Accelerating Critical Section Execution with Asymmetric Multicore Architectures

Contention for critical sections can reduce performance and scalability by causing thread serialization. The proposed accelerated critical sections mechanism reduces this limitation. ACS executes critical sections on the high-performance core of an asymmetric chip multiprocessor (ACMP), which can execute them faster than the smaller cores can.

M. Aater Suleman
University of Texas at Austin

Onur Mutlu
Carnegie Mellon University

Moinuddin K. Qureshi
IBM Research

Yale N. Patt
University of Texas at Austin

Extracting high performance from chip multiprocessors (CMPs) requires partitioning the application into threads that execute concurrently on multiple cores. Because threads cannot be allowed to update shared data concurrently, accesses to shared data are encapsulated inside critical sections. Only one thread executes a critical section at a given time; other threads wanting to execute the same critical section must wait. Critical sections can serialize threads, thereby reducing performance and scalability (that is, the number of threads at which performance saturates). Shortening the execution time inside critical sections can reduce this performance loss.

This article proposes the *accelerated critical sections* mechanism. ACS is based on the *asymmetric chip multiprocessor*, which consists of at least one large, high-performance core and many small, power-efficient cores (see the "Related work" sidebar for other work in this area). The ACMP was originally proposed to run Amdahl's serial bottleneck (where only a single thread exists) more quickly on the large core and the parallel program regions on the multiple small cores.[1-4] In addition to Amdahl's bottleneck, ACS runs selected critical sections on the large core, which runs them faster than the smaller cores.

ACS dedicates the large core exclusively to running critical sections (and the Amdahl's bottleneck). In conventional systems, when a core encounters a critical section, it acquires the lock for the critical section, executes the critical section, and releases the lock. In ACS, when a small core encounters a critical section, it sends a request to the large core for execution of that critical section and stalls. The large core executes the critical section and notifies the small core when it has completed the critical section. The small core then resumes execution. By accelerating critical section execution, ACS reduces serialization, lowering 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 percent compared to an equal-area 32-core

Published by the IEEE Computer Society

Related work

The most closely related work to accelerated critical sections (ACS) are the numerous proposals to optimize the implementation of lock acquire and release operations and the locality of shared data in critical sections using operating system and compiler techniques. We are not aware of any work that speeds up the execution of critical sections using more aggressive execution engines.

Improving locality of shared data and locks

Sridharan et al.[1] propose a thread-scheduling algorithm to increase shared data locality. When a thread encounters a critical section, the operating system migrates the thread to the processor with the shared data. Although ACS also improves shared data locality, it does not require the operating system intervention and the thread migration overhead incurred by their scheme. Moreover, ACS accelerates critical section execution, a benefit unavailable in Sridharan et al.'s work.[1] Trancoso and Torrellas[2] and Ranganathan et al.[3] improve locality in critical sections using software prefetching. We can combine these techniques with ACS to improve performance. Primitives (such as Test&Test&Set and Compare&Swap) implement lock acquire and release operations efficiently but do not increase the speed of critical section processing or the locality of shared data.[4]

Hiding the latency of critical sections

Several schemes hide critical section latency.[5-7] We compared ACS with TLR, a mechanism that hides critical section latency by overlapping multiple instances of the same critical section as long as they access disjoint data. ACS largely outperforms TLR because, unlike TLR, ACS accelerates critical sections whether or not they have data conflicts.

Asymmetric CMPs

Several researchers have shown the potential to improve the performance of an application's serial part using an asymmetric chip multiprocessor (ACMP).[8-11] We use the ACMP to accelerate critical sections as well as the serial part in multithreaded workloads. Ïpek et al. state that multiple small cores can be combined—that is, fused—to form a powerful core to speed up the serial program portions.[12] We can combine our technique with their scheme such that the powerful core accelerates critical sections. Kumar et al. use heterogeneous cores to reduce power and increase throughput for multiprogrammed workloads, not multithreaded programs.[13]

Remote procedure calls

The idea of executing critical sections remotely on a different processor resembles the RPC[14] mechanism used in network programming. RPC is used to execute client subroutines on remote server computers. ACS differs from RPC in that ACS runs critical sections remotely on the same chip within the same address space, not on a remote computer.

References

1. S. Sridharan et al., ''Thread Migration to Improve Synchronization Performance,'' *Proc. Workshop on Operating System Interference in High Performance Applications* (OSIHPA), 2006; http://www.kinementium.com/~rcm/doc/osihpa06.pdf.
2. P. Trancoso and J. Torrellas, ''The Impact of Speeding up Critical Sections with Data Prefetching and Forwarding,'' *Proc. Int'l Conf. Parallel Processing* (ICPP 96), IEEE CS Press, 1996, pp. 79-86.
3. P. Ranganathan et al., ''The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems,'' *Proc. Ann. Int'l Symp. Computer Architecture* (ISCA 97), ACM Press, 1997, pp. 144-156.
4. D. Culler et al., *Parallel Computer Architecture: A Hardware/Software Approach,* Morgan Kaufmann, 1998.
5. J.F. Martínez and J. Torrellas, ''Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications,'' *Proc. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 18-29.
6. R. Rajwar and J.R. Goodman, ''Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution,'' *Proc. Int'l Symp. Microarchitecture* (MICRO 01), IEEE CS Press, 2001, pp. 294-305.
7. R. Rajwar and J.R. Goodman, ''Transactional Lock-Free Execution of Lock-Based Programs,'' *Proc. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 5-17.
8. M. Annavaram, E. Grochowski, and J. Shen, ''Mitigating Amdahl's Law through EPI Throttling,'' *SIGARCH Computer Architecture News,* vol. 33, no. 2, 2005, pp. 298-309.
9. M. Hill and M. Marty, ''Amdahl's Law in the Multicore Era,'' *Computer,* vol. 41, no. 7, 2008, pp. 33-38.
10. T.Y. Morad et al., ''Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors,'' *IEEE Computer Architecture Letters,* vol. 5, no. 1, Jan. 2006, pp. 14-17.
11. M.A. Suleman et al., *ACMP: Balancing Hardware Efficiency and Programmer Efficiency,* HPS Technical Report, 2007.
12. E. Ïpek et al., ''Core Fusion: Accommodating Software Diversity in Chip Multiprocessors,'' *Proc. Ann. Int'l Symp. Computer Architecture* (ISCA 07), ACM Press, 2007, pp. 186-197.
13. R. Kumar et al., ''Heterogeneous Chip Multiprocessors,'' *Computer,* vol. 38, no. 11, 2005, pp. 32-38.
14. A.D. Birrell and B.J. Nelson, ''Implementing Remote Procedure Calls,'' *ACM Trans. Computer Systems,* vol. 2, no. 1, 1984, pp. 39-59.

symmetric CMP and by 23 percent compared to an equal-area ACMP. Moreover, for seven of the 12 workloads, ACS also increases scalability.

The problem

A multithreaded application consists of two parts. The *serial* part is the classical Amdahl's bottleneck.[5] The *parallel* part is where multiple threads execute concurrently. Threads operate on different portions of the same problem and communicate via shared memory. The *mutual exclusion* principle dictates that multiple threads may not update shared data concurrently. Thus, accesses to shared data are encapsulated in regions of code guarded by synchronization primitives (such as locks) called critical sections. Only one thread can execute a particular critical section at any given time. Critical sections differ from Amdahl's serial bottleneck. During a critical section's execution, other threads that do not need to execute that critical section can make progress. In contrast, no other thread exists in Amdahl's serial bottleneck.

We use a simple example to show critical sections' performance impact.

Figure 1a shows the code for a multithreaded kernel in which each thread dequeues a work quantum from the priority queue (PQ) and attempts to process it. If the thread cannot solve the problem, it divides the problem into subproblems and inserts them into the priority queue. This is a common technique for parallelizing many branch-and-bound algorithms.[6] In our benchmarks, we use this kernel to solve the 15-puzzle problem (see http://en.wikipedia.org/wiki/Fifteen_puzzle).

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 because 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.

Figure 1b shows the execution time of the kernel in Figure 1a on a 4-core CMP. After the serial part A, four threads (T1, T2, T3, and T4) are spawned, one on each core. When part B is complete, the serial part E is executed on a single core. We analyze the serialization caused by the critical section in the 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, which allows three threads (T1, T3, and T4) to 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 executing it but also the threads waiting to enter the critical section. For example, between t_2 and t_3, two threads (T1 and T4) are waiting for T3 to exit the critical section, without performing any useful work. Therefore, accelerating the critical section's execution not only improves T3's performance, but also reduces the wasteful waiting time of T1 and T4. Figure 1c 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 significantly improve performance.

On average, the critical section in Figure 1a 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. Because of its lengthy execution, this critical section incurs high contention. When the workload is executed with eight threads, on average four 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

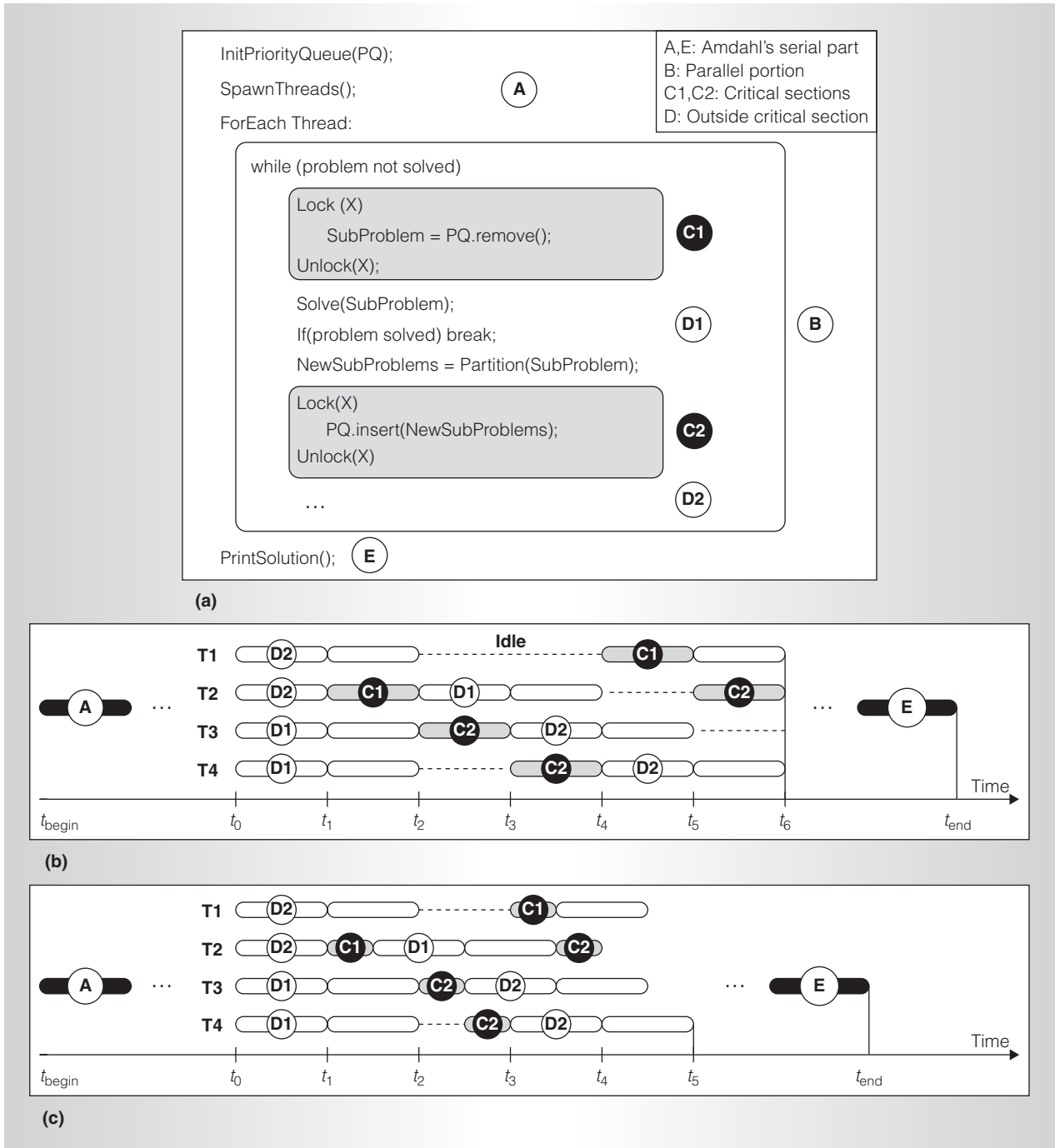

Figure 1. Amdahl's serial part, parallel part, and the critical section in a multithreaded 15-puzzle kernel: code example (a) and execution timelines on the baseline chip multiprocessor (CMP) (b) and accelerated critical sections (ACS) (c).

32 threads. In contrast, when we accelerate this critical section using ACS, the average number of waiting threads reduces to two and three for eight- and 32-threaded execution, respectively.

The task of shortening critical sections has traditionally been left to programmers. However, tuning code to minimize critical section execution requires substantial programmer time and effort. A mechanism that can

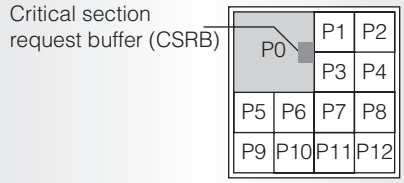

Figure 2. An example ACS architecture implemented on an asymmetric chip multiprocessor (ACMP). ACS consists of one high-performance core (P0) and several smaller cores.

shorten critical sections' execution time transparently in hardware, without requiring programmer support, can significantly impact both the software and the hardware industries.

ACS design

The ACS mechanism is implemented on a homogeneous-ISA, heterogeneous-core CMP that provides hardware support for cache coherence. ACS is based on the ACMP architecture,[3,4,7] which was proposed to handle Amdahl's serial bottleneck. ACS consists of one high-performance core and several small cores. The serial part of the program and the critical sections execute on the high-performance core, whereas the remaining parallel parts execute on the small cores.

Figure 2 shows an example ACS architecture implemented on an ACMP consisting of one large core (P0) and 12 small cores (P1 to P12). ACS executes the parallel threads on the small cores and dedicates P0 to the execution of critical sections (as well as serial program portions). A *critical section request buffer* (CSRB) in P0 buffers the critical section execution requests from the small cores. ACS introduces two new instructions—*critical section execution call* (CSCALL) and *critical section return* (CSRET)—which are inserted at the beginning and end of a critical section, respectively.

Figure 3 contrasts ACS with a conventional system. In conventional locking (Figure 3a), when a small core encounters a critical section, it acquires the lock protecting the critical section, executes the critical section, and releases the lock. In ACS (Figure 3b), when a small core executes a CSCALL, it sends the CSCALL to P0 and stalls until it receives a response. When P0 receives a CSCALL, it buffers it in the CSRB. P0 starts executing the requested critical section at the first opportunity and continues normal processing until it encounters a CSRET instruction, which signifies the end of the critical section. When P0 executes the CSRET instruction, it sends a *critical section done* (CSDONE) signal to the requesting small core. Upon receiving this signal, the small core resumes normal execution.

Our ASPLOS paper describes the ACS architecture in detail, focusing on hardware/ISA/compiler/library support, interconnect extensions, operating system support, ACS's handling of nested critical sections and interrupts/exceptions, and its accommodation of multiple large cores and multiple parallel applications.[8]

False serialization

ACS, as we have described it thus far, executes all critical sections on a single dedicated core. Consequently, ACS can serialize the execution of independent critical sections that could have executed concurrently in a conventional system. We call this effect *false serialization*. To reduce false serialization, ACS can dedicate multiple execution contexts for critical section acceleration. We can achieve this by making the large core a simultaneous multithreading (SMT) engine or by providing multiple large cores on the chip. With multiple contexts, different contexts operate on independent critical sections concurrently, thereby reducing false serialization.

Some workloads continue to experience false serialization even when more than one context is available for critical section execution. For such workloads, we propose *selective acceleration of critical sections* (SEL). SEL tracks false serialization experienced by each critical section and disables the accelerated execution of critical sections experiencing high false serialization.

To estimate false serialization, we augment the CSRB with a table of saturating counters (one per critical section) for tracking false serialization. We quantify false serialization by counting the number of critical sections in the CSRB for which the LOCK_ADDR differs from the incoming request's LOCK_ADDR. If this count is greater than 1 (that is, if the CSRB contains at least two independent critical sections), the estimation logic adds the count to the saturating

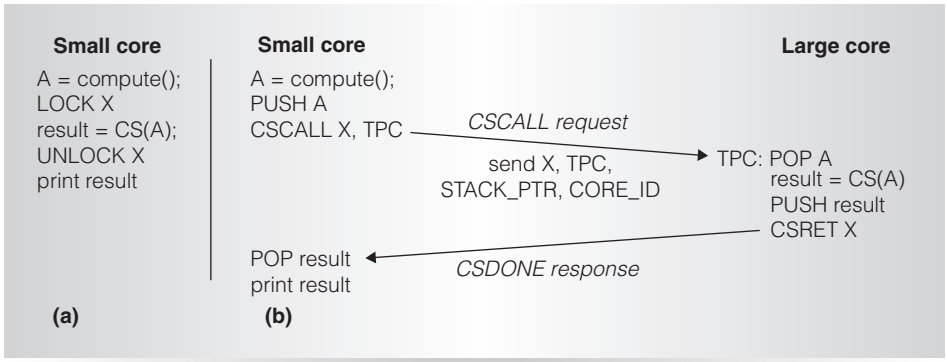

Figure 3. Source code and its execution: baseline (a) and ACS (b).

counter corresponding to the incoming request's LOCK_ADDR. If the count is 1 (that is, if the CSRB contains exactly one critical section), the estimation logic decrements the corresponding saturating counter. If the counter reaches its maximum value, the estimation logic notifies all small cores that ACS has been disabled for the particular critical section. Future CSCALLs to that critical section are executed locally at the small core. (Our SEL implementation 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. As our ASPLOS paper shows, SEL completely recovers the performance loss due to false serialization, and ACS with SEL outperforms ACS without SEL by 15 percent.[8]

Trade-off analysis: Why ACS works

ACS makes three performance trade-offs compared to conventional systems.

Faster critical sections versus fewer threads

First, ACS has fewer threads than conventional systems because it dedicates a large core, which could otherwise be used for more threads, to accelerating critical sections. ACS improves performance when the benefit of accelerating critical sections is greater than the loss due to the unavailability of more threads. ACS is more likely to improve performance when the number of cores on the chip increases for two reasons.

The marginal loss in parallel throughput due to the large core is smaller (for example, if the large core replaces four small cores, it eliminates 50 percent of the smaller cores in an 8-core system, but only 12.5 percent of cores in a 32-core system).

In addition, more cores (threads) increase critical section contention, thereby increasing the benefit of faster critical section execution.

CSCALL/CSDONE signals versus lock acquire/release

Second, ACS requires the communication of CSCALL and CSDONE transactions between a small core and a large core, an overhead not present in conventional systems. However, ACS can compensate for the overhead of CSCALL and CSDONE by keeping the lock at the large core, thereby reducing the cache-to-cache transfers incurred by conventional systems during lock acquire operations.[9] ACS actually has an advantage in that it can overlap the latency of CSCALL and CSDONE with the execution of another instance of the same critical section. A conventional locking mechanism can only acquire a lock after the critical section has been completed, which always adds a delay before critical section execution.

Cache misses due to private versus shared data

Finally, ACS transfers private data that is referenced in the critical section from the small core's cache to the large core's cache. Conventional locking does not incur this overhead. However, conventional systems incur overhead in transferring shared data. The shared data ''ping-pongs'' between caches as different threads execute the critical section. ACS eliminates transfers of shared data by keeping the data at the large core, which can offset the misses it causes to transfer private data into the large core.

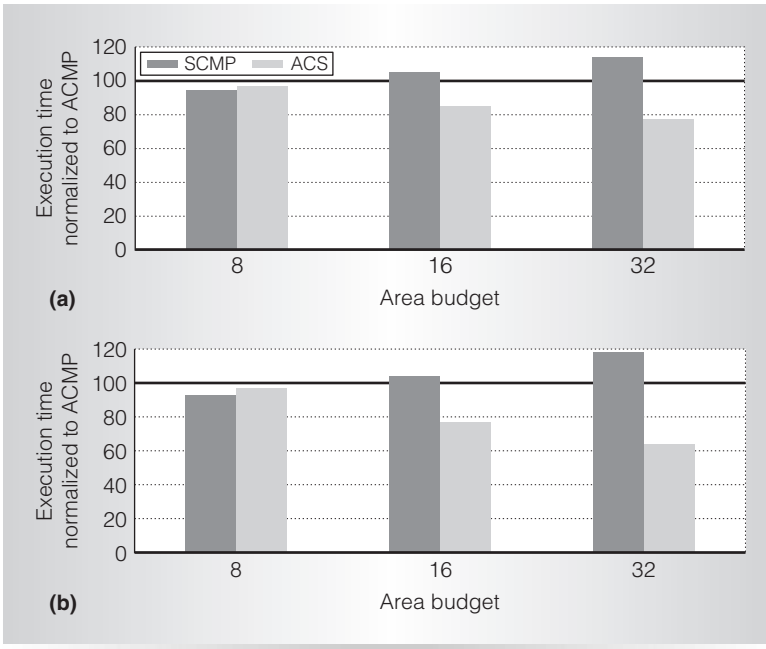

Figure 4. Execution time of symmetric CMP (SCMP) and ACS, normalized to ACMP, when the number of threads is equal to the optimal number of threads for each application (a) and when the number of threads is equal to the number of thread contexts (b).

By keeping shared data in the large core's cache, ACS provides less cache space for shared data than 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 large core's private cache is usually large enough. In fact, ACS decreases cache misses if the critical section accesses more shared data than private data.

ACS can improve performance even if there are equal or more accesses to private data than shared data because the large core can still improve the performance of other instructions and hide the latency of some cache misses using latency-tolerance techniques such as out-of-order execution.

In summary, ACS's performance benefits (faster critical section execution, improved lock locality, and improved shared data locality) are likely to outweigh its overhead (reduced parallel throughput, CSCALL and CSDONE overhead, and reduced private data locality). Our extensive experimental results on a wide variety of systems and quantitative analysis of ACS's performance trade-offs support this, as we demonstrate elsewhere.[8]

Results

We evaluate three CMP architectures: a symmetric CMP (SCMP) consisting of all small cores; an asymmetric CMP (ACMP) with one large core with two-way SMT and remaining small cores; and an ACMP augmented with support for ACS. All comparisons are at an equal-area budget. We use 12 critical-section-intensive parallel applications, including sorting, MySQL databases, data mining, IP packet routing, Web caching, and SPECjbb. Details of our methodology are available elsewhere.[8]

Performance at optimal number of threads

We evaluate ACS when the number of threads is equal to the optimal number of threads for each application-configuration pair. Figure 4a shows the execution time of SCMP and ACS normalized to ACMP at area budgets of 8, 16, and 32. ACS improves performance compared to ACMP at all area budgets. At an area budget of 8, SCMP outperforms ACS because ACS cannot compensate for the loss in throughput due to fewer parallel threads. However, this loss diminishes as the area budget increases. At area budget 32, ACS improves performance by 34 percent compared to SCMP, and 23 percent compared to ACMP.

Performance when the number of threads equals the number of contexts

When an estimate of the optimal number of threads in unavailable, most systems spawn as many threads as there are available thread contexts. Having more threads increases critical section contention, thereby further increasing ACS's performance benefit. Figure 4b shows the execution time of SCMP and ACS normalized to ACMP when the number of threads is equal to the number of contexts for area budgets of 8, 16, and 32. ACS improves performance compared to ACMP for all area budgets. ACS improves performance over SCMP for all area budgets except 8, where accelerating critical sections does not make up for the loss in throughput. For an area budget of 32, ACS outperforms both SCMP and ACMP by 46 and 36 percent, respectively.

Application scalability

We evaluate ACS's impact on application scalability (the number of threads at which performance saturates). Figure 5 shows the speedup curves of ACMP, SCMP, and ACS over one small core as the area budget varies 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 four small cores. Table 1 summarizes Figure 5 by showing the number of threads required to minimize execution time for each application using ACMP, SCMP, and ACS. For seven of the 12 applications, ACS improves scalability. For the remaining applications, scalability is not affected. We conclude that if thread contexts are available on the chip, ACS uses them more effectively than either ACMP and SCMP.

Table 1. Best number of threads for each configuration.

Workload	Symmetric CMP	Asymmetric CMP	Accelerated critical sections
ep	4	4	4
is	8	8	12
pagemine	8	8	12
puzzle	8	8	32
qsort	16	16	32
sqlite	8	8	32
tsp	32	32	32
iplookup	24	24	24
oltp-1	16	16	32
oltp-2	16	16	24
specjbb	32	32	32
webcache	32	32	32

ACS on symmetric CMP

Part of ACS's performance benefit is due to the improved locality of shared data and locks. This benefit can be realized even in the absence of a large core. We can implement a variant of ACS on a symmetric CMP, which we call *symmACS*. In symmACS, one of the small cores is dedicated to executing critical sections. On average, symmACS reduces execution time by only 4 percent, which is much lower than ACS's 34 percent performance benefit. We conclude that most of ACS's performance improvement comes from using the large core.

ACS versus techniques to hide critical section latency

Several proposals—for example, transactional memory,[10] speculative lock elision (SLE),[11] transactional lock removal (TLR),[9] and speculative synchronization[12]—try to hide a critical section's latency. These proposals execute critical sections concurrently with other instances of the same critical section as long as no data conflicts occur. In contrast, ACS accelerates all critical sections regardless of their length and the data they are accessing. Furthermore, unlike transactional memory, ACS does not require code modification; it incurs a significantly lower hardware overhead; and it reduces the cache misses for shared data, a feature unavailable in all previous schemes.

We compare the performance of ACS and TLR for critical-section-intensive applications. We implemented TLR as described elsewhere,[9] adding a 128-entry buffer to each small core to handle speculative memory updates. Figure 6 shows the execution time of an ACMP augmented with TLR and the execution time of ACS normalized to ACMP (the area budget is 32 and the number of threads is set to the optimal number for each system). ACS outperforms TLR on all benchmarks. TLR reduces average execution time by 6 percent whereas ACS reduces it by 23 percent. We conclude that ACS is a compelling, higher-performance alternative to state-of-the-art mechanisms that aim to transparently reduce critical section overhead.

Contributions and impact

Several concepts derived from our work on ACS can impact future research and development in parallel programming and CMP architectures.

Handling critical sections

Current systems treat critical sections as a part of the normal thread code and execute them in place—that is, on the same core where they are encountered. Instead of executing critical sections on the same core, ACS executes them on a remote high-performance core that executes them faster than the other smaller, cores. This idea of accelerating code remotely can be extended to other program portions.

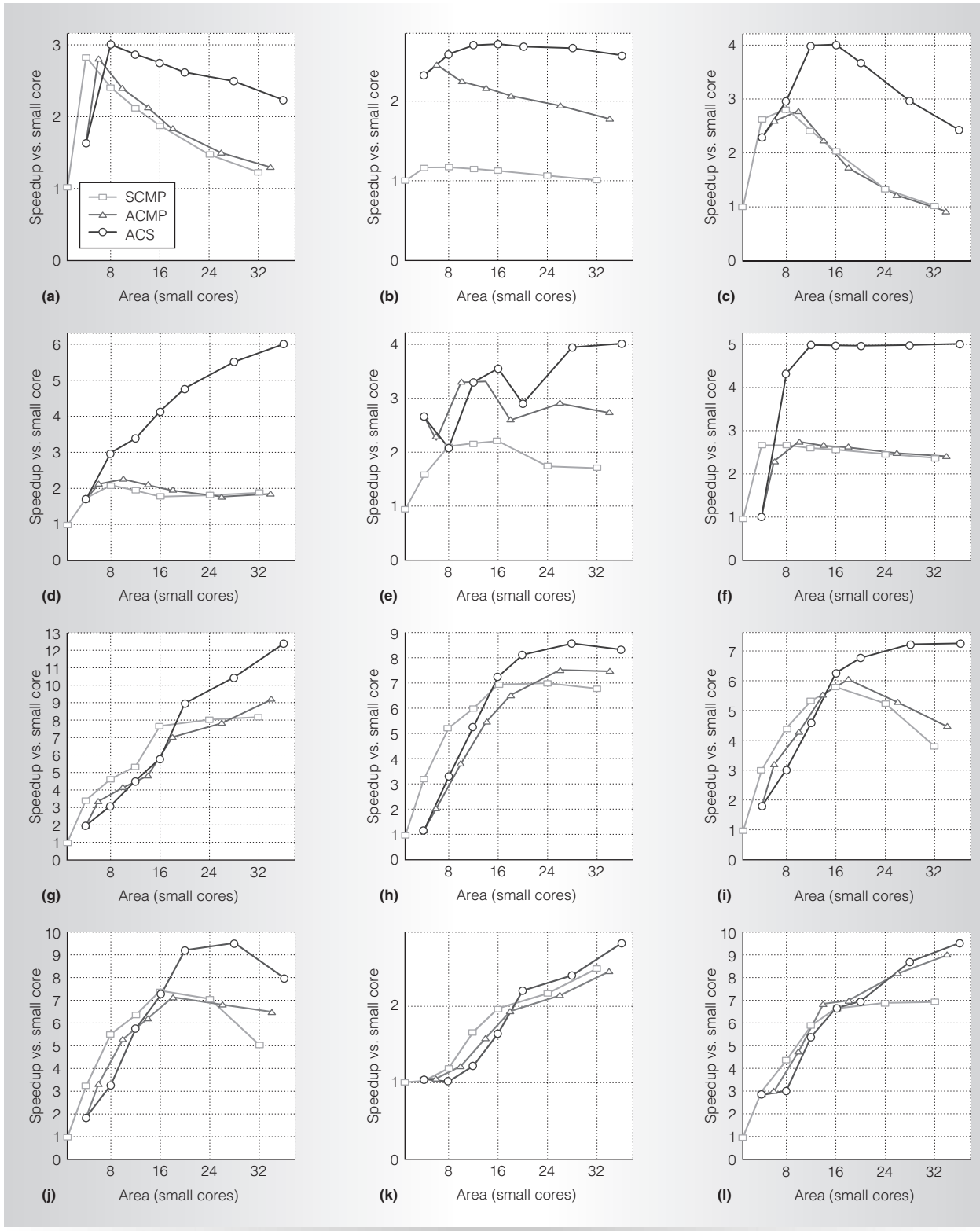

Figure 5. Speedup of ACMP, SCMP, and ACS over a single small core: ep (a), is (b), pagemine (c), puzzle (d), qsort (e), sqlite (f), tsp (g), iplookup (h), oltp-1 (i), oltp-2 (j), specjbb (k), and webcache (l).

Accelerating bottlenecks

Current proposals, such as transactional memory, try to parallelize critical sections. ACS accelerates them. This concept is applicable to other critical paths in multithreaded programs. For example, we could improve the performance of pipeline parallel (that is, streaming) workloads, which is dominated by the execution speed of the slowest pipeline stage, by accelerating the slowest stage using the large core.

Trade-off between shared and private data

Many believe that executing a part of the code at a remote core will increase the number of cache misses. ACS shows that this assumption is not always true. Remotely executing code can actually reduce the number of cache misses if more shared data than private data is accessed (as in the case of most critical sections). Compiler/runtime schedulers can use this trade-off to decide whether to run a code segment locally or remotely.

Impact on programmer productivity

Correctly inserting and shortening critical sections are among the most challenging tasks in parallel programming. By executing critical sections faster, ACS reduces the programmers' burden. Programmers can write code with longer critical sections (which is easy to get correct) and rely on hardware for critical section acceleration.

Impact on parallel algorithms

ACS could enable the use of more efficient parallel algorithms that traditionally are not used in favor of algorithms with shorter critical sections. For example, ACS could make practical the use of memoization tables (data structures to cache computed values for avoiding computation redundancy), which are avoided in parallel programs because their use introduces long critical sections with poor shared-data locality.

Impact on CMP architectures

Our previous work proposed ACMP with a single large core that runs the serial Amdahl's bottleneck.[4] Thus, without ACS, ACMP is inapplicable for parallel programs with nonexistent (or small) serial portions (such as server applications). ACS provides a mechanism to leverage one or more large cores in the parallel program portions, which makes the ACMP applicable to programs with or without serial portions. This makes ACMP practical for multiple market segments.

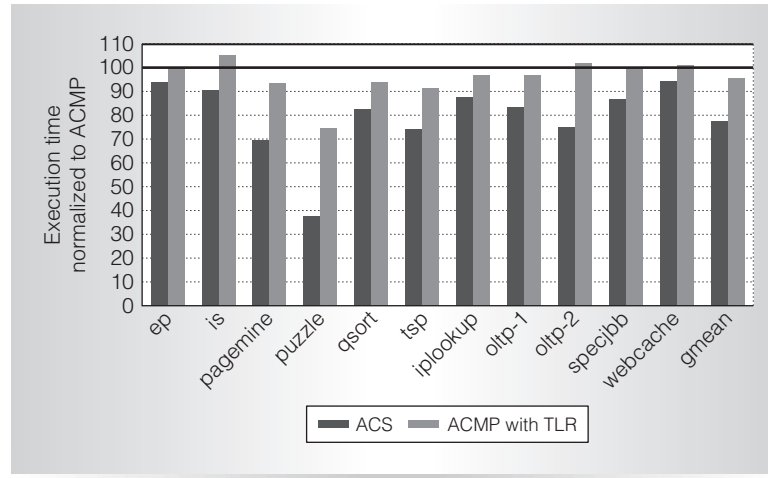

Figure 6. ACS vs. transactional lock removal (TLR) performance.

Alternate ACS implementations

We proposed a combined hardware/software implementation of ACS. Future research can develop other implementations of ACS for systems with different trade-offs and requirements. For example, we can implement ACS solely in software as a part of the runtime library. The software-only ACS requires no hardware support and can be used in today's systems. However, it has a higher overhead for sending CSCALLs to the remote core. Other implementations of ACS can be developed for SMP and multisocket systems.

The ACS mechanism provides a high-performance and simple-to-implement substrate that allows more productive and easier development of parallel programs—a key problem facing multicore systems and computer architecture today. As such, our proposal allows the software and hardware industries to make an easier transition to many-core engines. We believe that ACS will not only impact future CMP designs but also make parallel programming more accessible to the average programmer. MICRO

References

1. M. Annavaram, E. Grochowski, and J. Shen, ''Mitigating Amdahl's Law through EPI Throttling,'' *SIGARCH Computer Architecture News,* vol. 33, no. 2, 2005, pp. 298-309.
2. M. Hill and M. Marty, ''Amdahl's Law in the Multicore Era,'' *Computer,* vol. 41, no. 7, 2008, pp. 33-38.
3. T.Y. Morad et al., ''Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors,'' *IEEE Computer Architecture Letters,* vol. 5, no. 1, Jan. 2006, pp. 14-17.
4. M.A. Suleman et al., *ACMP: Balancing Hardware Efficiency and Programmer Efficiency,* HPS Technical Report, 2007.
5. G.M. Amdahl, ''Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,'' *Am. Federation of Information Processing Societies Conf. Proc.* (AFIPS), Thompson Books, 1967, pp. 483-485.
6. E.L. Lawler and D.E. Wood, ''Branch-and-Bound Methods: A Survey,'' *Operations Research,* vol. 14, no. 4, 1966, pp. 699-719.
7. E. Ïpek et al., ''Core Fusion: Accommodating Software Diversity in Chip Multiprocessors,'' *Proc. Ann. Int'l Symp. Computer Architecture* (ISCA 07), ACM Press, 2007, pp. 186-197.
8. M.A. Suleman et al., ''Accelerating Critical Section Execution with Asymmetric Multicore Architectures,'' *Proc. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 09), ACM Press, 2009, pp. 253-264.
9. R. Rajwar and J.R. Goodman, ''Transactional Lock-Free Execution of Lock-Based Programs,'' *Proc. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 5-17.
10. M. Herlihy and J. Moss, ''Transactional Memory: Architectural Support for Lock-Free Data Structures,'' *Proc. Ann. Int'l Symp. Computer Architecture* (ISCA 93), ACM Press, 1993, pp. 289-300.
11. R. Rajwar and J.R. Goodman, ''Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution,'' *Proc. Int'l Symp. Microarchitecture* (MICRO 01), IEEE CS Press, 2001, pp. 294-305.
12. J.F. Martínez and J. Torrellas, ''Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications,'' *Proc. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 02), ACM Press, 2002, pp. 18-29.

M. Aater Suleman is a PhD candidate of electrical and computer engineering at the University of Texas at Austin. His research interests include chip multiprocessor architectures and parallel programming. Suleman has a master's degree in electrical and computer engineering from the University of Texas at Austin. He is a member of the IEEE, ACM, and HKN.

Onur Mutlu is an assistant professor of electrical and computer engineering at Carnegie Mellon University. His research interests include computer architecture and systems. Mutlu has a PhD in electrical and computer engineering from the University of Texas at Austin. He is a member of the IEEE and ACM.

Moinuddin Qureshi is a research staff member at IBM Research. His research interest includes computer architecture, scalable memory system design, resilient computing, and analytical modeling of computer systems. Qureshi has a PhD in electrical and computer engineering from the University of Texas at Austin. He is a member of the IEEE.

Yale N. Patt is the Ernest Cockrell, Jr., Centennial Chair in Engineering at the University of Texas at Austin. His research interests include harnessing the expected fruits of future process technology into more effective microarchitectures for future microprocessors. Patt has a PhD in electrical engineering from Stanford University. He is a Fellow of both the IEEE and ACM.

Direct questions and comments to Aater Suleman, 1 University Station, C0803, Austin, TX 78712; suleman@hps.utexas.edu.