Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs

José A. Joao [†] M. Aater Suleman [‡†] Onur Mutlu [§] Yale N. Patt [†]

[†] ECE Department
The University of Texas at Austin
Austin, TX, USA
{joao, patt}@ece.utexas.edu

[‡] Flux7 Consulting
Austin, TX, USA
suleman@hps.utexas.edu

[§] Computer Architecture Laboratory
Carnegie Mellon University
Pittsburgh, PA, USA
onur@cmu.edu

ABSTRACT

Asymmetric Chip Multiprocessors (ACMPs) are becoming a reality. ACMPs can speed up parallel applications if they can identify and accelerate code segments that are critical for performance. Proposals already exist for using coarse-grained thread scheduling and fine-grained bottleneck acceleration. Unfortunately, there have been no proposals offered thus far to decide which code segments to accelerate in cases where both coarse-grained thread scheduling and fine-grained bottleneck acceleration could have value. This paper proposes Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs (UBA), a cooperative software/hardware mechanism for identifying and accelerating the most likely critical code segments from a set of multithreaded applications running on an ACMP. The key idea is a new Utility of Acceleration metric that quantifies the performance benefit of accelerating a bottleneck or a thread by taking into account both the criticality and the expected speedup. UBA outperforms the best of two state-of-the-art mechanisms by 11% for single application workloads and by 7% for two-application workloads on an ACMP with 52 small cores and 3 large cores.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)

General Terms

Design, Performance

Keywords

Multithreaded applications, critical sections, barriers, multicore, asymmetric CMPs, heterogeneous CMPs

1. INTRODUCTION

Parallel applications are partitioned into threads that can execute concurrently on multiple cores. Speedup is often limited when some threads are prevented from doing useful work concurrently because they have to wait for other code segments to finish. Asymmetric Chip Multi-Processors (ACMPs) with one or few large, fast cores and many small, energy-efficient cores have been proposed for accelerating the most performance critical code segments, which can lead to significant performance gains. However, this approach has heretofore had at least two fundamental limitations:

1. The problem of accelerating only one type of code segment. There are two types of code segments that can become performance limiters: (1) threads that take longer to execute than other threads because of load imbalance or microarchitectural mishaps such as cache misses, and (2) code segments, like contended critical sections, that make other threads wait. We call threads of the first type *lagging threads*. They increase execution time since the program cannot complete until all its threads have finished execution. Code segments of the second type reduce parallelism and can potentially become the critical path of the application. Joao et al. [10] call these code segments *bottlenecks*. Prior work accelerates *either* lagging threads [6, 5, 13] *or* bottlenecks [24, 10], but not both. Thus, these proposals benefit only the applications whose performance is limited by the type of code segments that they are designed to accelerate. Note that there is overlap between lagging threads and bottlenecks: lagging threads, if left alone, can eventually make other threads wait and become bottlenecks. However, the goal of the proposals that accelerate lagging threads is to try to prevent them from becoming bottlenecks.

Real applications often have both bottlenecks and lagging threads. Previous acceleration mechanisms prove suboptimal in this case. Bottleneck Identification and Scheduling (BIS) [10] does not identify lagging threads early enough and as a result it does not always accelerate the program's critical execution path. Similarly, lagging thread acceleration mechanisms do not accelerate consecutive instances of the same critical section that execute on different threads and as a result can miss the opportunity to accelerate the program's critical execution path. Combining bottleneck and lagging thread acceleration mechanisms is non-trivial because the combined mechanism *must* predict the relative benefit of accelerating bottlenecks and lagging threads. Note that this benefit depends on the input set and program phase, as well as the underlying machine. Thus a static solution would likely not work well. While the existing acceleration mechanisms are dynamic, they use different metrics to identify good candidates for acceleration; thus, their outputs cannot be compared directly to decide which code segments to accelerate.

2. The problem of not handling multiple multithreaded applications. In practice, an ACMP can be expected to run multiple multithreaded applications. Each application will have lagging threads and bottlenecks that benefit differently from acceleration. The previous work on

ISCA'13 Tel-Aviv, Israel

bottleneck acceleration [24, 10] and lagging thread acceleration [13] does not deal with multiple multithreaded applications, making their use limited in practical systems.

To make ACMPs more effective, we propose *Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs* (UBA). UBA is a general cooperative software/hardware mechanism to identify the most important code segments from one or multiple applications to accelerate on an ACMP to improve system performance. UBA introduces a new *Utility of Acceleration* metric for each code segment, either from a lagging thread or a bottleneck, which is used to decide which code segments to run on the large cores of the ACMP. The key idea of the utility metric is to consider both the acceleration expected from running on a large core and the criticality of the code segment for its application as a whole. Therefore, this metric is effective in making acceleration decisions for both single- and multiple-application cases. UBA also builds on and extends previous proposals to identify potential bottlenecks [10] and lagging threads [13].

This paper makes three main **contributions**:

1. It introduces a new Utility of Acceleration metric that combines a measure of the acceleration that each code segment achieves, with a measure of the criticality of each code segment. This metric enables meaningful comparisons to decide which code segments to accelerate regardless of the segment type. We implement the metric in the context of an ACMP where acceleration is performed with large cores, but the metric is general enough to be used with other acceleration mechanisms, e.g., frequency scaling.
2. It provides the first mechanism that can accelerate both bottlenecks and lagging threads from a single multithreaded application, using faster cores. It can also leverage ACMPs with any number of large cores.
3. It is the first work that accelerates bottlenecks in addition to lagging threads from multiple multithreaded applications.

We evaluate UBA on single- and multiple-application scenarios on a variety of ACMP configurations, running a set of workloads that includes both bottleneck-intensive applications and non-bottleneck-intensive applications. For example, on a 52-small-core and 3-large-core ACMP, UBA improves average performance of 9 multithreaded applications by 11% over the best of previous proposals that accelerate only lagging threads [13] or only bottlenecks [10]. On the same ACMP configuration, UBA improves average harmonic speedup of 2-application workloads by 7% over our aggressive extensions of previous proposals to accelerate multiple applications. Overall, we find that UBA significantly improves performance over previous work and its performance benefit generally increases with larger area budgets and additional large cores.

2. MOTIVATION

2.1 Bottlenecks

Joao et al. [10] defined *bottleneck* as any code segment that makes other threads wait. Bottlenecks reduce the amount of thread-level parallelism (TLP); therefore, a program running with significant bottlenecks can lose some or even all of the potential speedup from parallelization. Inter-thread synchronization mechanisms can create bottlenecks, e.g., contended critical sections, the last thread arriving to a barrier and the slowest stage of a pipeline-parallel program.

Figure 1 shows four threads executing non-critical-section segments (Non-CS) and a critical section CS (in gray). A critical section enforces mutual exclusion: only one thread can execute the critical section at a given time, making any other threads wanting to execute the same critical section wait, which reduces the amount of useful work that can be done in parallel.

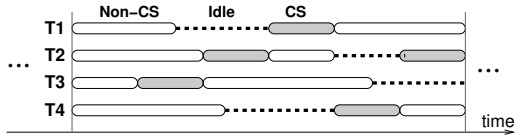


Figure 1: Example of a critical section.

The state-of-the-art in bottleneck acceleration on an ACMP is BIS [10], which consists of software-based annotation of potential bottlenecks and hardware-based tracking of *thread waiting cycles*, i.e., the number of cycles threads waited for each bottleneck. Then, BIS accelerates the bottlenecks that are responsible for the most thread waiting cycles. BIS is effective in accelerating critical sections that limit performance at different times. However, it accelerates threads arriving last to a barrier and slow stages of a pipeline-parallel program *only after* they have started making other threads wait, i.e., after accumulating a minimum number of thread waiting cycles. If BIS could start accelerating such lagging threads earlier, it could remove more thread waiting and further reduce execution time.

2.2 Lagging Threads

A parallel application is composed of groups of threads that split work and eventually either synchronize at a barrier, or finish and join. The thread that takes the most time to execute in a thread group determines the execution time of the entire group and we call that thread a *lagging thread*. Thread imbalance can appear at runtime for multiple reasons, e.g., different memory behavior that makes some threads suffer from higher average memory latency, and different contention for critical sections that makes some threads wait longer.

Figure 2 shows execution of four threads over time. Thread T2 becomes a lagging thread as soon as it starts making slower progress than the other threads towards reaching the barrier at time t2. Note that at time t1, T2 becomes the last thread running for the barrier and becomes a bottleneck. Therefore, lagging threads are potential future bottlenecks, i.e., they become bottlenecks if thread imbalance is not corrected in time. Also note that if there are multiple threads with approximately as much remaining work to do as the most lagging thread, all of them need to be accelerated to actually reduce total execution time. Therefore, all those threads have to be considered lagging threads.

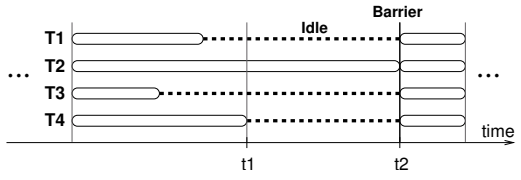


Figure 2: Example of a lagging thread (T2).

The state-of-the-art in acceleration of lagging threads are proposals that identify a lagging thread by tracking either thread progress [6, 13] or reasons for a thread to get delayed [5]. Meeting Points [6] tracks the threads that are

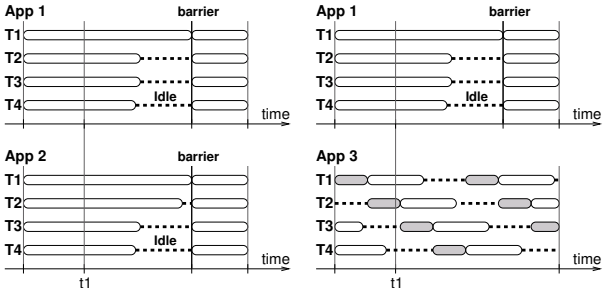
lagging in reaching a barrier by counting the number of loop iterations that have been completed. Thread Criticality Predictors [5] predict that the threads that suffer from more cache misses will be delayed and will become critical. Age-based Scheduling [13] accelerates the thread that is predicted or profiled to have more remaining work until the next barrier or the program exit, measured in terms of committed instructions. Once a lagging thread is identified, it can be accelerated on a large core of an ACMP.

2.3 Applications have both Lagging Threads and Bottlenecks

Joao et al. [10] showed that different bottlenecks can limit performance at different times. In particular, contention for different critical sections can be very dynamic. It is not evident upfront whether accelerating a critical section or a lagging thread leads to better performance. Therefore, it is fundamentally important to dynamically identify the code segments, either bottlenecks or lagging threads, that have to be accelerated at any given time.

2.4 Multiple Applications

Figure 3(a) shows two 4-thread applications running on small cores of an ACMP with a single large core. Let's assume that at time t1 the system has to decide which thread to accelerate on the large core to maximize system performance. With knowledge of the progress each thread has made towards reaching the next barrier, the system can determine that App1 has one lagging thread T1, because T1 has significantly more remaining work to do than the other threads, and App2 has two lagging threads T1 and T2, because both of them have significantly more work to do than T3 and T4. Accelerating T1 from App1 would directly reduce App1's execution time by some time Δt, while accelerating either T1 or T2 from App2 would not significantly reduce App2's execution time. It is necessary to accelerate both T1 during one quantum and T2 during another quantum to reduce App2's execution time by a similar Δt, assuming the speedups for all threads on the large core are similar. Therefore, system performance will increase more by accelerating T1 from App1.



(a) Lagging threads (b) Lagging threads and critical sections

Figure 3: Examples of lagging threads and critical sections.

Figure 3(b) shows the same App1 from the previous example and an App3 with a strongly-contended critical section (in gray). Every thread from App3 has to wait to execute the critical section at some time and the critical section is clearly on the critical path of execution (there is always one thread executing the critical section and at least one thread waiting for it). At time t1, App1 has a single lagging thread T1, which is App1's critical path. Therefore, every cycle saved by accelerating T1 from App1 would directly reduce App1's execution time. Similarly, every cycle saved by accelerating instances of the critical section from App3 on any of its threads would directly reduce App3's execution time. Ideally, the system should dynamically accelerate the code segment that gets a higher speedup from the large core, either a segment of the lagging thread from App1 or a sequence of instances of the critical section from App3.

These two examples illustrate that acceleration decisions need to consider both the criticality of code segments and how much speedup they get from the large core.

Our goal is to design a mechanism that decides which code segments, either from lagging threads or bottlenecks, to run on the available large cores of an ACMP, to improve system performance in the presence of a single multithreaded application or multiple multithreaded applications.

3. UTILITY-BASED ACCELERATION (UBA)

The core of UBA is a new Utility of Acceleration metric that is used to decide which code segments to accelerate at any given time. Utility combines an estimation of the acceleration that each code segment can achieve on a large core, and an estimation of the criticality of the code segment.

Figure 4 shows the three main components of UBA: Lagging Thread Identification unit, Bottleneck Identification unit, and Acceleration Coordination unit. Every scheduling quantum (1M cycles in our experiments), the Lagging Thread Identification (LTI) unit produces the set of Highest-Utility Lagging Threads (HULT), one for each large core, and the Bottleneck Acceleration Utility Threshold (BAUT). Meanwhile, the Bottleneck Identification (BI) unit computes the Utility of accelerating each of the most important bottlenecks and identifies those with Utility greater than BAUT, which we call Highest-Utility Bottlenecks (HUB). Only these bottlenecks are enabled for acceleration. Finally, the Acceleration Coordination (AC) unit decides which code segments to run on each large core, either a thread from the HULT set, or bottlenecks from the HUB set.

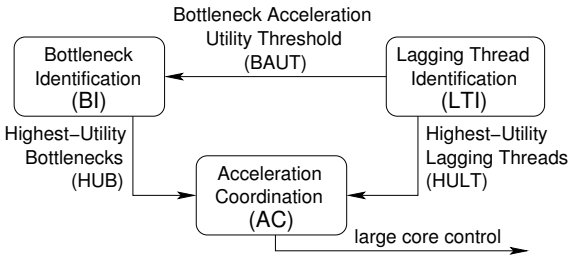


Figure 4: Block diagram of UBA.

3.1 Utility of Acceleration

We define *Utility of Accelerating a code segment c* as the reduction in the application's execution time due to acceleration of c relative to the application's execution time before acceleration. Formally,

$$U^c = \frac{\Delta T}{T}$$

where ΔT is the reduction in the *entire* application's execution time and T is the original execution time of the *entire* application.

If code segment c of length t cycles is accelerated by Δt, then after multiplying and dividing by t Δt, Utility of accelerating c can be rewritten as:

$$U^c = \frac{\Delta T}{T} = (\frac{\Delta t}{t})\ (\frac{t}{T})\ (\frac{\Delta T}{\Delta t}) = L \times R \times G$$

L: The first factor is the *Local Acceleration*, which is the reduction in the execution time of *solely* the code segment c due to running on a large core divided by the original execution time of c on the small core.

$$L = \frac{\Delta t}{t}$$

L depends on the net speedup that running on a large core can provide for the code segment, which is a necessary condition to improve the application's performance: if L is close to zero or negative, running on the large core is not useful and can be harmful.

R: The second factor is the *Relevance of Acceleration*, which measures how important code segment c is for the application as a whole: R is the execution time (in cycles) of c on the small core divided by the application's execution time (in cycles) before acceleration.

$$R = \frac{t}{T}$$

R limits the overall speedup that can be obtained by accelerating a single code segment. For example, let's assume two equally long serial bottlenecks from two different applications start at the same time and can be accelerated with the same L factor. One runs for 50% of its application's execution time, while the other runs for only 1%. Obviously, accelerating the first one is a much more effective use of the large core to improve system performance.

G: The third factor is the *Global Effect of Acceleration*, which represents how much of the code segment acceleration Δt translates into a reduction in execution time ΔT.

$$G = \frac{\Delta T}{\Delta t}$$

G depends on the criticality of code segment c: if c is on the critical path, $G = 1$, otherwise $G = 0$. In reasonably symmetric applications, multiple threads may arrive to the next barrier at about the same time and all of them must be accelerated to reduce the application's execution time, which makes each of the threads partially critical ($0 < G < 1$).

We will explain how we estimate each factor L, R and G in Sections 3.5.1, 3.5.2 and 3.5.3, respectively.

3.2 Lagging Thread Identification

The set of Highest-Utility Lagging Threads (HULT) is produced every scheduling quantum (i.e., Q cycles) by the LTI unit with the following steps:

1. Identify lagging threads. We use the same notion of progress between consecutive synchronization points as in [13]; i.e., we assume approximately the same number of instructions are expected to be committed by each thread between consecutive barriers or synchronization points, and use instruction count as a metric of thread progress.[1] A committed instruction counter *progress* is kept as part of each hardware context. After thread creation or when restarting the threads after a barrier, *progress* is reset with a simple command *ResetProgress*, implemented as a store to a reserved memory location.

Figure 5 shows the *progress* of several threads from the same application that are running on small cores. Thread

[1]Note that we are not arguing for instruction count as the best progress metric. In general, the best progress metric is application dependent and we envision a mechanism that lets the software define which progress metric to use for each application, including [6, 13, 5] or even each application periodically reporting how much progress each thread is making. UBA can be easily extended to use any progress metric.

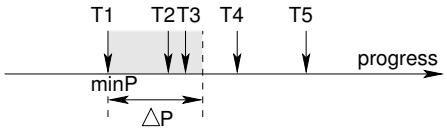


Figure 5: Lagging thread identification.

T1 has made the smallest progress *minP* and is a lagging thread. Let's assume that if thread T1 is accelerated during the next scheduling quantum, it will make ΔP more progress than the other non-accelerated threads. Therefore, T1 will leave behind threads T2 and T3, which then will also have to be accelerated to fully take advantage of the acceleration of T1. Therefore, we consider the initial set of lagging threads to be $\{T1, T2, T3\}$. We estimate ΔP as $AvgDeltaIPC \times Q$, where $AvgDeltaIPC$ is the difference between average IPC on the large cores and average IPC on the small cores across all threads in the application, measured over five quanta.

2. Compute Utility of Acceleration for each lagging thread. We will explain how UBA estimates each factor L, R and G in Sections 3.5.1, 3.5.2 and 3.5.3, respectively.

3. Find the Highest-Utility Lagging Thread (HULT) set. The HULT set consists of the lagging threads with the highest Utility. The size of the set is equal to the number of large cores. Lagging threads from the same application whose Utilities fall within a small range of each other[2] are considered to have the same Utility and are sorted by their *progress* instruction count (highest rank for lower *progress*) to improve fairness and reduce the impact of inaccuracies in Utility computations.

4. Determine the Bottleneck Acceleration Utility Threshold (BAUT). As long as no bottleneck has higher Utility than any lagging thread in the HULT set, no bottleneck should be accelerated. Therefore, the BAUT is simply the smallest Utility among the threads in the HULT set.

To keep track of the relevant characteristics of each thread, the LTI unit includes a **Thread Table** with as many entries as hardware contexts, indexed by software thread ID (tid).

3.3 Bottleneck Identification

The Highest-Utility Bottleneck set is continuously produced by the Bottleneck Identification (BI) unit. The BI unit is implemented similarly to BIS [10] with one fundamental change: instead of using BIS' thread waiting cycles as a metric to classify bottlenecks, the BI uses our Utility of Acceleration metric.[3]

Software support. 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 purpose of the BottleneckWait instruction is threefold: 1) it implements waiting for the value on a memory location to change, 2) it allows the hardware to keep track of which threads are waiting for each bottleneck and 3) it makes instruction count a more accurate measure of thread progress by removing the spinning loops that wait for synchronization and execute instructions that do not make progress.

Hardware support: Bottleneck Table. The hardware tracks which threads are executing or waiting for each bottleneck and identifies the critical bottlenecks with low overhead in hardware using a Bottleneck Table (BT) in the BI unit. Each BT entry corresponds to a bottleneck and collects all

[2]We find a range of 2% works well in our experiments.

[3]Our experiments (not shown due to space limitations) show BIS using Utility outperforms BIS using thread waiting cycles by 1.5% on average across our bottleneck-intensive applications.

the data required to compute the Utility of its acceleration.

Utility of accelerating bottlenecks. The Bottleneck Table computes the L factor once every quantum, as explained in Section 3.5.1. It recomputes the Utility of accelerating each bottleneck whenever its R (Section 3.5.2) or G (Section 3.5.3) factor changes. Therefore, Utility can change at any time, but it does not change very frequently because of how R and G are computed as explained later.

Highest-Utility Bottleneck (HUB) set. Bottlenecks with Utility above the Bottleneck Acceleration Utility Threshold (BAUT) are enabled for acceleration, i.e. they are part of the HUB set.

3.4 Acceleration Coordination

The candidate code segments for acceleration are the lagging threads in the HULT set, one per large core, provided by the LTI unit, and the bottlenecks in the HUB set, whose acceleration has been enabled by the BI unit.

Lagging thread acceleration. Each lagging thread in the HULT set is assigned to run on a large core at the beginning of each quantum. The assignment is based on affinity to preserve cache locality, i.e., if a lagging thread will continue to be accelerated, it stays on the same large core, and threads newly added to the HULT set try to be assigned to a core that was running another thread from the same application.

Bottleneck acceleration. When a small core executes 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 includes a local Acceleration Index Table (AIT) that caches the bottleneck ID (bid), acceleration enable bit and assigned large core for each bottleneck.[4] If acceleration is disabled, the small core executes the bottleneck locally. If acceleration is enabled, the small core sends a bottleneck execution request to the assigned large core and stalls waiting for a response. The large core enqueues the request into a Scheduling Buffer (SB), which is a priority queue based on Utility. The oldest instance of the bottleneck with highest Utility is executed by the large core until the BottleneckReturn instruction, at which point the large core sends a BottleneckDone signal to the small core. On receiving the BottleneckDone signal, the small core continues executing the instruction after the BottleneckCall.

The key idea of Algorithm 1, which controls each large core, is that each large core executes the assigned lagging thread, as long as no bottleneck is migrated to it to be accelerated. Only bottlenecks with higher Utility than the BAUT (the smallest Utility among all accelerated lagging threads, i.e., the HULT set) are enabled to be accelerated. Therefore, it makes sense for those bottlenecks to preempt a lagging thread with lower Utility.

The large core executes in one of two modes: accelerating the assigned lagging thread or accelerating bottlenecks from its Scheduling Buffer (SB), when they show up. After no bottleneck shows up for 50Kcycles, the assigned lagging thread is migrated back to the large core. This delay reduces the number of spurious lagging thread migrations, since bottlenecks like contended critical sections usually occur in bursts. The reasons to avoid finer-grained interleaving of lagging threads and bottlenecks are: 1) to reduce the impact of frequent migrations on cache locality and 2) to avoid excessive migration overhead. Both effects can significantly reduce or eliminate the benefit of acceleration.

[4]Initially all bottlenecks are assigned to the large core running the lagging thread with minimum Utility (equal to the BAUT threshold for bottleneck acceleration), but they can be reassigned as we will explain in Section 3.5.4. When a bottleneck is included in or excluded from the HUB set, the BT broadcasts the update to the AITs on all small cores.

Algorithm 1 Acceleration Coordination

```
while 1 do
  // execute a lagging thread
  migrate assigned lagging thread from small core
  while not bottleneck_in_SB do
    run assigned lagging thread
  end while
  migrate assigned lagging thread back to small core

  // execute bottlenecks until no bottleneck shows up for 50Kcycles
  done_with_bottlenecks = false
  while not done_with_bottlenecks do
    while bottleneck_in_SB do
      deque from SB and run a bottleneck
    end while
    delay = 0
    while not bottleneck_in_SB and (delay < 50Kcycles) do
      wait while incrementing delay
    end while
    done_with_bottlenecks = not bottleneck_in_SB
  end while
end while
```

3.5 Implementation Details

3.5.1 Estimation of L.

L is related to the speedup S due to running on a large core by:

$$L = \frac{\Delta t}{t} = \frac{t - t/S}{t} = 1 - \frac{1}{S}$$

Any existing or future technique that estimates performance on a large core based on information collected while running on a small core can be used to estimate S. We use Performance Impact Estimation (PIE) [27], the latest of such techniques. PIE requires measuring total cycles per instruction (CPI), CPI due to memory accesses, and misses per instruction (MPI) while running on a small core.

For code segments that are running on a large core, PIE also provides an estimate of the slowdown of running on a small core based on measurements on the large core. This estimation requires measuring CPI, MPI, average dependency distance between a last-level cache miss and its consumer, and the fraction of instructions that are dependent on the previous instruction (because they would force execution of only one instruction per cycle in the 2-wide in-order small core). Instead of immediately using this estimation of performance on the small core while running on the large core, our implementation remembers the estimated speedup from the last time the code segment ran on a small core, because it is more effective to compare speedups obtained with the same technique. After five quanta we consider the old data to be stale and we switch to estimate the slowdown on a small core based on measurements on the large core.

Each core collects data to compute L for its current thread and for up to two current bottlenecks to allow tracking up to one level of nested bottlenecks. When a bottleneck finishes and executes a BottleneckReturn instruction, a message is sent to the Bottleneck Table in the regular BIS implementation. We include the data required to compute L for the bottleneck on this message without adding any extra overhead. Data required to compute L for lagging threads is sent to the Thread Table at the end of the scheduling quantum.

3.5.2 Estimation of R.

Since acceleration decisions are made at least once every scheduling quantum, the objective of each decision is to

maximize Utility for one quantum at a time. Therefore, we estimate R (and Utility) only for the next quantum instead of for the whole run of the application, i.e., we use $T = Q$, the quantum length in cycles. During each quantum the hardware collects the number of active cycles, t_{lastQ}, for each thread and for each bottleneck to use as an estimate of the code segment length t for the next quantum. To that end, each Bottleneck Table (BT) entry and each Thread Table (TT) entry include an *active* bit and a *timestamp_active*. On the BT, the *active* bit is set between BottleneckCall and BottleneckReturn instructions. On the TT, the *active* bit is only reset while executing a BottleneckWait instruction, i.e., while the thread is waiting. When the *active* bit is set, *timestamp_active* is set to the current time. When the *active* bit is reset, the active cycle count t_{lastQ} is incremented by the difference between current time and *timestamp_active*.

Lagging thread activity can be easily inferred: a running thread is always active, except while running a BottleneckWait instruction.

$$R_{estimated} = \frac{t_{lastQ}}{Q} \qquad for\ lagging\ threads$$

Bottleneck activity is already reported to the Bottleneck Table for bookkeeping, off the critical path, after executing BottleneckCall and BottleneckReturn instructions. Therefore the *active* bit is set by BottleneckCall and reset by BottleneckReturn. Given that bottlenecks can suddenly become important, the Bottleneck Table also keeps an active cycle counter $t_{lastSubQ}$ for the last subquantum, an interval equal to 1/8 of the quantum. Therefore,

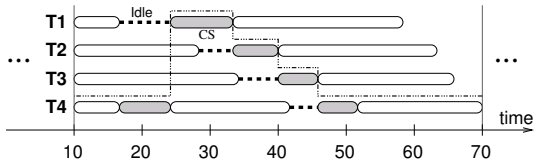
$$R_{estimated} = \max(\frac{t_{lastQ}}{Q}, \frac{t_{lastSubQ}}{Q/8}) \qquad for\ bottlenecks$$

3.5.3 Estimation of G.

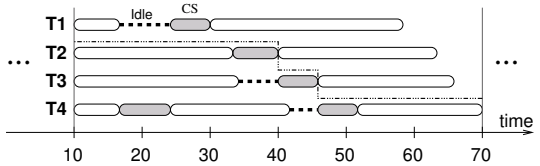
The G factor measures criticality of the code segment, i.e., how much of its acceleration is expected to reduce total execution time. Consequently, we estimate G for each type of code segment as follows:

Lagging threads. Criticality of lagging threads depends on the number of lagging threads in the application. If there are M lagging threads, all of them have to be evenly accelerated to remove the thread waiting they would cause before the next barrier or joining point. That is, all M lagging threads are potential critical paths with similar lengths. Therefore, $G_{estimated} = 1/M$ for each of the lagging threads. Amdahl's serial segments are part of the only thread that exists, i.e., a special case of lagging threads with $M = 1$. Therefore, each serial segment is on the critical path and has $G = 1$. Similarly, the last thread running for a barrier and the slowest stage of a pipelined program are also identified as single lagging threads and, therefore, have $G = 1$.

Critical sections. Not every contended critical section is on the critical path and high contention is not a necessary condition for being on the critical path. Let's consider two cases. Figure 6(a) shows a critical section that is on the critical path (dashed line), even though there is never more than one thread waiting for it. All threads have to wait at some time for the critical section, which makes the critical path jump from thread to thread following the critical section segments. We consider *strongly contended critical sections* those that have been making all threads wait in the recent past and estimate $G = 1$ for them. To identify strongly contended critical sections each Bottleneck Table entry includes a *recent_waiters* bit vector with one bit per hardware context. This bit is set on executing BottleneckWait for the corresponding bottleneck. Each Bottleneck Table entry also keeps a moving average for bottleneck length *avg_len*. If there are N active threads in the application, *recent_waiters* is evaluated every $N \times avg\_len$ cycles: if N bits are set, indicating that all active threads had to wait, the critical section is assumed to be strongly contended. Then, the number of ones in *recent_waiters* is stored in *past_waiters* (see the next paragraph) and *recent_waiters* is reset.



(a) Strongly contended critical section (all threads wait for it)



(b) Weakly contended critical section (T2 never waits for it)

Figure 6: Types of critical sections.

We call the critical sections that have not made all threads wait in the recent past *weakly contended critical sections* (see Figure 6(b)). Accelerating an instance of a critical section accelerates not only the thread that is executing it but also every thread that is waiting for it. If we assume each thread has the same probability of being on the critical path, the probability of accelerating the critical path by accelerating the critical section would be the fraction of threads that get accelerated, i.e., $G = (W + 1)/N$, when there are W waiting threads and a total of N threads. Since the current number of waiters W is very dynamic, we combine it with history (*past_waiters* from the previous paragraph). Therefore, we estimate G for weakly contended critical sections as $G_{estimated} = (max(W, past\_waiters) + 1)/N$.

3.5.4 False Serialization and Using Multiple Large Cores for Bottleneck Acceleration.

Instances of different bottlenecks from the same or from different applications may be accelerated on the same large core. Therefore, a bottleneck may get falsely serialized, i.e., it may have to wait for too long on the Scheduling Buffer for another bottleneck with higher Utility. Bottlenecks that suffer false serialization can be reassigned to a different large core, as long as their Utility is higher than that of the lagging thread assigned to run on that large core. Otherwise, a bottleneck that is ready to run but does not have a large core to run is sent back to its small core to avoid false serialization and potential starvation, as in BIS [10].

3.5.5 Reducing Large Core Waiting.

While a lagging thread is executing on a large core it may start waiting for several reasons. First, if the thread starts waiting for a barrier or is about to exit or be de-scheduled, the thread is migrated back to its small core and is replaced with the lagging thread with the highest Utility that is not running on a large core. Second, if the lagging thread starts waiting for a critical section that is not being accelerated, there is a situation where a large core is waiting for a small

Structure	Purpose	Location and entry structure (field sizes in bits in parenthesis)	Cost
Thread Table (TT)	To track threads, identify lagging threads and compute their Utility of Acceleration	LTI unit, one entry per HW thread (52 in this example). Each entry has 98 bits: tid(16), pid(16), is_lagging_thread(1), num_threads(8), timestamp_active(24), active(1), t_lastQ(16), Utility(16).	637 B
Bottleneck Table (BT)	To track bottlenecks [10] and compute their Utility of Acceleration	One 32-entry table on the BI unit. Each entry has 452 bits: bid(64), pid(16), executers(6), executer_vec(64), waiters(6), waiters_sb(6), large_core_id(2), PIE data(107), timestamp_active(24), active(1), t_lastQ(16), t_lastSubQ(13), timeoutG(24), avg_len(18), recent_waiters(64), past_waiters(6), Utility(16)	1808 B
Acceleration Index Tables (AIT)	To avoid accessing 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(2). Each AIT has 268 bytes.	13.6 KB
Scheduling Buffers (SB)	To store and prioritize bottleneck execution requests on each large core	One 52-entry buffer per large core. Each entry has 214 bits: *bid*(64), small core ID(6), target PC(64), stack pointer(64), Utility(16). Each SB has 1391 bytes.	4.1 KB
		Total	20.1 KB

Table 1: Hardware structures for UBA and their storage cost on an ACMP with 52 small cores and 3 large cores.

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: LC large cores, SC = N − 4 × LC small cores.	
ACMP [15, 16]	A large core always runs any single-threaded code. Max number of threads is $SC + LC$.
AGETS [13]	In each quantum, the large cores run the threads with more expected work to do. Max number of threads is $SC + LC$.
BIS [10]	The large cores run any single-threaded code and bottleneck code segments as proposed in [10]: 32-entry Bottleneck Table, each large core has an SC-entry Scheduling Buffer, each small core has a 32-entry Acceleration Index Table. Max number of threads is SC.
UBA	The large cores run the code segments with the highest Utility of Acceleration: BIS structures plus an SC-entry Thread Table. Max number of threads is SC.

Table 2: Baseline processor configuration.

core, which is inefficient. Instead, we save the context of the waiting thread on a shadow register alias table (RAT) and migrate the thread that is currently running the critical section from its small core to finish on the large core. Third, if the accelerated lagging thread wants to enter a critical section that is being accelerated on a different large core, it is migrated to the large core assigned to accelerate that critical section, to preserve shared data locality. Fourth, if acceleration of a critical section is enabled and there are threads waiting to enter that critical section on small cores, they are migrated to execute the critical section on the assigned large core. All these mechanisms are implemented as extensions of the behavior of the BottleneckCall and BottleneckWait instructions and use the information that is already on the Bottleneck Table.

3.5.6 Hardware Structures and Cost.

Table 1 describes the hardware structures required by UBA and their storage cost for a 52-small-core, 3-large-core ACMP, which is only 20.1 KB. UBA does not substantially increase storage cost over BIS, since it only adds the Thread Table and requires minor changes to the Bottleneck Table.

3.5.7 Support for Software-based Scheduling.

Software can directly specify lagging threads if it has better information than what is used by our hardware-based progress tracking. Software can also modify the quantum length Q depending on application characteristics (larger Q means less migration overhead, but also less opportunity to accelerate many lagging threads from the same application between consecutive barriers). Finally, software must be able to specify priorities for different applications, which would become just an additional factor in the Utility metric. Our evaluation does not include these features, and exploring them is part of our future work.

4. EXPERIMENTAL METHODOLOGY

We use an x86 cycle-level simulator that models asymmetric CMPs with small in-order cores modeled after the Intel Pentium processor and large out-of-order cores modeled after the Intel Core 2 processor. Our simulator faithfully models all latencies and core to core communication, including those due to execution migration. Configuration details are shown in Table 2. We compare UBA to previous work summarized in Table 3. Our comparison points for thread scheduling are based on two state-of-the-art proposals: Age-based Scheduling [13] (AGETS) and PIE [27]. We chose these baselines because we use similar metrics for progress and speedup estimation. Note that our baselines for multiple applications are aggressive extensions of previous proposals: AGETS combined with PIE to accelerate lagging threads, and an extension of BIS that dynamically shares all large cores among applications to accelerate any bottleneck based on relative thread waiting cycles.

We evaluate 9 multithreaded workloads with a wide range of performance impact from bottlenecks, as shown in Table 4. Our 2-application workloads are composed of all combinations from the 10-application set including the 9 multithreaded applications plus the compute-intensive *ft_nasp*, which is run with one thread to have a mix of single-threaded and multithreaded applications. Our 4-application workloads are 50 randomly picked combinations of the same 10 applications. We run all applications to completion. On the multiple-application experiments we run until the longest application finishes and meanwhile, we restart any application that finishes early to continue producing interference and contention for all resources, including large cores. We measure execution time during the first run of each application. We run each application with the optimal number of threads found when running alone. When the sum of the optimal number of threads for all applications is greater than

Mechanism	Description
ACMP	Serial portion runs on a large core, parallel portion runs on all cores [3, 15, 16].
AGETS	Age-based Scheduling algorithm for a single multithreaded application as described in [13].
AGETS+PIE	To compare to a reasonable baseline tor thread scheduling of multiple applications we use AGETS [13] to find the most lagging thread within each application. Then, we use PIE [27] to pick for each large core the thread that would get the largest speedup among the lagging threads from each application.
BIS	Serial portion and bottlenecks run on the large cores, parallel portion runs on small cores [10].
MA-BIS	To compare to a reasonable baseline, we extend BIS to multiple applications by sharing the large cores to accelerate bottlenecks from any application. To follow the key insights from BIS, we prioritize bottlenecks by thread waiting cycles normalized to the number of threads for each application, regardless of which application they belong to.
UBA	Our proposal.

Table 3: Experimental configurations.

Workload	Description	Source	Input set	# Bottl.	Bottleneck description
blacksch	BlackScholes option pricing	[18]	1M options	1	Final barrier after *omp parallel*
hist_ph	Histogram of RGB components	Phoenix [19]	S (small)	1	Crit. sections (CS) on map-reduce scheduler
iplookup	IP packet routing	[28]	2.5K queries	# thr.	CS on routing tables
is_nasp	Integer sort	NAS suite [4]	n = 64K	1	CS on buffer of keys
mysql	MySQL server [1]	SysBench [2]	OLTP-nontrx	18	CS on meta data, tables
pca_ph	Principal components analysis	Phoenix [19]	S (small)	1	CS on map-reduce scheduler
specjbb	JAVA business benchmark	[22]	5 seconds	39	CS on counters, warehouse data
tsp	Traveling salesman	[12]	8 cities	2	CS on termination condition, solution
webcache	Cooperative web cache	[26]	100K queries	33	CS on replacement policy
ft_nasp	FFT computation	NAS suite [4]	size = 32x32x32	1	Run as single-threaded application

Table 4: Evaluated workloads.

the maximum number of threads, we reduce the number of threads for the application(s) whose performance is(are) less sensitive to the number of threads. Unless otherwise indicated, we use harmonic mean to compute all the averages in our evaluation. To measure system performance with multiple applications [9] we use *Harmonic mean of Speedups (Hspeedup)* [14] and *Weighted Speedup (Wspeedup)*[21], defined below for N applications. T_i^{alone} is the execution time when the application runs alone in the system and T_i^{shared} is the execution time measured when all applications are running. We also report *Unfairness* [17] as defined below.

$$Hspeedup = \frac{N}{\sum_{i=0}^{N-1} \frac{T_i^{shared}}{T_i^{alone}}} \qquad Wspeedup = \sum_{i=0}^{N-1} \frac{T_i^{alone}}{T_i^{shared}}$$

$$Unfairness = \frac{\max(T_i^{alone}/T_i^{shared})}{\min(T_i^{alone}/T_i^{shared})}$$

5. EVALUATION

5.1 Single Application

We carefully choose the number of threads to run each application with, because that number significantly affects performance of multithreaded applications. We evaluate two situations: (1) number of threads equal to the number of available hardware contexts, i.e., maximum number of threads, which is a common practice for running non-I/O-intensive applications; and (2) optimal number of threads, i.e., the number of threads that minimizes execution time, which we find with an exhaustive search for each application on each configuration. Table 5 shows the average speedups of UBA over other mechanisms for different ACMP configurations. UBA performs better than the other mechanisms on every configuration, except for multiple large cores on a 16-core area budget. BIS and UBA dedicate the large cores to accelerate code segments, unlike ACMP and AGETS. Therefore, the maximum number of threads that can run on BIS and UBA is smaller. With an area budget of 16, BIS and UBA cannot overcome the loss of parallel throughput due to running significantly fewer threads. For example, with 3 large cores, AGETS and ACMP can execute applications with up to 7 threads (4 on small cores and 3 on large cores), while BIS and UBA can execute a maximum of 4 threads. Overall, the benefit of UBA increases with area budget and number of large cores.

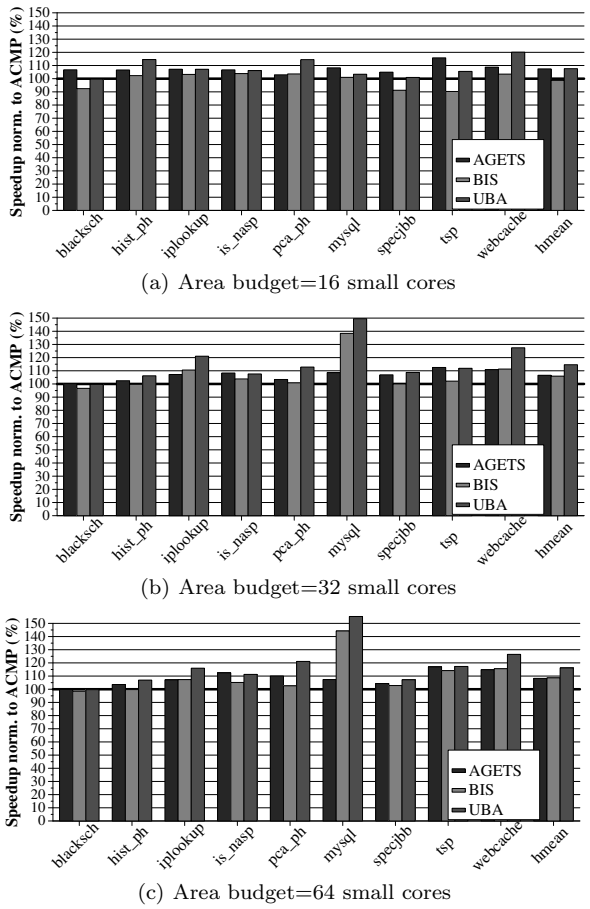
Config.		Opt. number of threads			Max. number of threads		
Area	LC	ACMP	AGETS	BIS	ACMP	AGETS	BIS
16	1	7.6	0.2	9.0	8.2	0.2	9.0
16	2	−6.3	−5.8	19.2	−5.9	−5.7	19.2
16	3	−43.4	−11.7	37.8	−43.3	−11.6	37.8
32	1	14.6	7.5	8.2	16.1	6.2	4.1
32	2	15.3	4.7	13.2	21.7	9.0	13.9
32	3	14.5	2.2	16.0	22.2	5.7	15.4
64	1	16.2	7.3	6.9	20.5	6.2	5.5
64	2	21.6	9.8	9.5	30.1	18.3	10.5
64	3	23.1	11.0	11.3	33.5	24.0	13.0

Table 5: Average speedup (%) of UBA over ACMP, AGETS and BIS.

5.1.1 Single-Large-Core ACMP

Figure 7 shows the speedup of AGETS, BIS and UBA over ACMP, which accelerates only the Amdahl's serial bottleneck. Each application runs with its optimal number of threads for each configuration. We show results for 16, 32 and 64-small-core area budgets and a single large core. On average, our proposal improves performance over ACMP by 8%/15%/16%, over AGETS by 0.2%/7.5%/7.3% and over BIS by 9%/8%/7% for area budgets of 16/32/64 small cores. We make three observations.

First, as the number of cores increases, AGETS, BIS and UBA provide higher average performance improvement over ACMP. Performance improvement of UBA over AGETS increases with the number of cores because, unlike AGETS, UBA can accelerate bottlenecks, which have an increasingly larger impact on performance as the number of cores increases (as long as the number of threads increases). However, performance improvement of UBA over BIS slightly decreases with a higher number of cores. The reason is that the benefit of accelerating lagging threads in addition to bottlenecks gets smaller for some benchmarks as the number of threads increases, depending on the actual amount of thread imbalance that UBA can eliminate. Since BIS and UBA dedicate the large core to accelerate code segments, they can run one fewer thread than ACMP and AGETS. With an area budget of 16, the impact of running one fewer thread is significant for BIS and UBA, but UBA is able to overcome



(a) Area budget=16 small cores

(b) Area budget=32 small cores

(c) Area budget=64 small cores

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

that disadvantage with respect to ACMP and AGETS by accelerating both bottlenecks and lagging threads.

Second, as the number of threads increases, *iplookup*, *is_nasp*, *mysql*, *tsp* and *webcache* become limited by contended critical sections and significantly benefit from BIS. UBA improves performance more than AGETS and BIS because it is able to accelerate both lagging threads and bottlenecks. *Hist_ph* and *pca_ph* are MapReduce applications with no significant contention for critical sections where UBA improves performance over AGETS because its shorter scheduling quantum and lower-overhead hardware-managed thread migration accelerates all three parallel portions (map, reduce and merge) more efficiently.

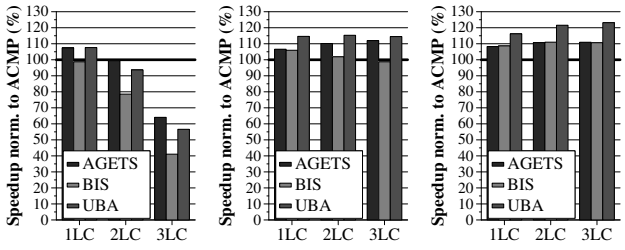
Third, *blacksch* is a very scalable workload with neither significant bottlenecks nor significant thread imbalance, which is the worst-case scenario for all the evaluated mechanisms. Therefore, AGETS produces the best performance because it accelerates all threads in round-robin order and it can run one more thread than BIS or UBA, which dedicate the large core to acceleration. However, AGETS' performance benefit for *blacksch* decreases as the number of cores (and threads) increases because the large core is time-multiplexed among all threads, resulting in less acceleration on each thread and a smaller impact on performance. Note that in a set of symmetric threads, execution time is reduced only by the minimum amount of time that is saved from any thread, which requires accelerating all threads evenly. UBA efficiently accelerates all threads, similarly to AGETS, but is penalized by having to run with one fewer thread.

We conclude that UBA improves performance of applications that have lagging threads, bottlenecks or both by a larger amount than AGETS, a previous proposal to accelerate only lagging threads, and BIS, a previous proposal to accelerate only bottlenecks.

5.1.2 Multiple-Large-Core ACMP

Figure 8 shows the average speedups across all workloads on the different configurations with the same area budgets, running with their optimal number of threads. The main observation is that replacing small cores with large cores on a small area budget (16 cores, Figure 8(a)) has a very large negative impact on performance due to the loss of parallel throughput. With an area budget of 32 (Figure 8(b)) AGETS can take advantage of the additional large cores to increase performance, but BIS cannot, due to the loss of parallel throughput. UBA performs about the same with 1, 2 or 3 large cores, but still provides the best overall performance of all three mechanisms. With an area budget of 64 (Figure 8(c)) there is no loss of parallel throughput, except for *blacksch*. Therefore, both AGETS and BIS can take advantage of more large cores. However, average performance of UBA improves more significantly with additional large cores. According to per-benchmark data not shown due to space limitations, the main reason for this improvement is that *iplookup*, *mysql* and *webcache* benefit from additional large cores because UBA is able to concurrently accelerate the most important critical sections and lagging threads.

We conclude that as the area budget increases UBA becomes more effective than ACMP, AGETS and BIS in taking advantage of additional large cores.



(a) Area=16 (b) Area=32 (c) Area=64

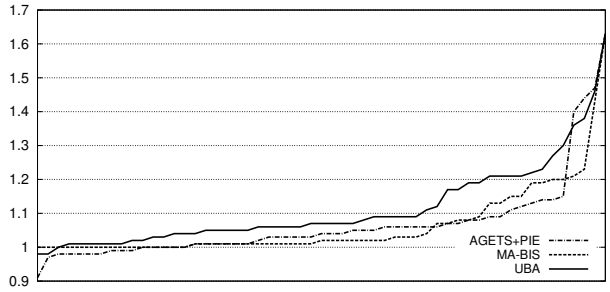
Figure 8: Average speedups with multiple large cores, normalized to ACMP with 1 large core.

5.2 Multiple Applications

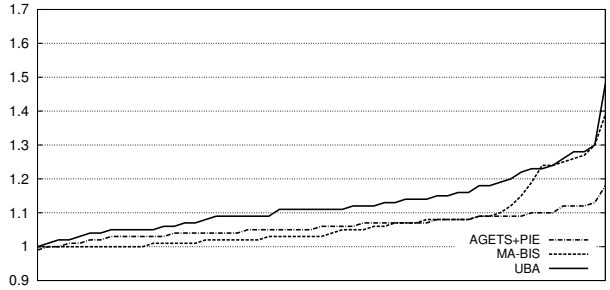
Figure 9 shows the sorted harmonic speedups of 55 2-application workloads with each mechanism: the extensions of previous proposals to multiple applications (AGETS+PIE and MA-BIS) and UBA, normalized to the harmonic speedup of ACMP, with an area budget of 64 small cores. Performance is generally better for UBA than for the other mechanisms and the difference increases with additional large cores. Results for 2-application and 4-application workloads with an area budget of 128 small cores show a similar pattern and are not shown in detail due to space limitations, but we show the averages. Tables 6 and 7 show the improvement in average weighted speedup, harmonic speedup and unfairness with UBA over the other mechanisms. Average *Wspeedup* and *Hspeedup* for UBA are better than for the other mechanisms on all configurations. Unfairness is also reduced, except for three cases with 4 applications. UBA's unfairness measured as maximum slowdown [7] is even lower than with the reported metric (by an average -4.3% for 2 ap-

plications and by an average -1.7% for 4 applications, details not shown due to space limits).

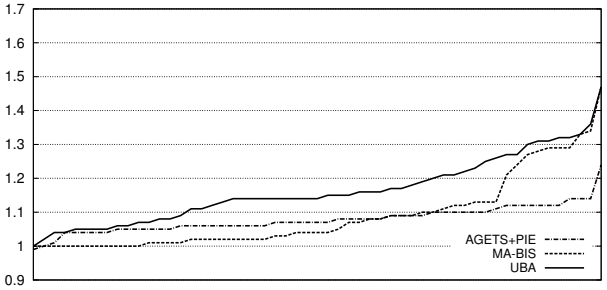
We conclude that as the area budget and the number of large cores increase, the performance advantage of UBA generally increases because UBA utilizes the large cores more effectively than the other mechanisms, i.e., UBA identifies and accelerates the most important bottlenecks and lagging threads for each application.



(a) Area budget=64, 1LC



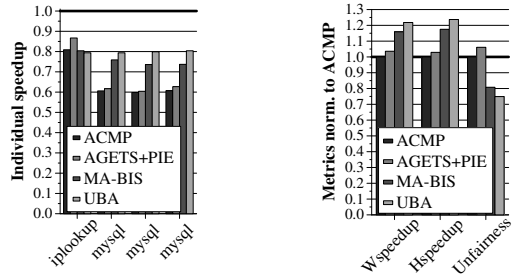
(b) Area budget=64, 2LC



(c) Area budget=64, 3LC

Figure 9: Sorted harmonic speedups for 55 2-application workloads, normalized to ACMP, on Area budget=64.

Figure 10 shows detailed results for one of the 4-application workloads, consisting of *iplookup*, which has many critical sections with medium criticality, and three instances of *mysql*, which has many critical sections with high criticality. The experiment ran on an area budget of 128, with a single large core. Figure 10(a) shows the individual speedups for each application, relative to each application running alone with the best configuration, i.e., UBA. AGETS improves performance on *iplookup* but not much on *mysql*, because *mysql* is critical-section limited and does not have much thread imbalance. Both BIS and UBA focus on the more important critical sections from *mysql* and do not improve *iplookup*'s performance. However, UBA is more effective in choosing the critical sections whose acceleration can improve performance for each application and is also more fair. Therefore, UBA has the best weighted and harmonic speedups and the lowest unfairness, as shown in Figure 10(b).

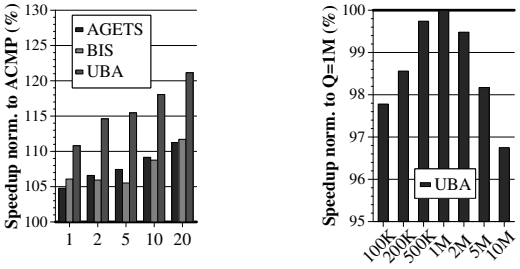


(a) Individual speedups (b) Average metrics

Figure 10: A case study with multiple applications.

5.3 Other Results

Figure 11(a) shows the average speedups for all mechanisms relative to ACMP on a 28-small-core, 1-large-core ACMP, running with optimal number of threads, for different interconnect hop latencies. On average, the benefit of UBA over the best of AGETS and BIS increases with hop latency (4.4%/7.5%/7.5%/8.1%/8.4% for 1/2/5/10/20 cycles). This is because UBA is more effective than previous proposals in selecting the code segments that provide higher performance benefit from acceleration.



(a) Interconnect Hop Latency (cycles) (b) Scheduling Quantum Length (Q) (cycles)

Figure 11: Sensitivity studies.

Figure 11(b) shows the average speedups for UBA for different values of the scheduling quantum Q on the same system configuration. A shorter quantum enables finer-grained scheduling of lagging threads between consecutive barriers, i.e., even when the number of identified lagging threads is not small, all of them have an opportunity to be accelerated before the next barrier. A longer quantum better amortizes the costs of migration, which could otherwise reduce or even negate the benefit of acceleration. For our workloads, Q = 1Mcycles is the overall best value.

We also analyzed the effect of acceleration proposals on cache misses. Our workloads are not memory intensive (LLC MPKI is less than 2.5). Code segment migration does not significantly affect L1 misses and L3 (i.e., LLC) misses. The only significant change is in L2 misses, more specifically in core-to-core data transfers for *mysql*. In *mysql*, the amount of private data that must be migrated to the large core running critical sections is significant. Therefore, BIS and UBA increase L2 MPKI by 16% and 18%, respectively, since UBA slightly increases the number of critical sections that run on the large core. Using Data Marshaling [23] can significantly reduce this problem, further increasing *mysql*'s speedup over BIS (by 10.8%) and UBA (by 8.8%).

5.4 Discussion

UBA improves performance of a single application over AGETS or BIS, because it can accelerate both bottlenecks

Configuration	Average Weighted speedup			Average Harmonic speedup			Average Unfairness		
	ACMP	AGETS+PIE	MA-BIS	ACMP	AGETS+PIE	MA-BIS	ACMP	AGETS+PIE	MA-BIS
Area=64, 1 LC	8.7	3.9	3.6	10.2	4.1	4.4	−13.4	−3.9	−6.3
Area=64, 2 LC	11.5	5.4	5.3	12.1	5.8	5.4	−7.1	−6.1	−0.6
Area=64, 3 LC	14.7	6.7	6.9	15.6	7.3	7.0	−8.5	−6.7	−0.8
Area=128, 1 LC	8.4	4.5	2.4	10.1	4.9	3.1	−13.3	−4.6	−4.8
Area=128, 2 LC	11.8	6.6	4.2	12.7	7.2	4.3	−9.6	−7.4	−2.0
Area=128, 3 LC	14.9	8.3	6.1	16.4	9.0	6.2	−12.6	−8.4	−1.9

Table 6: Average improvement (%) for each metric with UBA over each baseline on each configuration with 2 applications.

Configuration	Average Weighted speedup			Average Harmonic speedup			Average Unfairness		
	ACMP	AGETS+PIE	MA-BIS	ACMP	AGETS+PIE	MA-BIS	ACMP	AGETS+PIE	MA-BIS
Area=128, 1 LC	3.0	3.1	0.9	4.1	2.6	1.7	−12.9	0.6	−9.1
Area=128, 2 LC	7.3	3.8	1.9	9.6	4.2	2.5	−18.9	−4.4	−5.7
Area=128, 3 LC	7.4	3.7	3.4	9.3	4.1	3.0	−14.2	−3.3	4.6
Area=128, 4 LC	7.6	4.0	2.8	8.9	4.7	2.1	−10.7	−6.4	8.4

Table 7: Average improvement (%) for each metric with UBA over each baseline on each configuration with 4 applications.

and lagging threads, depending on which ones are more critical for performance at any time. UBA is also better at taking advantage of multiple large cores and accelerating multiple applications.

The relative performance benefit of UBA over other mechanisms is somewhat reduced as the number of concurrent applications increases. This is due to two reasons. First, our baselines for multiple applications are new contributions of this paper and are very aggressive: AGETS+PIE combines AGETS with PIE, and MA-BIS thoughtfully extends BIS. Second, as the number of applications increases, it is easier to find at any given time a bottleneck or lagging thread that is clearly critical for one of the applications and should be accelerated. Therefore, even the simpler mechanisms can improve performance as the number of applications increases. For the same reasons, the performance improvement of UBA does not consistently increase with additional large cores when running 4 applications.

Figure 9 shows that UBA is not the best proposal for every workload. We identify two reasons why UBA does not always accelerate the code segments that provide the highest system performance improvement. First, UBA uses PIE [27] to predict performance on a large core for code segments running on small cores. In this mode of operation, PIE is simple but does not distinguish between code segments that do not access main memory. Second, our L factor does not consider the costs of migration. Therefore, UBA sometimes accelerates critical sections that require migration of a significant amount of private data, instead of a lagging thread or bottleneck from another concurrent application, whose acceleration would have been more profitable. Extending our Utility of Acceleration metric to consider the costs of migration is part of our future work.

6. RELATED WORK

Our major contribution is a new Utility of Acceleration metric that allows meaningful comparisons to decide which code segments to accelerate, both bottlenecks and lagging threads, in a single- or multiple-application scenario. The most closely related work is a set of proposals to accelerate bottlenecks or to accelerate lagging threads using large cores of an ACMP. Also related are software-based thread scheduling proposals for ACMPs.

6.1 Accelerating Bottlenecks

Several proposals accelerate Amdahl's serial bottleneck. Annavaram et al. [3] use frequency throttling, Morad et al. use the large core of an ACMP for a single application [15] and for multiple applications [16]. Suleman et al. [24] accelerate critical sections on a large core of an ACMP. Suleman et al. [25] reduce stage imbalance on pipeline-parallel programs by allocating cores to stages. Joao et al. [10] accelerate bottlenecks of multiple types on large cores of an ACMP, using the amount of thread waiting each bottleneck causes as an estimate of its criticality. These proposals accelerate only specific bottlenecks, improving performance of applications that are limited by those bottlenecks, but do not take advantage of the large cores of an ACMP for applications that do not have the bottlenecks that they target. Additionally, these proposals are not designed to accelerate bottlenecks from multiple applications, except for [16], which is limited to serial bottlenecks. In contrast, our proposal always accelerates code segments, either bottlenecks or lagging threads, fully utilizing any number of large cores to improve performance of any kind of single and multiple applications.

Ebrahimi et al. [8] prioritize memory requests produced by cores that are executing important critical sections or barriers to accelerate their execution. This work is orthogonal to ours, and our Utility metric could also be used to influence memory scheduling decisions.

6.2 Accelerating Lagging Threads on Single Multithreaded Applications

Several proposals try to find the most lagging thread using different progress metrics. Meeting Points [6] uses hints from the software to count executed loop iterations and determine which thread is lagging, assuming all threads have to execute the same number of iterations. Thread Criticality Predictors [5] use a combination of L1 and L2 cache miss counters to predict thread progress, considering that in a set of balanced threads, those that are affected by more cache misses are more likely to lag behind. Age-based Scheduling [13] assumes balanced threads have to execute the same number of instructions and finds the thread that has executed fewest instructions. Only [13] explicitly accelerates the most lagging thread on a large core of an ACMP, but the other proposals could also be used for the same purpose, e.g., as [6] is used for memory scheduling in [8]. These approaches are limited to reducing or eliminating thread imbalance, but cannot identify and accelerate limiting critical sections. Age-based Scheduling is implemented as a coarse-grained O/S-based thread scheduler and therefore has higher scheduling and context switch overheads than our proposal, which relies on hardware-controlled execution migration.

6.3 Accelerating Multiple Applications

Koufaty [11] schedules on a large core threads with fewer off-core memory accesses and fewer core front-end stalls. Saez [20] proposes a utility factor that includes a speedup

factor based on LLC miss rates and a function of the total number of threads (TLP) and the number of accelerated threads. The utility factor is used to classify threads into three priority classes and the large cores are assigned starting with the high-priority class and are time-multiplexed if there are more threads in a class than available large cores. Sequential phases are given the highest priority, as long as their utility factor qualifies for the high-priority class.

UBA overcomes four major limitations of these past proposals. First, their speedup estimation is based on LLC misses or stalls, while our Utility of Acceleration metric predicts the speedup from executing a thread on a large core using PIE [27], which considers not only memory accesses but also instruction-level parallelism (ILP) and memory-level parallelism (MLP). Second, they do not distinguish lagging threads within each application and treat all threads in an application with the same importance in terms of criticality, assuming balanced parallel phases. In contrast, UBA identifies lagging threads. If the number of lagging threads is significantly smaller than the total number of threads in an application, focusing acceleration on those lagging threads has higher potential to improve performance [13]. Third, they do not consider accelerating critical sections that are likely to be on the critical path, which UBA accelerates when critical sections have higher Utility of Acceleration than lagging threads. Fourth, [11] and [20] are coarse-grained O/S-based proposals and have higher scheduling and context switch overheads than our proposal, which manages execution migration in hardware.

Van Craeynest et al. [27] develop a simple model to estimate performance on different core types, and use it to schedule multiple single-threaded applications on an ACMP. We use their proposal (PIE) as part for our Utility of Acceleration metric to estimate the acceleration of executing on a large core. Since our Utility metric also takes into account the criticality of code segments, it can be used to identify the most important bottlenecks and lagging threads within each application. Therefore UBA can accelerate any combination of single- and multi-threaded applications.

7. CONCLUSION

We propose *Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs* (UBA), the first mechanism that can accelerate both bottleneck code segments and lagging threads on an Asymmetric CMP (ACMP). Our proposal can accelerate multiple multithreaded applications and can leverage an ACMP with any number of large cores. We show that UBA improves performance of a set of workloads including bottleneck-intensive and non-bottleneck-intensive applications. UBA outperforms state-of-the-art thread scheduling and bottleneck acceleration proposals for single applications, and their aggressive extensions for multiple applications. The benefit of UBA increases with more cores. We conclude that UBA is a comprehensive proposal to effectively use ACMPs to accelerate any number of parallel applications with reasonable hardware cost and without requiring programmer effort.

ACKNOWLEDGMENTS

We thank Khubaib, Ben Lin, Rustam Miftakhutdinov, Carlos Villavieja, other members of the HPS research group and the anonymous reviewers for their comments and suggestions. We gratefully acknowledge the support of the Cockrell Foundation and Intel Corporation.

REFERENCES

[1] "MySQL database engine 5.0.1," http://www.mysql.com.
[2] "SysBench: a system performance benchmark v0.4.8," http://sysbench.sourceforge.net.
[3] M. Annavaram, E. Grochowski, and J. Shen, "Mitigating Amdahl's law through EPI throttling," in *ISCA*, 2005.
[4] D. H. Bailey *et al.*, "The NAS parallel benchmarks," NASA Ames Research Center, Tech. Rep. RNR-94-007, 1994.
[5] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *ISCA*, 2009.
[6] 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.
[7] R. Das, O. Mutlu, T. Moscibroda, and C. Das, "Application-aware prioritization mechanisms for on-chip networks," in *MICRO*, 2009.
[8] 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.
[9] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, 2008.
[10] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *ASPLOS*, 2012.
[11] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *EuroSys*, 2010.
[12] H. Kredel, "Source code for traveling salesman problem (TSP)," http://krum.rz.uni-mannheim.de/ba-pp-2007/java/index.html.
[13] N. B. Lakshminarayana, J. Lee, and H. Kim, "Age based scheduling for asymmetric multiprocessors," in *SC*, 2009.
[14] K. Luo, J. Gummaraju, and M. Franklin, "Balancing thoughput and fairness in SMT processors," in *ISPASS*, 2001.
[15] T. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. Ayguade, "Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors," *Comp. Arch. Letters*, 2006.
[16] T. Morad, A. Kolodny, and U. Weiser, "Scheduling multiple multithreaded applications on asymmetric and symmetric chip multiprocessors," in *PAAP*, 2010.
[17] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *MICRO*, 2007.
[18] NVIDIA Corporation, "CUDA SDK code samples," 2009.
[19] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *HPCA*, 2007.
[20] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, "A comprehensive scheduler for asymmetric multicore systems," in *EuroSys*, 2010.
[21] A. Snavely and D. M. Tullsen, "Symbiotic job scheduling for a simultaneous multithreading processor," in *ASPLOS*, 2000.
[22] *Welcome to SPEC*, The Standard Performance Evaluation Corporation, http://www.specbench.org/.
[23] M. A. Suleman, O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt, "Data marshaling for multi-core architectures," *ISCA*, 2010.
[24] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," *ASPLOS*, 2009.
[25] M. A. Suleman, M. K. Qureshi, Khubaib, and Y. N. Patt, "Feedback-directed pipeline parallelism," in *PACT*, 2010.
[26] Tornado Web Server, http://tornado.sourceforge.net, 2008.
[27] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *ISCA*, 2012.
[28] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," in *SIGCOMM*, 1997.