 12 small

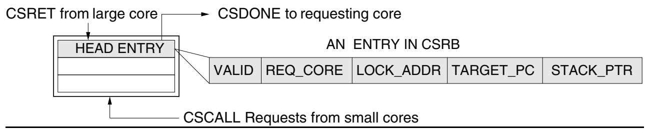


Figure 7. Critical Section Request Buffer (CSRB)

cores, the CSRB has 12 entries, each of which is 25 bytes[3] each. Thus, the storage overhead of the CSRB is 300 bytes.

3.4.3 Modifications to the large core

When the large core receives an entry from the CSRB, it loads its stack pointer register with STACK_PTR and acquires the lock corresponding to LOCK_ADDR (as specified by program code). It then redirects the program counter to TARGET_PC and starts executing the critical section. When the core retires the CSRET instruction, it releases the lock corresponding to LOCK_ADDR and removes the HEAD ENTRY from the CSRB. Thus, ACS executes a critical section similar to a conventional processor by acquiring the lock, executing the instructions, and releasing the lock. However, it does so at a higher performance because of the aggressive configuration of the large core.

3.4.4 Interconnect Extensions

ACS introduces two new transactions on the on-chip interconnect: CSCALL and CSDONE. The interconnect transfers the CSCALL request (along with its arguments) from the smaller core to the CSRB and the CSDONE signal from the CSRB to the smaller core. Similar transactions already exist in the on-chip interconnects of current processors. For example, Sun Niagara-1 [22] uses such transactions to interface cores with the shared floating point unit.

3.5 Operating System Support

ACS requires modest support from the operating system (OS). When executing on an ACS architecture, the OS allocates the large core to a single application and does not schedule any threads onto it. Additionally, the OS sets the control registers of the large core to the same values as the small cores executing the application. As a result, the program context (e.g. processor status registers, and TLB entries) of the application remains the same in all cores, including the large core. Note that ACS does not require any special modifications because such support already exists in current CMPs to execute parallel applications [20].

Handling Multiple Parallel Applications: When multiple parallel applications are executing concurrently, ACS can be used if the CMP provides multiple high-performance contexts of execution (multiple large cores or simultaneous multithreading (SMT) [48] on the large core). Alternatively, the OS can time-share the large core between multiple applications taking performance and fairness into account. ACS can be enabled only for the application that is allocated the large core and disabled for the others. This paper introduces the concept and implementation of ACS; resource allocation policies are part of our future work.

3.6 Reducing False Serialization in ACS

Critical sections that are protected by different locks can be executed concurrently in a conventional CMP. However, in ACS, their execution gets serialized because they are all executed sequentially on the single large core. This "false serialization" reduces concurrency and degrades performance.

[3] Each CSRB entry has one valid bit, 4-bit REQ_CORE, 8 bytes each for LOCK_ADDR, TARGET_PC, and STACK_PTR.

We reduce false serialization using two techniques. First, we make the large core capable of executing multiple critical sections concurrently,[4] using simultaneous multithreading (SMT) [48]. Each SMT context can execute CSRB entries with different LOCK_ADDR. Second, to reduce false serialization in workloads where a large number of critical sections execute concurrently, we propose *Selective Acceleration of Critical Sections (SEL)*. The key idea of SEL is to estimate the occurrence of false serialization and adaptively decide whether or not to execute a critical section on the large core. If SEL estimates false serialization to be high, the critical section is executed locally on the small core, which reduces contention on the large core.

Implementing SEL requires two modifications: 1) a bit vector at each small core that contains the ACS_DISABLE bits and 2) logic to estimate false serialization. The ACS_DISABLE bit vector contains one bit per critical section and is indexed using the LOCK_ADDR. When the smaller core encounters a CSCALL, it first checks the corresponding ACS_DISABLE bit. If the bit is 0 (i.e., false serialization is low), a CSCALL request is sent to the large core. Otherwise, the CSCALL and the critical section is executed locally.

False serialization is estimated at the large core by augmenting the CSRB with a table of saturating counters, which track the false serialization incurred by each critical section. We quantify false serialization by counting the number of critical sections present in the CSRB for which the LOCK_ADDR is different from the LOCK_ADDR of the incoming request. If this count is greater than 1 (i.e. if there are at least two independent critical sections in the CSRB), the estimation logic adds the count to the saturating counter corresponding to the LOCK_ADDR of the incoming request. If the count is 1 (i.e. if there is exactly one critical section in the CSRB), the corresponding saturating counter is decremented. If the counter reaches its maximum value, the ACS_DISABLE bit corresponding to that lock is set by sending a message to all small cores. Since ACS is disabled infrequently, the overhead of this communication is negligible. To adapt to phase changes, we reset the ACS_DISABLE bits for all locks and halve the value of the saturating counters periodically (every 10 million cycles). We reduce the hardware overhead of SEL by hashing lock address into a small number of sets. Our implementation of SEL hashes lock addresses into 16 sets and uses 6-bit counters. The total storage overhead of SEL is 36 bytes: 16 counters of 6-bits each and 16 ACS_DISABLE bits for each of the 12 small cores.

3.7 Handling Nested Critical Sections

A nested critical section is embedded within another critical section. Such critical sections can cause deadlocks in ACS with SEL.[5] To avoid deadlocks without extra hardware complexity, our design does not convert nested critical sections to CSCALLs. Using simple control-flow analysis, the compiler identifies the critical sections that can possibly become nested at run-time. Such critical sections are not converted to CSCALLs.

[4] Another possible solution to reduce false serialization is to add additional large cores and distribute the critical sections across these cores. Further investigation of this solution is an interesting research direction, but is beyond the scope of this paper.

[5] For example, consider three nested critical sections: the outermost (*O*), inner (*N*), and the innermost (*I*). ACS is disabled for *N* and enabled for *O* and *I*. The large core is executing *O* and another small core is executing executing *N* locally (because ACS was disabled). The large core encounters *N*, and waits for the small core to finish *N*. Meanwhile, the small core encounters *I*, sends a CSCALL request to the large core, and waits for the large core to finish *I*. Therefore, deadlock ensues.

3.8 Handling Interrupts and Exceptions

ACS supports precise interrupts and exceptions. If an interrupt or exception happens outside a critical section, ACS handles it similarly to the baseline. If an interrupt or exception occurs on the large core while it is executing the critical section, the large core disables ACS for all critical sections, pushes the CSRB on the stack, and handles the interrupt or exception. If the interrupt is received by the small core while it is waiting for a CSDONE signal, it delays servicing the interrupt until the CSDONE signal is received. Otherwise, the small core may miss the CSDONE signal as it is handling the interrupt, leading to a deadlock.

Because ACS executes critical sections on a separate core, temporary register values outside the critical section are not visible inside the critical section and vice versa. This is not a concern in normal program execution because the compiler removes any register dependencies between the critical section and the code outside it. If visibility to temporary register values outside the critical section is required inside the critical section, e.g. for debugging purposes, the compiler can ensure the transfer of all register values from the small core to the large core by inserting additional stack operations in the debug version of the code.

3.9 Accommodating Multiple Large Cores

We have described ACS for an ACMP that contains only one large core. ACS can also leverage multiple large cores in two ways: 1) to execute different critical sections from the same multi-threaded application, thereby reducing "false serialization," 2) to execute critical sections from different applications, thereby increasing system throughput. Evaluation of ACS using multiple large cores is out of the scope of this paper.

4. Performance Trade-offs in ACS

There are three key performance trade-offs in ACS that determine overall system performance:

1. Faster critical sections vs. fewer threads: ACS executes selected critical sections on a large core, the area dedicated to which could otherwise be used for executing additional threads. ACS could improve performance if the performance gained by accelerating critical sections (and serial program portions) outweighs the loss of throughput due to the unavailability of additional threads. ACS's performance improvement becomes more likely when the number of cores on the chip increases because of two reasons. First, the marginal loss in parallel throughput due to the large core becomes relatively small (for example, if the large core replaces four small cores, then it eliminates 50% of the smaller cores in a 8-core system but only 12.5% of cores in a 32-core system). Second, more cores allow concurrent execution of more threads, which increases contention by increasing the probability of each thread waiting to enter the critical section [36]. When contention is high, faster execution of a critical section reduces not only critical section execution time but also the contending threads' waiting time.

2. CSCALL/CSDONE signals vs. lock acquire/release: To execute a critical section, ACS requires the communication of CSCALL and CSDONE transactions between a small core and a large core. This communication over the on-chip interconnect is an overhead of ACS, which the conventional lock acquire/release operations do not incur. On the other hand, a lock acquire operation often incurs cache misses [33] because the lock needs to be transferred from one cache to another. Each cache-to-cache transfer requires two transactions on the on-chip interconnect: a request for the cache line and the response, which has similar latency to the CSCALL and CSDONE transactions. ACS can reduce such cache-to-cache transfers by keeping the lock at the large core, which can compensate for the overhead of CSCALL and CSDONE. ACS actually has an advantage in that the latency of CSCALL and CSDONE can be overlapped with the execution of another instance of the same critical section. On the other hand, in conventional locking, a lock can only be acquired after the critical section has been completed, which *always* adds a delay before critical section execution. Therefore, the overhead of CSCALL/CSDONE is likely not as high as the overhead of lock acquire/release.

3. Cache misses due to private data vs. cache misses due to shared data: In ACS, private data that is referenced in the critical section needs to be transferred from the cache of the small core to the cache of the large core. Conventional locking does not incur this cache-to-cache transfer overhead because critical sections are executed at the local core and private data is often present in the local cache. On the other hand, conventional systems incur overheads in transferring shared data: in such systems, shared data "ping-pongs" between caches as different threads execute the critical section and reference the shared data. ACS eliminates the transfers of shared data by keeping it at the large core,[6] which can offset the misses it causes to transfer private data into the large core. In fact, ACS can decrease cache misses if the critical section accesses more shared data than private data. Note that ACS can improve performance even if there are equal or more accesses to private data than shared data because the large core can still 1) improve performance of other instructions and 2) hide the latency of some cache misses using latency tolerance techniques like out-of-order execution.

In summary, ACS can improve overall performance if its performance benefits (faster critical section execution, improved lock locality, and improved shared data locality) outweigh its overheads (reduced parallel throughput, CSCALL and CSDONE overhead, and reduced private data locality). Next, we will evaluate the performance of ACS on a variety of CMP configurations.

5. Experimental Methodology

Table 1 shows the configuration of the simulated CMPs, using our in-house cycle-accurate x86 simulator. The large core occupies the same area as four smaller cores: the smaller cores are modeled after the Intel Pentium processor [19], which requires 3.3 million transistors, and the large core is modeled after the Intel Pentium-M core, which requires 14 million transistors [12]. We evaluate three different CMP architectures: a symmetric CMP (SCMP) consisting of all small cores; an asymmetric CMP (ACMP) with one large core with 2-way SMT and remaining small cores; and an ACMP augmented with support for the ACS mechanism (ACS). Unless specified otherwise, all comparisons are done at equal area budget. We specify the area budget in terms of the number of small cores. Unless otherwise stated, the number of threads for each application is set equal to the number of threads that minimizes the execution time for the particular configuration; e.g. if the best performance of an application is obtained on an 8-core SCMP when it runs with 3 threads, then we report the performance with 3 threads. In both ACMP and SCMP, conventional lock acquire/release operations are implemented using the Monitor/Mwait instructions, part of the SSE3 extensions to the x86 ISA [17]. In ACS, lock acquire/release instructions are replaced with CSCALL/CSRET instructions.

[6] By keeping shared data in the large core's cache, ACS reduces the cache space available to shared data compared to conventional locking (where shared data can reside in any on-chip cache). This can increase cache misses. However, we find that such cache misses are rare and do not degrade performance because the private cache of the large core is large enough.

Small core	2-wide In-order, 2GHz, 5-stage. L1: 32KB write-through. L2: 256KB write-back, 8-way, 6-cycle access
Large core	4-wide Out-of-order, 2GHz, 2-way SMT, 128-entry ROB, 12-stage, L1: 32KB write-through. L2: 1-MB write-back, 16-way, 8-cycle
Interconnect	64-bit wide bi-directional ring, all queuing delays modeled, ring hop latency of 2 cycles (latency between one cache to the next)
Coherence	MESI, On-chip distributed directory similar to SGI Origin [26], cache-to-cache transfers. # of banks = # of cores, 8K entries/bank
L3 Cache	8MB, shared, write-back, 20-cycle, 16-way
Memory	32 banks, bank conflicts and queuing delays modeled. Row buffer hit: 25ns, Row buffer miss: 50ns, Row buffer conflict: 75ns
Memory bus	4:1 cpu/bus ratio, 64-bit wide, split-transaction, pipelined bus, 40-cycle latency
Area-equivalent CMPs. Area = N small cores. N varies from 1 to 32	
SCMP	N small cores, One small core runs serial part, all N cores run parallel part, conventional locking (Max. concurrent threads = N)
ACMP	1 large core and N-4 small cores; large core runs serial part, 2-way SMT on large core and small cores run parallel part, conventional locking (Maximum number of concurrent threads = N-2)
ACS	1 large core and N-4 small cores; (N-4)-entry CSRB on the large core, large core runs the serial part, small cores run the parallel part, 2-way SMT on large core runs critical sections using ACS (Max. concurrent threads = N-4)

Table 1. Configuration of the simulated machines

5.1 Workloads

Our main evaluation focuses on 12 critical-section-intensive workloads shown in Table 2. We define a workload to be critical-section-intensive if at least 1% of the instructions in the parallel portion are executed within critical sections. We divide these workloads into two categories: workloads with coarse-grained locking and workloads with fine-grained locking. We classify a workload as using coarse-grained locking if it has at most 10 critical sections. Based on this classification, 7 out of 12 workloads use coarse-grain locking and the remaining 5 use fine-grain locking. All workloads were simulated to completion. A description of the benchmarks whose source code is not publicly available is provided in [42].

Locks	Workload	Description	Source	Input set
Coarse	ep	Random number generator	[7]	262144 nums.
	is	Integer sort	[7]	n = 64K
	pagemine	Data mining kernel	[31]	10Kpages
	puzzle	15-Puzzle game	[50]	3x3
	qsort	Quicksort	[11]	20K elem.
	sqlite	sqlite3 [3] database engine	[4]	OLTP-simple
	tsp	Traveling salesman prob.	[23]	11 cities
Fine	iplookup	IP packet routing	[49]	2.5K queries
	oltp-1	MySQL server [1]	[4]	OLTP-simple
	oltp-2	MySQL server [1]	[4]	OLTP-complex
	specjbb	JAVA business benchmark	[40]	5 seconds
	webcache	Cooperative web cache	[45]	100K queries

Table 2. Simulated workloads

6. Evaluation

We make three comparisons between ACMP, SCMP, and ACS. First, we compare their performance on systems where the number of threads is set equal to the optimal number of threads for each application under a given area constraint. Second, we compare their performance assuming the number of threads is set equal to the number of cores in the system, a common practice employed in many existing systems.

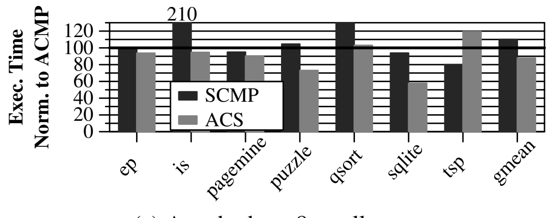
Third, we analyze the impact of ACS on application scalability i.e., the number of threads over which performance does not increase.

6.1 Performance with the Optimal Number of Threads

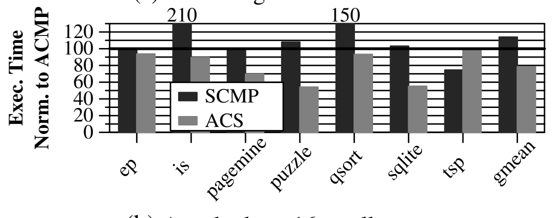
Systems sometimes use profile or run-time information to choose the number of threads that minimizes execution time [43]. We first analyze ACS with respect to ACMP and SCMP when the optimal number of threads are used for each application on each CMP configuration.[7] We found that doing so provides the best baseline performance for ACMP and SCMP, and a performance comparison results in the lowest performance improvement of ACS. Hence, this performance comparison penalizes ACS (as our evaluations in Section 6.2 with the same number of threads as the number of thread contexts will show). We show this performance comparison separately on workloads with coarse-grained locks and those with fine-grained locks.

6.1.1 Workloads with Coarse-Grained Locks

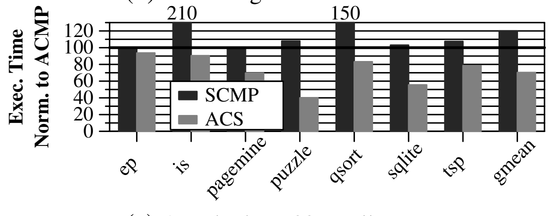
Figure 8 shows the execution time of each application on SCMP and ACS normalized to ACMP for three different area budgets: 8, 16, and 32. Recall that when area budget is equal to N, SCMP, ACMP, and ACS can execute up to N, N-2, and N-4 parallel threads respectively. In the ensuing discussion, we ask the reader to refer to Table 3, which shows the characteristics of critical sections in each application, to provide insight into the performance results.



(a) Area budget=8 small cores



(b) Area budget=16 small cores



(c) Area budget=32 small cores

Figure 8. Execution time of workloads with coarse-grained locking on ACS and SCMP normalized to ACMP

[7] We determine the optimal number of threads for an application by simulating all possible number of threads and using the one that minimizes execution time. The interested reader can obtain the optimal number of threads for each benchmark and each configuration by examining the data in Figure 10. Due to space constraints, we do not explicitly quote these thread counts.

Systems area-equivalent to 8 small cores: When area budget equals 8, ACMP significantly outperforms SCMP for workloads with high percentage of instructions in the serial part (85% in `is` and 29% in `qsort` as Table 3 shows). In `puzzle`, even though the serial part is small, ACMP improves performance because it improves cache locality of shared data by executing two of the six threads on the large core, thereby reducing cache-to-cache transfers of shared data. SCMP outperforms ACMP for `sqlite` and `tsp` because these applications spend a very small fraction of their instructions in the serial part and sacrificing two threads for improved serial performance is not a good trade-off. Since ACS devotes the two SMT contexts on the large core to accelerate critical sections, it can execute only four parallel threads (compared to 6 threads of ACMP and 8 threads of SCMP). Despite this disadvantage, ACS reduces the average execution time by 22% compared to SCMP and by 11% compared to ACMP. ACS improves performance of five out of seven workloads compared to ACMP. These five workloads have two common characteristics: 1) they have high contention for the critical sections, 2) they access more shared data than private data in critical sections. Due to these characteristics, ACS reduces the serialization caused by critical sections and improves locality of shared data.

Why does ACS reduce performance in `qsort` and `tsp`? The critical section in `qsort` protects a stack that contains indices of the array to be sorted. The insert operation pushes two indices (private data) onto the stack by changing the stack pointer (shared data). Since indices are larger than the stack pointer, there are more accesses to private data than shared data. Furthermore, contention for critical sections is low. Therefore, `qsort` can take advantage of additional threads in its parallel portion and trading off several threads for faster execution of critical sections lowers performance. The dominant critical section in `tsp` protects a FIFO queue where an insert operation reads the node to be inserted (private data) and adds it to the queue by changing only the head pointer (shared data). Since private data is larger than shared data, ACS reduces cache locality. In addition, contention is low and the workload can effectively use additional threads.

Systems area-equivalent to 16 and 32 small cores: Recall that as the area budget increases, the overhead of ACS decreases. This is due to two reasons. First, the parallel throughput reduction caused by devoting a large core to execute critical sections becomes smaller, as explained in Section 4. Second, more threads increases contention for critical sections because it increases the probability that each thread is waiting to enter the critical section. When the area budget is 16, ACS improves performance by 32% compared to SCMP and by 22% compared to ACMP. When the area budget is 32, ACS improves performance by 42% compared to SCMP and by 31% compared to ACMP. In fact, the two benchmarks (`qsort` and `tsp`) that lose performance with ACS when the area budget is 8 experience significant performance gains with ACS over both ACMP and SCMP for an area budget of 32. For example, ACS with an area budget of 32 provides 17% and 22% performance improvement for `qsort` and `tsp` respectively over an equal-area ACMP. With an area budget of at least 16, ACS improves the performance of *all* applications with coarse-grained locks. We conclude that ACS is an effective approach for workloads with coarse-grained locking even at small area budgets. However, ACS becomes even more attractive as the area budget in terms of number of cores increases.

Workload	% of instr. in Serial Part	% of parallel instr. in critical sections	# of disjoint critical sections	What is Protected by CS?	Avg. instr. in critical section	Shared/Private (at 4 threads)	Contention 4	8	16	32
ep	13.3	14.6	3	reduction into global data	620618.1	**1.0**	1.4	1.8	4.0	8.2
is	84.6	8.3	1	buffer of keys to sort	9975.0	**1.1**	2.3	4.3	8.1	16.4
pagemine	0.4	5.7	1	global histogram	531.0	**1.7**	2.3	4.3	8.2	15.9
puzzle	2.4	69.2	2	work-heap, memoization table	926.9	**1.1**	2.2	4.3	8.3	16.1
qsort	28.5	16.0	1	global work stack	127.3	0.7	1.1	3.0	9.6	25.6
sqlite	0.2	17.0	5	database tables	933.1	**2.4**	1.4	2.2	3.7	6.4
tsp	0.9	4.3	2	termination cond., solution	29.5	0.4	1.2	1.6	2.0	3.6
iplookup	0.1	8.0	# of threads	routing tables	683.1	0.6	1.2	1.3	1.5	1.9
oltp-1	2.3	13.3	20	meta data, tables	277.6	0.8	1.2	1.2	1.5	2.2
oltp-2	1.1	12.1	29	meta data, tables	309.6	0.9	1.1	1.2	1.4	1.6
specjbb	1.2	0.3	39	counters, warehouse data	1002.8	0.5	1.0	1.0	1.0	1.2
webcache	3.5	94.7	33	replacement policy	2257.0	**1.1**	1.1	1.1	1.1	1.4

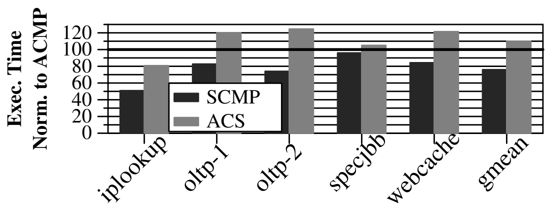
Table 3. Characteristics of Critical Sections. Shared/Private is the ratio of *shared* data (number of cache lines that are transferred from caches of other cores) to *private* data (number of cache lines that hit in the private cache) accessed inside a critical section. Contention is the average number of threads waiting for critical sections when the workload is executed with 4, 8, 16, and 32 threads on the SCMP.

6.1.2 Workloads with Fine-Grained Locks

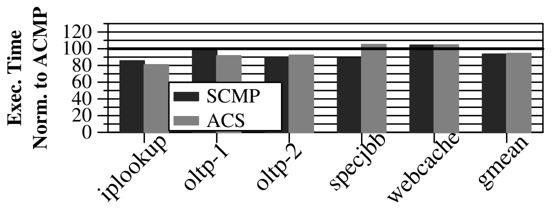
Figure 9 shows the execution time of workloads with fine-grained locking for three different area budgets: 8, 16, and 32. Compared to coarse-grained locking, fine-grained locking reduces contention for critical sections and hence the serialization caused by them. As a result, critical section contention is negligible at low thread counts, and the workloads can take significant advantage of additional threads executed in the parallel section. When the area budget is 8, SCMP provides the highest performance (as shown in Figure 9(a)) for all workloads because it can execute the most number of threads in parallel. Since critical section contention is very low, ACS essentially wastes half of the area budget by dedicating it to a large core because it is unable to use the large core efficiently. Therefore, ACS increases execution time compared to ACMP for all workloads except `iplookup`. In `iplookup`, ACS reduces execution time by 20% compared to ACMP but increases it by 37% compared to SCMP. The critical sections in `iplookup` access more private data than shared data, which reduces the benefit of ACS. Hence, the faster critical section execution benefit of ACS is able to overcome the loss of 2 threads (ACMP) but is unable to provide enough improvement to overcome the loss of 4 threads (SCMP).

As the area budget increases, ACS starts providing performance improvement over SCMP and ACMP because the loss of parallel throughput due to the large core reduces. With an area budget of 16, ACS performs similarly to SCMP (within 2%) and outperforms ACMP (by 6%) on average. With an area budget of 32, ACS's performance improvement is the highest: 17% over SCMP and 13% over ACMP; in fact, ACS outperforms both SCMP and ACMP on all workloads. Hence, we conclude that ACS provides the best performance compared to the alternative chip organizations, even for critical-section-intensive workloads that use fine-grained locking.

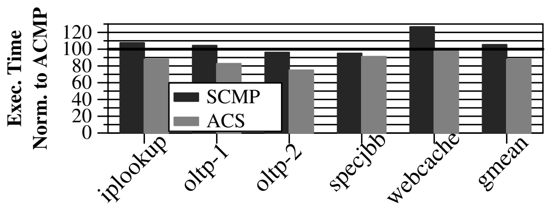
Depending on the scalability of the workload and the amount of contention for critical sections, the area budget required for ACS to provide performance improvement is different. Table 4 shows the area budget required for ACS to outperform an equivalent-area ACMP and SCMP. In general, the area budget ACS requires to outperform SCMP is higher than the area budget it requires to outperform ACMP. However, `webcache` and `qsort` have a high percentage of serial instructions; therefore ACMP becomes significantly more effective than SCMP for large area budgets. For all workloads with fine-grained locking, the area budget ACS requires to outperform an area-equivalent SCMP or ACMP is less than or equal to 24 small cores. Since chips with 8 and 16 small cores are already in the market [22], and chips with 32 small cores are being built [47, 38], we believe ACS can be a feasible and effective option to improve the performance of workloads that use fine-grained locking in near-future multi-core processors.



(a) Area budget=8 small cores



(b) Area budget=16 small cores



(c) Area budget=32 small cores

Figure 9. Execution time of workloads with fine-grained locking on ACS and SCMP normalized to ACMP

	ep	is	pagemine	puzzle	qsort	sqlite	tsp	iplookup	oltp-1	oltp-2	specjbb	webcache
ACMP	6	6	6	4	12	6	10	6	14	10	18	24
SCMP	6	4	6	4	8	6	18	14	14	16	18	14

Table 4. Area budget (in terms of small cores) required for ACS to outperform an equivalent-area ACMP and SCMP

Summary: Based on the observations and analyses we made above for workloads with coarse-grained and fine-grained locks, we conclude that ACS provides significantly higher performance than both SCMP and ACMP for both types of workloads, except for workloads with fine-grained locks when the area budget is low. ACS's performance benefit increases as the area budget increases. In future systems with a large number of cores, ACS is likely to provide the best system organization among the three choices we examined. For example, with an area budget of 32 small cores, ACS outperforms SCMP by 34% and ACMP by 23% averaged across all workloads, including both fine-grained and coarse-grained locks.

6.2 Performance with Number of Threads Set Equal to the Number of Available Thread Contexts

In the previous section, we used the optimal number of threads for each application-configuration pair. When an estimate of the optimal number of threads is not available, many current systems use as many threads as there are available thread contexts [18, 32]. We now evaluate ACS assuming the number of threads is set equal to the number of available contexts. Figure 10 shows the speedup curves of ACMP, SCMP, and ACS over one small core as the area budget is varied from 1 to 32. The curves for ACS and ACMP start at 4 because they require at least one large core which is area-equivalent to 4 small cores.

Number of threads	No. of max. thread contexts			Optimal		
Area Budget	8	16	32	8	16	32
SCMP	0.93	1.04	1.18	0.94	1.05	1.15
ACS	0.97	0.77	0.64	0.96	0.83	0.77

Table 5. Average execution time normalized to area-equivalent ACMP

Table 5 summarizes the data in Figure 10 by showing the average execution time of ACS and SCMP normalized to ACMP for area budgets of 8, 16, and 32. For comparison, we also show the data with optimal number of threads. With an area budget of 8, ACS outperforms both SCMP and ACMP on 5 out of 12 benchmarks. ACS degrades average execution time compared to SCMP by 3% and outperforms ACMP by 3%. When the area budget is doubled to 16, ACS outperforms both SCMP and ACMP on 7 out of 12 benchmarks, reducing average execution time by 26% and 23%, respectively. With an area budget of 32, ACS outperforms both SCMP and ACMP on all benchmarks, reducing average execution time by 46% and 36%, respectively. Note that this performance improvement is significantly higher than the performance improvement ACS provides when the optimal number of threads is chosen for each configuration (34% over SCMP and 23% over ACMP). Also note that when the area budget increases, ACS starts to consistently outperform both SCMP and ACMP. This is because ACS tolerates contention among threads better than SCMP and ACMP. Table 6 compares the contention of SCMP, ACMP, and ACS at an area budget of 32. For ep, on average more than 8 threads wait for each critical section in both SCMP and ACMP. ACS reduces the waiting threads to less than 2, which improves performance by 44% (at an area budget of 32).

We conclude that, even if a developer is unable to determine the optimal number of threads for a given application-configuration pair and chooses to set the number of threads at a point beyond the saturation point, ACS provides significantly higher performance than both ACMP and SCMP. In fact, ACS's performance benefit is even higher in systems where the number of threads is set equal to number of thread contexts because ACS is able to tolerate contention for critical sections significantly better than ACMP or SCMP.

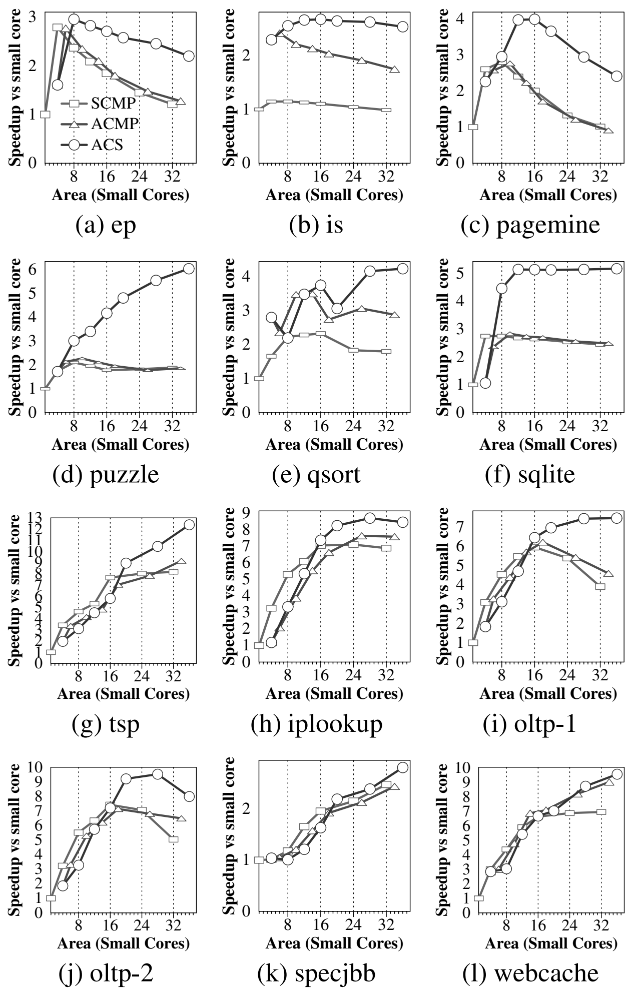


Figure 10. Speedup over a single small core

Workload	ep	is	pagemine	puzzle	qsort	sqlite	tsp	iplookup	oltp-1	oltp-2	specjbb	webcache
SCMP	8.2	16.4	15.9	16.1	25.6	6.4	3.6	1.9	2.2	1.6	1.2	1.4
ACMP	8.1	14.9	15.5	16.1	24.0	6.2	3.7	1.9	1.9	1.5	1.2	1.4
ACS	1.5	2.0	2.0	2.5	1.9	1.4	3.5	1.8	1.4	1.3	1.0	1.2

Table 6. Contention (see Table 3 for definition) at an area budget of 32 (Number of threads set equal to the number of thread contexts)

6.3 Application Scalability

We examine the effect of ACS on the number of threads required to minimize the execution time. Table 7 shows number of threads that provides the best performance for each application using ACMP, SCMP, and ACS. The best number of threads were chosen by executing each application with all possible threads from 1 to 32. For 7 of the 12 applications (`is`, `pagemine`, `puzzle`, `qsort`, `sqlite`, `oltp-1`, and `oltp-2`) ACS improves scalability: it increases the number of threads at which the execution time of the application is minimized. This is because ACS reduces contention due to critical sections as explained in Section 6.2 and Table 6. For the remaining applications, ACS does not change scalability.[8] We conclude that if thread contexts are available on the chip, ACS uses them more effectively compared to ACMP and SCMP.

Workload	ep	is	pagemine	puzzle	qsort	sqlite	tsp	iplookup	oltp-1	oltp-2	specjbb	webcache
SCMP	4	8	8	8	16	8	32	24	16	16	32	32
ACMP	4	8	8	8	16	8	32	24	16	16	32	32
ACS	4	12	12	32	32	32	32	24	32	24	32	32

Table 7. Best number of threads for each configuration

6.4 Performance of ACS on Critical Section Non-Intensive Benchmarks

We also evaluated all 16 benchmarks from the NAS [7] and SPLASH [51] suites that are not critical-section-intensive. These benchmarks contain regular data-parallel loops and execute critical sections infrequently (less than 1% of the executed instructions). Detailed results of this analysis are presented in [42]. We find that ACS does not significantly improve or degrade the performance of these applications. When area budget is 32, ACS provides a modest 1% performance improvement over ACMP and 2% performance reduction compared to SCMP. As area budget increases, ACS performs similar to (within 1% of) SCMP. We conclude that ACS will not significantly affect the performance of critical section non-intensive workloads in future systems with large number of cores.

7. Sensitivity of ACS to System Configuration

7.1 Effect of SEL

ACS uses the SEL mechanism (Section 3.6) to selectively accelerate critical sections to reduce false serialization of critical sections. We evaluate the performance impact of SEL. Since SEL does not affect the performance of workloads that have negligible false serialization, we focus our evaluation on the three workloads that experience false serialization: `puzzle`, `iplookup`, and `webcache`. Figure 11 shows the normalized execution time of ACS with and without SEL for the three workloads when the area budget is 32. For `iplookup` and `webcache`, which has the highest amount of false serialization, using SEL improves performance by 11% and 5% respectively over the baseline. The performance improvement is due to acceleration of *some* critical sections which SEL allows to be sent to the large core because they do not experience false serialization. In `webcache`, multiple threads access pages of different files stored in a shared cache. Pages from each file are protected by a different lock. In a conventional system, these critical sections can execute in parallel, but ACS without SEL serializes the execution of these critical sections by forcing them to execute on a single large core. SEL disables the acceleration of 17 out of the 33 locks, which eliminates false serialization and reduces pressure on the large core. In `iplookup`, multiple copies of the routing table (one for each thread) are protected by disjoint critical sections that get serialized without SEL. `puzzle` contains two critical sections protecting a heap object (PQ in Figure 1) and a memoization table. Accesses to PQ are more frequent than to the memoization table, which results in false serialization for the memoization table. SEL detects this serialization and disables the acceleration of the critical section for the memoization table. On average, across all 12 workloads, ACS with SEL outperforms ACS without SEL by 15%. We conclude that SEL can successfully improve the performance benefit of ACS by eliminating false serialization without affecting the performance of workloads that do not experience false serialization.

[8] Note that Figure 10 provides more detailed information on ACS's effect on the scalability of each application. However, unlike Table 7, the data shown on the x-axis is area budget and not number of threads.

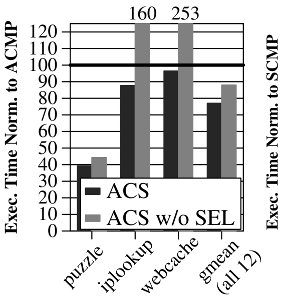


Figure 11. Impact of SEL.

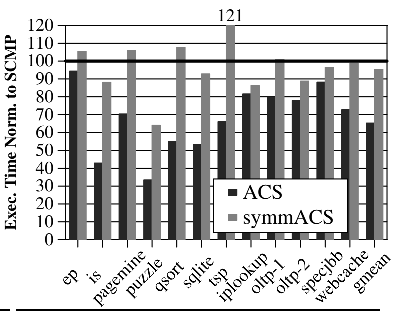


Figure 12. ACS on symmetric CMP.

7.2 ACS on Symmetric CMPs: Effect of Only Data Locality

Part of the performance benefit of ACS is due to improved locality of shared data and locks. This benefit can be realized even in the absence of a large core. A variant of ACS can be implemented on a symmetric CMP, which we call *symmACS*. In symmACS, one of the small cores is dedicated to executing critical sections. This core is augmented with a CSRB and executes the CSCALL requests and CSRET instructions. Figure 12 shows the execution time of symmACS and ACS normalized to SCMP when area budget is 32. SymmACS reduces execution time by more than 5% compared to SCMP in `is`, `puzzle`, `sqlite`, and `iplookup` because more shared data is accessed than private data in critical sections.[9] In `ep`, `pagemine`, `qsort`, and `tsp`, the overhead of CSCALL/CSRET messages and transferring private data offsets the shared data/lock locality advantage of ACS. Thus, overall execution time increases. On average, symmACS reduces execution time by only 4% which is much lower than the 34% performance benefit of ACS. Since the performance gain due to improved locality alone is relatively small, we conclude that most of the performance improvement of ACS comes from accelerating critical sections using the large core.

8. Related Work

The major contribution of our paper is a comprehensive mechanism to accelerate critical sections using a large core. The most closely related work is the numerous proposals to optimize the implementation of lock acquire/release operations and the locality of shared data in critical section using OS and compiler techniques. We are not aware of any work that speeds up the execution of critical sections using more aggressive execution engines. To our knowledge, this is the first paper that comprehensively accelerates critical sections by improving both the execution speed of critical sections and locality of shared data/locks.

[9] Note that these numbers do not correspond to those shown in Table 3. The Shared/Private ratio reported in Table 3 is collected by executing the workloads with 4 threads. On the other hand, in this experiment, the workloads were run with the optimal number of threads for each configuration.

8.1 Improving Locality of Shared Data and Locks

Sridharan et al. [39] propose a thread scheduling algorithm for SMP machines to increase shared data locality in critical sections. When a thread encounters a critical section, the operating system migrates the thread to the processor that has the shared data. This scheme increases cache locality of shared data but incurs the substantial overhead of migrating complete thread state on every critical section. ACS does not migrate thread contexts and therefore does not need OS intervention. Instead, it sends a CSCALL request with minimal data to the core executing the critical sections. Moreover, ACS accelerates critical section execution, a benefit unavailable in [39]. Trancoso and Torrellas [46] and Ranganathan et al. [35] improve locality in critical sections using software prefetching. These techniques can be combined with ACS for improved performance.

Several primitives (e.g., Test&Test&Set, Compare&Swap) were proposed to efficiently implement lock acquire and release operations [10]. Recent research has also studied hardware and software techniques to reduce the overhead of lock operations [16, 5]. The Niagara-2 processor improves cache locality of locks by executing the "lock acquire" instructions [13] remotely at the cache bank where the lock is resident. However, none of these techniques increase the speed of critical section processing or the locality of shared data.

8.2 Hiding the Latency of Critical Sections

Several proposals try to hide the latency of a critical section by executing it speculatively with other instances of the same critical section *as long as they do not have data conflicts with each other*. Examples include transactional memory (TM) [14], speculative lock elision (SLE) [33], transactional lock removal (TLR) [34], and speculative synchronization (SS) [29]. SLE is a hardware technique that allows multiple threads to execute the critical sections speculatively without acquiring the lock. If a data conflict is detected, only one thread is allowed to complete the critical section while the remaining threads roll back to the beginning of the critical section and try again. TLR improves upon SLE by providing a timestamp-based conflict resolution scheme that enables lock-free execution. ACS is partly orthogonal to these approaches due to three major reasons:

1. TLR/SLE/SS/TM improve performance when the concurrently executed instances of the critical sections do not have data conflicts with each other. In contrast, ACS improves performance even for critical section instances that have data conflicts. If data conflicts are frequent, TLR/SLE/SS/TM can degrade performance by rolling back the speculative execution of all but one instance to the beginning of the critical section. In contrast, ACS's performance is not affected by data conflicts in critical sections.
2. TLR/SLE/SS/TM amortize critical section latency by concurrently executing non-conflicting critical sections, but they do not reduce the latency of each critical section. In contrast, ACS reduces the execution latency of critical sections.
3. TLR/SLE/SS/TM do not improve locality of lock and shared data. In contrast, as Section 7.2 showed, ACS improves locality of lock and shared data by keeping them in a single cache.

We compare the performance of ACS and TLR. Figure 13 shows the execution time of an ACMP augmented with TLR[10] and the execution time of ACS normalized to ACMP (area budget is 32 and number of threads set to the optimal number for each system). TLR reduces average execution time by 6% while ACS reduces it by 23%. In applications where critical sections often access disjoint data (e.g., `puzzle`, where the critical section protects a heap to which accesses are disjoint), TLR provides large performance improvements. However, in workloads where critical sections conflict with each other (e.g., `is`, where each instance of the critical section updates all elements of a shared array), TLR degrades performance. ACS outperforms TLR on all benchmarks, and by 18% on average because ACS accelerates many critical sections, whether or not they have data conflicts, thereby reducing serialization.

[10] TLR was implemented as described in [34]. We added a 128-entry buffer to each small core to handle speculative memory updates.

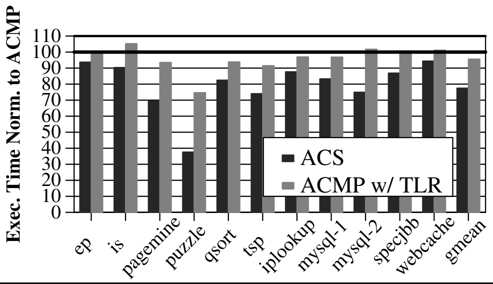


Figure 13. ACS vs. TLR performance.

8.3 Asymmetric CMPs

CMPs with heterogeneous cores have been proposed to reduce power consumption and improve performance. Morad et al. [30] proposed an analytic model of a CMP with one large core and multiple small, low-performance cores. The large core would be used to accelerate the serial bottleneck. Suleman et al. [41] show that the ACMP model can improve programmer efficiency. Hill at al. [15] further show that there is potential in improving the performance of the serial part of an application. Kumar et al. [24] use heterogeneous cores to reduce power and increase throughput for multi-programmed workloads. We use the ACMP to accelerate critical sections as well as the serial part in multi-threaded workloads.

Ipek et al. [21] propose Core Fusion, where multiple small cores can be combined, i.e. fused, to form a powerful core at runtime. They apply Core Fusion to speed up the serial portion of programs. Our technique can be combined with Core Fusion. A powerful core can be built by fusing multiple small cores to accelerate critical sections.

8.4 Other Related Work

The idea of executing critical sections remotely on a different processor resembles the *Remote Procedure Call (RPC)* [8] mechanism used in network programming to ease the construction of distributed, client-server based applications. RPC is used to execute (client) subroutines on remote (server) computers. In ACS, the small cores are analogous to the "client," and the large core is analogous to the "server" where the critical sections are remotely executed. ACS has two major differences from RPC. First, ACS executes "remote" critical section calls within the same address space and the same chip as the callee, thereby enabling the accelerated execution of shared-memory multi-threaded programs. Second, ACS's purpose is to accelerate shared-memory parallel programs and hence reduce the burden of parallel programming, whereas RPC's purpose is to ease network programming.

9. Conclusion

We proposed *Accelerated Critical Sections (ACS)* to improve the performance and scalability of multi-threaded applications. ACS accelerates execution of critical sections by executing them on the large core of an Asymmetric CMP (ACMP). Our evaluation with 12 critical section intensive workloads shows that ACS reduces the average execution time by 34% compared to an equal-area baseline with 32-core symmetric CMP and by 23% compared to an equal-area ACMP. Furthermore, ACS improves the scalability of 7 of the 12 workloads. As such, ACS is a promising approach to overcome the performance bottlenecks introduced by critical sections.

Acknowledgments

We thank Khubaib and Rustam Miftakhutdinov for their help. Special thanks to Eric Sprangle, Anwar Rohillah, Hyesoon Kim, and Jose Joao for helpful technical discussions and the Cockrell Foundation, IBM, Intel, and Microsoft for their generous support.

References

[1] MySQL database engine 5.0.1. http://www.mysql.com, 2008.

[2] Opening Tables scalability in MySQL. MySQL Performance Blog. http://www.mysqlperformanceblog.com/2006/11/21/-opening-tables-scalability, 2006.

[3] SQLite database engine version 3.5.8. 2008.

[4] SysBench: a system performance benchmark version 0.4.8. http://sysbench.sourceforge.net, 2008.

[5] S. Adve et al. Replacing locks by higher-level primitives. Technical Report TR94-237, Rice University, 1994.

[6] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS*, 1967.

[7] D. H. Bailey et al. NAS parallel benchmarks. Technical Report Tech. Rep. RNR-94-007, NASA Ames Research Center, 1994.

[8] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[9] C. Brunschen et al. OdinMP/CCp - a portable implementation of OpenMP for C. *Concurrency: Prac. and Exp.*, 2000.

[10] D. Culler et al. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann, 1998.

[11] A. J. Dorta et al. The OpenMP source code repository. In *Euromicro*, 2005.

[12] S. Gochman et al. The Intel Pentium M processor: Microarchitecture and performance. 7(2):21–36, May 2003.

[13] G. Grohoski. Distinguished Engineer, Sun Microsystems. Personal communication, November 2007.

[14] M. Herlihy and J. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA-20*, 1993.

[15] M. Hill and M. Marty. Amdahl's law in the multicore era. *IEEE Computer*, 41(7), 2008.

[16] R. Hoffmann et al. Using hardware operations to reduce the synchronization overhead of task pools. *ICPP*, 2004.

[17] Intel. Prescott New Instructions Software Dev. Guide. 2004.

[18] Intel. Source code for Intel threading building blocks.

[19] Intel. *Pentium Processor User's Manual Volume 1: Pentium Processor Data Book*, 1993.

[20] Intel. IA-32 Intel Architecture Software Dev. Guide, 2008.

[21] E. Ipek et al. Core fusion: accommodating software diversity in chip multiprocessors. In *ISCA-34*, 2007.

[22] P. Kongetira et al. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, 2005.

[23] H. Kredel. Source code for traveling salesman problem (tsp). http://krum.rz.uni-mannheim.de/ba-pp-2007/java/index.html.

[24] R. Kumar et al. Heterogeneous chip multiprocessors. *IEEE Computer*, 38(11), 2005.

[25] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *CACM*, 17(8):453–455, August 1974.

[26] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *ISCA*, pages 241–251, 1997.

[27] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.

[28] C. Liao et al. OpenUH: an optimizing, portable OpenMP compiler. *Concurr. Comput. : Pract. Exper.*, 2007.

[29] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X*, 2002.

[30] T. Morad et al. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *CAL*, 2006.

[31] R. Narayanan et al. MineBench: A Benchmark Suite for Data Mining Workloads. In *IISWC*, 2006.

[32] Y. Nishitani et al. Implementation and evaluation of OpenMP for Hitachi SR8000. In *ISHPC-3*, 2000.

[33] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO-34*, 2001.

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

[35] P. Ranganathan et al. The interaction of software prefetching with ILP processors in shared-memory systems. In *ISCA-24*, 1997.

[36] C. Rossbach et al. TxLinux: using and managing hardware transactional memory in an operating system. In *SOSP'07*, 2007.

[37] M. Sato et al. Design of OpenMP compiler for an SMP cluster. In *EWOMP*, Sept. 1999.

[38] L. Seiler et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 2008.

[39] S. Sridharan et al. Thread migration to improve synchronization performance. In Workshop on OSIHPA, 2006.

[40] The Standard Performance Evaluation Corporation. *Welcome to SPEC.* http://www.specbench.org/.

[41] M. Suleman et al. ACMP: Balancing Hardware Efficiency and Programmer Efficiency. Technical Report TR-HPS-2007-001, 2007.

[42] M. Suleman et al. An Asymmetric Multi-core Architecture for Accelerating Critical Sections. Technical Report TR-HPS-2008-003, 2008.

[43] M. Suleman et al. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *ASPLOS XIII*, 2008.

[44] J. M. Tendler et al. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[45] Tornado Web Server. http://tornado.sourceforge.net/.

[46] P. Trancoso and J. Torrellas. The impact of speeding up critical sections with data prefetching and forwarding. In *ICPP*, 1996.

[47] M. Tremblay et al. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *ISSCC*, 2008.

[48] D. M. Tullsen et al. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA-22*, 1995.

[49] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. In *SIGCOMM*, 1997.

[50] Wikipedia. Fifteen puzzle. http://en.wikipedia.org/wiki/-Fifteenpuzzle.

[51] S. C. Woo et al. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA-22*, 1995.

[52] P. Zhao and J. N. Amaral. Ablego: a function outlining and partial inlining framework. *Softw. Pract. Exper.*, 37(5):465–491, 2007.