Opportunistic Competition Overhead Reduction for Expediting Critical Section in NoC based CMPs

Yuan Yao
KTH Royal Institute of Technology
Stockholm, Sweden
yuanyao@kth.se

Zhonghai Lu
KTH Royal Institute of Technology
Stockholm, Sweden
zhonghai@kth.se

ABSTRACT

With the degree of parallelism increasing, performance of multi-threaded shared variable applications is not only limited by serialized critical section execution, but also by the serialized competition overhead for threads to get access to critical section. As the number of concurrent threads grows, such competition overhead may exceed the time spent in critical section itself, and become the dominating factor limiting the performance of parallel applications.

In modern operating systems, queue spinlock, which comprises a low-overhead spinning phase and a high-overhead sleeping phase, is often used to lock critical sections. In the paper, we show that this advanced locking solution may create very high competition overhead for multithreaded applications executing in NoC-based CMPs. Then we propose a software-hardware cooperative mechanism that can opportunistically maximize the chance that a thread wins the critical section access in the low-overhead spinning phase, thereby reducing the competition overhead. At the OS primitives level, we monitor the remaining times of retry (RTR) in a thread's spinning phase, which reflects in how long the thread must enter into the high-overhead sleep mode. At the hardware level, we integrate the RTR information into the packets of locking requests, and let the NoC prioritize locking request packets according to the RTR information. The principle is that the smaller RTR a locking request packet carries, the higher priority it gets and thus quicker delivery.

We evaluate our opportunistic competition overhead reduction technique with cycle-accurate full-system simulations in GEM5 using PARSEC (11 programs) and SPEC OMP2012 (14 programs) benchmarks. Compared to the original queue spinlock implementation, experimental results show that our method can effectively increase the opportunity of threads entering the critical section in low-overhead spinning phase, reducing the competition overhead averagely by 39.9% (maximally by 61.8%) and accelerating the execution of the Region-of-Interest averagely by 14.4% (maximally by 24.5%) across all 25 benchmark programs.

Keywords– *Critical Section; CMP; NoC; OS*

1. INTRODUCTION

As the core count grows quickly, there is a greater potential to speed up the execution of parallel and concurrent programs. While this trend helps to linearly expedite the concurrent execution part, the ultimate application speedup is limited by the sequential execution part of programs, as reflected in the well-known Amdahl's law [11]. For multi-threaded shared variable applications, entering and executing *critical section* that contains shared data need to be synchronized and must be mutually exclusive, meaning that only one thread can get access to a critical section at a time. As shown in previous studies [21, 12, 13, 17, 22, 8, 23, 14], the time spent in accessing/synchronizing shared data among different threads is usually the most significant source of serialization in parallel applications. As such, reducing the time a thread spends in accessing critical section is thus of critical importance for increasing the performance of multi-threaded programs.

Modern state-of-the-art operating systems such as Linux 4.2 and Unix BSD 4.4 provide an advanced solution to lock critical sections by queue spinlock. This solution combines the advantages of both spinlock and queuing lock. It works as follows: If a thread cannot lock the critical section for the first try, instead of going into a sleep mode immediately, the queue spinlock will spin for a few more times. Only if the critical section has still not been obtained after a certain times of retry, the thread's critical section request is sent to a request queue, and the thread is put into the sleep mode until being woken up and securing the lock when the critical section is unlocked by the holding thread.

The queue spinlock is an architecture-oblivious solution for critical section access. In general it improves locking performance over spinlock or queuing lock alone. However, when running multi-threaded PARSEC and SPEC OMP2012 programs in NoC-based CMPs, we observe that it incurs considerable *competition overhead* (COH) for threads to gain access to critical sections, contributing significantly to the sequential execution time of applications. Compared to the critical section execution itself which incurs small time, it is the competition overhead which dominates the critical section performance in the CMPs. This is the motivation of the paper in developing an effective technique to reduce the competition overhead for multi-threaded applications running on NoC-based multi/many-cores.

Being different from previous works, our technique focuses on reducing the blocking time spent by threads competing with each other to enter into critical section, instead of accelerating the critical section execution itself. To achieve such a goal, we propose a software-hardware cooperative mechanism that can opportunistically maximize the chance that a thread enters the critical section in the low-overhead

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.33

spinning phase. At the OS software level, we modify the queue spinlock primitives by monitoring the *Remaining Times of Retry* (RTR) in a thread's low-overhead spinning phase. At the hardware level, we integrate the RTR information into the critical section locking requests of threads, and the routers prioritize locking requests from different threads according to their RTR values. The principle is that the smaller RTR a thread has, the higher priority its locking requests get, because the thread is about to enter into the sleep mode sooner. However, when a thread has already entered into the sleep mode, it should get the critical section at a later time, since waking the thread up will introduce considerable overhead and it is thus preferable to allow threads in the low-overhead spinning phase to win the critical section access.

The rest of the paper is organized as follows. Section 2 precisely defines the competition overhead and quantitatively illustrates its criticality in comparison with the critical section execution itself. Section 3 analyzes the competition overhead in queue spinlock and describes our design principle. Section 4 gives design details from both OS software and hardware perspectives. In Section 5 we report experiments and results. After discussing related work in Section 6, we finally conclude in Section 7.

2. BACKGROUND AND MOTIVATION

2.1 Critical section and competition overhead

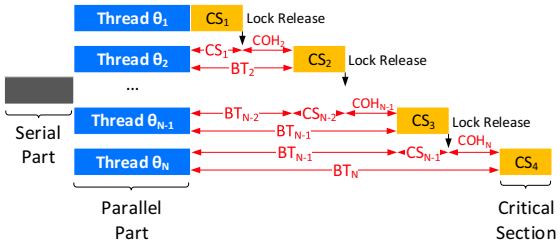


Figure 1: Execution of a parallel program with mutex synchronization illustrating thread block time (BT), critical section execution time (CS) and competition overhead (COH)

Figure 1 shows the typical execution of a multithreaded program. After an initial serial part, N threads start parallel executions and encounter a synchronization point where each thread competes to get access to a critical section. Without loss of generality, we assume that the threads enter the critical section in the order θ_1, θ_2, $\cdots$, θ_N. As the figure shows, the blocking time BT_i of thread i ($i \in [1,N]$) comprises other earlier entering threads' critical section completion time plus the COH, which is caused by either thread spin interval or thread wake-up time (see detailed illustrations in Section 3.2). For example, the blocking time BT_2 of thread θ_2 can be modeled as: $BT_2 = BT_1 + CS_1 + COH_2$, where $BT_1 = 0$ by our assumption. Recursively applying the model, one can get the blocking time BT_i for thread i as:

$$BT_i = BT_{i-1} + CS_{i-1} + COH_i = \sum_{j=1}^{i-1} CS_j + \sum_{j=1}^{i} COH_j \quad (1)$$

Equation 1 shows that the performance of a parallel program is not only limited by the critical section execution time ($\sum CS$), but also fundamentally by the competition overhead ($\sum COH$) when threads compete to get access to critical section.

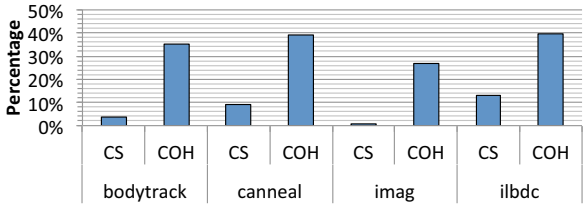


Figure 2: Criticality of competition overhead

2.2 Queue spinlock in operating systems

OS provides primitive functions to support critical section synchronization for concurrent programs. Traditionally, *queuing lock* and *spinlock* are two common methods.

In *queuing lock*, when a thread tries to lock a variable protecting a critical section and does not succeed, it will store its locking request into a lock queue and go to the sleep mode, allowing another thread to run on the same core. The thread will continue to sleep until being woken up when the critical section is unlocked by the holding thread. Although queuing lock can increase the core utilization, putting a thread to sleep and waking it up are both expensive operations, which incur considerable context switching overhead.

In *spinlock*, when a thread tries to lock a variable protecting a critical section and does not succeed, it will continuously re-try locking it usually after a short spin interval, until success. During the OS runtime quantum (usually in the order of hundreds of milliseconds) of the spinning thread, it will not allow another thread to use the core, leading to a waste of core processing time. However, with persistent polling on a lock variable, the spinlock can be quickly successful once the lock is released, and this is achieved with a small polling interval overhead and without large context switching overhead.

Due to their pros and cons, neither a programmer nor an OS kernel can know in advance whether queuing lock or spinlock will perform better on a particular hardware architecture. Most modern OSes such as Linux 4.2 and Unix BSD 4.4 adopt an advanced *queue spinlock* scheme, which combines the advantages of both methods. Queue spinlock behaves as a spinlock at first and then a queuing lock. If a thread cannot lock the critical section, it won't go to the sleep mode immediately, hoping that the critical section might get unlocked soon. Instead the queue spinlock will first behave exactly like a spinlock. Only if the critical section has still not been obtained after a certain times of retry, the thread's request is sent to a lock queue and the thread is put into the sleep mode.

2.3 Criticality of competition overhead

To examine the performance of queue spinlock used in parallel programs running on a NoC-based multicore, we experiment with a 64-core full-system simulation architecture in GEM5 (Section 5 describes our experimental setup.), with each application running alone in 64 concurrent threads. Figure 2 shows the percentage of program *ROI (Region of Interest) finish time* spent in critical sections (CS) and competi-

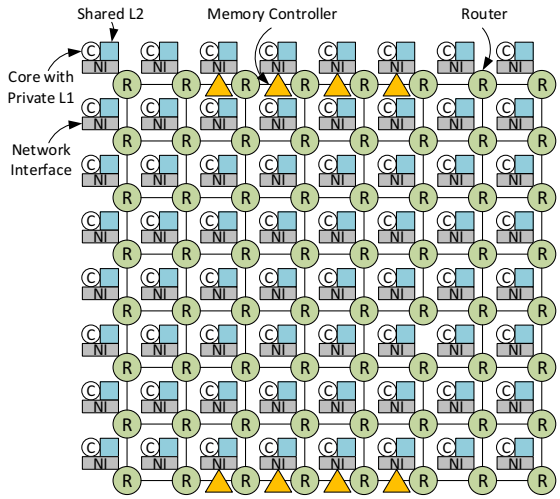


Figure 3: A 64-core 8-by-8 CMP architecture

tion overheads (COH) in two PARSEC benchmarks *bodytrack* (memory non-intensive) and *canneal* (memory intensive) and two SPEC OMP2012 benchmarks *imag* (memory non-intensive) and *ilbdc* (memory intensive). As the figure shows, in the two memory non-intensive benchmarks *bodytrack* and *imag*, critical section consumes about 3.8% and 1.1% of total application ROI finish time. However, the time spent in competition overhead is nearly 35.1% and 27.4% of total application ROI finish time, respectively. In memory intensive benchmarks *canneal* and *ilbdc*, the same phenomenon has been consistently observed. While the execution of critical section itself takes about 9.3% and 13.3%, the competition overhead consumes 38.6% and 39.7% of application ROI finish time, respectively. The results highlight that the competition overhead can be much larger than the critical section execution time and thus become the bottleneck of serialized execution.

3. OPPORTUNISTIC COH REDUCTION: PRINCIPLE AND EFFECT

3.1 Target CMP architecture

We first introduce the target CMP architecture we exploit through all experiments in the paper. Figure 3 shows a typical architecture for a 64-core CMP, where the NoC interconnects processor nodes (a CPU core with its private L1 cache), secondary on-chip shared L2 cache banks, and on-chip memory controllers together. Routers are organized in a popular mesh topology on which the XY routing algorithm is implemented to achieve simplicity and deadlock-free. There are eight memory controllers, which are connected to the middle four nodes on the top and bottom rows for architectural symmetry. To support cache coherency, a directory based MOESI cache coherence protocol is implemented. On the target CMP, once a core issues a memory request, the private L1 is first checked for the presence of the requested data. If this fails, depends on the location of the data block, the request is then forwarded to the local shared L2 or via the NoC to remote shared L2/DRAM.

3.2 Competition overhead in queue spinlock

Queue spinlock incurs different competition overhead in its spinning and sleeping phases. To illustrate the difference, Figure 4 shows its operation under cache coherence, in which we assume that thread 2's core is a sharer of the lock variable at time T_1.

Figure 4a shows that thread θ_2 is in the spinning phase. At time T_1, assume that two threads θ_1 and θ_2 compete to lock the same critical section and the lock variable is stored in the *Home* node. At time T_2, request from θ_1 reaches the home node before thread θ_2, and the critical section is thus granted to thread θ_1. An invalidation is then sent from the home node to thread θ_2 to notify it that its locking request is failed. Because thread θ_2 is in the spinning phase, it retries to lock the critical section at time T_5 and T_7. At time T_5, locking retry from θ_2 fails because θ_1 is still inside the critical section. However, at time T_7, the locking retry from θ_2 succeeds since the critical section is released by thread θ_1.

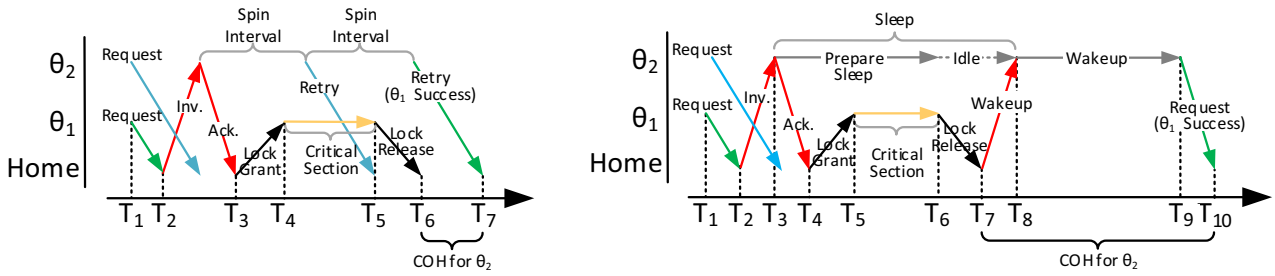
Figure 4b shows that thread θ_2 is in the sleeping phase. At time T_3, when thread θ_2 receives the invalidation from the home node, instead of continuously polling on the lock variable, θ_2 will directly go to the sleep mode. When the critical section is released at time T_7, the home node will wake up thread θ_2, which will then enter into the critical section after a successful locking request.

Comparing Figure 4a to Figure 4b, we can observe that, due to intensive retries with short interval and large wakeup overhead, competition overhead in the spinning phase can be much lower than that in the sleeping phase.

3.3 The basic principle

When locking requests are sent to the location where the lock variable is stored, the arrival order of the atomic locking requests directly affects the order of threads entering into the critical section. Since the locking requests experience contention for physical/virtual channels in the network, it is important to expedite those locking requests critical to the competition overhead. Thus it is desirable that the NoC delivers more critical requests faster.

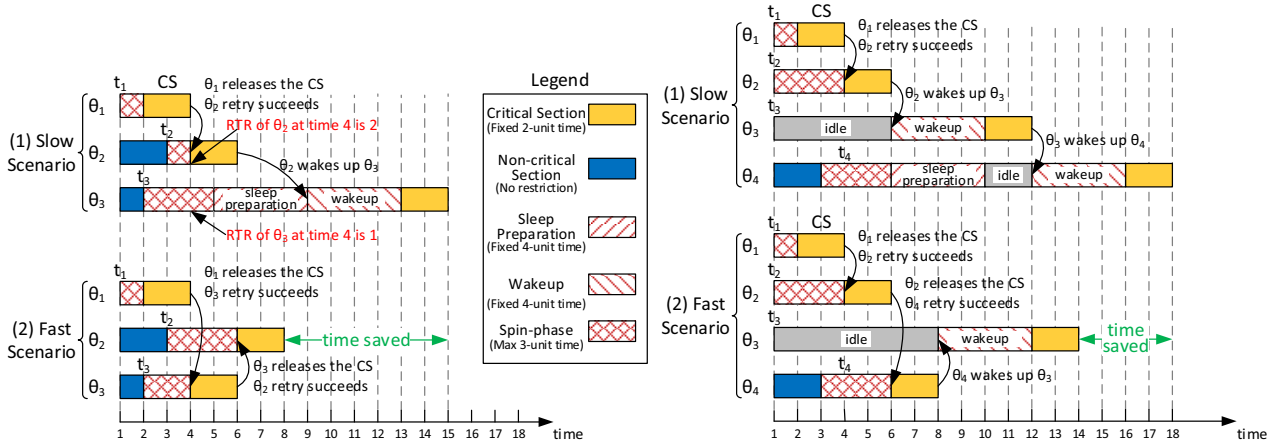
The basic principle of our technique contains two aspects. First, when a thread is in the low-overhead spinning phase, we maximize the opportunity that the thread gets the critical section. Second, when a thread already enters into the sleeping phase, we minimize the opportunity that the thread enters the critical section. The consideration is that an already slept thread should get the critical section at a later time, since waking it up will introduce considerable overhead and thus granting the access right to another competing thread in its spinning phase is preferred. To realize this scheme, our technique involves both software-level and hardware-level mechanisms. In the software level, we check the remaining times of retry (RTR) in a thread's spinning phase. This is done by modifying the implementation of queue spinlock function of the OSes. In the hardware level, we integrate the RTR information into the network packets of threads' locking requests, and let the NoC prioritize these requests from different threads based on the RTR they carry. Because RTR reflects in how long a thread must enter into the sleep mode, the smaller RTR a request packet carries, the higher priority it gets. Finally, the wakeup requests sent from the home



(a) Low competition overhead of thread θ_2

(b) High competition overhead of thread θ_2

Figure 4: Operation of queue spinlock under cache coherence, where in Figure 4a thread θ_2 is in the low-overhead spinning phase, and in Figure 4b the high-overhead sleeping phase.



(a) Granting lower RTR thread earlier

(b) Granting sleeping thread later

Figure 5: Locking success scenarios with different competition overheads. (a) Granting a thread with lower RTR earlier leads to a fast scenario; (b) Granting a sleeping thread later leads to a fast scenario.

node to sleeping threads are assigned with lowest priority.

3.4 Effect on competition overhead reduction

Figure 5 shows the effects of our opportunistic competition overhead reduction technique. Assume that each critical section takes two time units to finish, each thread spins for three times in the spinning phase (with each retry interval 1 time unit), and both the sleep-preparation and wake-up procedures of a thread consume 4 time units each.

As Figure 5a shows, three threads, θ_1, θ_2 and θ_3 compete to lock on a critical section. After one time unit in the spinning phase, θ_1 gets the lock and enters into the critical section, which lasts for 2 time units. At time 4, thread θ_1 releases the critical section, and θ_2 and θ_3 compete to get the lock. As shown in the figure, because θ_2 has retried once in the spinning phase, its RTR value is 2 (3-1=2) at time 4. Similarly, because thread θ_3 has spent two time units in the spinning phase, its RTR value is 1 (3-2=1). At time 4, two scenarios can happen. In the *slow* scenario, as shown in Figure 5a(1), thread θ_2 gets the critical section before thread θ_3, which will force θ_3 to enter into the sleeping phase. However, θ_3 does not actually sleep for any time since the critical section is released at time 6, and θ_3 is still performing the preparation procedure for putting thread into the sleep mode. Thus, θ_3 wakes up right away at time 9 because the critical section has been released. Hence, most of the application run time is wasted on the procedures of sleep-preparation and wake-up. In the *fast* scenario, as shown in Figure 5a(2), at time 4, the critical section is granted to thread θ_3 instead of thread θ_2, based on the fact that θ_3 has a smaller RTR value than θ_2. Thus, no thread has entered into the sleep mode, and the competition overhead is reduced.

Figure 5b shows that our technique can also be beneficial by having an already slept thread get the critical section at a later time. In the figure, four threads, θ_1, θ_2, θ_3 and θ_4 compete to get access to a critical section, and θ_1 first gets the access right. In both cases, we suppose that θ_3 has been in the sleep mode before time 1, and θ_2 will get the critical section when θ_1 releases the lock at time 4. In our mechanism, since θ_3 has already entered into the sleep mode when θ_2 releases the critical section, we give higher priority to the locking request from θ_4 at time 6, which will achieve the *fast* scenario in Figure 5b(2). Otherwise, prioritizing θ_3 over θ_4 when thread θ_2 releases the critical section will cause thread θ_4 to go to the sleeping phase, thus incurring additional larger overhead, as shown in the *slow* scenario in Figure 5b(1).

In essence, our technique aims to maximize the likelihood that the fast scenarios in Figure 5a and Figure 5b happen so as to effectively shrink the competition overhead.

Algorithm 1 Enhanced `q_spinlock_lock` primitive

```
1: function q_spinlock_lock(shared_lock *lock) {
2:     int c = 0;
3:     /* Spinning phase begins */
4:     for (i=0; i < MAX_SPIN_COUNT; i++) do {  /* Spinning phase */
5:         int RTR = MAX_SPIN_COUNT - i;
6:         write_local_reg(RTR, get_thread_PCB()->PROG);
7:         c = atomic_try_lock(lock);                    /* Atomic locking */
8:         if (!c) return 0;              /* Atomic locking succeeds, return */
9:         cpu_relax();          /* Otherwise, delay and retry the locking */
10:    }
11:    /* Sleeping phase begins, call Linux kernel function sys_futex to
       send the locking request to the lock queue, and put the thread into the
       sleep mode */
12:    sys_futex(lock, FUTEX_WAIT);
13: }
```

Algorithm 2 Enhanced `q_spinlock_unlock` primitive

```
1: function q_spinlock_unlock(shared_lock *lock) {
2:     atomic_release(lock);
3:     get_thread_PCB()->PROG++;
4:     write_local_reg(get_thread_PCB()->PROG);
5:     /* Call Linux kernel function sys_futex to send wakeup request to
       the home node, wake up a thread in the sleep mode */
6:     sys_futex(lock, FUTEX_WAKE);
7: }
```

4. DESIGN DETAILS

4.1 Software support in OS

To support our scheme, we need to pass the RTR value from application to hardware. This is done by modifying the OS's queue spinlock implementation. In our case, we realize it in a state-of-the-art Linux OS, version 4.2.

Enhanced queue spinlock implementation. Algorithms 1 and 2 illustrate the principle implementations of the queue spinlock's lock (denoted `q_spinlock_lock`) and unlock (denoted `q_spinlock_unlock`) primitives in Linux 4.2, together with our enhancements highlighted with bold text. Algorithm 1 shows that the `q_spinlock_lock` function consists of a spinning phase and a sleeping phase. In the spinning phase, the queue spinlock spins for a period of time that is determined by the value `MAX_SPIN_COUNT`. During the spin phase, the queue spinlock tries to get the exclusive access to the shared lock variable through an `atomic_try_lock` function, which can be implemented by a set of `test_and_set` like primitives in the hardware. If a thread fails to get the critical section within `MAX_SPIN_COUNT` retries, the thread will enter into the sleeping phase after registering its locking request to the lock queue via the Linux system call `sys_futex`.

As shown in Line 5 of Algorithm 1, we utilize the RTR value to monitor the remaining times of retry in a queue spinlock's spinning phase, which can be dynamically computed by subtracting the already passed times of retry from the total retry times. The RTR value together with a thread's progress information (PROG, which is used to avoid starvation for low-priority requests and associated threads, as will be explained later) are then written into special local registers (shown in Line 6 of Algorithm 1) in the CPU that the thread runs on. Further, the CPU informs the network interface (NI) to assign the RTR value and the progress information to the network packets of atomic *locking requests*, before they are sent to the NoC. In the sleeping phase, the `sys_futex(lock, FUTEX_WAIT)` system call provides a method for a thread to register its locking request into the lock queue in the home node where the lock variable locates. Then the thread goes into the sleep mode and waits for wakeup until the lock variable is released.

```
1: function pthread_mutex_lock(shared_lock *lock) {
2:     ...
3:     q_spinlock_lock(lock);
4:     ...
5: }
```

```
1: function pthread_mutex_unlock(shared_lock *lock) {
2:     ...
3:     q_spinlock_unlock(lock);
4:     ...
5: }
```

Figure 6: Modified pthread mutex lock/unlock functions.

Once a thread finishes the critical section, the `q_spinlock_unlock` function is called to release the lock variable and wake up one of the sleeping threads in two steps. At first, an atomic_release function is called to release the shared lock variable. Then the `sys_futex(lock, FUTEX_WAKE)` function is called to send a `WAKE_UP` request to the request queue in the home node. In our mechanism, we give the lowest priority to the `WAKE_UP` request to implement the "Wakeup Request Last" rule in Table 1, Section 4.2.

Starvation avoidance. Although prioritizing packets with small RTR can increase the opportunity of a thread entering into critical section in the low-overhead spin phase, such prioritization may however cause starvation to higher RTR value threads. To avoid the starvation, our mechanism tracks the progress of threads, and packets belonging to slow progress threads are prioritized over packets from fast progress ones. This ensures that locking requests from low progress threads always get served first in the NoC routers.

As shown in Algorithm 1, our queue spinlock tracks the progress of a thread and records its value to a local register in the core, which will further be integrated into the locking request packet header for routers to do priority-based scheduling. To this end, we add an additional field, denoted `PROG`, in the Linux thread handler (called Process Control Block, PCB) to dynamically track the progress of a thread. There are plenty of ways to determine per-thread progress information. For our purpose, we use the progress value to specify the times that a thread has successfully entered into critical sections. During thread initialization, the progress value `PROG` of all threads is set to 0. Whenever a thread has successfully finished the jobs inside a critical section, `PROG` is increased by 1 when the critical section is released (as shown in Line 3 of Algorithm 2).

Compatibility with existing mutex synchronization. It is worth noting that the new implementation of the queue spinlock in the OS remains the underlying technique to implement user-space synchronization primitives such as the *pthread mutex* [16] or *OpenMP critical directives* [4]. Figure 6 shows that the proposed queue spinlock can be used to implement `pthread_mutex_lock` and `pthread_mutex_unlock` functions in the pthread library. More importantly, there is *no need* to modify the source code of programs that are already written with the pthreads or OpenMP libraries to use our technique. Therefore our enhanced queue spinlock is transparent to application programs.

Table 1: Priority rules of the NoC routers

Rules in priority-based NoC routers
1: **Slow Progress First.** Packets from a slower progress thread are prioritized over packets from a faster progress thread.
2: **Locking Request Packet First.** Among threads with the same progress, both locking request packets from the spinning phase and wakeup requests from the sleeping phase are prioritized over normal data and cache coherence packets.
3: **Least RTR First.** Among threads with the same progress, locking request packets that have smaller RTR values are prioritized over locking request packets that have larger RTR values.
4: **Wakeup Request Last.** Locking requests from the spinning phase are prioritized over wakeup requests from the sleeping phase.

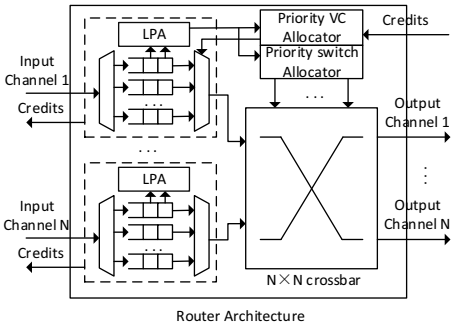


Figure 7: Priority-based router architecture

4.2 Hardware support in NoC

Router micro-architecture. In hardware, after obtaining the RTR and progress information from the OS, routers in the NoC prioritize locking request packets with smaller RTR over locking request packets with larger RTR to maximize the opportunity that a thread wins the critical section access right during the spinning phase. To achieve this, we develop a router with priority-based virtual channel (VC) and switch allocation, as shown in Figure 7. Our baseline router is a 2-stage pipelined speculative router [19] in which the first stage performs Route Computation (RC), VC Allocation (VA), and Switch Allocation (SA) in parallel and the second stage is Switch Traversal (ST). Further, Table 1 summarizes the prioritization rules that are applied to both VC and switch allocation for packet scheduling.

Additional head fields in locking & wakeup request packets. To integrate the RTR value as well as the thread progress information into the request packet, we add three additional fields, *priority check bit*, *priority bits*, and *progress bits*, to the request packet header. As shown in Figure 8, before a locking request packet is sent into the network, the network interface (NI) integrates these additional fields into the request packet header. The priority check bit then differentiates a lock/wakeup request packet from normal data and coherence packets. In our scheme, we first prioritize packets from slow progress threads over fast progress ones to avoid starvation. Then, among threads with the same progress, we prioritize both lock and wakeup request packets over normal data and cache coherence packets. Further, we prioritize locking packets with small RTR values over packets with larger RTR values, and we set the priority of wakeup requests in the sleeping phase to the lowest level to delay critical section grants to sleeping threads. The packets prioritization rules are summarized in Table 1). As shown in Figure 8, the priority bits reflect the priority level of a packet based on the RTR value. To minimize the priority comparison overhead in routers, we use one-hot coding to represent the priority of a packet as in [6], with one priority level mapped to one bit in the priority bits, and the lowest priority level of the wakeup request packet to one additional bit. However, such one-priority-level one-bit mapping may introduce considerable hardware overhead. In state-of-the-art Linux 4.2 environment, the maximum spinning times of a queue spinlock is 128[1], thus 129 bits (128 RTR priority bits + 1 additional bit) are required for the priority bits in one-hot coding. To achieve low-overhead design, instead of mapping one RTR value to one priority bit (thus K RTR values K bits), we evenly divide the maximum spin time-span into 8 segments, with each segment containing 16 spin times. Further, we map one such spin time segment into one priority bit, thus only 9 bits (8 RTR priority bits + 1 additional bit) are needed to cover all the priority levels. During packetization, only when the RTR value shifts to another segment, the priority of the outgoing locking request packets is updated. The same principle is applied to assign the progress bits. As shown in [6], by adopting the one-hot priority mapping mechanism, there is no need to use comparators in routers to achieve priority-based arbitration, ensuring limited hardware overhead. For wakeup request packets, they have always the lowest priority indicated by the additional bit.

Priority-based packet scheduling. Figure 8 shows the effects of the prioritization rules in Table 1 on the packet scheduling order in routers. For narration convenience, we use $R^i j$ to denote the locking request and W^i the wake-up request from a thread, where i denotes the progress value (PROG) of the issuing thread, and j the RTR value carried by the packet header. Assume that threads θ_1, θ_2 and θ_3 with progress value a send three locking requests with RTR values of 1, 2, and 3 to the home node of the lock variable, respectively. Threads η_1, η_2 and η_3 with progress value b also send three locking requests with RTR values of 1, 2, and 3 to the lock variable's home node. The locations of the issuing threads and the home node, and the routing path of the request packets are shown in the example NoC in Figure 8. Further assume that $a < b$, which implies that threads η_1, η_2 and η_3 have progressed faster than threads θ_1, θ_2 and θ_3. Suppose that the previous critical section has been held by thread θ_p, which has finished the critical section access and advances to progress b. After releasing the critical section, thread θ_p sends a `wake-up` request (as denoted by W^b) to the lock queue in the home node to wake up one of the sleeping threads. Figure 8 illustrates the arrival and departure order of the above requests at the highlighted router R. In comparison with two possible round-robin (the default scheduling policy in the baseline router) scheduling policy results starting from the east VC and the west VC, Figure 8 gives an example of our priority-based packet scheduling policy, where the request packets of smaller RTR are prioritized over request packet with larger RTR, and the wake-up request of a thread gets the lowest priority.

Fairness of the priority-based scheduling scheme. As shown in Figure 8, although our priority-based packet schedul-

[1]Based on the Linux 4.2 kernel from www.linux.com

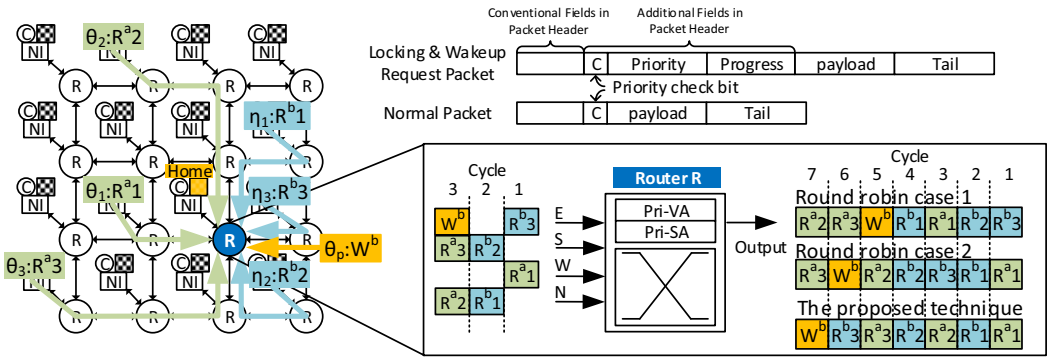


Figure 8: Additional packet fields and illustrative comparison of our priority-based scheduling and round-robin scheduling.

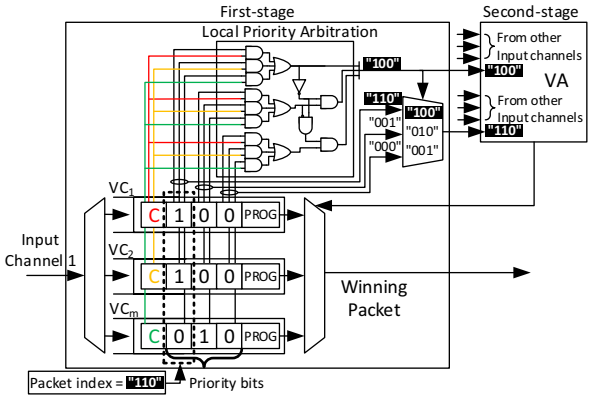


Figure 9: Hardware implementation of the local priority arbitration (LPA) logic, where C denotes the priority check bit

ing scheme is in favor of packets from slow-progress threads, this priority is however not dominating the packet service in the router, because within the same VC packets are served in FIFO order. As the figure shows, the three packets (in green color, with superscript "a") from slow progress threads (θ_1, θ_2, θ_3) indeed do not block the three packets (in blue color, with superscript "b") from fast progress threads (η_1, η_2, η_3). Since the priority-based packet scheduling scheme is built on top of FIFO fairness, our mechanism achieves a balanced treatment on priority and fairness.

Design and overhead of low-cost LPA. To achieve low overhead in the priority-based VC and switch allocation, we adopt a 2-stage arbitration approach, as sketched in Figure 7 and detailed in 9. In the first stage, a *Local Priority Arbiter* (LPA) logic is added to each input channel to select the highest priority packets in the local VCs. As shown in Figure 9, the per-packet priority check bit acts as the *enable* signal for checking priority bits in the arbitration process. Only when a packet has the priority check bit of 1 (which means a locking or wake-up request packet), its priority bits are compared. As the figure shows, we exploit one-hot coding to implement different priorities. Each LPA generates two outputs to the second stage where the local highest priority packets are further compared with each other to win the global VC/switch allocation. The first output is the highest priority level in the local VCs, and the second output is an index of the packet(s) with the highest priority.

We exemplify the LPA operation with Figure 9. For narration convenience, we assume three priority levels for locking & wake-up requests, and thus three priority bits in each request packet. Assume that each input channel contains m VCs (here $m = 3$). Three packets with priority bits "100" (highest priority), "100" (highest priority) and "010" (middle priority), arrive at the VCs of input channel 1. The first output of the LPA is thus "100", indicating that the highest priority from input channel 1 is '100". Further, the index of the three packets in input channel 1 are output in the bit combination of "110", indicating the exact locations of the two packets with the highest priority. The outputs of LPA are then input to the second step for global VA and SA. After the allocation, the packet index is used to select the winning packet. If there are more than one packet with the highest priority, a random selection is made.

5. EXPERIMENTS

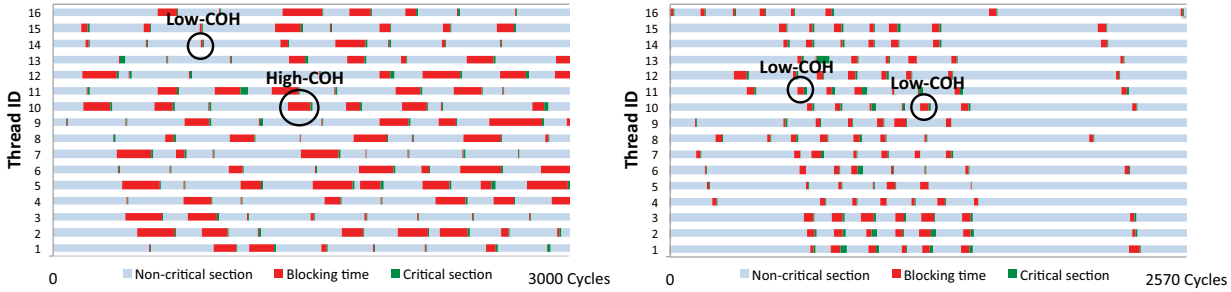
5.1 Methodology

We evaluate our Opportunistic Competition Overhead Reduction (**OCOR**) technique with timing-detailed full-system simulations using both PARSEC (11 programs) [2] and SPEC OMP2012 (14 programs) [18] benchmarks. We implement and integrate our design in GEM5 [3], on which the OS kernel (Linux 4.2) and the embedded network GARNET [1] are enhanced according to our technique. The details of the simulation platform setup are shown in Table 2. As the table shows, our CMP uses the Alpha based out-of-order cores, which supports Alpha-ev67 ISA.

At the software level, we enhance the default queue spinlock (`mutex`) implementation of Linux 4.2, as illustrated in Algorithm 1 and 2 in Section 4.1. For this modification to be visible to user-space, we further revise the corresponding critical section locking/unlocking functions in *pthread* and *OpenMP* libraries, which are implemented on top of the Linux's `mutex`. This involves integrating the enhanced queue spinlock in Section 4.1 to the `pthread_mutex_lock`, `pthread_mutex_unlock` functions in the *pthread* library and the `#pragma omp critical` directive in the *OpenMP* library. At the hardware-level, we modify the default VA and SA (class VCallocator and SWallocator) of the GARNET VC router in GEM5 to support our priority based arbitration.

Table 2: Simulation platform configuration

Item	Amount	Description
Processor	64 cores	Alpha based 2.0 GHz out-of-order cores. 32-entry instruction queue, 64-entry load queue, 64-entry store queue, 128-entry reorder buffer (ROB).
L1-Cache	64 banks	Private, 32 KB per-core, 4-way set associative, 128 B block size, 2-cycles latency, split I/D caches, 32 MSHRs.
L2-Cache	64 banks	Chip-wide shared, 1 MB per-bank, 16-way set associative, 128 B block size, 6-cycles latency, 32 MSHRs.
Memory	8 ranks	4 GB DRAM, 512 MB per-rank, up to 16 outstanding requests for each processor, 8 memory controllers.
NoC	64 nodes	8×8 mesh network. Each node consists of 1 router, 1 network interface (NI), 1 core, 1 private L1 cache, and 1 shared L2 cache. X-Y dimensional routing. Router is 2-stage pipelined, 6 VCs per port, 4 flits per VC. 128-bit data path. Directory based MOESI cache coherence protocol. One cache block consists of 1 packet, which consists of 8 flits. One coherence control message consists of 1 single-flit packet.
priority	-	128 retry times in the spinning phase. 9 priority levels, 8 higher levels for requests in the spinning phase, each priority level is mapped to 16 retry times; 1 lowest priority level for wakeup requests.
Benchmark	-	PARSEC and SPEC OMP2012. To scale the benchmarks well to 64 cores, we choose large input sets for all programs in PARSEC. For data validity, we only report results obtained from the multi-threaded execution phase called Region-of-Interest (ROI) in the experiments.



(a) *bodytrack* execution without OCOR (b) *bodytrack* execution with OCOR

Figure 10: Comparison of the execution profile of benchmark *bodytrack* without and with our technique. In the figure, High/Low-COH denotes high/low competition overhead featured thread blocking time.

Unless otherwise specified, each PARSEC and OMP2012 program runs on 64 cores with one core running one thread. Moreover, eight priority levels are used for locking requests, and one level for wakeup requests. We compare results obtained without OCOR (**Original**) and with OCOR. The platform without OCOR is the baseline which uses the original Linux queue spinlock without any modification in all programs.

5.2 Experimental results

5.2.1 *Competition overhead reduction*

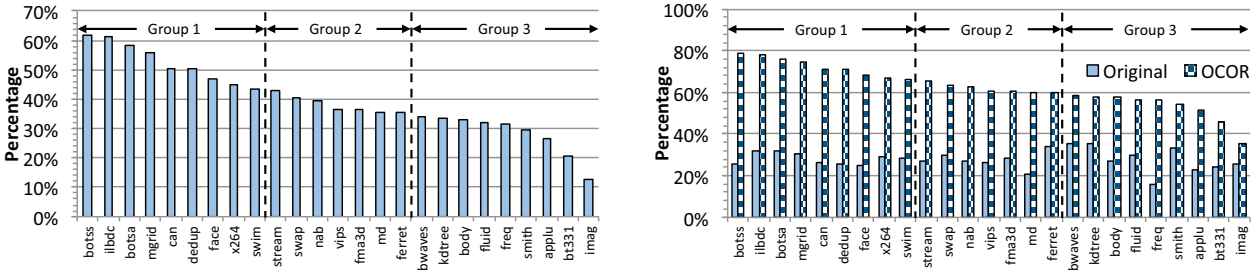
Figure 10 illustrates the execution profile of a PARSEC benchmark *bodytrack*. Due to space limitation and clarity concern, we select to report the first 3000 cycles of execution time of the first 16 threads in Figure 10. The figure divides the thread execution time into three parts: 1) parallel execution part, where threads perform concurrent computation tasks, 2) thread blocking time which includes competition overhead in the locking process (see Equation 1), and 3) critical section execution, where threads execute the critical section code. Based on the figure, we can make the following observations. First, each thread spends very little time in the critical section. This is due to the fact that parallel programs optimize the critical section code to small size in order to shorten the serial execution of the program. In the above threads, the time spent in critical section is about 5.2% of total application execution time. Second, in terms of the competition overhead each thread spends to wait for entering into the critical section, the time is much longer than the execution time of the critical section itself. As Figure 10a shows, without OCOR, high competition overhead featured thread blocking time consumes 33.2% of the total application execution time. However, with OCOR, competition overhead among threads is significantly reduced which decreases the thread blocking time to 18.7%, accelerating the execution of the *bodytrack* segment code by about 14.5% (from 3000 to 2570 cycles).

Figure 11a shows the COH reduction across all 25 benchmarks, in which we sort the percentage of improvement from most to least. After applying the proposed technique, competition overhead is constantly reduced across all benchmarks. As shown in the figure, benchmark *botss* achieves the maximum COH improvement (61.8%) and *imag* the minimum (12.5%). The average COH improvement is 40.4% for PARSEC and 39.3% for OMP2012 programs.

Figure 11b further illustrates that the proposed technique increases the chance of a thread getting the critical section in the low-overhead spinning phase. For consistency purpose, the benchmark order in Figure 11b remains the same as in Figure 11a. The maximum improvement is also achieved by benchmark *botss*, where without OCOR, only 25.7% of critical sections are obtained in the low-overhead spinning phase. With OCOR, the percentage increases to 79.2%, where an improvement of 53.5% is achieved. Benchmark *imag* achieves the minimum improvement which is 9.5%. Overall, the average improvement across all benchmarks is 33.1%.

5.2.2 *Application-dependent improvement*

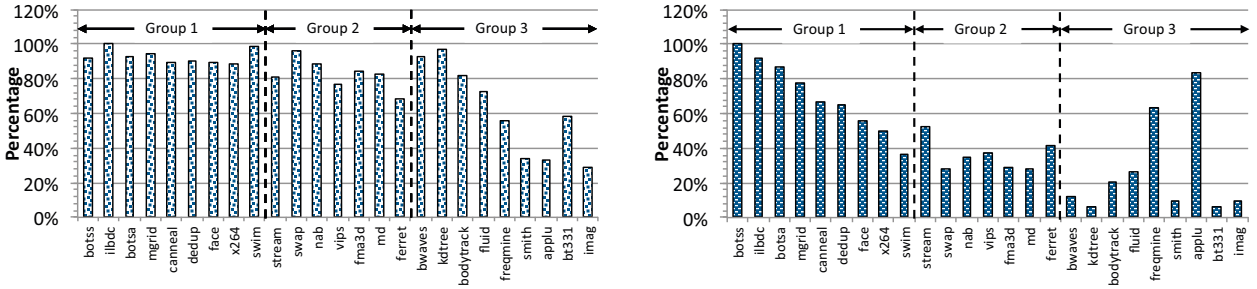
The improvements in competition overhead and the chance that a thread gets the critical section in the low-overhead spinning phase depends heavily on a benchmark's characteristics. Figure 12 shows the characteristics of the PARSEC and SPEC OMP2012 benchmarks in terms of *critical section*



(a) Competition overhead improvement with OCOR

(b) Fraction of critical section accesses in the spinning phase

Figure 11: Improvements in COH and the fraction of threads entering critical section in the low-overhead spinning phase.



(a) Relative critical section access rate

(b) Relative network utilization

Figure 12: Characteristics of all benchmarks in which we normalize the maximum value in each case to 100%.

access rate and *network utilization*. We keep the order of the benchmarks in both Figure 12a and Figure 12b the same as in Figure 11a. Figure 12a shows the normalized critical section access rate, which is measured in the average critical section locking packet injection rate, and reflects how often a thread needs to access a critical section. As the figure shows, benchmark *ilbdc* has the maximum critical section access rate and *imag* the minimum. For comparison convenience, we normalized the value obtained from the maximum one *ilbdc* to 100%. Figure 12b shows the normalized network utilization of all benchmarks, which is measured in the average network packet injection rate, and reflects an application's network communication demand. As the figure shows, benchmark *botss* achieves the maximum network utilization and *ketree* the minimum. For comparison convenience, we normalized the value obtained from *botss* to 100%.

From Figure 12, we can observe that the higher critical section access rate and the higher network utilization of a benchmark, the higher competition overhead reduction is achieved by our technique. This is in accordance to the intuition that applications with higher network packet injection rate and higher critical section access rate tend to overload the NoC heavier, creating more severe competition among threads and increasing the occurrence likelihood of the slow scenarios in Figure 5. Therefore it offers greater opportunity for our technique to achieve larger improvement. For example, in benchmarks *botss*, *ilbdc* and *botsa*, both the network injection rate and critical section access rate are high, and our technique achieves 61.8%, 61.5%, and 58.2% competition overhead reduction, respectively. In benchmark *img*,

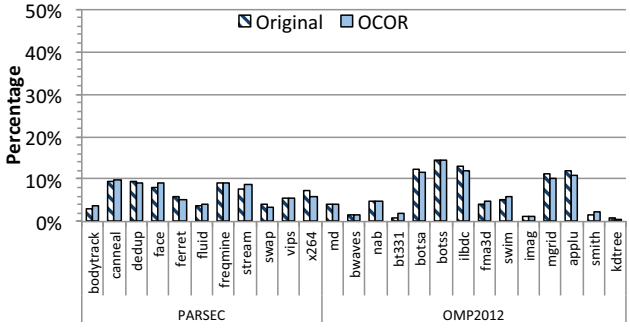
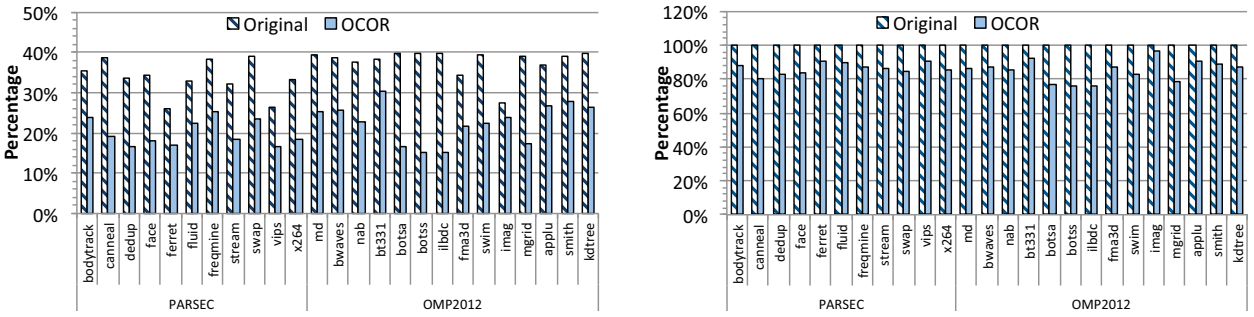


Figure 13: Relative critical section execution time

although both the network utilization and critical section access rate are low, our technique still achieves 12.6% reduction in competition overhead.

To analyze the effect of critical section access rate and network utilization on the COH improvement, we divide the benchmarks in Figure 11a and Figure 12 into 3 groups. In Group 1, critical section access rates across different benchmarks remain quite stable, but network utilization decreases. The COH improvement thus decreases accordingly, as shown in Figure 11a. In Group 2, both critical section access rate and network utilization stay relatively stable, the improvements of COH show small variations. The most interesting cases occur in Group 3, where both critical section access rate and network utilization fluctuate largely. As Figure 12 shows, benchmarks (for example *bodytrack* and *kdtree*) that have higher critical section access rate and lower network utilization achieve more COH improvements than bench-



(a) Percentage of COH in ROI finish time

(b) Comparison in ROI finish time

Figure 14: Improvement in COH leading to reducing ROI finish time.

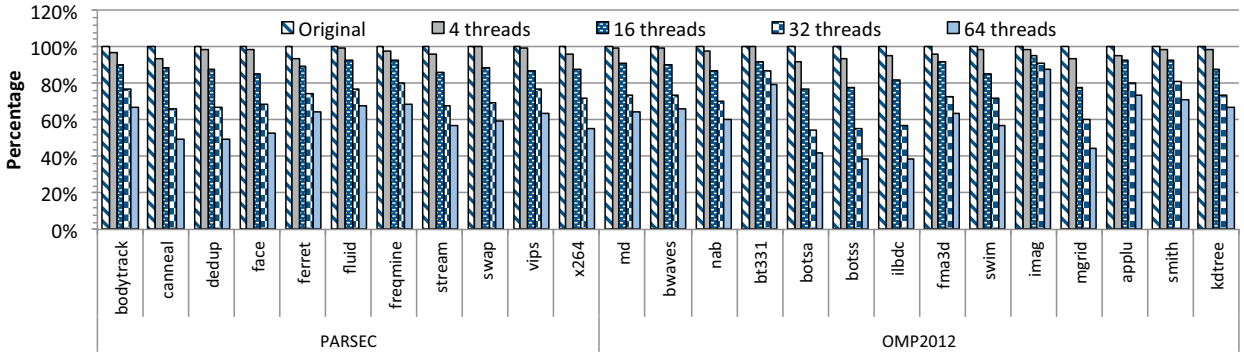


Figure 15: Comparison of the percentage of time spent in COH for all benchmarks running in 4, 16, 32, 64 threads.

marks (*freqmine* and *applu*) that have lower critical section access rate and higher network utilization. This shows that critical section access rate exerts a relatively heavier impact on the COH reduction. This makes sense because, only when a thread issues more critical section locking requests into the network, the network utilization starts to show its effects on COH reduction. Otherwise, even if the network is overloaded, there are low competitions among threads to get access to critical section, and thus there is little space in the COH improvement.

5.2.3 *ROI finish time reduction*

We first illustrate the influence of OCOR on critical section execution through Figure 13, in which we report the percentage of ROI execution time spent in critical sections across different benchmarks. As can be observed in the figure, instead of reducing the execution of critical section itself, our OCOR aims to reduce the competition overhead in which threads compete to enter into critical sections. Thus, compared with the original design, applying OCOR negligibly effects critical section execution time across all benchmark programs.

Figure 14a depicts the improvements in percentage of COH in ROI finish time, and Figure 14b reports the improvements in ROI finish time across all benchmarks. As shown in Figure 14a, the fraction of COH in the ROI finish time differs in different benchmarks. Benchmark *kdtree* in SPEC OMP2012 has the most competition overhead fraction which is 39.8% of the program's ROI finish time, and benchmark *ferret* in PARSEC has the least fraction, which is 26.1%. The fraction of competition overhead in the ROI execution directly influences the effects of our technique on the overall ROI finish time improvement. The higher fraction of competition overhead, the larger improvement achieved. As Figure 14b shows, in benchmark *ilbdc*, the maximum improvement in ROI finish time achieved is about 24.5% (61.5% COH reduction, where the COH is 39.7% of the ROI execution time), and in benchmark *imag*, the least improvement is achieved, which is about 3.4% (12.5% COH reduction, where the COH is 27.4% of the ROI execution time). Further, the average ROI finish time improvement is 13.7% for PARSEC and 15.1% for OMP2012 benchmarks.

5.2.4 *Scalability evaluation*

To evaluate scalability, Figure 15 shows the results of our technique applied to 4, 16, 32, 64 threads running on 4, 16, 32, 64 cores, respectively. In all the cases, we normalize the competition overhead without OCOR to 100%. As shown in the figure, our mechanism constantly reduces the competition overhead across all benchmarks. Further, the more threads spawned, the larger COH reduction.

In the case of 4 threads, improvements of the proposed mechanism are limited across all benchmarks. This is because in the fewer threads situation, competition among threads is low, thus little time has been spent in the competition overhead. However, with the growing number of threads, competition among threads also increase, thus the more improvement our technique achieves. As shown in Figure 15, the scalable improvement of our technique is also affected by the characteristics of benchmarks. For example, in low network utilization and low critical section access rate benchmark such as *imag*, the improvements in 4, 16, 32, 64 threads

are 1.7%, 4.9%, 9.2% and 12.5%, respectively. However, in high network utilization and high critical section access rate benchmark such as *ilbdc*, the corresponding improvements are 5.0%, 18.3%, 43.5%, and 61.5%.

5.2.5 *Sensitivity to the number of priority levels*

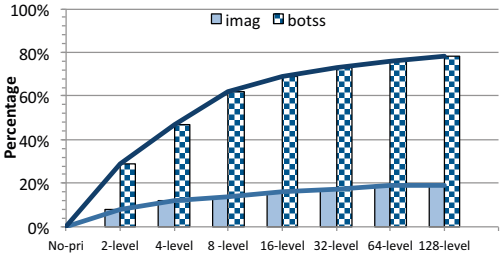


Figure 16: COH improvements with different priority levels for two extreme cases

The number of priority levels for locking request packets is a design parameter, which determines the prioritization granularity of our technique. In general, more priority levels will enable finer scheduling accuracy and increase the performance. However, adding more priority levels will also add more hardware overhead. Therefore a proper priority level should be chosen. To investigate the impact of different priority levels on the COH improvement, we look into the two extremes, one program *botss* showing best improvement and the other program *imag* the minimum improvement. Figure 16 shows their COH improvements as the number of priority levels grows. As can be observed, increasing the priority level increases the improvement, but the improvement acceleration decreases, meaning non-proportional benefit increase until saturation. Program benefiting more from our technique has a larger improvement potential. This figure also justifies the selection of the eight priority levels as our default architecture configuration.

5.2.6 *Summary of main results*

Table 3: Result summary for the 64-thread case

PARSEC	Characteristics		COH	ROI	OMP2012	Characteristics		COH	ROI
	CS Rate	Net. Util.	Impro.	Impro.		CS Rate	Net. Util.	Impro.	Impro.
ferret	low	low	35.5%	9.3%	imag	low	low	12.5%	3.4%
vips	high	low	36.7%	9.7%	bt331	low	low	20.8%	8.0%
fluid	low	low	32.0%	10.5%	applu	low	high	26.8%	9.9%
body	high	low	33.0%	11.7%	smith	low	low	29.4%	11.5%
freq	low	high	31.6%	13.1%	fma3d	high	low	36.5%	12.5%
stream	high	high	43.1%	13.9%	bwaves	high	low	33.9%	13.2%
x264	high	high	44.8%	14.9%	kdtree	high	low	33.4%	13.3%
swap	high	low	40.3%	15.7%	md	high	low	35.6%	14.1%
face	high	high	46.9%	16.0%	nab	high	low	39.7%	14.9%
dedup	high	high	50.3%	16.9%	swim	high	low	43.8%	17.1%
can	high	high	50.5%	19.5%	mgrid	high	high	55.7%	21.8%
					botsa	high	high	58.1%	23.2%
					botss	high	high	61.8%	24.4%
					ilbdc	high	high	61.5%	24.5%
PARSEC average			**40.4%**	**13.7%**	**OMP2012 average**			**39.3%**	**15.1%**
Overall avg. improvement in COH is 39.9%, in ROI finish time 14.4%.									

We summarize the improvements in COH and ROI finish time for all benchmarks in the 64-thread case in Table 3, where we also list their critical session access rate (CS Rate) and network utilization characteristics. The results are ordered by the ROI finish time improvement from up to down, with the lowest improvement on the top line. As shown in the table, in the PARSEC benchmarks, the average COH reduction reaches 40.4% and the average improvement in ROI finish time is 13.7%. The maximum COH reduction reaches 50.5% and the maximum improvement in ROI finish time is 19.5%, both for *canneal*. In SPEC OMP2012, the average COH reduction reaches 39.3% and the average improvement in ROI finish time is 15.1%. The maximum COH reduction reaches 61.8% for *botss* and the maximum improvement in ROI finish time is 24.5% for *ilbdc*. Across all 25 benchmarks, the average COH reduction reaches 39.9% and the average ROI finish time improvement is 14.4%.

6. RELATED WORK

Previously proposed techniques in expediting parallel application execution emphasize on accelerating serialized critical section execution. This involves finding the serialized part of parallel applications, and then accelerating its execution by exploiting various mechanisms such as running serialized codes on the fat cores in an asymmetric CMP ([21, 12, 13, 17, 22]), or by predicting and prioritizing threads that are executing serialized sections [8, 23, 14].

Identifying and analyzing critical threads: Amdahl's law is exploited with the notion of critical section in [10]. An analytic model is proposed to show that critical section which requires mutual excluded execution leads to serialization and puts a fundamental limit on parallel programs' performance. In [5], contention probability and hot critical section are used to evaluate the impact of critical sections on multi-threaded applications. Work in [9] proposes speedup stack, which quantifies the impact of various scaling delimiters on multi-threaded applications such as LLC, memory subsystem interference, workload imbalance and cache coherency, etc. In [7], a metric for assessing thread criticality, which considers both how much time a thread is performing useful work and how many co-running threads are waiting, is proposed. Further, the proposed metric is exploited to create criticality stacks that break the total execution time of a thread into criticality-related components, allowing accelerating critical threads using voltage/frequency scaling to improve performances.

Accelerating critical section execution: In [21], the authors propose a mechanism to accelerate critical section execution, in which selected critical sections are executed on the large core of an asymmetric CMP. Further, a mechanism called "Selective Acceleration of Critical Sections" is proposed to reduce false serialization in workloads. If different critical sections frequently appear at the same time, they are not sent to the large core for false serialization prevention. [12] proposes a cooperative software-hardware mechanism to identify and accelerate the most critical bottlenecks, which can be critical sections, barrier, or software pipelines, in multi-threaded programs executing on CMPs. The rationale is to measure the number of cycles spent by threads waiting for each bottleneck, and the one that keeps CPUs waiting for the longest time is the critical bottleneck. In [13], utility-based acceleration of multi-threaded applications is proposed, which relies on a new utility of acceleration metric that can both identify and accelerate the most critical code segments from multi-threaded applications. The mechanism is based on the observation that two types of code segments

that can become performance limits: 1) Threads that take longer to execute than other threads because of load imbalance. 2) Code segments, like contended critical sections, that make other threads wait. A memory-interference induced application slowdown estimation (MISE) model is introduced in [20], which can be utilized to achieve both QoS guarantee and system fairness enhancement in CMPs. It is shown in the work that performance of memory-intensive (memory-bound) application is roughly proportional to memory request service rate. The faster its memory requests are served, the faster the application's progress is. An application experiences little memory-level interference when its memory requests get the highest propriety. This observation is used to calculate an important factor called application's alone-request-service-rate in the MISE model. In [15], a dual voltage supply technique is exploited to speed up threads that hold locks.

In contrast to the above studies [21, 12, 13, 17, 22, 8, 23, 14]) which focus on accelerating critical section execution, our technique concentrates on reducing the competition overhead in multi/many-core when the queue spinlock in modern operating systems is used for locking critical sections in multi-threaded programs.

7. CONCLUSION

In the paper, we have shown that competition overhead as a major source of thread's blocking time can largely exceed the execution time of critical section itself and becomes the dominating factor that limits the performance of parallel programs. After analyzing the behavior of queue spinlock, we have proposed a software-hardware cooperative technique with implementation simplicity to effectively reduce the competition overhead of threads accessing critical sections. The basic idea is to opportunistically maximize the chance that a thread wins the critical section access right in the low-overhead spinning phase. This is achieved by embedding a thread's remaining times of retry (RTR) before being switched into the high-overhead sleeping phase as a priority parameter into the locking requests, and routers use the priority level to resolve physical/virtual channel contention so as to expedite the locking request packets with smaller RTR. As such, threads with a smaller RTR have a higher chance of reaching the lock variable earlier and hence can secure the lock earlier, thus reducing the thread blocking time.

With extensive full-system experiments in GEM5 running both PARSEC and SPEC OMP2012 benchmarks, we have demonstrated the effectiveness of our technique in reducing the competition overhead without influencing the critical section execution itself and thus improving the ROI finish time for all benchmark programs. We have also shown the scalable gains of our technique and how a proper priority level has been chosen for high performance improvements with low hardware overheads.

8. REFERENCES

[1] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A Detailed On-chip Network Model Inside a Full-system Simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Aug. 2008.

[3] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[4] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, Oct. 2007.

[5] G. Chen and P. Stenstrom, "Critical Lock Analysis: Diagnosing Critical Section Bottlenecks in Multithreaded Applications," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[6] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., 2003.

[7] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, "Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior," in *International Symposium on Computer Architecture (ISCA)*, 2013.

[8] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt, "Parallel Application Memory Scheduling," in *International Symposium on Microarchitecture (MICRO)*, 2011.

[9] S. Eyerman, K. Du Bois, and L. Eeckhout, "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012.

[10] S. Eyerman and L. Eeckhout, "Modeling Critical Sections in Amdahl's Law and Its Implications for Multicore Design," in *International Symposium on Computer Architecture (ISCA)*, 2010.

[11] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," in *IEEE Computer*, July 2008.

[12] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck Identification and Scheduling in Multithreaded Applications," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[13] ——, "Utility-based Acceleration of Multithreaded Applications on Asymmetric CMPs," in *International Symposium on Computer Architecture (ISCA)*, 2013.

[14] N. B. Lakshminarayana, J. Lee, and H. Kim, "Age Based Scheduling for Asymmetric Multiprocessors," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2009.

[15] T. N. Miller, X. Pan, R. Thomas, N. Sedaghati, and R. Teodorescu, "Booster: Reactive Core Acceleration for Mitigating the Effects of Process Variation and Application Imbalance in Low-Voltage Chips," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2012.

[16] M. Mitchell, J. Oldham, and A. Samuel, *Advanced Linux Programming*. New Riders Publishing, June 2001.

[17] T. Morad, A. Kolodny, and U. Weiser, "Scheduling Multiple Multithreaded Applications on Asymmetric and Symmetric Chip Multiprocessors," in *International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, 2010.

[18] M. S. Müller, J. Baron, W. C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, P. Shelepugin, M. van Waveren, B. Whitney, and K. Kumaran, "SPEC OMP2012 – An Application Benchmark Suite for Parallel Systems Using OpenMP," in *International Conference on OpenMP in a Heterogeneous World (IWOMP)*, 2012.

[19] L.-S. Peh and W. J. Dally, "A Delay Model and Speculative Architecture for Pipelined Routers," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2001.

[20] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu, "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2013.

[21] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating Critical Section Execution with Asymmetric Multi-core Architectures," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2009.

[22] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt, "Feedback-directed Pipeline Parallelism," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010.

[23] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling Heterogeneous Multi-Cores Through Performance Impact Estimation (PIE)," in *International Symposium on Computer Architecture (ISCA)*, 2012.