Bottleneck Identification and Scheduling in Multithreaded Applications

José A. Joao
ECE Department
The University of Texas at Austin
joao@ece.utexas.edu

M. Aater Suleman
Calxeda Inc.
aater.suleman@calxeda.com

Onur Mutlu
Computer Architecture Lab.
Carnegie Mellon University
onur@cmu.edu

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

Abstract

Performance of multithreaded applications is limited by a variety of bottlenecks, e.g. critical sections, barriers and slow pipeline stages. These bottlenecks serialize execution, waste valuable execution cycles, and limit scalability of applications. This paper proposes *Bottleneck Identification and Scheduling* (BIS), a cooperative software-hardware mechanism to identify and accelerate the most critical bottlenecks. BIS identifies which bottlenecks are likely to reduce performance by measuring the number of cycles threads have to wait for each bottleneck, and accelerates those bottlenecks using one or more fast cores on an Asymmetric Chip Multi-Processor (ACMP). Unlike previous work that targets specific bottlenecks, BIS can identify and accelerate bottlenecks regardless of their type. We compare BIS to four previous approaches and show that it outperforms the best of them by 15% on average. BIS' performance improvement increases as the number of cores and the number of fast cores in the system increase.

Categories and Subject Descriptors C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures (Multiprocessors)

General Terms Design, Performance

Keywords Critical sections, barriers, pipeline parallelism, multicore, asymmetric CMPs, heterogeneous CMPs

1. Introduction

Speeding up a single application using Chip Multi-Processors (CMPs) requires that applications be split into *threads* that execute concurrently on multiple cores. In theory, a CMP can provide speedup equal to the number of cores, however, the speedup is often lower in practice because of the thread waiting caused by serializing bottlenecks. These bottlenecks not only limit performance but also limit application scalability, e.g., performance of the *mysql* database peaks at 16 threads (Section 5.1.1).

Programmers are often burdened with the task of identifying and removing serializing bottlenecks. This solution is too costly because (a) writing correct parallel programs is already a daunting task, and (b) serializing bottlenecks change with machine configuration, program input set, and program phase (as we show in Section 2.2), thus, what may seem like a bottleneck to the programmer may not be a bottleneck in the field and vice versa. Hence, a solution that can identify and remove these bottlenecks at run-time (without requiring programmer effort) can not only increase parallel program performance and scalability but also reduce the work of the programmers.

An Asymmetric CMP (ACMP) consists of at least one large, fast core and several small, slow cores.[1] Previous research has shown that faster cores on an ACMP can be used to accelerate serializing bottlenecks. However, these proposals lack generality and fine-grained adaptivity. For example, several proposals [5, 13, 21, 26] use the fast core to accelerate Amdahl's serial portions (i.e., portions of a program where only one thread exists) but they do not handle serializing bottlenecks that happen in the parallel portions of the code. Accelerated Critical Sections (ACS) [36] accelerates critical sections –code regions that only one thread can execute at a given time–, but it does not always accelerate the critical sections that are currently *most critical* for performance and is limited to ACMPs with a single large core. Meeting Points [8] accelerates the thread with the longest expected completion time in the parallel region to reduce barrier synchronization overhead. However, it only applies to barriers in statically scheduled workloads, where the work to be performed by each thread is known before runtime. Feedback-Directed Pipelining (FDP) [34, 37], a software-only library, assigns threads to cores to balance stage throughput on pipelined workloads and improve performance or reduce power consumption, but it is slow to adapt to phase changes because it is software-based. Realistic workloads contain multiple different bottlenecks that can become critical at different execution phases, as we show in this paper. Therefore, it is important to have a single mechanism that is general and adaptive enough to accelerate the *most critical* bottlenecks at any given time.

We propose *Bottleneck Identification and Scheduling* (BIS), a general cooperative software-hardware framework to *identify* the most critical serializing bottlenecks at runtime, and *accelerate* them using one or multiple large cores on an ACMP. The key insight for identification is that *the most critical serializing bottlenecks are likely to make other threads wait for the greatest amount of time*. The programmer, compiler or library delimits potential bottlenecks using BottleneckCall and BottleneckReturn instructions, and replaces the code that waits for bottlenecks with a BottleneckWait instruction. The hardware uses these instructions to measure thread waiting cycles (i.e., number of cycles threads wait due to the bottleneck) for each bottleneck and tracks them in a *Bottleneck Table*. The bottlenecks with the highest number of thread waiting cycles are selected for acceleration on one or more large cores. On executing a BottleneckCall instruction, the small core checks if the bottleneck has been selected for acceleration. If it has, the small core enqueues an execution request into the *Scheduling Buffer* of a large core. The large core dequeues each request from its Scheduling Buffer, executes the bottleneck and notifies the small core once it encounters the BottleneckReturn instruction.

ASPLOS'12 March 3–7, 2012, London, England, UK.
Copyright © 2012 ACM 978-1-4503-0759-8/12/03…$10.00
Reprinted from ASPLOS'12, Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, March 3–7, 2012, London, England, UK.

[1] We use the terms fast and large, and slow and small, interchangeably, to refer to the two types of cores on an ACMP.

This paper makes three main **contributions**:

1. We propose a cooperative hardware-software mechanism to identify the most critical bottlenecks of different types, e.g., critical sections, barriers and pipeline stages, allowing the programmer, the software or the hardware to remove or accelerate those bottlenecks. To our knowledge, this is the first such proposal.
2. We propose an automatic acceleration mechanism that decides which bottlenecks to accelerate and where to accelerate them. We use the fast cores on an ACMP as hardware-controlled fine-grain accelerators for serializing bottlenecks. By using hardware support, we minimize the overhead of execution migration and allow for quick adaptation to changing critical bottlenecks.
3. This is the first paper to explore the trade-offs of bottleneck acceleration using ACMPs with multiple large cores. We show that multiple large cores improve performance when multiple bottlenecks are similarly critical and need to be accelerated simultaneously.[2]

Our evaluation shows that BIS improves performance by 37% over a 32-core Symmetric CMP and by 32% over an ACMP (which only accelerates serial sections) with the same area budget. BIS also outperforms recently proposed dynamic mechanisms (ACS for non-pipelined workloads and FDP for pipelined workloads) on average by 15% on a 1-large-core, 28-small-core ACMP. We also show that BIS' performance improvement increases as the number of cores and the number of large cores increase.

2. Motivation

2.1 Bottlenecks: What They Are and Why They Should Be Accelerated

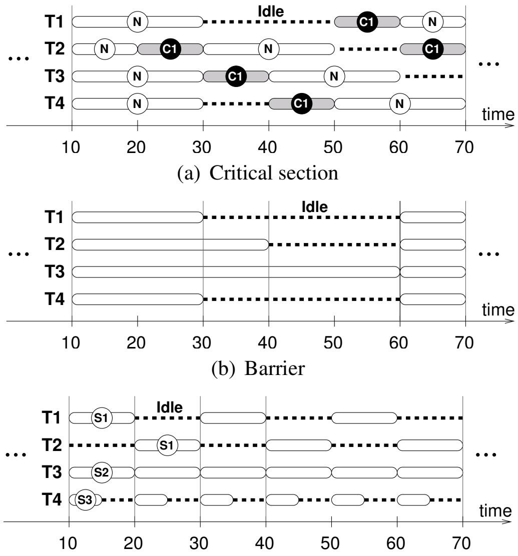
We define *bottleneck* as any code segment for which threads contend. A bottleneck may consist of a single thread or multiple threads that need to reach a synchronization point before other threads can make progress. We call the threads that stall due to a bottleneck *waiters* and the threads executing a bottleneck *executers*. A single instance of a bottleneck can be responsible for one or many waiters. The effect of bottlenecks on performance can be substantial because every cycle a bottleneck is running is a cycle wasted on *every one of the waiters* of that bottleneck. Bottlenecks cause thread serialization. Thus, a parallel program that spends a significant portion of its execution in bottlenecks can lose some or even all of the speedup that could be expected from parallelization. Next we describe examples of bottlenecks.

2.1.1 Amdahl's Serial Portions

When only one thread exists [4], it is on the critical path and should be scheduled on the fastest core to minimize execution time. Meanwhile, all other cores are idle.

2.1.2 Critical Sections

Critical sections ensure mutual exclusion, i.e. only one thread can execute a critical section at a given time, and any other thread wanting to execute the critical section must wait. That is, there is only one executer and possibly multiple waiters. Figure 1(a) shows the execution of four threads (T1-T4) that run a simple loop that consists of a critical section C1 and a non-critical section N. Threads can execute in parallel until time 30, at which T3 acquires the lock and prevents T1 and T4 from doing so. Then, T1 and T4 become idle, waiting on T3. Accelerating the execution of C1 on T3 during this time would not only accelerate T3 and reduce the total execution time (because this instance of code segment C1 is on the critical path), but also reduce the wait for both T1 and T4. However, accelerating the same C1 on T2 between times 20 and 30 would not help because all the other threads are running useful work and no thread is waiting for the lock.



(a) Critical section

(b) Barrier

(c) Pipeline stages

Figure 1. Examples of bottleneck execution.

2.1.3 Barriers

Threads that reach a barrier must wait until all threads reach the barrier. There can be multiple executers and multiple waiters. Figure 1(b) shows the execution of four threads reaching a barrier. T1 and T4 reach the barrier at time 30 and start waiting on T2 and T3. T2 reaches the barrier at time 40, leaving T3 as the only running thread until time 60, when T3 reaches the barrier and all four threads can continue. Therefore, every cycle saved by accelerating T3 gets subtracted from the total execution time, up to the difference between the arrivals of T2 and T3 at the barrier.

2.1.4 Pipeline Stages

In a pipelined parallel program, loop iterations are split into stages that execute on different threads. Threads executing other pipeline stages wait for the slowest pipeline stages. There can be multiple executers and multiple waiters. Figure 1(c) shows the execution of a pipelined program with three stages (S1-S3) on four threads. S1 is scalable and runs on both T1 and T2. S2 is not scalable, so it runs on a single thread (T3). S3 is shorter and runs on T4. The throughput of a pipeline is the throughput of the slowest stage. Since S2 has a smaller throughput than S1, S2's input queue gets full and S1 has to stall to produce work for S2 at the pace S2 can deal with it, which introduces idle time on T1 and T2. On the other side, S3 is faster than S2 and finishes each iteration before S2 can feed it more work to do, which introduces idle time on T4. Therefore, T1, T2 and T4 are all waiters on T3 at some point. If S2 could be accelerated by a factor of 2, the pipeline throughput would be balanced (i.e., each stage would deal with one iteration every 5 time units) and all idle time would be eliminated. Since S2 is not scalable, this acceleration is only feasible with higher single-thread performance on T3. Note that only accelerating the actual slowest stage improves performance. Accelerating any other stage does not. Also note that the best way to accelerate a *scalable* stage is to add more threads (cores) to work on it.

From these examples, we can conclude that *accelerating the bottlenecks that are making other threads wait is likely to improve multithreaded program performance*.

[2] We also evaluate large cores with Simultaneous Multithreading (SMT) in Section 5.2.

2.2 How Bottlenecks Change Over Time

Let's first consider a simple example, Program 1, where elements X are moved from linked list A to linked list B. Removal from A and insertion into B are protected by critical sections to avoid races that could corrupt the lists. The amount of work performed inside each of those critical sections varies over time, because list A is shrinking while list B is growing. Figure 2 shows that contention (i.e. the number of threads waiting to enter each critical section) with 32 threads changes as follows: in the first phase, critical section A is the bottleneck, while in the second phase, critical section B becomes the bottleneck. Successful acceleration of this program at runtime requires dynamic identification of which code is the current critical bottleneck and then running it on a fast core.

Program 1 Example of time-variant critical sections

```
A=full linked list; B=empty linked list
repeat
  Lock A
    Traverse list A
    Remove X from A
  Unlock A
  Compute on X
  Lock B
    Traverse list B
    Insert X into B
  Unlock B
until A is empty
```

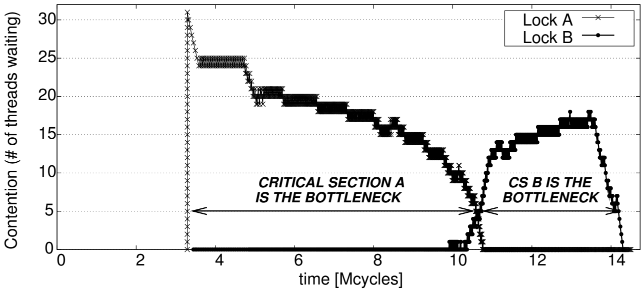


Figure 2. Contention for each critical section on Program 1.

Similar examples exist in important real-life workloads. For example, the most contended critical section in the widely used *mysql* database changes over short periods of time. The two most contended critical sections in *mysql* with the OLTP-nontrx input set are *LOCK_open* and *LOCK_log*. *LOCK_open* is used to ensure mutual exclusion while opening and closing tables in the *open_table* and *close_thread_tables* functions. *LOCK_log* is used to protect access to the log file, which happens in the *MYSQL_LOG::write* function.

Figure 3 shows contention for *LOCK_open* and *LOCK_log* over the first 8M cycles of *mysql* with 16 threads (the optimal number of threads) while being driven by SysBench [3] with the OLTP-nontrx input set[3] (*mysql-3* in our benchmark set, Table 6). The graph shows three clearly different phases: A) the initial use of *LOCK_log* by every thread, B) the heavy use of *LOCK_open* while all threads simultaneously want to open the tables they need, C) the steady state period where transactions continue executing and *LOCK_log* is usually the most contended, but *LOCK_open* sporadically gets highly contended (e.g., see arrows around 4.8Mcycles and 7.7Mcycles). This last phase continues for the rest of the program: the most contended lock changes intermittently for the duration of this phase. The temporal behavior of the critical sections depends entirely on the database operations that are executed, i.e., the input set. For example, with the OLTP-complex input set, *LOCK_open* gets highly contended even more often.

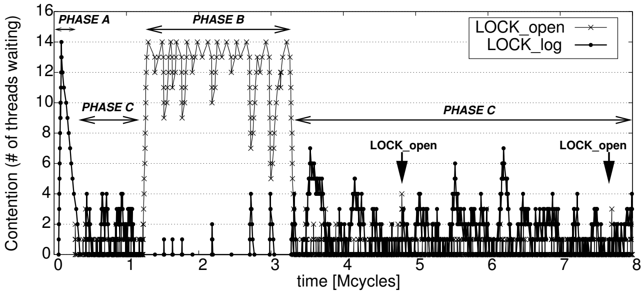


Figure 3. Most contended critical sections on *mysql*.

Previously proposed ACS [36] can accelerate only one critical section at a time on a single large core (or two critical sections on a 2-way SMT large core, but this reduces the benefit of acceleration due to resource sharing between SMT threads). When there are multiple critical sections contending to be accelerated, ACS disables acceleration of critical sections that are less frequent, in order to avoid false serialization (i.e., serialized execution of different critical sections at the large core). Periodically (every 10M instructions in [36]), ACS re-enables acceleration of all critical sections, allowing for some false serialization while it relearns which critical sections should be disabled. This mechanism works well when contention for critical sections is stable for long periods but it is suboptimal for applications like *mysql*, because it ignores the frequent changes on which critical section is currently the most limiting bottleneck, as shown in Figure 3. In addition, ACS is specific to critical sections and cannot handle other types of bottlenecks such as barriers and slow pipeline stages.

Our goal is to design a mechanism that identifies at any given time which bottlenecks of any type are currently limiting performance, and accelerates those bottlenecks.

3. Bottleneck Identification and Scheduling

3.1 Key Insight

The benefit of parallel computing comes from concurrent execution of work; the higher the concurrency, the higher the performance. Every time threads have to wait for each other, less work gets done in parallel, which reduces parallel speedup and wastes opportunity. To maximize performance and increase concurrency, it is pivotal to minimize thread waiting as much as possible. Consequently, a large amount of effort is invested by parallel programmers in reducing bottlenecks that can lead to thread waiting. For example, programmers spend an enormous amount of effort in reducing false sharing, thread synchronization, thread load imbalance, etc. Previous research on computer architecture also targets thread waiting, e.g., transactional memory [12], primitives like test-and-test-and-set [31], lock elision [29], speculative synchronization [24], etc. We use this conventional wisdom –that reducing thread waiting improves performance– to identify the best bottlenecks for acceleration. Our key idea is thus simple: measure the number of cycles spent by threads waiting for each bottleneck and accelerate the bottlenecks responsible for the highest thread waiting cycles.

Our proposal, BIS, consists of two parts: identification of critical bottlenecks and acceleration of those bottlenecks.

3.2 Identifying Bottlenecks

Identification of critical bottlenecks is done in hardware based on information provided by the software. The programmer, compiler or library assigns a unique bottleneck ID (*bid*) to each potential serializing bottleneck, and encapsulates the bottleneck code between *BottleneckCall* and *BottleneckReturn* instructions. Using these instructions, the hardware keeps track of the bottlenecks and which hardware threads (core ID and SMT context) they execute on.

[3] SysBench is a widely used multi-threaded benchmark to measure system and database performance.

```
          ...
          BottleneckCall bid, targetPC
          ...
targetPC: while cannot acquire lock
            BottleneckWait bid
          acquire lock
            ...
          release lock
          BottleneckReturn bid
```

(a) Critical section

```
          ...
          BottleneckCall  bid, targetPC
          ...
targetPC: code running for the barrier
          ...
          enter barrier
          while not all threads in barrier
            BottleneckWait  bid
          exit barrier
          BottleneckReturn  bid
```

(b) Barrier

```
          while not done
            BottleneckCall bid, targetPC
          ...
targetPC: while empty queue                 // previous stage
            BottleneckWait prev_bid         // caused the waiting
          dequeue work
          do the work ...
          while full queue                  // next stage
            BottleneckWait next_bid         // caused the waiting
          enqueue next work
          BottleneckReturn bid
```

(c) Pipeline

Figure 4. Modifications to code implementing bottlenecks.

To identify which bottlenecks cause thread waiting, we use a new instruction *BottleneckWait*. The purpose of this instruction is twofold: it implements waiting for a memory location to change (similarly to existing instructions like *mwait* in x86 [17]), and it allows the hardware to keep track of how many cycles each thread is waiting for each bottleneck. Typically, when threads are waiting for a bottleneck, they are waiting for the value of a particular memory location *watch_addr* to satisfy a condition. For example, for critical sections, *watch_addr* can be the lock address; for barriers it can be the address of the counter of threads that have reached the barrier; and for pipeline stages it can be the address of the size of the queue where work quanta are read from [37]. In most cases *bid* can be the same as *watch_addr*, but we separate them to provide the flexibility required for some implementations, e.g. MCS locks [25]. Table 1 summarizes the format and purpose of the new instructions required by BIS.

BottleneckCall bid, *targetPC*
Marks the beginning of the bottleneck identified by *bid* and calls the bottleneck subroutine starting at targetPC
BottleneckReturn bid
Marks the end of the bottleneck identified by *bid* and returns from the bottleneck subroutine
BottleneckWait bid[, *watch_addr*[, *timeout*]]
Waits for a maximum of *timeout* cycles for the content of memory address *watch_addr* associated with bottleneck *bid* to change, while keeping track of the number of waiting cycles

Table 1. ISA support for BIS.

The new instructions required by BIS do not affect the semantics or the implementation of the bottlenecks and can be easily added to any existing bottleneck implementation. The compiler encapsulates potential bottlenecks into functions using *function outlining* [40] and inserts BottleneckCall, BottleneckReturn and BottleneckWait instructions. Figure 4 shows the use of these instructions for critical sections, barriers and pipeline stages. Since programming primitives that become bottlenecks are usually implemented inside libraries, the changes can be hidden so that the programmer does not need to change the program.

We track waiting cycles for each bottleneck, and identify the critical bottlenecks with low overhead in hardware using a **Bottleneck Table (BT)**. Each BT entry corresponds to a bottleneck and includes a thread waiting cycles (TWC) field. TWC is computed by aggregating the number of waiters on the bottleneck (i.e., the number of threads currently running a BottleneckWait instruction) over time. The exact computation is described in Sections 3.5.1 and 3.5.2. The number of waiters is another field in the BT entry, and is incremented when a BottleneckWait instruction starts executing for that *bid* and decremented when the BottleneckWait instruction commits, i.e., after the wait for the bottleneck ends. The Bottleneck Table contains logic to identify the most critical bottlenecks, i.e. the entries with top N highest thread waiting cycles. We provide more details about the implementation of the BT in Section 3.5.2.

3.3 Accelerating Bottlenecks

There are multiple ways to accelerate a bottleneck, e.g. increasing core frequency [5], giving a thread higher priority in shared hardware resources [9], or migrating the bottleneck to a faster core with a more aggressive microarchitecture or higher frequency [26, 36]. The bottleneck identification component of BIS can work with any of these schemes. As an example, we show how our identification mechanism can use the fast/large cores of an ACMP as fine-grain hardware-controlled bottleneck accelerators, and the slow/small cores for running the remaining parallel code. We first assume a single large core context and in Section 3.4 we extend the design to multiple large core contexts.

When the BT updates thread waiting cycles (TWC), bottlenecks with TWC greater than a threshold are enabled for acceleration and the rest are disabled.[4]

When a small core encounters a BottleneckCall instruction, it checks whether or not the bottleneck is enabled for acceleration. To avoid accessing the global BT on every BottleneckCall, each small core is augmented with an **Acceleration Index Table** (AIT) that caches the *bid* and acceleration enable bit of bottlenecks.[5] If a *bid* is not present in the AIT, it is assumed to be disabled for acceleration. If the bottleneck is disabled for acceleration, the small core executes the bottleneck locally. If acceleration is enabled, the small core sends a bottleneck execution request to the large core and stalls waiting for a response.

When the large core receives a bottleneck execution request from a small core, it enqueues the request into a new structure called **Scheduling Buffer** (SB). Each SB entry has: *bid*, originating small core/hardware thread ID, target PC, stack pointer and thread waiting cycles (TWC, copied from the BT entry). The SB works as a priority queue based on TWC: the first bottleneck that runs on the large core is the oldest instance of the bottleneck with the highest TWC. When the large core executes the BottleneckReturn instruction, it sends a BottleneckDone signal to the small core. On receiving BottleneckDone, the small core continues executing the instruction after the BottleneckCall. As we will explain in Section 3.3.2, the large core may also abort the request while it is on the SB. If that happens, the large core removes the request from the SB and sends a BottleneckCallAbort signal to the small core. On receiving BottleneckCallAbort, the small core executes the bottleneck locally.

3.3.1 Threads Waiting on the Scheduling Buffer

A thread executing a BottleneckCall instruction that was sent to the large core and is waiting on the SB also incurs thread waiting cycles that must be attributed to the bottleneck. To account for these waiting cycles, when a bottleneck execution request is enqueued into the SB, the large core sends an update to the BT, incrementing waiters. It also increments a new field waiters_sb, which separately

[4] A threshold of 1024 works well for our workloads.

[5] When a bottleneck gets enabled or disabled for acceleration, the BT broadcasts the updates to the AITs on all small cores.

keeps track of the number of waiters on the Scheduling Buffer for purposes we explain in Section 3.3.2. When the bottleneck execution request is dequeued from the SB, to be executed or aborted, the large core updates the BT, decrementing waiters and waiters_sb.

3.3.2 Avoiding False Serialization and Starvation

Since multiple bottlenecks can be scheduled on a large core, a bottleneck may have to wait for another bottleneck with higher priority (i.e. higher thread waiting cycles) for too long, while it could have been executing on the small core. This problem is called *false serialization*[6] and BIS mitigates it by aborting a request enqueued on the SB if the bottleneck: (a) does not have the highest thread waiting cycles in the SB, and (b) would be *ready to run* on its original small core. A bottleneck is considered *ready to run* if no other instance of that bottleneck is currently executing and all the waiters are on the SB. This situation results in unnecessary delaying of all those waiters and can be easily detected when the BT is updated. To perform this detection, each BT entry keeps the number of executers (incremented by BottleneckCall and decremented by BottleneckReturn), the number of waiters (incremented while a BottleneckWait instruction is executing or a bottleneck is waiting on the SB), and the number of waiters on the SB (waiters_sb, incremented while a request is waiting on the SB). If executers is zero and waiters is non-zero and equal to waiters_sb, the BT sends a BottleneckReady signal to the SB. Upon receiving this signal, the SB dequeues the oldest request for the bottleneck and sends a BottleneckCallAbort to the originating small core. This mechanism also prevents starvation of requests in the SB. As a failsafe mechanism to avoid starvation, the SB also aborts requests that have been waiting for more than 50K cycles.

3.3.3 Preemptive Acceleration

Bottleneck instances can vary in length. Since our mechanism uses history to decide whether or not to accelerate a bottleneck, it is possible for a long instance to get scheduled on a small core and later become responsible for high thread waiting cycles. To fix this problem, we introduce a preemptive mechanism: if upon updating thread waiting cycles, the BT detects that a bottleneck running on a small core has become the bottleneck with the highest thread waiting cycles, the bottleneck is shipped to the large core. BT sends a preempt signal to the small core informing it to stall execution, push the architectural state on the stack and send a request to the large core to continue execution. The large core pops the architectural state from the stack and resumes execution until BottleneckReturn. Acceleration of barriers relies on this preemptive mechanism.

3.4 Support for Multiple Large Core Contexts

Our mechanism can scale to multiple large cores or a large core with multiple hardware contexts (Simultaneous Multithreading, SMT) with only three modifications:

First, each large core has its own Scheduling Buffer. SMT contexts at a large core share the Scheduling Buffer but execute different bottlenecks because otherwise they would wait for each other.

Second, each bottleneck that is enabled for acceleration is assigned to a fixed large core context to preserve cache locality and avoid different large cores having to wait for each other on the same bottlenecks. To that effect, each BT entry and Acceleration Index Table entry is augmented with the ID of the large core the bottleneck is assigned to. If there are N large core contexts, the Bottleneck Table assigns each of the top N bottlenecks to a different large core context to eliminate false serialization among them. Then, the rest of the bottlenecks that are enabled for acceleration are assigned to a random large core.

Third, the preemptive mechanism described in Section 3.3.3 is extended so that in case a bottleneck scheduled on small cores (a) becomes the top bottleneck, and (b) its number of executers is less than or equal to the number of large core contexts, the BT sends signals to preemptively migrate those threads to the large cores. This mechanism allows acceleration on N large core contexts of the last N threads running for a barrier.

3.5 Implementation Details

3.5.1 Tracking Dependent and Nested Bottlenecks

Sometimes a thread has to wait for one bottleneck while it is executing another bottleneck. For example, in a pipelined workload, a thread executing stage S2 may wait for stage S1. Similar situations occur when bottlenecks are nested. For example, a thread T1 is running B1 but stalls waiting for a nested bottleneck B2. Concurrently, T2 also stalls waiting for B1, which means it is indirectly waiting for B2. In these scenarios, the thread waiting cycles should be attributed to the bottleneck that is the root cause of the wait, e.g., waiting cycles incurred by T2 should not be attributed to bottleneck B1 which is not really running, but to bottleneck B2.

In general, to determine the bottleneck Bj that is the root cause of the wait for each bottleneck Bi, we need to follow the dependency chain between bottlenecks until a bottleneck Bj is found not to be waiting for a different bottleneck. Then, the current number of waiters for Bi, i.e. the required increment in TWC, is added to the TWC for Bj. To follow the dependency chain we need to know (a) which threads is executing a bottleneck and (b) which bottleneck, if any, that thread is currently waiting for. To know (a), we add an executer_vec bit vector on each BT entry (one bit per hardware thread) that records all current executers of each bottleneck (set on BottleneckCall and reset on BottleneckReturn). To know (b), we add a small **Current Bottleneck Table** (CBT) associated with the BT and indexed with hardware thread ID that gives the *bid* that the thread is currently waiting for (set during the execution of BottleneckWait, or zero at all other times). To summarize, the BT updates TWC for each bottleneck Bi with the following algorithm:

```
foreach Bi in BottleneckTable:
  B = Bi; done = false
  while (not done):
    Bj = B
    E = randomly picked Executer of B // uniformly distribute TWC
    B = CurrentBottleneck(E)
    done = (B == 0) or // Bj is running or
           (B == Bj)   // waiting for another instance of itself
  TWC(Bj) += waiters(Bi)
```

The basic idea of the algorithm is to follow the dependence chain between bottlenecks until a bottleneck Bj is found not to be waiting for a different bottleneck. This algorithm is also applicable when there are no dependencies, i.e. the loop will execute once and will result in Bj = Bi. Note that the algorithm is not exact and only attempts to assign thread waiting cycles to a bottleneck that is actually running and making other threads wait, so that if that bottleneck were accelerated, the cycles spent by other threads waiting would also be reduced. The computation of TWC does not need to be perfect because BIS is solely for performance and does not affect correctness.

3.5.2 Bottleneck Table

The new instructions update the Bottleneck Table after committing and off the critical path of execution. Table 2 describes each BT entry. When a BottleneckCall or BottleneckWait instruction is executed with a *bid* that is not in the BT, a new entry is inserted in the BT. Since BIS is solely for performance and not for correctness, the BT only needs enough entries to track the major bottlenecks in each phase of a program. Our studies (not included here due to space constraints) show that a BT with 32 entries is sufficient for our workloads.

[6] False serialization can also introduce deadlocks on a deadlock-free program [36].

Field	Description
bid	Bottleneck ID
pid	Process ID
executers	Current number of threads that are running the bottleneck, i.e., executing between a BottleneckCall and a BottleneckReturn
executer_vec	Bit vector of threads executing the bottleneck, each bit set on BottleneckCall and reset on BottleneckReturn
waiters	Current number of threads that are waiting for the bottleneck, i.e. executing a BottleneckWait or enqueued on the SB
waiters_sb	Current number of threads that are waiting on the SB, i.e. executing a BottleneckCall that was sent to a large core
TWC	Thread waiting cycles
large_core_id	ID of the large core assigned for acceleration

Table 2. Bottleneck Table entry.

We implement the BT as an associative cache. The replacement algorithm evicts the entry with the smallest thread waiting cycles. We update thread waiting cycles for each bottleneck by incrementing it by the number of waiters every cycle (for practicality we add 32 * number of waiters every 32 cycles). We halve the thread waiting cycles for all entries every 100K cycles to allow for the replacement of stale entries. We find that a centralized BT structure is not a bottleneck, however, if need arises, the BT can be partitioned without much effort using the principles of a distributed directory.

3.5.3 Hardware Cost

Figure 5 shows the location of the new structures required by BIS on an ACMP, and Table 3 describes their storage cost, which is only 18.7 KB for a CMP with 2 large cores and 56 small cores (area budget of 64). The storage cost for a CMP with 2 large cores and 120 small cores (area budget of 128) would be 39.6 KB.

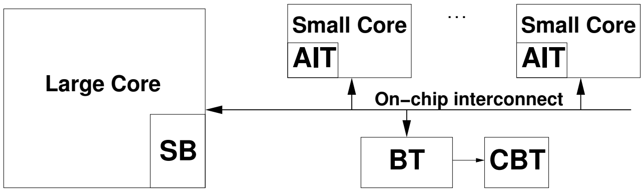


Figure 5. Hardware structures for BIS.

3.5.4 Handling Interrupts

The Operating System may interrupt the cores, e.g., on time quantum expiration or for I/O. If a small core gets an interrupt while waiting for a large core to execute a bottleneck, it does not service the interrupt until a BottleneckDone or BottleneckCallAbort is received. If a small core gets an interrupt while executing a BottleneckWait instruction, the BottleneckWait instruction is forced to finish, which removes the waiter from the BT, but it is reexecuted after returning from the interrupt. If a large core gets an interrupt while accelerating a bottleneck, it aborts all bottlenecks in its Scheduling Buffer, finishes the current bottleneck, and then services the interrupt.

3.5.5 Transfer of Cache State to the Large Core

A bottleneck executing remotely on the large core may require data that resides in the small core, thereby producing cache misses that reduce the benefit of acceleration. Data Marshaling [35] (DM) has been proposed to reduce these cache misses, by identifying and marshaling the cache lines required by the remote core. It is easy to integrate DM with our proposal and we evaluate our proposal with and without DM in Section 5.4.

3.5.6 Compiler Support

The compiler can trivially convert bottlenecks that start and end in the same scope to use BottleneckCall/BottleneckReturn. Complex bottlenecks spanning multiple functions or modules may require programmer intervention and certainly BIS may not be applicable when not even the programmer can perform function outlining on the bottlenecks. A more flexible ISA with two separate instructions BottleneckBegin and BottleneckEnd to delimit bottlenecks without encapsulating them in functions can expand the applicability of BIS at the expense of potentially higher overheads, but we have not explored it. The consequences of failing to mark code segments as potential bottlenecks depend on their actual criticality. There is no penalty for failing to mark non-critical bottlenecks. However, if critical bottlenecks are not marked, BIS misses the opportunity to improve performance. Marking too many non-critical bottlenecks does not significantly reduce performance because the Bottleneck Table retains only the currently most critical bottlenecks.

We have targeted same-ISA ACMPs. However, BIS could be extended to heterogeneous-ISA ACMPs by including two binary implementations for each bottleneck in the executable image, one to be executed on the small core and the other to be executed on the large core. This is part of our future work.

3.5.7 Support for Program Tuning

BIS can increase programmer productivity by providing feedback to programmers about the most critical bottlenecks. Programmers can then decide which bottlenecks to optimize to get the best results from their effort. We propose adding a set of performance counters to the BT to store *bid* and cumulative thread waiting cycles (TWC) for the bottlenecks with highest TWC. These counters can be accessed by tools such as Vtune and Intel Parallel Studio [18] to guide the programmer. Due to space limitations we do not describe the structure and the interface, but these are straightforward since the BT already has the required information.

4. Experimental Methodology

We use an x86 cycle-level simulator that models symmetric and asymmetric CMPs. Symmetric CMPs (SCMPs) consist of only small cores. The small in-order cores are modeled after the Intel Pentium processor and the fast cores are modeled after the Intel Core 2 architecture. Details are shown in Table 4. We model all latency overheads associated with our mechanism faithfully. We evaluate our proposal on different processor configurations with equal area budget. We compare our proposal to previous work, summarized in Table 5. SCMP and ACMP use different thread scheduling policies for non-pipelined and pipelined workloads. We apply ACS and MC-ACS only to non-pipelined workloads, while FDP only applies to pipelined workloads by design. BIS is general and applies to both. Our evaluation is based on the 12 workloads shown in Table 6, which we simulate to completion. We use *mysql* with three different input sets because (a) *mysql* is a widely used commercial workload, and (b) its behavior significantly depends on the input set.

5. Evaluation

5.1 Single Large Core

It is difficult to determine before runtime the optimal number of threads (and this number varies dynamically), thus it is common practice to run applications with as many threads as available cores [16, 28]. On applications that are not I/O-intensive, this policy can maximize resource utilization without oversubscribing the cores, but may result in inferior performance if the application performance peaks at fewer threads than the number of cores. Thus, we evaluate two situations: number of threads equal to the number of cores and optimal number of threads, i.e., the number of threads at which performance is maximum. Table 7 summarizes the average speedups of BIS over other techniques in both situations for different area budgets.

Structure	Purpose	Location and entry structure (field sizes in parenthesis)	Cost
Bottleneck Table (BT)	To decide which bottlenecks are more critical and where to accelerate them	One global 32-entry table for the CMP. Each entry has 192 bits: *bid*(64), pid(16), executers(6), executer_vec(64), waiters(6), waiters_sb(6), TWC(24), large_core_id(6)	768 B
Current Bottleneck Table (CBT)	To help track the root cause of thread waiting on nested or dependent bottlenecks	Attached to the BT, 64-entries. Each entry has 64 bits, the *bid* of the bottleneck the thread is waiting for, and is indexed by hardware thread ID.	512 B
Acceleration Index Tables (AIT)	To reduce global accesses to BT to find if a bottleneck is enabled for acceleration	One 32-entry table per small core. Each entry has 66 bits: *bid*(64), enabled (1), large core ID (1). Each AIT has 264 bytes.	14.4 KB
Scheduling Buffers (SB)	To store and prioritize bottleneck execution requests on each large core	One 56-entry buffer per large core. Each entry has 222 bits: *bid*(64), small core ID (6), target PC(64), stack pointer (64), thread waiting cycles (24). Each SB has 1554 bytes.	3.1 KB
		Total	18.7 KB

Table 3. Description of hardware structures for BIS and storage cost on a 56-small-core, 2-large-core CMP.

Small core	2-wide, 5-stage in-order, 4GHz, 32 KB write-through, 1-cycle, 8-way, separate I and D L1 caches, 256KB write-back, 6-cycle, 8-way, private unified L2 cache
Large core	4-wide, 12-stage out-of-order, 128-entry ROB, 4GHz, 32 KB write-through, 1-cycle, 8-way, separate I and D L1 caches, 1MB write-back, 8-cycle, 8-way, private unified L2 cache
Cache coherence	MESI protocol, on-chip distributed directory, L2-to-L2 cache transfers allowed, 8K entries/bank, one bank per core
L3 cache	Shared 8MB, write-back, 16-way, 20-cycle
On-chip interconnect	Bidirectional ring, 64-bit wide, 2-cycle hop latency
Off-chip memory bus	64-bit wide, split-transaction, 40-cycle, pipelined bus at 1/4 of CPU frequency
Memory	32-bank DRAM, modeling all queues and delays, row buffer hit/miss/conflict latencies = 25/50/75ns

CMP configurations with area equivalent to N small cores.	
SCMP	N small cores
ACMP	1 large core and N-4 small cores, the large core always runs any single-threaded code
ACS	1 large core and N-4 small cores, the large core always runs any single-threaded code and accelerates critical sections as proposed in [36]
BIS	L large cores and S=N-4*L small cores, one large core always runs any single-threaded code, 32-entry Bottleneck Table, N-entry Current Bottleneck Table, each large core is equipped with an S-entry Scheduling Buffer, each small core is equipped with a 32-entry Acceleration Index Table

Table 4. Baseline processor configuration.

Mechanism	Non-pipelined workloads	Pipelined workloads
SCMP	One thread per core	Best static integer schedule described in [37]
ACMP	Serial portion on large core, parallel portion on all cores [5, 26]	Best static integer schedule including large cores [34]
ACS	Serial portion and critical sections on large core, parallel portion on small cores [36]	Not used, because in our workloads it is more profitable to schedule the large core/s to accelerate pipeline stages using FDP, as described in [34], instead of critical sections within stages
MC-ACS	To compare to the best possible baseline, we extend ACS to multiple large cores by retrying once more the acceleration of falsely serialized critical sections on a different large core before disabling their acceleration	
FDP	Not applicable (designed for pipelined workloads)	Dynamic scheduling of all cores to stages described in [34]
BIS	Our proposal	

Table 5. Experimental configurations.

Workload	Description	Source	Input set	# of Bottl.	Bottleneck description
iplookup	IP packet routing	[39]	2.5K queries	# of threads	Critical sections (CS) on routing tables
mysql-1	MySQL server [1]	SysBench [3]	OLTP-complex	29	CS on meta data, tables
mysql-2	MySQL server [1]	SysBench [3]	OLTP-simple	20	CS on meta data, tables
mysql-3	MySQL server [1]	SysBench [3]	OLTP-nontrx	18	CS on meta data, tables
specjbb	JAVA business benchmark	[33]	5 seconds	39	CS on counters, warehouse data
sqlite	sqlite3 [2] DB engine	SysBench [3]	OLTP-simple	5	CS on database tables
tsp	Traveling salesman	[20]	11 cities	2	CS on termination condition, solution
webcache	Cooperative web cache	[38]	100K queries	33	CS on replacement policy
mg	Multigrid solver	NASP [6]	S input	13	Barriers for omp single/master
ft	FFT computation	NASP [6]	T input	17	Barriers for omp single/master
rank	Rank strings based on similarity to another string	[37]	800K strings	3	Pipeline stages
pagemine	Computes a histogram of string similarity	Rsearch [27]	1M pages	5	Pipeline stages

Table 6. Evaluated workloads.

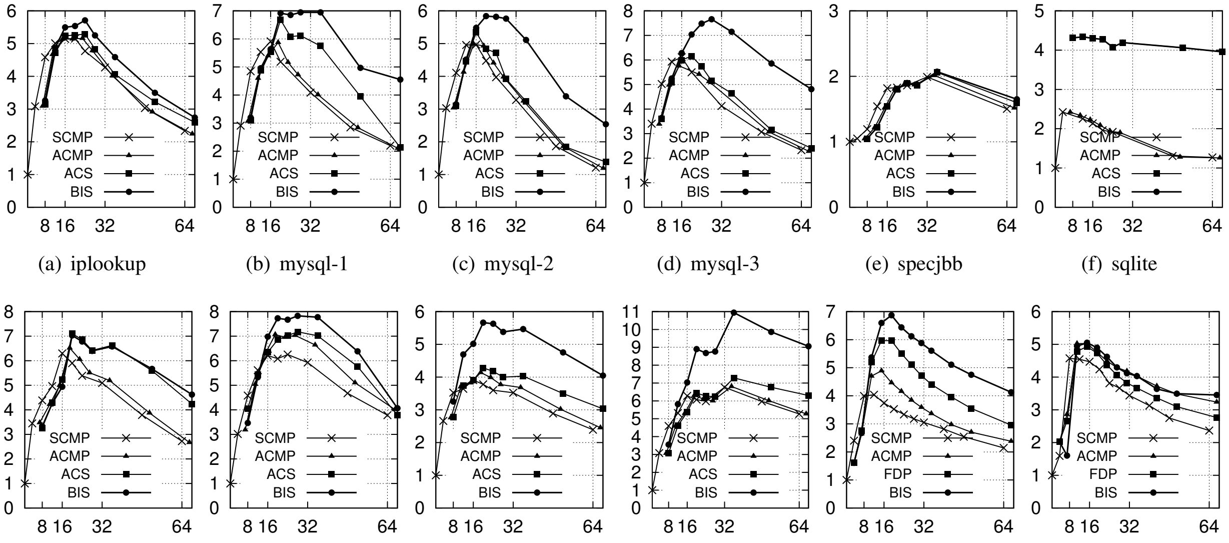
Num. of threads	Equal to number of cores				Optimal			
Area Budget	8	16	32	64	8	16	32	64
SCMP	-20	10	45	75	-20	10	37	43
ACMP	6.9	13	43	70	1.4	12	32	36
ACS/FDP	2.4	6.2	17	34	2.4	6.2	15	19

Table 7. Average speedups (%) of BIS over SCMP, ACMP and ACS/FDP.

5.1.1 Number of Threads Equal to Number of Cores

Figure 6 shows speedups over a single small core for different area budgets (measured in equivalent number of small cores). The figure shows the speedups of SCMP, ACMP and ACS (for non-pipelined workloads) or FDP (for pipelined workloads) and BIS. Our proposal improves performance over ACMP and SCMP and matches or improves over ACS/FDP on all benchmarks for an area budget of 32 cores. On average, it improves performance by 43% over ACMP, 45% over SCMP and 17% over ACS/FDP. Three key observations are in order.

First, as the number of cores increases, so does the benefit of BIS. On average, BIS outperforms ACS/FDP by 2.4%/6.2%/17%/34% for 8/16/32/64-core area budget. Recall from Table 4 that the large core replaces 4 small cores, reducing the maximum number of threads by 3 for ACMP and FDP (because they run only one thread on the large core), and by 4 for ACS and BIS (because they dedicate the large core for critical sections and bottlenecks) with respect to SCMP. With a small area budget, this loss of parallel throughput cannot be offset by the benefit of acceleration because contention for bottlenecks is smaller at fewer threads. Therefore, with a small area budget, BIS performs similar to ACS/FDP and underperforms SCMP (e.g. for an 8-core area



(a) iplookup (b) mysql-1 (c) mysql-2 (d) mysql-3 (e) specjbb (f) sqlite

(g) tsp (h) webcache (i) mg (j) ft (k) rank (l) pagemine

Figure 6. Speedup over a single small core, x-axis is area budget (in equivalent small cores).

budget). As the area budget increases, the marginal loss of parallel throughput due to the large core becomes relatively smaller, and with more threads running, contention for serializing bottlenecks increases. Therefore, the dynamic mechanisms that accelerate those bottlenecks (ACS, FDP and BIS) start to prevail.

Second, with more threads, the benefit of BIS over ACS/FDP increases on workloads where the critical bottlenecks change over time (*iplookup*, all *mysql*, *webcache*, *mg*, *ft* and *rank*), because BIS is better at choosing which bottlenecks to accelerate and quickly adapts to changes. Additionally, both critical sections and barriers are limiting bottlenecks in *mg* and *ft* at different times, and BIS accelerates both types of bottlenecks, unlike ACS. On workloads where the critical bottlenecks do not change over time (*specjbb*, *sqlite* and *tsp*), BIS matches the performance of ACS. *Pagemine* has two very similar non-scalable pipeline stages (the reduction of local histograms into a global histogram) that are frequently the limiting bottlenecks. Therefore, BIS with a single large core is unable to significantly improve performance over FDP because that would require both stages to be simultaneously accelerated. However, BIS is able to improve performance when multiple large cores are employed, as we will show in Section 5.2.

Third, BIS increases the scalability (the number of threads at which performance peaks) of four of the workloads (all versions of *mysql* and *rank*) beyond the already-improved scalability of ACS/FDP over ACMP or SCMP. By accelerating bottlenecks more effectively than previous proposals, BIS reduces contention and thread waiting, allowing the workloads to take advantage of a larger number of threads before performance gets limited by bottlenecks.

We conclude that BIS improves performance and scalability of applications over ACS/FDP. As the area budget increases, so does the performance benefit of BIS.

5.1.2 Optimal Number of Threads

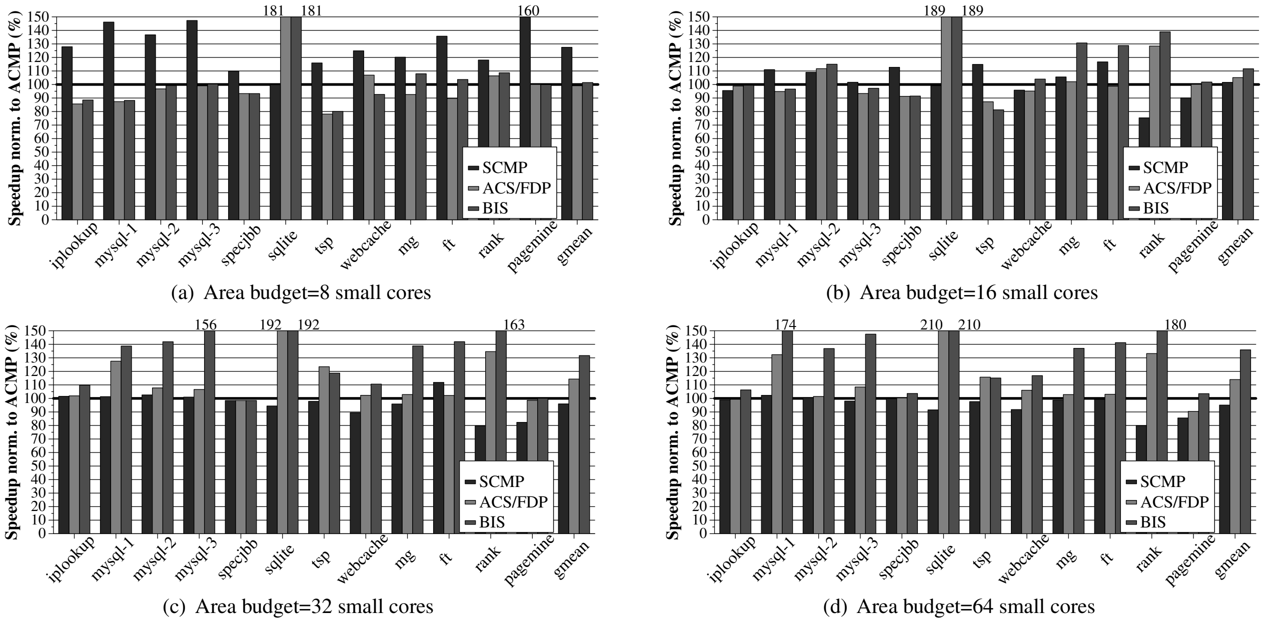
Figure 7 shows the speedups of our proposal for different area budgets with the optimal number of threads, relative to the ACMP configuration. We determine the optimal number of threads for each application-configuration pair by simulating all possible numbers of threads and reporting results for the one that minimizes execution time. For example, the optimal number of threads for iplookup-BIS on the 32-core area budget is 20.

With an area budget of 8 cores, all benchmarks except *sqlite* run with the maximum number of threads (8 for SCMP, 5 for ACMP/FDP and 4 for ACS and BIS) and with low contention for bottlenecks. Therefore, SCMP performs the best because the loss of throughput on the configurations with a large core is greater than the benefit of acceleration. The exception is *sqlite*, which becomes critical-section limited at 4 threads and can take full advantage of the acceleration mechanisms. Since *sqlite* has only one dominant critical section during the whole execution, both ACS and BIS provide the same performance improvement. On average, ACS, BIS and ACMP perform within 2% of each other.

With higher area budgets, the relative impact of running fewer threads due to the presence of a large core becomes less significant, which progressively reduces the advantage of SCMP and favors ACMP, ACS/FDP and BIS. On average, BIS outperforms SCMP by 10%/37%/43%, ACMP by 12%/32%/36% and ACS/FDP by 6.2%/15%/19% on a 16/32/64-core budget. As contention increases, it becomes more important to focus the acceleration effort on the *most critical* bottlenecks. While BIS actively identifies and accelerates the bottlenecks that are stopping more threads from making progress, ACS disables acceleration of critical sections that suffer from false serialization, even though such critical sections can be causing the most thread waiting. Additionally, BIS is more adaptive to changes in the important bottlenecks, while both ACS and FDP maintain the decisions they make for a fixed and relatively long period (10Mcycles for ACS and until the next major phase change for FDP). As a result of these two factors, BIS significantly outperforms ACS and FDP.

The only benchmark where BIS does not outperform ACS is *tsp* because BIS accelerates many more bottlenecks than ACS, increasing core-to-core cache misses. Since those bottlenecks are only 52-instruction long on average, the benefit of accelerating them does not overcome the cache miss penalty, which causes BIS' performance to be lower than that of ACS. BIS is able to improve performance of *tsp* over ACS with the Data Marshaling mechanism [35] that reduces cache misses incurred by the bottlenecks, as we show in Section 5.4.

We conclude that even with optimal number of threads, BIS improves performance and its benefit over all other mechanisms increases as the number of cores (area budget) increases.



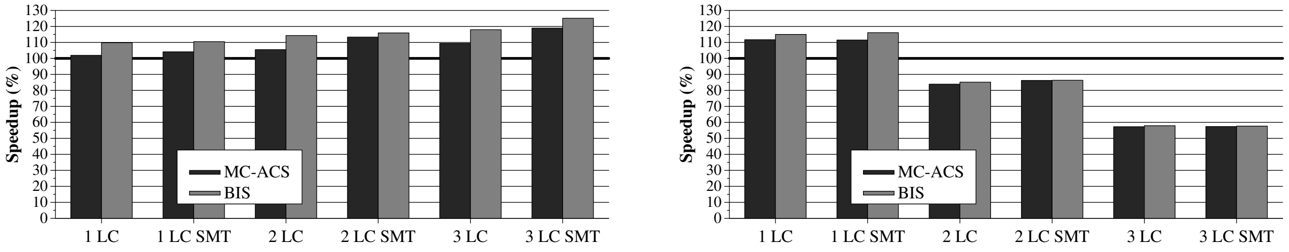
(a) Area budget=8 small cores

(b) Area budget=16 small cores

(c) Area budget=32 small cores

(d) Area budget=64 small cores

Figure 7. Speedup for optimal number of threads, normalized to ACMP.



(a) iplookup at 32-core area budget

(b) mysql-2 at 16-core area budget

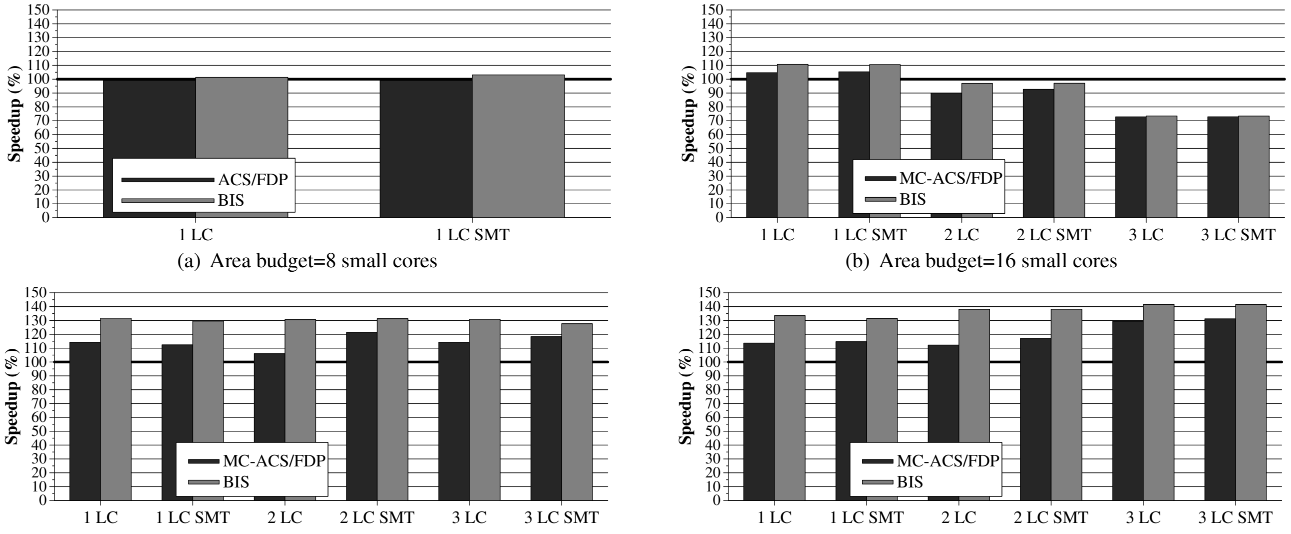
Figure 8. Speedup with multiple large core contexts, normalized to ACMP with a single large core (LC: large core).

5.2 Multiple Large Core Contexts

We first explain the trade-offs of multiple large cores with two examples. Figure 8(a) shows the performance of *iplookup* with optimal number of threads as we increase the number of large cores while keeping the same area budget. The figure also shows results for large cores with 2-way Simultaneous Multithreading. *Iplookup* executes a large number of independent critical sections that are contended. Therefore, it can significantly benefit from extra large cores that can simultaneously accelerate more of those equally-critical bottlenecks. The performance improvement of BIS over ACMP steadily grows from 10% for one large core to 25% with three large 2-way SMT cores. On the other hand, Figure 8(b) shows that for *mysql-2* additional large cores decrease performance with MC-ACS or BIS because the benefit of simultaneously accelerating multiple critical sections that are not heavily contended is much smaller than the cost of reducing the number of threads that can be executed concurrently to accommodate the additional large cores. The conclusion is that the best number of large cores dedicated to bottleneck acceleration significantly depends on the application and number of threads. Therefore, achieving the best performance requires either software tuning to tailor bottleneck behavior to what the available hardware configuration can efficiently accelerate, or building reconfigurable "large" cores that can either run more threads with lower performance or a single thread with higher performance. Both are interesting avenues for future work.

Figure 9 shows the geometric mean of speedups for all of our benchmarks on different core configurations with the same area budget, ranging from an area budget of 8 small cores to 64 small cores. The workloads run with the optimal number of threads for each configuration. An area budget of 8 small cores only allows for a single-large-core ACMP configuration, and on average there is not much contention for bottlenecks that either ACS/FDP or BIS can provide benefit for. With an area budget of 16 cores, performance decreases with additional large cores because the benefit of acceleration on the extra large cores is not enough to compensate for the loss of parallel performance due to the reduced number of threads. When the area budget is 32 cores, the average performance benefit of BIS remains stable for different numbers of large cores. With an area budget of 64 cores, there is no loss of parallel throughput with 2 or 3 large cores, because Figure 6 shows that performance peaks at fewer than 40 threads on all workloads. Therefore, BIS' performance increases by 3.4% and 6% with 2 and 3 large cores, respectively, with respect to a single large core. We conclude that having more large cores increases the benefit of BIS at large area budgets.

Having SMT on the large core does not change BIS' performance by more than 2% in any configuration. The trade-off is that a 2-way SMT large core can accelerate two different bottlenecks concurrently, but the two hardware threads share all processor resources and therefore the acceleration benefit of each thread is lower than if the thread were running alone on the large core. Two effects reduce the acceleration benefit when SMT is used on the large core. First, since local data is frequently accessed inside critical sections and pipeline stages and stays resident in the cache, our



(a) Area budget=8 small cores

(b) Area budget=16 small cores

(c) Area budget=32 small cores

(d) Area budget=64 small cores

Figure 9. Speedup with multiple large core contexts, normalized to ACMP with a single large core (LC: large core).

workloads rarely need to access main memory (average L3 MPKI is 0.79 on BIS with the 32-core budget), hence SMT's main benefit on an out-of-order core, cache miss latency tolerance, rarely gets exercised. Second, tight resource sharing between threads degrades performance of each thread. We conclude that using SMT to increase the number of large core contexts available for bottleneck acceleration is not useful for BIS.

5.3 Analysis

Figure 10 shows the normalized execution time spent accelerating code segments on and off the critical path for ACS/FDP and BIS on a 32-core area budget running 28 threads. To avoid changing the critical path on the fly due to acceleration, we disable acceleration and only perform bottleneck identification in this experiment. Then the actual critical path is computed off-line and compared to the code segments identified as bottlenecks by ACS/FDP and BIS. We define *coverage* as the fraction of the critical path that is actually identified as bottleneck. BIS matches or improves coverage of ACS/FDP on every benchmark and the average coverage of BIS (59%) is significantly better than the average coverage of ACS/FDP (39%). We define *accuracy* as the fraction of execution time of identified bottlenecks that is actually on the critical path. The average accuracy of BIS (73.5%) is slightly better than that of ACS/FDP (72%). We conclude that the significantly better coverage and similar accuracy of BIS vs. ACS/FDP results in the performance improvements we showed in Sections 5.1 and 5.2.

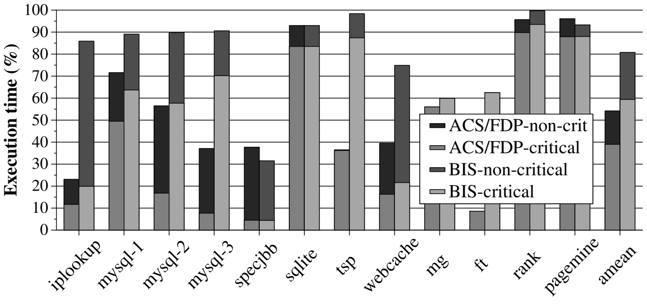


Figure 10. Bottleneck identification on and off the critical path.

Table 8 shows the number of dynamic instances of annotated bottlenecks that are executed and identified for acceleration, along with their average length in number of instructions. Bottlenecks are most numerous and only 52-instruction long on *tsp*, which increases the overhead since even a single cache-to-cache transfer due to migration introduces a penalty in the same order of magnitude as the benefit of acceleration, resulting in a performance loss unless Data Marshaling is used, as we will show next.

	Annotated Bottlenecks		Average Length
Workload	Executed	Identified	(Instructions)
iplookup	19184	888 (4.6%)	2130
mysql-1	18475	9342 (50.6%)	391
mysql-2	11761	1486 (12.6%)	378
mysql-3	25049	2593 (10.4%)	453
specjbb	1652	1008 (61.0%)	1468
sqlite	174	116 (66.7%)	29595
tsp	21932	12704 (57.9%)	52
webcache	15118	2099 (13.9%)	1908
mg	516	515 (99.8%)	11585
ft	228	227 (99.6%)	39804
rank	3003	1024 (34.1%)	1800
pagemine	5005	1030 (20.6%)	1160

Table 8. Bottleneck characterization.

5.4 Interaction with Data Marshaling

Bottlenecks that are accelerated on a large core may need to access data that was produced in the code segment running on the small core before execution migrated to the large core. These *inter-segment* memory accesses become cache misses that may limit the performance benefit of acceleration. To reduce these cache misses, Suleman et al. [35] proposed Data Marshaling (DM), which can be easily integrated with BIS. DM identifies and marshals the data needed by a bottleneck to the large core using compiler and hardware support. The compiler uses profiling to find the store instructions that generate the data to be produced in the small core and consumed by a bottleneck. The hardware keeps track of the cache lines modified by those special store instructions and pushes those cache lines into the large core's cache as soon as the bottleneck is shipped to the large core. When the bottleneck starts executing, after waiting in the Scheduling Buffer, the inter-segment data is likely to be available in the large core's cache.

Figure 11 shows performance of BIS and ACS/FDP with and without DM on 32-core area budget with 28 threads, normalized to ACMP. In general, DM improves ACS and BIS, but it improves BIS more, because BIS migrates bottlenecks to a large core more frequently than ACS. In particular, BIS underperforms ACS on *tsp*

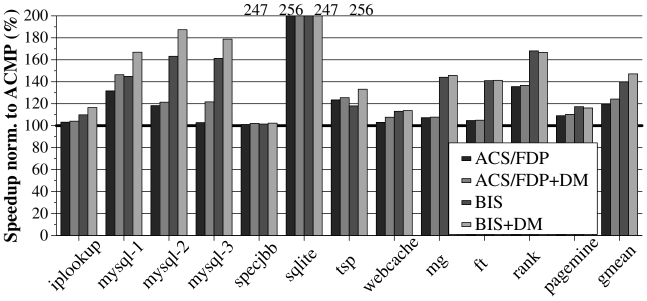


Figure 11. Speedup with and without Data Marshaling.

due to inter-segment cache misses, which are eliminated by DM, which results in BIS+DM outperforming ACS+DM. On average on the 32-core budget, BIS+DM improves performance by 5.2% over BIS, while ACS+DM improves by 3.8% over ACS. We conclude that BIS interacts positively with techniques that can reduce cache misses incurred due to bottleneck acceleration.

6. Related Work

Our proposal is a comprehensive mechanism to identify and accelerate bottlenecks (using heterogeneous cores) in a multithreaded program. To our knowledge, this is the first proposal that reduces thread waiting caused by bottlenecks regardless of bottleneck type. The mechanism is unique as it requires no significant programmer support and uses the fast cores on the ACMP to accelerate the bottlenecks. Most related to our work are previous proposals to reduce the performance impact of bottlenecks and other techniques to accelerate specific bottlenecks using heterogeneous cores.

6.1 Reducing the Impact of Bottlenecks

Removing or alleviating serializing bottlenecks has been a topic of wide interest for decades. For example, test-and-test-and-set [31] was proposed to reduce the serialization caused by thread communication. These primitives are only used at the start and the end of the bottleneck (lock acquire/release). As such, they are orthogonal to BIS since BIS accelerates the code inside the bottleneck.

Several mechanisms have been proposed to overlap execution of bottlenecks as long as they do not modify shared data. The first of these approaches is Transactional Memory (TM) [12], which targets critical sections. TM's benefit heavily depends on bottlenecks not having data conflicts, i.e. if two instances of a transaction have a data conflict, TM cannot execute them in parallel. In contrast, BIS speeds up bottlenecks regardless of their data access patterns. BIS can also complement TM as it can be used to accelerate transactions that cannot be overlapped due to their length or conflicts.

Other proposals like Speculative Lock Elision [29], Transactional Lock Removal [30], and Speculative Synchronization (SS) [24] can overlap the execution of non-conflicting instances of the same critical section without requiring programmers to write code using transactions. In addition to critical sections, SS can also reduce the overhead of barriers by speculatively allowing threads to execute past the barrier as long as there are no data conflicts. These approaches fundamentally differ from BIS for four reasons. First, they cannot hide the latency of bottlenecks that have data conflicts, whereas BIS can accelerate such bottlenecks. Second, they best apply to short bottlenecks as long bottlenecks complicate conflict detection hardware and are more likely to conflict, while BIS can accelerate arbitrarily long bottlenecks. Third, these techniques do not take criticality of the bottleneck into account, which is a main feature of BIS. Fourth, they are limited to particular types of bottlenecks (e.g., these approaches cannot be applied to bottleneck pipeline stages), whereas BIS is more general. BIS can complement these proposals by accelerating bottlenecks to which they are not applicable.

Previously proposed techniques [15, 37] to reduce load imbalance among stages of a parallel pipeline do so by either re-pipelining the workload via compiler/programmer support or by choosing the thread-to-stage allocation to reduce imbalance. Their performance benefit is limited because the decisions are coarse-grain and they are unable to speedup stages that do not scale with the number of cores. In BIS, we can leverage these previous mechanisms to manage our thread-to-core allocation but we also identify and accelerate the critical bottlenecks using a large core at a fine granularity. Using the faster cores allows us to improve performance even if the bottleneck pipeline stage does not scale.

In general, our proposal is complementary to mechanisms that reduce the impact of bottlenecks. Evaluating their interactions is part of our future work.

6.2 Accelerating Bottlenecks

Annavaram et al. [5] proposed to make the core running the serial bottleneck faster using frequency throttling. Morad et al. [26] proposed accelerating Amdahl's serial bottleneck [4] using the fast core of an ACMP while running the parallel phases on the small cores. While accelerating the single-threaded bottleneck is provably beneficial for performance, it does not reduce serialization in parallel portions of programs. We use their mechanism as a baseline and provide significant performance improvement on top of it by also accelerating bottlenecks in the parallel portions.

Suleman et al. [36] use the faster cores to accelerate critical sections. Their scheme differs from ours because (1) it is limited to critical sections while BIS targets thread waiting for any bottleneck, (2) it does not consider critical section criticality when deciding which critical sections to accelerate while BIS prioritizes bottlenecks that are predicted to be most critical, and (3) it only supports a single large core (which can execute only a single bottleneck) while BIS can schedule multiple bottlenecks for acceleration on any arbitrary number of fast cores. The generality and fine-grained adaptivity of BIS allow it to significantly outperform ACS, as we showed in Section 5.

Meeting Points [8], Thread Criticality Predictors (TCP) [7] and Age-Based Scheduling [22] try to reduce the overhead of barriers by detecting and accelerating the threads which are likely to reach the barrier last. Age-Based Scheduling uses history from the previous instance of the loop to choose the best candidates for acceleration. Thus, their scheme is restricted to iterative kernels that have similar behavior across multiple invocations. TCP uses a combination of L1 and L2 cache miss counters to predict which threads are likely to arrive early at a barrier and slows them down to save power, but it can also be used to guide acceleration of the thread that is lagging. Meeting Points uses hints from the software to determine which thread is lagging (thus likely to be a bottleneck). We can leverage these schemes to enhance our bottleneck identification mechanism, because they may be able to identify threads that need acceleration before they start making other threads wait, allowing their acceleration to start earlier and be more effective.

6.3 Critical Path Prediction

Hollingsworth [14] was the first to propose an online algorithm to compute the critical path of a message-passing parallel program. Fields et al. [10, 11] proposed a data- and resource-dependency-based token-passing algorithm that predicts instruction criticality and slack in single-threaded applications. In contrast, our approach predicts criticality at the code segment (i.e., bottleneck) granularity in multithreaded applications by only focusing on thread waiting. Dependencies between instructions in a single-threaded application and between bottlenecks in a multithreaded application are conceptually different. While instructions obey well-defined data-flow dependencies, bottlenecks have more complicated and dynamic dependencies. For example, critical sections with the same lock can execute in any order as long as they do not execute concurrently, e.g., block A may wait for block B in one dynamic instance while block B may wait for block A in another instance of execution, depending on the timing of each thread's execution. With many threads, dependencies between bottlenecks become more complicated and their execution order becomes less repeatable. Therefore, we found that methods to predict instruction criticality in single-threaded applications similar to Fields et al. [11] are not straightforwardly applicable to predict bottlenecks in multithreaded applications, although future work may find a way of doing it. Li et al. [23] proposed an efficient *off-line* DAG-based algorithm to find instruction-level criticality in multithreaded applications. Our problem is different in that it requires *online* prediction of critical bottlenecks to enable their acceleration.

6.4 Coarse-Grain Thread Scheduling

Several proposals schedule threads to large cores of an ACMP at coarse granularity, i.e. the operating system level. For example, Saez et al. [32] proposed running on large cores the threads that can make better use of them, according to a utility factor that considers both memory-intensity and thread-level parallelism. Koufaty et al. [19] proposed assigning a large core to the thread that is expected to have a larger speedup running on the large core, i.e. those that have fewer off-core memory requests and fewer core front-end stalls. In contrast, BIS does fine-grain code segment scheduling in hardware, running only serial and parallel bottlenecks on large cores, i.e. the portions of *any thread* that are adaptively identified as performance limiters.

7. Conclusion

We propose *Bottleneck Identification and Scheduling* (BIS), the first generalized mechanism to identify the most critical bottlenecks that cause thread waiting on multithreaded applications and to accelerate those bottlenecks using one or more large cores of an ACMP. We show that BIS improves performance of a set of bottleneck-intensive applications on average by 15% over ACS, which is the state-of-the-art technique for critical section acceleration, and FDP, a software technique to dynamically assign pipeline stages to cores, on a 1-large-core, 28-small-core ACMP. The benefit of BIS increases with the number of cores and BIS improves scalability of 4 out of 12 workloads compared to ACS/FDP. We are also the first to propose bottleneck acceleration using multiple large cores and find that performance improves with multiple large cores since there are independent bottlenecks that need to be simultaneously accelerated. We conclude that BIS provides a comprehensive approach to fine-grain bottleneck acceleration on future ACMPs that can help existing and future applications take advantage of a larger number of cores while reducing programmer effort.

Acknowledgments

We thank Eiman Ebrahimi, Veynu Narasiman, Santhosh Srinath, other members of the HPS research group, our shepherd Ras Bodik and the anonymous reviewers for their comments and suggestions. We gratefully acknowledge the support of the Cockrell Foundation, Intel Corporation, Oracle Corporation and the Gigascale Systems Research Center. José Joao was supported by an Intel PhD Fellowship.

References

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

[2] SQLite database engine version 3.5.8. 2008.

[3] SysBench: a system performance benchmark v0.4.8. http://sysbench.sourceforge.net.

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

[5] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttling. In *ISCA*, 2005.

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

[7] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *ISCA*, 2009.

[8] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *PACT*, 2008.

[9] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt. Parallel application memory scheduling. In *MICRO*, 2011.

[10] B. Fields, R. Bodik, and M. Hill. Slack: maximizing performance under technological constraints. In *ISCA*, 2002.

[11] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *ISCA*, 2001.

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

[13] M. D. Hill and M. R. Marty. Amdahl's law in Multicore Era. Technical Report CS-TR-2007-1593, Univ. of Wisconsin, 2007.

[14] J. K. Hollingsworth. An online computation of critical path profiling. In *SIGMETRICS Symposium on Parallel and Distributed Tools*, 1996.

[15] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *PACT*, 2009.

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

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

[18] Intel. Getting Started with Intel Parallel Studio, 2011.

[19] D. Koufaty, D. Reddy, and S. Hahn. Bias scheduling in heterogeneous multi-core architectures. In *EuroSys*, 2010.

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

[21] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 2005.

[22] N. B. Lakshminarayana, J. Lee, and H. Kim. Age based scheduling for asymmetric multiprocessors. In *SC*, 2009.

[23] T. Li, A. R. Lebeck, and D. J. Sorin. Quantifying instruction criticality for shared memory multiprocessors. In *SPAA*, 2003.

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

[25] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *ASPLOS*, 1991.

[26] T. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *Computer Architecture Letters*, 2006.

[27] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A. Choudhary. MineBench: A benchmark suite for data mining workloads. In *IISWC*, 2006.

[28] Y. Nishitani, K. Negishi, H. Ohta, and E. Nunohiro. Implementation and evaluation of OpenMP for Hitachi SR8000. In *ISHPC*, 2000.

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

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

[31] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *ISCA*, 1984.

[32] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *EuroSys*, 2010.

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

[34] M. A. Suleman. *An Asymmetric Multi-core Architecture for Efficiently Accelerating Critical Paths in Multithreaded Programs*. PhD thesis, UT-Austin, 2010.

[35] M. A. Suleman, O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt. Data marshaling for multi-core architectures. *ISCA*, 2010.

[36] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. *ASPLOS*, 2009.

[37] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt. Feedback-directed pipeline parallelism. In *PACT*, 2010.

[38] Tornado Web Server. Source code. http://tornado.sourceforge.net/, 2008.

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

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