An Asymmetric Multi-core Architecture for Accelerating Critical Sections

M. Aater Suleman   Onur Mutlu   Moinuddin Qureshi   Yale N. Patt

High Performance Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712-0240

TR-HPS-2008-003
September 2008

An Asymmetric Multi-core Architecture for Accelerating Critical Sections

M. Aater Suleman   Onur Mutlu   Moinuddin Qureshi   Yale N. Patt

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. As a result, 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 out of the 12 workloads, ACS improves scalability by increasing the number of threads at which performance saturates.*

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-Multiprocessor (CMP) architectures [23, 46, 43] 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 [26]. 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 [38]), 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 [25, 15, 32] 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 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. This is done by 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 the following 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 the proposed architecture 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 baseline ACMP.

[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.

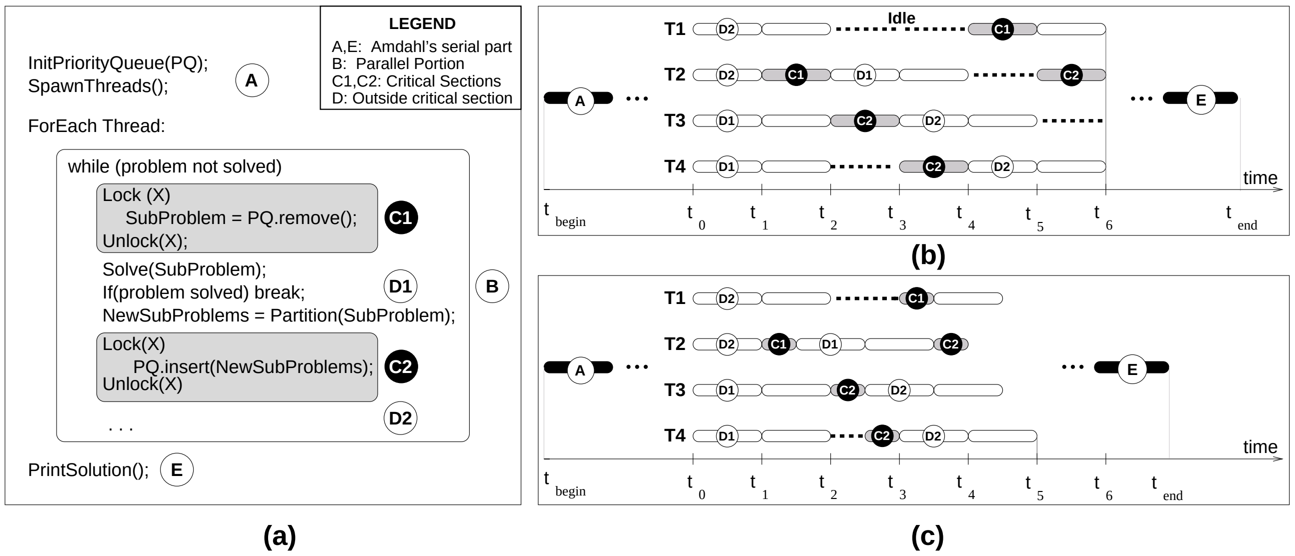
[2]We simulated a CMP with private L1 and L2 caches and a shared L3 cache. Section 5 describes our experimental methodology.

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 [28]. 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.



```
InitPriorityQueue(PQ);                          A
SpawnThreads();

ForEach Thread:
    while (problem not solved)
        Lock (X)
            SubProblem = PQ.remove();           C1
        Unlock(X);
        Solve(SubProblem);
        If(problem solved) break;               D1      B
        NewSubProblems = Partition(SubProblem);
        Lock(X)
            PQ.insert(NewSubProblems);          C2
        Unlock(X)
        . . .                                   D2

PrintSolution();                                E
```

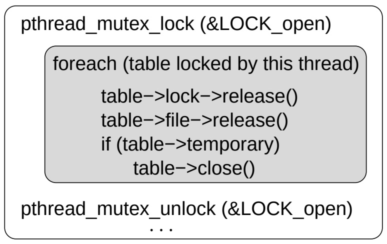
Figure 1: Serial part, parallel part, and critical section in a multi-threaded kernel of 15-puzzle (a) Code example (b) Execution timeline on the baseline CMP (c) Execution timeline with accelerated critical sections.

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 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 exit the critical section. 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. 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 also 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 closes all the tables opened by a particular transaction. A similar function (not shown) exists to open the tables and is protected by the same lock. On average, this critical section executes 670 instructions. The average length of each transaction (for the `oltp-simple` input set) is 40K instructions. As a result, critical sections account for 3% of the total instructions which leads to high contention. The serialization caused by 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.

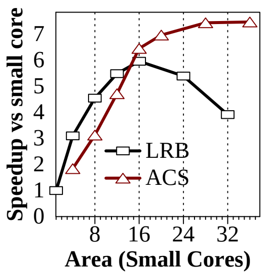


Figure 3: Scalability of MySQL.

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 [32, 25, 15], 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). Similar to previous ACMP proposals [25, 15, 32], 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.

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

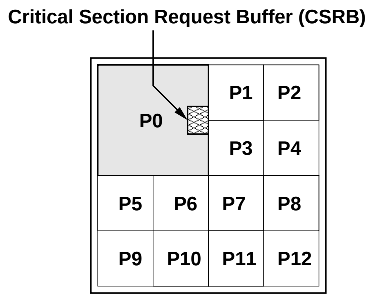
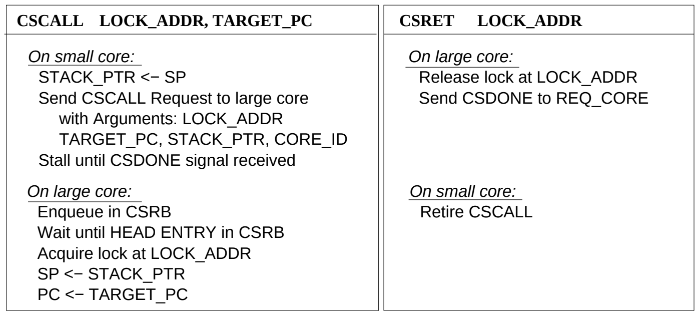


Figure 4: ACS on ACMP with 1 large core and 12 small cores



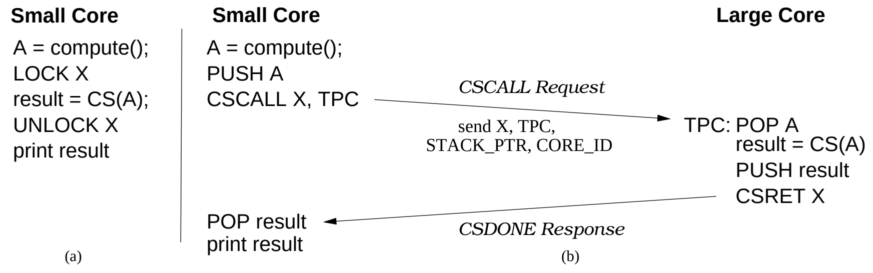
```
CSCALL   LOCK_ADDR, TARGET_PC
  On small core:
    STACK_PTR <- SP
    Send CSCALL Request to large core
       with Arguments: LOCK_ADDR
       TARGET_PC, STACK_PTR, CORE_ID
    Stall until CSDONE signal received
  On large core:
    Enqueue in CSRB
    Wait until HEAD ENTRY in CSRB
    Acquire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

CSRET   LOCK_ADDR
  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

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 the critical section in a separate function and ensuring that all input and output parameters of the function are communicated via the stack. Several OpenMP compilers already do function outlining for critical sections [30, 39, 9]. Therefore, compiler modifications are 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).



```
A = compute();
LOCK X
result = CS(A);
UNLOCK X
print result
```

```
A = compute();
PUSH A
CSCALL X, TPC
POP result
print result
```

```
TPC: POP A
     result = CS(A)
     PUSH result
     CSRET X
```

Figure 6: Source code and its execution: (a) baseline (b) with 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 [23] 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 cores, the CSRB has 12 entries, 25-bytes[3] each. Thus, the storage overhead of the CSRB is 300 bytes.

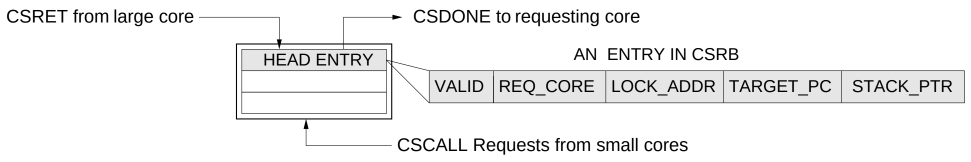


Figure 7: Critical Section Request Buffer (CSRB)

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 [23] 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.

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

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 [21].

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) [47] 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. 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) [47]. 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

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

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.

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 similar 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

[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.

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 reduces 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 [38]. 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 [35] 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.

[6]By keeping all 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.

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 [20], 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 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.

Table 1: Configuration of the simulated machines

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 [27], 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 where area is equal to N small cores. We vary N 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)

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.

We briefly describe the benchmarks whose source code is not publicly available.[7] `iplookup` is an Internet Protocol (IP) packet routing algorithm [49]. Each thread maintains a copy of the routing table, each with a separate lock. On a lookup, a thread locks and searches its own routing table. On an update, a thread locks and updates all routing tables. Thus, the updates, although infrequent, cause substantial serialization and disruption of data locality.

[7]The source code of these benchmarks will be made available publicly on our website.

Table 2: Simulated workloads

Locks	Workload	Description	Source	Input set	# of disjoint critical sections	What is Protected by CS?
Coarse	ep	Random number generator	NAS suite [7]	262144 nums.	3	reduction into global data
	is	Integer sort	NAS suite [7]	n = 64K	1	buffer of keys to sort
	pagemine	Data mining kernel	MineBench [33]	10Kpages	1	global histogram
	puzzle	15-Puzzle game	[50]	3x3	2	work-heap, memoization table
	qsort	Quicksort	OpenMP SCR [11]	20K elem.	1	global work stack
	sqlite	sqlite3 [3] database engine	SysBench [4]	OLTP-simple	5	database tables
	tsp	Traveling salesman prob.	[24]	11 cities	2	termination cond., solution
Fine	iplookup	IP packet routing	[49]	2.5K queries	# of threads	routing tables
	oltp-1	MySQL server [1]	SysBench [4]	OLTP-simple	20	meta data, tables
	oltp-2	MySQL server [1]	SysBench [4]	OLTP-complex	29	meta data, tables
	specjbb	JAVA business benchmark	[42]	5 seconds	39	counters, warehouse data
	webcache	Cooperative web cache	[44]	100K queries	33	replacement policy

`puzzle` solves a 15-Puzzle problem [50] using a branch-and-bound algorithm. There are two shared data structures: a work-list implemented as a priority heap and a memoization table to prevent threads from duplicating computation. Priority in the work-list is based on the Manhattan distance from the final solution. The work-list (heap) is traversed every iteration, which makes the critical sections long and highly contended for. `webcache` implements a shared software cache used for caching "pages" of files in a multi-threaded web server. Since, a cache access can modify the contents of the cache and the replacement policy, it is encapsulated in a critical section. One lock is used for every file with at least one page in the cache. Accesses to different files can occur concurrently. `pagemine` is derived from the data mining benchmark `rsearchk` [33]. Each thread gathers a local histogram for its data set and adds it to the global histogram inside a critical section.

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.

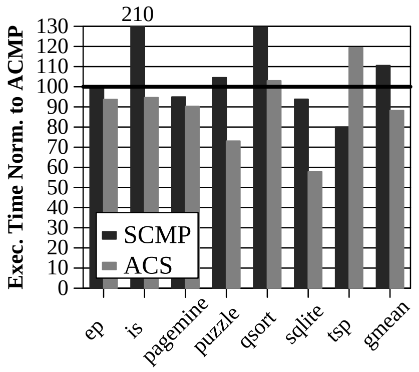
6.1. ACS Performance with the Optimal Number of Threads

Developers sometimes use profile information to choose the number of threads that minimizes execution time [19]. 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.[8] 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.

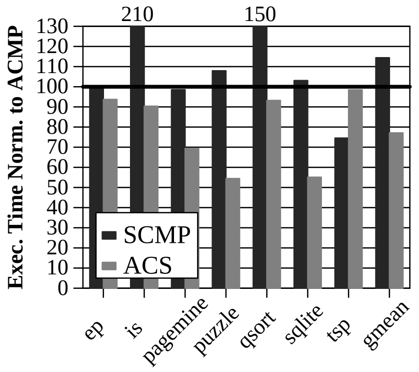
6.1.1. Workloads with Coarse-Grained Locks Figure 8 shows the execution time of each application on SCMP and ACS normalized to ACMP for three different area budgets: 8, 16, and 32. Recall that when area

[8]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.

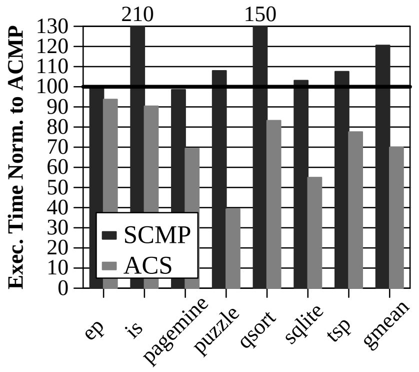
budget is equal to N, SCMP, ACMP, and ACS can execute up to N, N-2, and N-4 parallel threads respectively. In the ensuing discussion, we refer to Table 3, which shows the characteristics of critical sections in each application, to provide insight into the performance results.



(a) Area budget=8 small cores



(b) Area budget=16 small cores



(c) Area budget=32 small cores

Figure 8: Normalized execution time of ACS and SCMP on workloads with coarse-grained locking.

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`). 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.

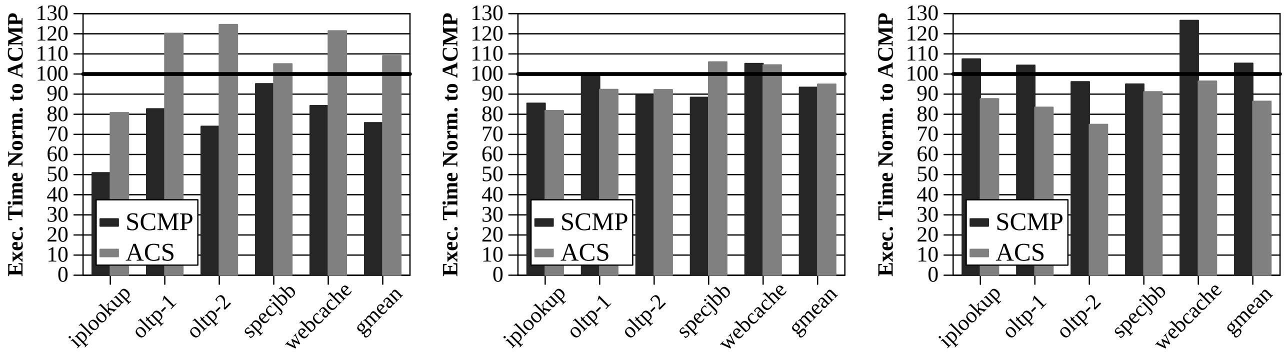
Table 3: Characteristics of Critical Sections. Shared/Private is the ratio of *shared* data (number of cache lines that are transferred from caches of other cores) to *private* data (number of cache lines that hit in the private cache) accessed inside a critical section. Contention is the average number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

Workload	% of total instr. in Serial part	% of parallel instr. in critical sections	# of disjoint critical sections	Avg. instr. in critical section	Shared/Private (at 4 threads)	Contention 4	Contention 8	Contention 16	Contention 32
ep	13.3	14.6	3	620618.1	**1.0**	1.4	1.8	4.0	8.2
is	84.6	8.3	1	9975.0	**1.1**	2.3	4.3	8.1	16.4
pagemine	0.4	5.7	1	531.0	**1.7**	2.3	4.3	8.2	15.9
puzzle	2.4	69.2	2	926.9	**1.1**	2.2	4.3	8.3	16.1
qsort	28.5	16.0	1	127.3	0.7	1.1	3.0	9.6	25.6
sqlite	0.2	17.0	5	933.1	**2.4**	1.4	2.2	3.7	6.4
tsp	0.9	4.3	2	29.5	0.4	1.2	1.6	2.0	3.6
iplookup	0.1	8.0	4	683.1	0.6	1.2	1.3	1.5	1.9
oltp-1	2.3	13.3	20	277.6	0.8	1.2	1.2	1.5	2.2
oltp-2	1.1	12.1	29	309.6	0.9	1.1	1.2	1.4	1.6
specjbb	1.2	0.3	39	1002.8	0.5	1.0	1.0	1.0	1.2
webcache	3.5	94.7	33	2257.0	**1.1**	1.1	1.1	1.1	1.4

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, executing 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.

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).



(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

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 similar 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 small cores are already in the market [23], chips with 16 and 32 small cores are being built [46, 40], and chips with 80 small cores are already prototyped [48], 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.

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

	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

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.

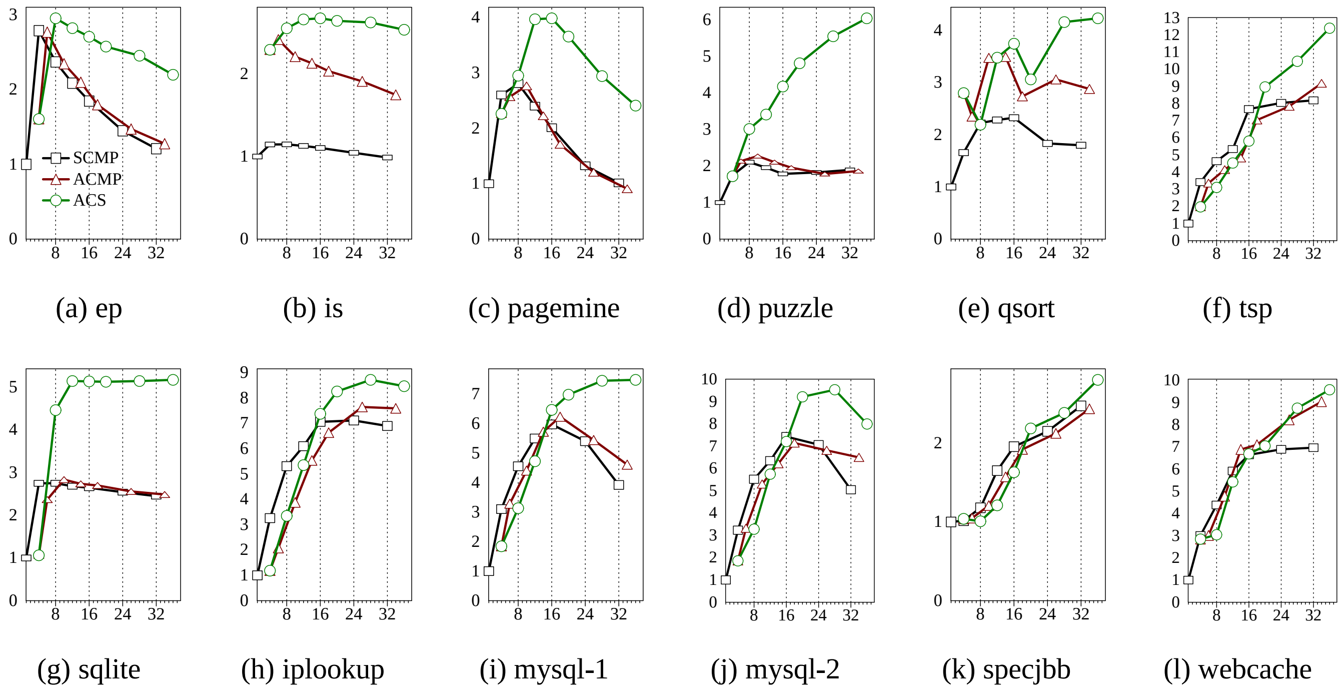


Figure 10: Speedup over a single thread running on a single small core.

6.2. ACS 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, 34]. 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.

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.

Table 5: Average Execution time normalized to area-equivalent ACMP.

Number of threads	No. of max. thread contexts			Optimal		
Area Budget	8	16	32	8	16	32
SCMP	93	104	118	94	105	115
ACS	97	77	64	96	83	77

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 not in a position 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 critical-section related inter-thread contention significantly better than ACMP or SCMP.

Table 6: Contention at an area budget of 32 (Number of threads set equal to the number of thread contexts)

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

6.3. Effect of ACS on Application Scalability

We examine the effect of ACS on the number of threads required to minimize the execution time of each application. 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.[9] We conclude that if thread contexts are available on the chip, ACS uses them more effectively compared to ACMP and SCMP.

Table 7: Best number of threads for each configuration.

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

6.4. Performance of ACS on Critical Section Non-Intensive Benchmarks

We have 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). We find that ACS does not significantly improve or degrade the performance of any of these application compared to ACMP. ACS provides a modest 1% performance improvement over ACMP and 2% performance reduction compared to SCMP. At increased area budgets, ACS performs similarly to (within 1% of) SCMP and ACMP. We conclude that ACS will not significantly affect the performance of critical section non-intensive workloads in future systems with large number of cores. (We do not present detailed results due to space limitations. We will include them in an extended technical report.)

[9]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.

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) 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.

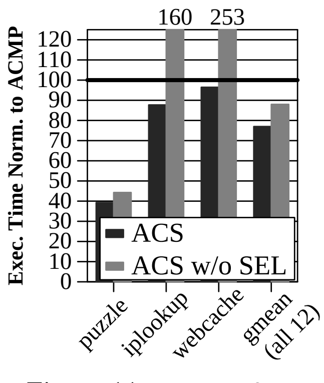


Figure 11: Impact of SEL.

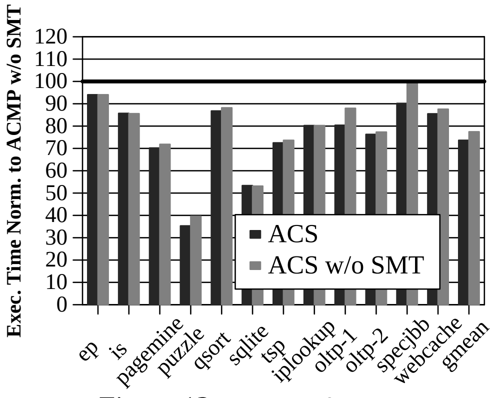


Figure 12: Impact of SMT.

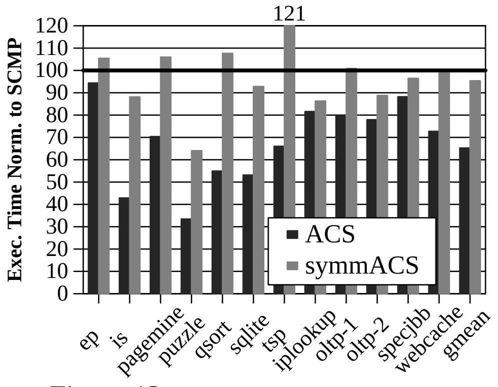


Figure 13: ACS on symmetric CMP.

7.2. Effect of using SMT on the Large Core

We have shown that ACS significantly improves performance over SCMP and ACMP when the large core provides support for SMT. The added SMT context provides ACS with the opportunity to concurrently execute critical sections that are protected by different locks on the high performance core. When the large core does not support SMT, contention for the large core can increase and lead to false serialization. Since SMT is not a

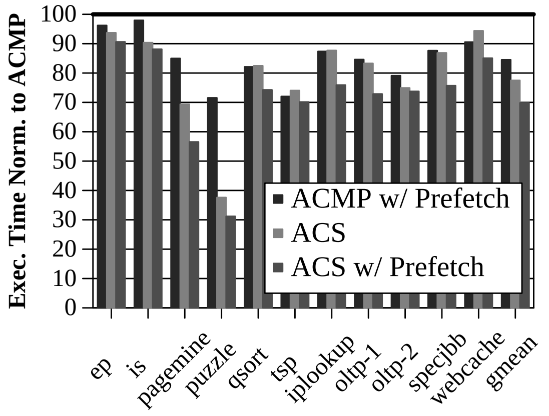


Figure 14: Impact of prefetching

requirement for ACS, we evaluate ACS on an ACMP where the large core does not support SMT and executes only one thread. Figure 12 shows the execution time of ACS without SMT normalized to ACMP without SMT when the area budget is 32. On average, ACS without SMT reduces execution time by 22% whereas ACS with SMT by 26%. Thus, SMT provides 4% performance benefit by reducing false serialization of critical sections.

7.3. 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 can execute the CSCALL requests and CSRET instructions. Figure 13 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.[10] 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 section execution using the large core.

7.4. Interaction of ACS with Hardware Prefetching

Part of the performance benefit of ACS comes from improving shared data/lock locality, which can also be partially improved by data prefetching [45, 37]. To study the effect of prefetching, we augment each core with a L2 stream prefetcher [43] (32 streams, up to 16 lines ahead). Figure 14 shows the execution time of ACMP with a prefetcher, ACS (with and without a prefetcher), all normalized to an ACMP without a prefetcher (area budget is 32). On all benchmarks, prefetching improves the performance of both ACMP and ACS, and ACS with a prefetcher outperforms ACMP with a prefetcher. However, in `puzzle`, `qsort`, `tsp`, and `oltp-2`, ACMP benefits more from prefetching than ACS because these workloads contain shared data structures that

[10]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.

lend themselves to prefetching. For example, in `tsp`, one of the critical sections protects an array. All elements of the array are read, and often updated, inside the critical section which leads to cache misses in ACMP. The stream prefetcher successfully prefetches this array, which reduces the execution time of the critical sections. As a result, ACMP with a prefetcher is 28% faster. Because ACS already reduces the misses for the array by keeping it at the large core, the improvement from prefetching is modest compared to ACMP (4%). On average, ACS with prefetching reduces execution time by 18% compared to ACMP with prefetching and 10% compared to ACS without prefetching. Thus, ACS interacts positively with a stream prefetcher and both schemes can be employed together.

8. Related Work

The major contribution of our paper is a comprehensive mechanism that accelerates the execution of 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.

8.1. Related Work in Improving Locality of Shared Data and Locks

Sridharan et al. [41] 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 operating system 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 [41]. Trancoso et al. [45] and Ranganathan et al. [37] improve locality in critical sections using software prefetching. These techniques can be combined with ACS for improved performance.

Several primitives (e.g., Test&Set, 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, 29, 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. Related Work in 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) [35], transactional lock removal (TLR) [36], and speculative synchronization (SS) [31]. 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

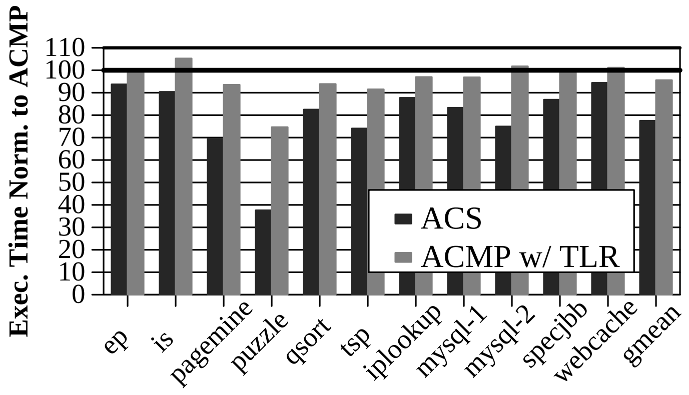


Figure 15: ACS vs. TLR performance.

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 our results in Section 7.3 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 15 shows the execution time of an ACMP augmented with TLR[11] 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. This is because ACS accelerates many critical sections regardless of whether or not they have data conflicts, thereby reducing serialization.

8.3. Related Work in Asymmetric CMPs

CMPs with heterogeneous cores have been proposed to reduce power and improve performance. Morad et al. [32] proposed an Asymmetric Chip Multiprocessor (ACMP) with one large core and multiple small, low-performance cores. The large core was used to accelerate the serial bottleneck. Hill at al. [15] build on the

[11]TLR was implemented as described in [36]. We added a 128-entry buffer to each small core to handle speculative memory updates.

ACMP model and further show that there is potential in improving the performance of the serial part of an application. Kumar et al. [25] propose heterogeneous cores to reduce power and increase throughput for multi-programmed workloads. Our proposal utilizes the ACMP architecture to accelerate the execution of critical sections as well as the serial part in parallel workloads.

Ipek et al. [22] propose Core Fusion, which consists of multiple small cores that can be combined, i.e. fused, to form a powerful core at runtime if parallelism is low. They also apply Core Fusion to speed up serial portion of programs. Our technique can be adapted to work on a Core Fusion architecture, where multiple execution engines can be combined to form a powerful execution engine 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, whereas RPC's purpose is to ease network programming.

9. Conclusion

This paper proposed *Accelerated Critical Sections (ACS)*, a novel technique 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. Our future work will examine resource allocation in ACS architecture and extending/generalizing the ACS paradigm to accelerate critical program paths.

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. http:/www.sqlite.org, 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.*, 12(12), 2000.
[10] D. Culler, J. Singh, and A. Gupta. *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. E. B. 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. http://cache-www.intel.com/cd/00/00/06/67/66753_66753.pdf, 2004.
[18] Intel. Source code for Intel threading building blocks. http://threadingbuildingblocks.org/uploads/78/75/2.0/tbb20_014oss_src.tar.gz.
[19] Intel. Threading methodology: Principles and practices. www.intel.com/cd/ids/developer/asmo-na/eng/219349.htm, 2003.
[20] Intel. *Pentium Processor User's Manual Volume 1: Pentium Processor Data Book*, 1993.
[21] Intel. IA-32 Intel Architecture Software Dev. Guide, 2008.
[22] E. Ipek et al. Core fusion: accommodating software diversity in chip multiprocessors. In *ISCA-34*, 2007.
[23] P. Kongetira et al. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, 2005.
[24] H. Kredel. Source code for traveling salesman problem (tsp). http://krum.rz.uni-mannheim.de/ba-pp-2007/java/index.html.
[25] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 38(11), 2005.
[26] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *CACM*, 17(8):453–455, August 1974.
[27] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *ISCA*, pages 241–251, 1997.
[28] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.
[29] U. Legedza and W. E. Weihl. Reducing synchronization overhead in parallel simulation. In *PADS '96*, 1996.
[30] C. Liao et al. OpenUH: an optimizing, portable OpenMP compiler. *Concurr. Comput. : Pract. Exper.*, 19(18):2317–2332, 2007.
[31] J. F. Martínez and J. Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X*, 2002.
[32] T. Morad et al. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *Comp Arch Lttrs*, 2006.
[33] R. Narayanan et al. MineBench: A Benchmark Suite for Data Mining Workloads. In *IISWC*, 2006.
[34] Y. Nishitani et al. Implementation and evaluation of OpenMP for Hitachi SR8000. In *ISHPC-3*, 2000.
[35] R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *MICRO-34*, 2001.
[36] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS-X*, 2002.
[37] P. Ranganathan et al. The interaction of software prefetching with ILP processors in shared-memory systems. In *ISCA-24*, 1997.
[38] C. Rossbach et al. TxLinux: using and managing hardware transactional memory in an operating system. In *SOSP'07*, 2007.
[39] M. Sato et al. Design of OpenMP compiler for an SMP cluster. In *EWOMP*, Sept. 1999.
[40] L. Seiler et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 2008.
[41] S. Sridharan et al. Thread migration to improve synchronization performance. In Workshop on OS Interference in High Performance Applications, 2006.
[42] The Standard Performance Evaluation Corporation. *Welcome to SPEC*. http://www.specbench.org/.
[43] J. M. Tendler et al. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.
[44] Tornado Web Server. Source code. http://tornado.sourceforge.net/.
[45] P. Trancoso and J. Torrellas. The impact of speeding up critical sections with data prefetching and forwarding. In *ICPP*, 1996.
[46] M. Tremblay et al. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC Processor. In *ISSCC*, 2008.
[47] D. M. Tullsen et al. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA-22*, 1995.
[48] S. Vangal et al. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *IEEE Journal of Solid-State Circuits*, 2008.
[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/Fifteen_puzzle.
[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.