Criticality Stacks: Identifying Critical Threads in Parallel Programs using Synchronization Behavior*

Kristof Du Bois Stijn Eyerman Jennifer B. Sartor Lieven Eeckhout
ELIS Department, Ghent University, Belgium
{kristof.dubois,stijn.eyerman,jennifer.sartor,leeckhou}@elis.UGent.be

ABSTRACT

Analyzing multi-threaded programs is quite challenging, but is necessary to obtain good multicore performance while saving energy. Due to synchronization, certain threads make others wait, because they hold a lock or have yet to reach a barrier. We call these *critical threads*, i.e., threads whose performance is determinative of program performance as a whole. Identifying these threads can reveal numerous optimization opportunities, for the software developer and for hardware.

In this paper, we propose a new metric for assessing thread criticality, which combines both how much time a thread is performing useful work and how many co-running threads are waiting. We show how thread criticality can be calculated online with modest hardware additions and with low overhead. We use our metric to create *criticality stacks* that break total execution time into each thread's criticality component, allowing for easy visual analysis of parallel imbalance.

To validate our criticality metric, and demonstrate it is better than previous metrics, we scale the frequency of the most critical thread and show it achieves the largest performance improvement. We then demonstrate the broad applicability of criticality stacks by using them to perform three types of optimizations: (1) program analysis to remove parallel bottlenecks, (2) dynamically identifying the most critical thread and accelerating it using frequency scaling to improve performance, and (3) showing that accelerating only the most critical thread allows for targeted energy reduction.

*Stijn Eyerman is supported through a postdoctoral fellowship by the Research Foundation – Flanders (FWO). Additional support is provided by the FWO project G.0179.10N, the UGent-BOF project 01Z04109, the ICT Department of Ghent University, and the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295.

ISCA'13 Tel-Aviv, Israel

1. INTRODUCTION

In order to take advantage of today's ubiquitous multicore processors, software has to provide enough parallel work to make use of the available resources in order to continue the trend of ever-improving performance. Multi-threaded programs that try to use these resources inherently introduce synchronization to ensure correct execution. Typical synchronization examples are barriers (a thread cannot go beyond a certain point in the code until all threads have reached that point), critical sections (only one thread can execute a certain critical section, to prevent hazardous parallel updates of data), and consumer-producer synchronization in pipelined programs (a thread can only proceed with its calculation after the needed data is produced by another thread). While synchronization is necessary, it results in threads waiting for each other, stalling program progress, limiting performance, and wasting energy.

Identifying critical threads in a parallel program is important, because these threads cause others to wait (by holding a lock or not yet reaching a barrier, etc.), and largely determine overall performance. Threads identified as critical can be targeted for performance optimization, through software re-design or through hardware techniques. Speeding up critical threads can speed up the whole program. Or inversely, slowing down non-critical threads has almost no impact on performance, which enables a more energy-efficient execution. Speeding up a thread can be done by migrating it to a faster core in a heterogeneous multicore [28], by temporarily boosting the frequency of the core it executes on [2], by raising the fetch priority of that thread in an SMT context [7], by allowing more task stealing from this thread in a task stealing context [3], etc. All of these examples allow for only one or a few threads to be sped up, so it is important to identify the most critical thread(s).

Key contribution: Criticality stack. In this paper, we propose a novel metric to measure thread criticality in parallel programs using synchronization behavior, and we present a hardware implementation for dynamically measuring thread criticality at low overhead. Our criticality metric measures how much time a thread is performing useful work and how many threads are concurrently waiting. The metric gathers information for program execution intervals delineated by synchronization behavior (critical sections, barriers and pipes). A thread has a larger criticality component when more threads wait concurrently on it, and thus it is more determinative of program running time.

Combining different threads' components into a *critical-*

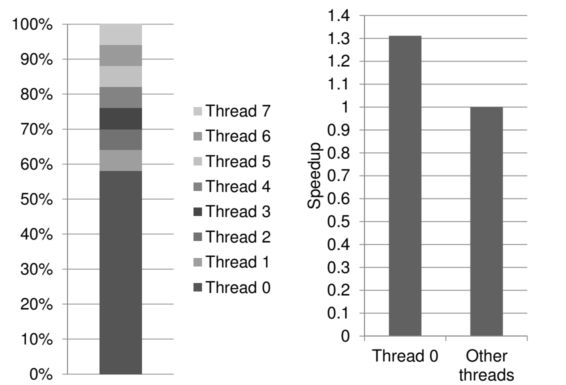

Figure 1: BFS's criticality stack and total program speedups from accelerating the identified critical and non-critical threads.

ity stack facilitates easy comparison of parallel (im)balance. The criticality stack is a stacked bar graph that divides the program's execution time (100%) into each thread's criticality component. If all threads have approximately the same criticality, then no one thread is critical, and no performance gain can be obtained by speeding up a thread. If, however, certain threads have larger criticality than other threads, they reveal themselves as parallel bottlenecks.

We validate criticality stacks by experimentally showing that speeding up the most critical thread (if one exists) results in significant performance speedups; accelerating identified non-critical threads on the other hand does not affect performance. Figure 1 illustrates this for the BFS benchmark: the criticality stack at the left shows that thread 0 is much more critical than all other threads. The graph at the right shows the program speedup when each of the threads is accelerated individually, running at twice the clock frequency. We present results for thread 0 and the maximum of all other threads (as they all result in no program speedup).

Comparison to prior work. Previous closely related work on detecting thread criticality tries to address load imbalance caused by barriers [3]. We reimplement their technique that predicts thread criticality based on cache misses, and find that our criticality metric more accurately identifies the thread most critical to running time. Bottleneck Identification and Scheduling (BIS) [15] tries to identify parallel bottlenecks, and migrates threads executing these bottlenecks to a big core in a heterogeneous multicore. While they accelerate bottlenecks that limit parallel performance, we instead find the thread(s) most critical to overall performance. A bottleneck could be on the critical path for one thread, but not for others. Therefore, accelerating bottlenecks does not necessarily improve performance, and could even needlessly accelerate threads, reducing energy efficiency. We reimplement their technique, scaling the frequency of the identified bottleneck instead of migrating it, and show that our metric achieves higher performance by accelerating the critical thread.

Optimization applications. Using the information from dynamically calculated criticality stacks, we present three optimization use cases. (1) We demonstrate how criticality stacks can help programmers address performance problems, and present how a simple software optimization derived from the criticality stack for one of our benchmarks (BFS), yields a 1.67× and 2.16× speedup for 8 and 16 cores, respectively. (2) We speed up the most critical thread (if any) per time slice from 2 to 2.5 GHz with per-core frequency scaling, reacting to phase behavior. This approach improves overall parallel performance by on average 4.6% for 8 threads and 4.2% for 16 threads, and up to 17%. Our dynamic algorithm almost doubles BIS's [15] performance improvement. (3) We demonstrate that criticality stacks enable targeting parallel optimization to reduce energy consumption. Our dynamic algorithm which speeds up only one thread at a time, reduces energy consumption by on average 3.2% for 16 threads and 2.8% for 8 threads, and up to 12.6%.

Significance and impact. Overall, criticality stacks are a novel, insightful and intuitive performance metric for parallel programs, enabling a multitude of applications, ranging from program analysis, to software optimization, to dynamic optimization for performance and power.

2. DEFINING THREAD CRITICALITY

A thread's criticality depends on both if it is doing useful work[1], and if other threads are waiting for it. We say a thread is critical if its progress at a certain point determines the progress of the whole program. One example is when all threads but one have reached a barrier. Because all other threads are waiting, the progress of the one thread that is still executing equals the progress of the whole program, and therefore this thread is critical.

In general, identifying the most critical thread in parallel programs is non-trivial. Figure 2 shows an example program with 4 threads that has both barrier (horizontal line across all threads) and critical section (darker vertical bar) synchronization. Thread 3 has the largest running time ($t_0+t_1+t_2+t_3+t_4+t_5+t_6=17$) and therefore performs most useful work; thread 0 on the other hand waits the longest for acquiring the critical section and keeps all other threads waiting at the barrier. It is not obvious which thread is most critical to overall performance.

To comprehensively compute thread criticality, we propose our criticality metric that takes into account both running time and number of waiting threads. Execution time is divided into a number of intervals. A new interval begins whenever any thread changes state, from active to inactive or vice versa, as a result of synchronization behavior. (Section 3 defines how we identify running threads.) Each active thread's criticality number gets a portion of interval time t. In other words, time t is divided by the number of threads doing useful work, and this is added to each thread's criticality sum (see Figure 2). This metric essentially weights time, adding more to active threads for which many threads wait, and less to active threads when no threads are waiting.

We formalize the criticality metric in the following way. Suppose that for a time interval t, r out of n threads are running. For the r threads that are running we add $\frac{t}{r}$ to their respective criticality counter. For the other $n-r$ threads, we add nothing. In each interval, the set of running threads

[1]When a thread is spinning or busy waiting, we assume that it is not performing useful work. In the remainder of the paper, we will denote a thread that is performing useful work as 'running', 'active', or 'executing', excluding spinning.

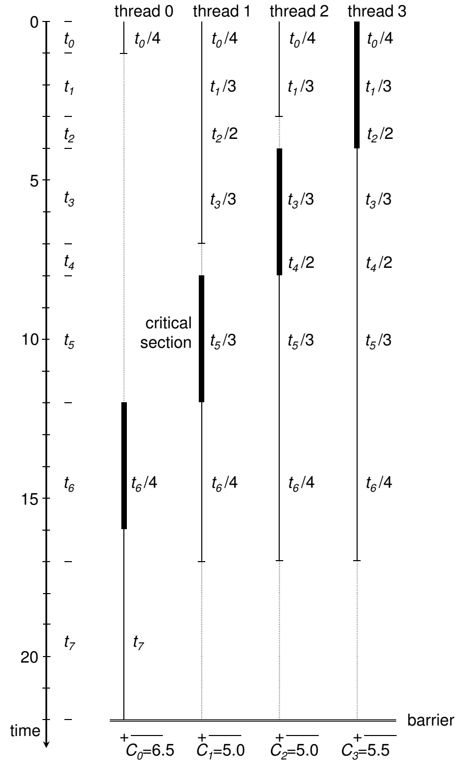

Figure 2: Criticality calculation example.

is fixed. Assume there are N such intervals over the whole program (or a phase of the program), t_i is the duration of interval i, r_i is the number of running threads in that interval and R_i is the set containing the thread IDs of the running threads (therefore $|R_i| = r_i$). Then the total criticality of thread j equals

$$C_j = \sum_{i=0}^{N-1} \begin{cases} \frac{t_i}{r_i}, & \text{if } j \in R_i \\ 0, & \text{if } j \notin R_i \end{cases} \tag{1}$$

Figure 2 shows an example of how the criticality metric is calculated. Thread 0 has a total criticality of $t_0/4 + t_6/4 + t_7$=6.5. Threads 1, 2, and 3 all have lower criticality sums at 5, 5, and 5.5, respectively. Therefore, thread 0 is determined to be the most critical thread in this example. This might seem counter-intuitive because it has the smallest total running time ($t_0 + t_6 + t_7$=11) compared to all other threads (thread 1 = 16, thread 2 = 16, and thread 3 = 17). Accelerating thread 3 would reduce the execution time of the critical section, and as a result, threads 0, 1 and 2 would enter their critical sections sooner, however, thread 0 would still reach the barrier much later than the other threads, resulting in only a small speedup. Speeding up thread 0 on the other hand results in a much larger speedup, because it is guaranteed to reduce the barrier waiting time of all other threads, so thread 0 is indeed more critical as detected by the criticality metric. By taking into account the number of active threads, our metric illustrates differences in criticality between threads clearly.

An important characteristic of this metric is that the sum of all threads' criticalities equals the total execution time. Formally, if T is the total execution time of the parallel program (or a phase), then

$$\sum_{j=0}^{n-1} C_j = T. \tag{2}$$

This is intuitive, as for every interval r times $\frac{t_i}{r}$ is accounted, which gives a total of t_i over all threads, and $\sum_{i=0}^{N-1} t_i = T$. This property allows us to divide each criticality sum by T to obtain each thread's normalized criticality component. We represent these components in a stacked bar, yielding the *criticality stack*, which breaks up a program's total execution time into each thread's criticality percentage.

3. COMPUTING THREAD CRITICALITY

In order to dynamically measure thread criticality, we need to determine at every moment in time how many threads are performing useful work. We now first detail how to identify which threads are active, which also delineates time intervals. We then describe our dedicated hardware implementation for calculating thread criticality in an efficient manner using very little energy and without interfering with the running program.

3.1 Identifying running threads

There are two main causes why a thread is not performing useful work: either it is scheduled out by the operating system, or it is spinning (using a waiting loop, constantly checking the synchronization variable). The operating system can easily communicate when it schedules threads in and out. Spinning is more difficult to detect, since the thread is executing instructions, albeit useless ones.

Either software or hardware can detect spinning. Software solutions involve adding extra instructions that denote spinning threads. These extra instructions are typically inserted in threading libraries (e.g., Pthreads) so that programmers do not have to explicitly add them. Hardware solutions use tables in the processor to keep track of backward branches [20], which possibly belong to a spinning loop, or repetitive loads [30], which are possibly loading a condition variable. Spinning is detected if specific conditions are met, i.e., no architectural state changes since the last branch or an update from another core to the repetitive load's address.

Both approaches have their advantages and disadvantages. A hardware solution can detect all types of spinning, including user-level spinning. On the other hand, a hardware solution detects spinning later (e.g., only after a certain threshold is reached), which can have an impact on the effectiveness of the technique that needs spinning information, and there is a chance to have false positives (e.g., a non-captured architectural state change) or false negatives (e.g., when the number of spinning iterations is under a certain threshold).

Software solutions on the other hand use semantic information from the program itself, and will only detect true spinning loops. Of course, user-level spinning that is not instrumented cannot be detected. However, if correctly instrumented, software accurately detects the start of the spinning, and can immediately indicate the end of the spinning.

For this study we use a software solution, since software detects spinning in a more timely manner and is easier to implement. The benchmarks we evaluate only use threading libraries to perform synchronization (Pthreads and OpenMP). We instrument all Pthread and OpenMP primitives that involve spinning (locks, barriers and condition variables).

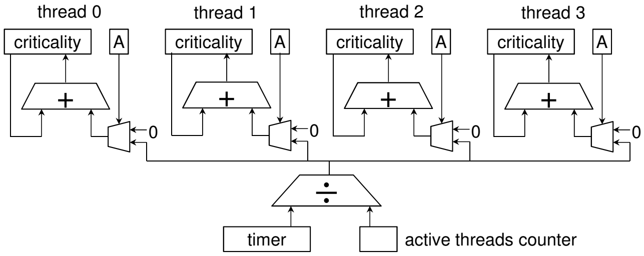

Figure 3: Hardware device for online criticality calculation ('A' is the active bit per thread).

When the program enters and exits a spinning loop, we insert a call-down to notify hardware that the thread becomes inactive or active, respectively. The next section explains how hardware performs an online calculation of criticality based on these call-downs.

3.2 Calculating criticality

To calculate the criticality as defined in Equation 1, we need to know for each time interval which threads are performing useful work. To that end, we propose a small hardware component that keeps track of the running threads and the criticality of each thread. There is one criticality counter per thread (64 bit) and an 'active' bit that indicates whether the thread is running or not (see Figure 3). Each thread's criticality active bit is set or reset through the thread (de)activate calls that are sent from software. The cores or hardware contexts receive the calls from software, and send a signal to update the criticality state. These signals coming from the cores can either be transmitted over dedicated lines (a single line is sufficient for setting one bit), or through the existing interconnection network. In both cases, they do not incur much overhead, because the signal is only one bit and is sent relatively infrequently (we discuss frequency later in this section).

In addition to the per-thread counters and active bits, there is a counter that holds the number of active threads and a timer (see the bottom of Figure 3). The active thread counter is simply incremented when an activate call is received, and is decremented when a thread deactivates. The timer keeps track of absolute time (hence it is independent of a core's frequency) since the previous synchronization event and is reset whenever an activate or deactivate call is received. Initially when a software call is received, the timer holds the duration of the past interval. Thus, before updating state, we add the result of the timer divided by the active thread counter to each thread's criticality counter for which the active bit is set. Then, the active bits and counter are updated and the timer is reset, indicating the start of a new time interval.

While conceptually we need a counter per thread, we can implement one counter per core or hardware context in reality (even when there are more threads than hardware contexts). Only while threads are running do their criticality counters need to be updated (inactive threads do not receive criticality anyway); thus, keeping one hardware counter plus active bit per core or hardware context allows running threads to update their criticality state. Upon a context switch, the operating system saves the criticality state for the thread being scheduled out, and initializes the core or context's criticality state to that of the thread becoming active. Thus, our implementation works with more threads than cores.

The advantage of using a dedicated hardware component is that it has negligible impact on the performance of a running application. The application just sends the (asynchronous) activate/deactivate calls and can continue its execution without waiting for an answer. In terms of hardware overhead, we need 65 bits per thread (a 64-bit timer plus the 'active' bit). For sixteen threads, this amounts to a total of 1,108 bits. Additionally, we need one integer divider (the interval duration is usually much larger than the number of threads, so the fraction after the decimal point can easily be ignored), and one 64-bit adder per thread. (Note the divider and adders can be low-performance, low-power units because they are off the processor's critical path.) In other words, the hardware overhead for computing criticality stacks is limited.

To calculate the power overhead, we recorded the number of updates per 10 ms time slice. For 16 threads, there are 1,920 updates per time slice on average, with a maximum of 31,776 updates. On every update, we need to perform an integer division and at most 16 additions (assuming 16 threads). According to Wattch [6], an integer division consumes approximately 0.65 nJ and an addition consumes 0.2 nJ in a 100 nm chip technology; energy consumption is likely to be (much) lower in more recent chip technologies, hence these estimates are conservative. This implies a maximum of 3.85 nJ per update, and by taking into account the number of updates per unit of time, this leads to an average 7.39 μW power consumption, and 0.12 mW at most, which is very small compared to the power consumed by modern-day high-end processors (around 100+ W).

no. of cores	8, 16
core type	4-wide out-of-order
base frequency	2 GHz
L1 D-cache	64 KB, private, 2 cycles
L1 I-cache	64 KB, private, 2 cycles
L2 cache	512 KB, private, 10 cycles
L3 cache	8 MB, shared, 10 ns
memory bus	32 GB/s
memory access	100 ns

Table 1: Simulated processor configurations.

4. EXPERIMENTAL SETUP

We conduct full-system simulations using gem5 [5]. Table 1 shows the configurations of the simulated multicore processors. We consider eight- and sixteen-core processors, running eight- and sixteen-threaded versions of the parallel benchmarks, respectively. Each core is a four-wide superscalar out-of-order core, with private L1 and L2 caches, and a last-level L3 cache that is shared among cores. The OS that we run is Linux version 2.6.27; a thread is pinned onto a core to improve data locality and reduce the impact of context switching.

We consider benchmarks from the SPLASH-2 [32], PARSEC [4] and Rodinia [8] benchmark suites, see Table 2. We evaluate those benchmarks from the suites that correctly execute on our simulator for both eight and sixteen threads, and for which thread-to-core pinning could be done reliably (i.e., there is a unique thread-to-core mapping). The bench-

Suite	*Benchmark*	*Input*
SPLASH-2	Cholesky	tk29.O
	FFT	4,194,304 points
	FMM	32,768 particles
	Lu cont.	1024×1024 matrix
	Lu non-cont.	1024×1024 matrix
	Ocean cont.	1026×1026 ocean
	Ocean non-cont.	1026×1026 ocean
PARSEC	Canneal	Simmedium
	Facesim	Simmedium
	Fluidanimate	Simmedium
	Streamcluster	Simmedium
Rodinia	BFS	1,000,000 nodes
	Srad	2048×2048 matrix
	Lud_omp	512×512 matrix
	Needle	4096×4096 matrix

Table 2: Evaluated benchmarks.

marks were compiled using gcc 4.3.2 and glibc 2.6.1. Our experimental results are gathered from the parallel part of the benchmarks. Profiling starts in the main thread just before threads are spawned and ends just after the threads join (however, there is the possibility of sequential parts of code within this region). This approach factors out the impact of the trivial case of speeding up the sequential initialization and postprocessing parts of the program, and allows us to use criticality information to analyze the challenging parallel part of the program.

While our evaluation is limited to these programs, which have both critical sections and barriers, criticality stacks could also be useful for analyzing heterogeneous applications. The criticality stack for pipelined parallel programs can reveal the thread or pipeline stage that most dominates running time. Similarly, our criticality metric could reveal imbalances in a task stealing context as well. In addition, the criticality metric can be calculated for setups with more threads than cores.

5. THREAD CRITICALITY VALIDATION AND ANALYSIS

We now present criticality stacks for our parallel applications. We computed criticality for each thread for all benchmarks with 8 and 16 thread configurations, and present stacks that summarize thread criticality. We validate our criticality metric in the next section using frequency scaling of individual threads. We then compare our speedups to those achieved by scaling a thread identified to be critical by previous work that is based on cache misses. Finally, we show the variance when scaling over a range of frequencies.

5.1 Validation of criticality stacks

Figure 4(a) shows the criticality stacks for all benchmarks when executed on 8 cores (we omit 16-core stacks for space and readability considerations), with 100% of the execution time broken up into each thread's criticality percentage. For some benchmarks, all criticality components are approximately equal-sized (Cholesky, FFT, Lu cont., Ocean cont., Ocean non-cont., Canneal, and Srad). There is no critical thread in these cases, and thus we expect speeding up any single thread will yield no performance gain. For the other benchmarks, one thread has a significantly larger fraction of criticality compared to the others: thread 2 for FMM, Lu non-cont. and Streamcluster; thread 0 for Facesim, BFS, Lud-omp and Needle; and thread 5 for Fluidanimate. This is the most critical thread, and it is expected that speeding it up will result in a considerable performance gain, while speeding up other threads will have no significant performance impact. If an application's stack revealed more than one most critical thread, we would expect speeding up each of those would improve performance.

We evaluate the validity of criticality stacks by checking that accelerating the most critical thread (if one exists) results in program speedups. Each simulation speeds up one thread by raising the core's frequency from 2 GHz to 4 GHz[2], and we present speedup results versus a baseline of all threads at 2 GHz in Figure 4(b) for 8 threads. For each benchmark we present the speedup obtained by accelerating each of the three threads that have the largest components in the criticality stack. For the other threads, the speedup was equal to or lower than the speedup of the third largest component.

Figure 4(b) shows that for the benchmarks that have equal-sized components in the criticality stack (see Figure 4(a)), there is no single thread that when accelerated results in a significant program speedup, which is in line with expectations. For the other benchmarks, speeding up the thread that has a significantly larger criticality than the other threads results in a considerable speedup for the whole program (e.g., Lu non-cont. and BFS have speedups over 20% and over 30%, respectively). Moreover, speeding up the thread with the largest component results in the largest speedup, while speeding up threads with smaller, roughly-equal components yields little or no speedup. One interesting phenomenon is Streamcluster, which has a few other threads besides thread 2 that have slightly larger criticality percentages, and thus each of the three threads show some speedup after being scaled up. This validates that criticality stacks provide useful and accurate information that can be used to guide optimizations.

FMM is an exception to the rule because the criticality stack reveals that thread 2 is more critical than the others, but there is no speedup when this thread is accelerated. In fact, speeding up any single thread for this program never yields a significant speedup. Looking at the criticality stack after speeding up thread 2 revealed that that thread's component was reduced, but thread 7's component had grown significantly. FMM is an anomaly; for other benchmarks, speeding up the most critical thread resulted in a criticality stack with more equal-sized components. The criticality of the second thread for FMM is hidden, or overlapped, by the criticality of the first thread. Accelerating one thread just makes the other become more critical. However, we will show in Section 7.1 that by dynamically alternating the accelerated thread per time slice, we get a larger speedup for this benchmark. For the rest of the results, we focus on the benchmarks that have a most-critical thread, and thus parallel imbalance, that can be targeted for acceleration.

5.2 Comparison to prior criticality metric

We compare the performance improvement of speeding up one thread that is identified as most critical for various ways of identifying the critical thread in Figure 5. We

[2]This frequency raise is not intended to resemble a practical situation, it serves only as a way to validate the criticality stacks. We present a more realistic frequency scaling policy in Section 7.

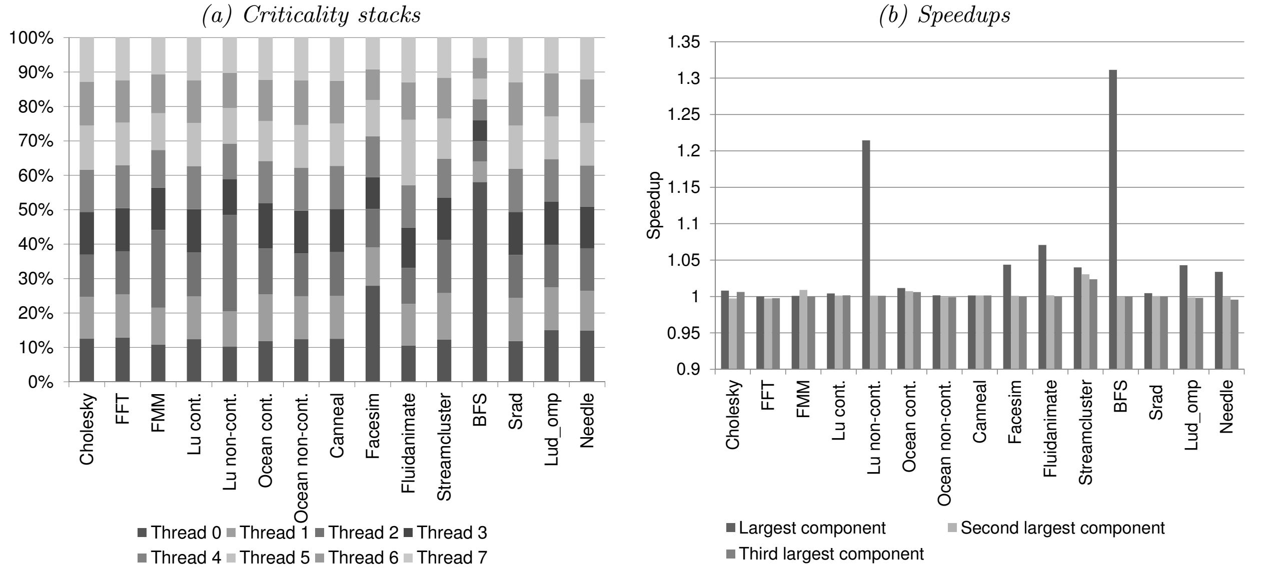

Figure 4: Criticality stacks for all benchmarks for 8 threads and corresponding speedups by accelerating one thread.

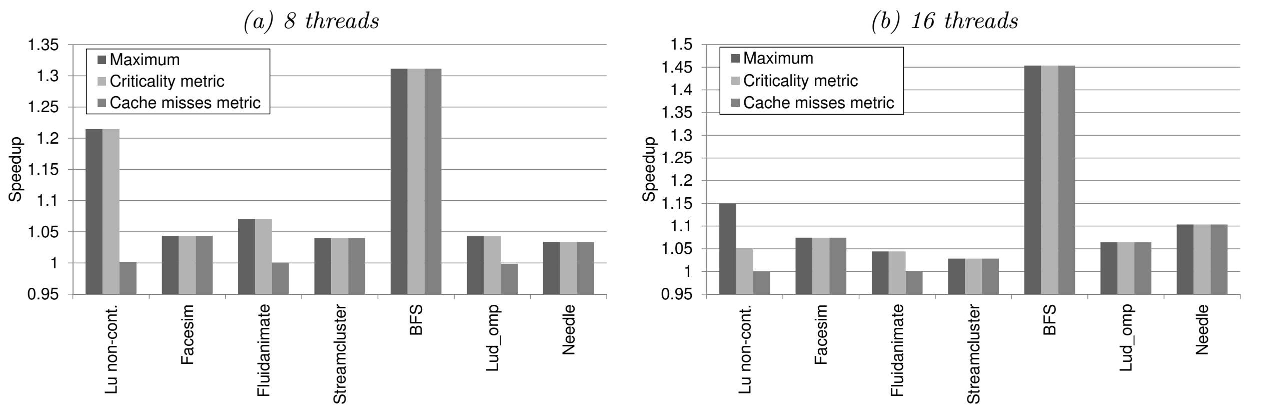

Figure 5: Comparison between our and a prior metric, and the maximum achievable speedup by accelerating one thread.

limit ourselves to accelerating one thread here because most of our benchmarks have only one most critical thread, but if more were detected, more threads could be accelerated. We present speedup results for the benchmarks that have a critical thread, i.e., speeding up a single thread results in a speedup of at least 3%. We present the results using a theoretical technique that takes the maximum speedup gained when accelerating each thread individually. We compare this with our criticality metric as defined in Section 2 and with previous work that uses cache misses to define criticality [3]. The cache miss metric takes a weighted average of the number of L1, L2 and L3 cache misses[3], with the relative latency as a weighting factor.

Our newly proposed criticality metric achieves the same speedup as the maximum achievable speedup in all cases but one. For the 16-threaded version of Lu non-cont., there are two criticality stack components that are significantly larger than the others (thread 0 and thread 2). The maximum speedup is achieved by accelerating the second largest component (thread 2). A detailed analysis reveals that in the beginning of the program, thread 0 is executing alone for a while, spawning threads and distributing data, resulting in a large criticality component. However, this process is very memory-intensive and results in many cache misses. Since the access time to memory is constant, raising the frequency of that core does not yield a significant speedup. After initialization, thread 2 becomes more critical, but its criticality does not exceed the accumulated criticality of thread 0. Although it is not the largest component, accelerating thread 2 yields the largest overall speedup. We will show in Section 7.1 that by dynamically changing the accelerated core, Lu non-cont. achieves a slightly higher speedup than accelerating only the one most critical thread for the entire program.

Figure 5 reveals that using cache misses to identify critical threads is less accurate at identifying critical threads and does not lead to any performance gains for three benchmarks in the 8-thread configuration, and for two benchmarks with

[3]We adapted the original formula in [3] to three levels of cache for our configuration.

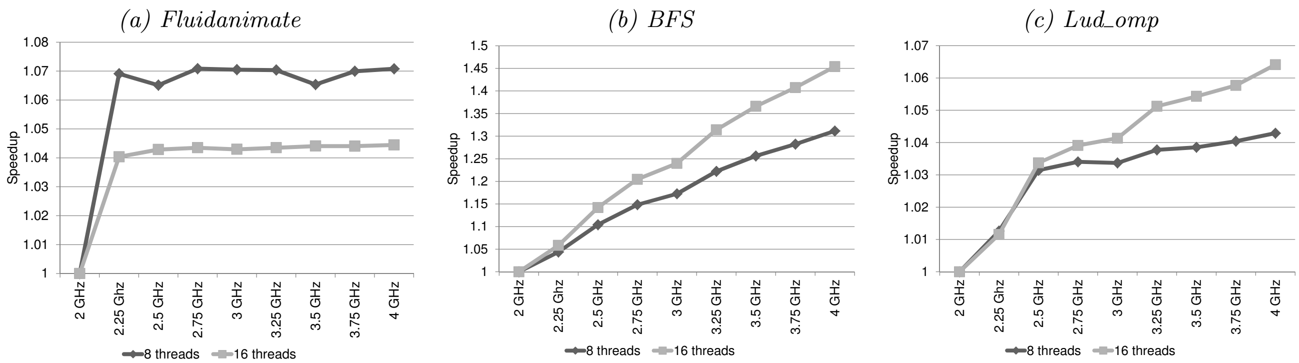

Figure 6: Impact of frequency scaling on achieved speedup.

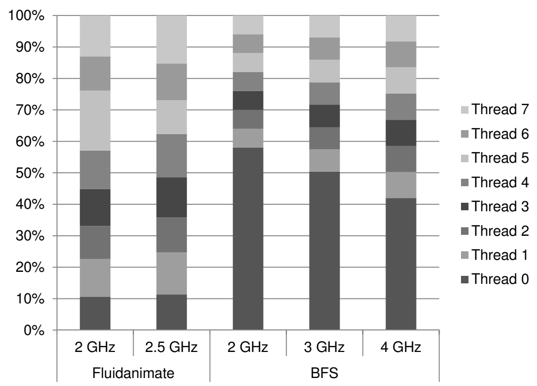

Figure 7: Impact of frequency scaling on criticality stacks.

16 threads, while our criticality metric always improves performance. The cache miss metric has been proven effective in barrier-synchronized parallel programs, while our new metric covers all types of synchronization. We conclude that our newly proposed metric is most effective at finding the thread most critical to performance.

5.3 Varying the amount of frequency scaling

In the previous experiments, we raised the frequency of one thread from 2 GHz to 4 GHz, and now we explore more realistic frequencies between 2 GHz and 4 GHz, at increments of 0.25 GHz. We use our metric to find the most critical thread to speed up, and evaluate the impact of frequency scaling on the total program speedup, which reveals interesting insights about the applications. Figure 6 shows the resulting speedups for three representative benchmarks, and Figure 7 shows the criticality stacks for a subset of these frequencies.

These three benchmarks show different behavior as we scale frequency up. For Fluidanimate, Figure 6(a) shows that program speedup increases from 2 to 2.25 GHz, but remains constant when the frequency is raised further. This is a typical case of inter-thread synchronization criticality. Once the thread that other threads are waiting for is sped up enough such that the other threads do not have to wait anymore, no further speedup can be attained despite a faster core. This is also reflected in the change between the two criticality stacks on the left of Figure 7: after speeding up the most critical thread (thread 5), its criticality component shrinks, making the thread non-critical, and thus no further speedup can be obtained.

For BFS in Figure 6(b), the performance continues to improve as the frequency increases. BFS includes an inherently sequential part where only one thread is running, which continues to see performance improvements when sped up to higher and higher frequencies. When looking at the three criticality stacks for BFS on the right side of Figure 7, we see that after accelerating the most critical thread, this thread's component decreases, but remains the largest component.

In Figure 6(c), Lud_omp displays a mix of the behavior of the two previous cases: in the beginning the speedup raises considerably, while after a certain frequency (2.5 GHz), speedup goes up at a slower pace. This benchmark's critical thread shows both inter-thread synchronization criticality and sequential criticality. For applications such as this, setting the frequency of the critical thread to the place where speedup slows, yields the best performance and energy consumption balance.

6. USE CASE #1: SOFTWARE OPTIMIZATION

Having validated criticality stacks, we now consider three use cases in order to illustrate the broad applicability of criticality stacks, ranging from software optimization, to dynamically accelerating critical threads and saving energy. We consider software analysis and optimization in this section, and the other two applications in two subsequent sections.

To illustrate the use case of criticality stacks for facilitating program analysis and optimization, we refer to the right side of Figure 7 which shows that BFS suffers from excessive critical imbalance, even when the most critical thread is sped up to a high frequency. We investigated this benchmark further to determine whether, as predicted, there is some sequential part of the program that slows down progress. The main work of BFS, which does breadth-first search of a tree data structure, is performed in a `do-while` loop. Inside the loop are two `for` loops that loop over all of the nodes in the tree. Only the first is parallelized. The first loop visits the edges of each node, potentially updating data. The second, unparallelized loop goes over each node of the tree, checking if it was updated. If there were updates, it sets the `do-while` flag to loop again, otherwise the `do-while` loop can terminate. We surmise that the most critical thread identified with our stacks, thread 0, is responsible for performing the second `for`-loop, which runs sequentially.

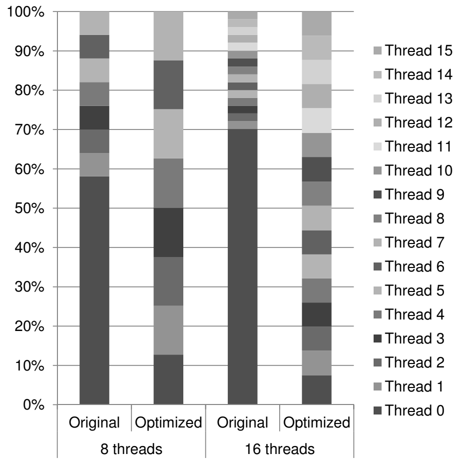

Figure 8: Example of using criticality stacks as a guide for software optimization (BFS benchmark).

We analyzed the second loop, determined it has no dependencies between iterations, and optimized it by parallelizing the loop. After this small program change, Figure 8 presents the comparison between the unoptimized and optimized BFS criticality stacks, for both 8 and 16 threads. While Figure 7 shows that scaling to even large frequencies did not remove the criticality bottleneck, with software analysis and editing, we achieve balanced criticality stacks, as seen on the right in Figure 8. After this code change, BFS achieves a 1.67× and 2.16× speedup for 8 and 16 cores, respectively. These improvements are significantly better than the 31% and 45% speedups that are achieved through frequency scaling alone to 4 GHz (in Figure 6(b)). This use case illustrates that criticality stacks are a useful tool to assist software programmers in analyzing and fixing parallel imbalance.

7. USE CASE #2: CRITICAL THREAD ACCELERATION

Our second use case dynamically accelerates a critical thread during execution using frequency scaling. We evaluate an algorithm that dynamically measures thread criticality over a time interval, and scales up the identified most critical thread in the next interval. While we evaluate frequency scaling on only one thread, scaling multiple threads could be an option if criticality stacks reveal that this could be worth the energy cost. This dynamic optimization requires no offline analysis, reacts to phase behavior, and improves parallel program execution time.

We first detail our dynamic algorithm, then compare results of our dynamic approach with those gathered for our offline approach in Section 5.2. We compare our dynamic approach with prior work called BIS, showing we almost double their performance improvements. In Section 8, we will show that this dynamic algorithm leads to a more energy-efficient execution of our parallel applications.

Exploring a large frequency range in Section 5.3 showed that using a 2.5 GHz frequency achieves the largest speedups relative to the amount of scaling, while not overly consuming energy. In these experiments, we raise a critical thread's frequency to 2.5 GHz. We assume a multicore processor with a base frequency of 2 GHz, where the processor's Thermal

if $f = \emptyset$ **then**
  if $\max(C_i)/\min(C_i) > \alpha$ **then**
    $f :=\text{maxindex}(C_i)$
  end
else
  if $\max(C_i)/\min_{i \neq f}(C_i) > \alpha$ **then**
    $f :=\text{maxindex}(C_i)$
  else
    if $C_f/\max(C_i) < \beta$ **then**
      $f := \emptyset$
    end
  end
end

Algorithm 1: Dynamic frequency scaling algorithm. f is the currently accelerated core; C_i is the criticality for each thread; 'maxindex' finds the index of the core with maximum criticality.

Design Power (TDP) allows one and only one core's frequency to be increased. We use a time interval of 10 ms for our dynamic algorithm. At the start of a new time interval, we reset criticality counters. Over the time interval, the hardware calculates each thread's criticality sum using the method described in Section 3.2. Algorithm 1 details how these criticality numbers are used to decide which (if any) core to scale up in the next time interval.

Algorithm description. Initially, we check if there is currently an accelerated core, tracked with f. Calculated criticality numbers are stored in C_i for each thread. If no core is accelerated ($f = \emptyset$), we calculate the ratio between the largest and the smallest criticality. If the result is larger than a certain threshold α (the base value is 1.2), then the frequency of the core running the thread with the largest criticality component is raised (by setting f to the index of the core with maximum C_i).

If a core was accelerated in the previous interval, we check the ratio of the largest criticality to the smallest criticality that is not the currently accelerated core (taking the second-smallest criticality if the smallest is for the accelerated thread). We perform this check to prevent constantly scaling up and down a core, since speeding up a thread will usually result in a smaller criticality component. If this ratio is above our α threshold, we raise the frequency of the core running the most critical thread (slowing down the previously accelerated thread if it is different). If the ratio is not larger than the threshold, the algorithm calculates the ratio of the criticality of the thread running on the accelerated core to the largest criticality. If this ratio is smaller than a β threshold (with a base value of 0.8), then the accelerated thread is slowed down again. This check prevents continuously accelerating a core without seeing a performance benefit, as a thread that was initially critical can eventually become non-critical. We performed experiments in which we vary time interval duration, and the α and β parameters, but found little performance difference as compared to using the base values.

In addition to this proactive algorithm, we implemented two straightforward reactive mechanisms to further reduce energy consumption and improve performance. First, when an accelerated thread is scheduled out, we reduce the frequency of that core to the base frequency, as speeding up

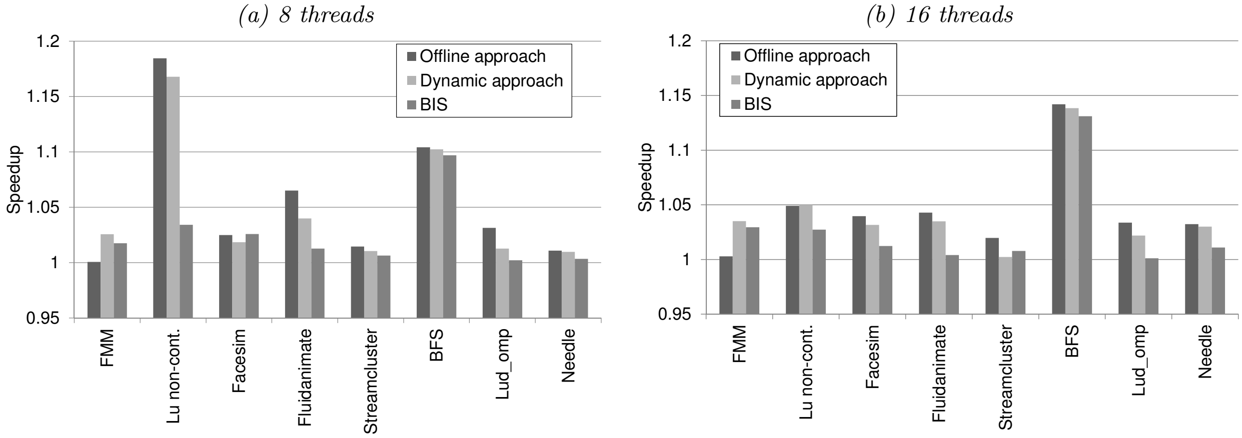

Figure 9: Results for the dynamic frequency scaling policy.

that thread has no performance benefit. Secondly, when there is only one thread active, and that thread is currently not accelerated, we scale up the frequency of the core running that thread. In this case, the running thread is by definition the most critical thread, and should be accelerated. Our dynamic optimization algorithm reacts to changing circumstances and variation in thread criticality over a program run.

7.1 Effectiveness of dynamic optimization

Figure 9 shows the performance results of our dynamic frequency scaling technique for both 8 and 16-threaded configurations[4]. Because FMM's total program criticality stack did reveal a most critical thread, we include it again in our dynamic results, despite the fact that speeding up one thread over the whole execution did not improve performance. Each benchmark has three bars. The first bar is the speedup obtained by the offline approach, i.e., profiling the program and running the program again while speeding up the most critical thread over the whole program execution. The next bar shows the speedup obtained by our dynamic approach. The last bar shows the results for BIS, which we discuss in the next section.

For both 8 and 16-threaded runs, FMM achieves larger speedups with our dynamic approach than the offline approach. Although the offline approach could not improve FMM's performance, our dynamic approach deals better with the overlapping criticality, and improved performance by about 3%. Also, as discussed in Section 5.2, the offline approach could not solve 16-threaded Lu non-cont.'s problem that one thread was most critical initially and another was critical later in the program. The dynamic approach slightly improves upon the performance of Lu non-cont. with 16 threads, adapting to the most critical thread during each program phase.

For the other benchmarks, the speedups of the dynamic approach are slightly smaller than those of the offline approach. This is due to the reactiveness of the dynamic algorithm: frequency is only scaled up after a critical thread is detected in the previous time interval. However, the dynamic approach achieves similar program speedups with more energy-efficiency run by not always scaling up the frequency. On average, the dynamic approach adapts to phase behavior, obtaining a speedup of 4.4%, compared to 4.8% for the offline approach, while speeding up one thread from 2 GHz to 2.5 GHz for 71% of the time on average.

7.2 Comparison to BIS

We compare the results of our dynamic frequency scaling algorithm to the best-performing previous work which accelerates synchronization bottlenecks instead of threads, called Bottleneck Identification and Scheduling (BIS) [15]. They focus on accelerating the most critical bottleneck, e.g., a critical section that is heavily contended or a barrier with many threads waiting for a significant amount of time. When a thread encounters such a bottleneck, it is temporarily migrated to a faster core in a heterogeneous system. We reimplemented their technique but instead of thread migration, we use core frequency scaling (to 2.5 GHz) in our experimental setup.

Figure 9 presents the speedup for each benchmark using our dynamic algorithm against those obtained using the BIS technique. For 8-threaded benchmarks, in Figure 9(a), we see our criticality metric outperforms BIS in all but one benchmark, significantly outperforming BIS for Lu non-cont. by speeding up the benchmark 17% compared to 3% for BIS. Similarly, our dynamic algorithm improves upon BIS's speedup in all 16-threaded benchmarks except for Streamcluster. We found that our technique is more effective at speeding up programs that have many barriers, because we speed up more of the whole thread's execution instead of only when a single thread that has yet to reach the barrier. For programs with many heavily contending critical sections, BIS might achieve better performance. Overall, our dynamic scheme achieves an average of 4.6% speedup in comparison with BIS's 2.4% for 8 threads. For 16 threads, we speed up on average by 4.2%, almost doubling BIS's improvement of 2.7%.

8. USE CASE #3: ENERGY OPTIMIZATION

While we have shown that criticality stacks are useful for identifying parallel thread imbalance, and accelerating the most critical thread achieves program speedups, we now also

[4]We do account for the (small) overhead incurred when scaling frequency in our simulation experiments.

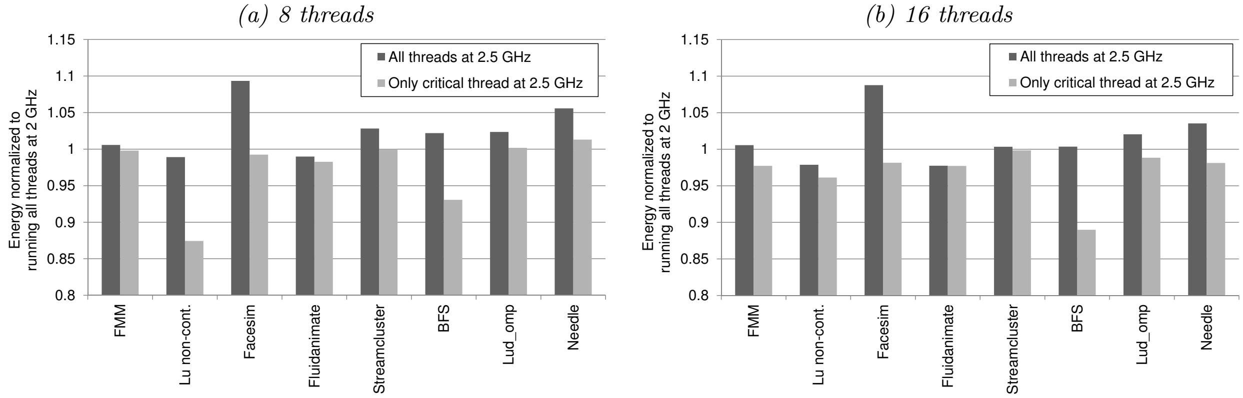

Figure 10: Comparison of energy consumed when running all threads at 2.5 GHz and only the most critical at 2.5 GHz using our dynamic scheme, compared to running all threads at 2 GHz.

demonstrate that criticality stacks are good for targeting optimization towards saving energy. While we want to achieve maximum performance for parallel programs, power and energy are first-order concerns in modern systems, including the embedded and server domains.

We perform an experiment to compare the energy consumed when running our multi-threaded benchmarks at various frequencies. We run once with all threads at 2 GHz, once with all threads at 2.5 GHz, and once using our dynamic technique to accelerate only the most critical thread to 2.5 GHz. Obviously, running all threads at the higher frequency will result in a larger power output. Figure 10 presents the energy consumed, which is power multiplied by execution time, for our benchmarks with 8 and 16 threads. We present energy numbers for all threads at 2.5 GHz, and only the critical thread at 2.5 GHz, normalized to the energy consumption for all threads at 2 GHz. We estimate power consumption using McPAT [18] (assuming a 32 nm technology).

Figure 10 shows that accelerating all threads to the higher frequency consumes more energy than accelerating only one thread for all of our benchmarks. For both Lu non-cont. and Fluidanimate, running with all threads at 2.5 GHz consumes slightly less energy than with all threads at 2 GHz, because it results in large program speedups. However, if energy is of prime concern, we see the best result comes from targeting acceleration only at the most critical thread. For almost all benchmarks, using our dynamic algorithm reduces the energy consumed from all threads at 2 GHz. Particularly for BFS with 16 threads, and Lu non-cont. with 8 threads, we reduce the energy consumed by 11% and 12.6%, respectively. Also, targeting acceleration to the thread identified as most critical by our metric particularly benefits Facesim, which consumes about 10% more energy when all threads are accelerated. Overall, when all threads are executed at 2.5 GHz, the total energy consumption *increases* by 1.3% for 16 threads and 2.5% for 8 threads. In comparison, by accelerating only the critical thread, the total energy consumption is *reduced* by 3.2% on average for 16 threads and 2.8% for 8 threads.

In summary, through this use case, we have demonstrated that our criticality stacks are good at not only informing dynamic optimization to improve parallel program performance, but also at targeting this optimization to minimize the critical resource of energy.

9. RELATED WORK

A significant amount of prior work exists that tries to understand and optimize criticality. However, prior work focuses on other forms of criticality or aims at optimizing thread waiting time in parallel programs. We instead propose an intuitive metric for thread criticality in parallel, shared-memory programs that is a useful tool to optimize performance.

9.1 Criticality analysis

Understanding program criticality is challenging because of various interaction and overlap effects across concurrent events, be it instructions or threads. Fields et al. [12] and Tune et al. [31] proposed offline techniques to analyze instruction criticality and slack based on data and resource dependencies in sequential programs. Li et al. [19] extended offline approach to shared-memory programs. Hollingsworth [14] proposed an online mechanism to compute the critical path of a message-passing parallel program. Saidi et al. [27] use critical path analysis to detect bottlenecks in networking applications. More recently, Cheng and Stenström [9] propose an offline analysis to detect critical sections on the critical path. None of this prior work addressed thread criticality in parallel, shared-memory programs with general synchronization primitives (including critical section, barrier and pipelined synchronization). Thread criticality stacks as proposed in this paper can be computed both offline and online.

9.2 Parallel program analysis

Tallent et al. [29] use an online profiling tool to measure idle time and attribute it to the specific locks that caused this idle time. As discussed in [9], idle time does not always point to the most critical locks. Speedup stacks [10] present an analysis of the causes of why an application does not achieve perfect scalability. Speedup stacks measure the impact of synchronization and interference in shared hardware resources, and attribute the gap between achieved and ideal speedup to the different possible performance delimiters. However, speedup stacks present no data on which thread could be the cause, and do not suggest how to overcome the scalability limitations they identify. Criticality stacks point to the threads that are most critical and should be targeted for optimization.

9.3 Reducing thread waiting time

Improving parallel performance by reducing thread waiting time is a well-known optimization paradigm. Many previously proposed mechanisms apply this conventional wisdom for specific performance idioms. Our novel criticality stack can steer optimizations in an energy-efficient way to only the most critical threads.

Threads wait for several reasons. The most obvious case is serial execution parts of a parallel program [1]. When there is only one thread active doing useful work, optimizing its performance is likely to yield substantial performance benefits. Annavaram et al. [2] optimize serial code by running at a higher clock frequency; Morad et al. [24] run serial code on a big core in a heterogeneous multicore.

Critical sections guarantee mutual exclusion and lead to serialization, which puts a fundamental limit on parallel performance [11]. Removing or alleviating serialization because of critical sections has been a topic of wide interest for many years. Transactional Memory (TM) aims to overlap the execution of critical sections as long as they do not modify shared data [13]. Speculative Lock Elision [25], Transactional Lock Removal [26] and Speculative Synchronization [22] apply similar principles to traditional lock-synchronized programs. Suleman et al. [28] use the big core in a heterogeneous multicore to accelerate critical sections.

Several techniques have been proposed to improve performance and/or reduce energy consumption of barriers, which all threads have to reach before the program proceeds. In thrifty barriers [17], a core is put into a low-power mode when it reaches a barrier with a predicted long stall time. Liu et al. [21] improve on that by reducing the frequency of cores running threads that are predicted to reach a barrier much sooner than other threads, even when they are still executing. Cai et al. [7] keep track of how many iterations of a parallel loop each thread has executed, delaying those that have completed more, and giving more resources to those with fewer in an SMT context. Age-based scheduling [16] uses history from the previous instance of the loop to choose the best candidate for acceleration. While previous works all target a specific synchronization paradigm (barriers and parallel loops), our metric is independent of the type of synchronization, and can profile every (instrumented) stall event due to synchronization.

As discussed in the introduction, thread-criticality prediction (TCP) [3] aims at estimating load imbalance in barrier-synchronized parallel programs by correlating criticality to cache misses. Their predictions are used to steal work from critical threads to improve performance, or to reduce the frequency of cores running non-critical threads. We showed in Section 5.2 that our metric finds threads that are more critical, and when accelerated, result in higher speedups.

Turbo Boost[5] increases the core frequency when there are few active cores. As such, for multi-threaded programs, it increases thread performance when parallelism is low. Booster [23] speeds up threads that hold locks or that are active when other threads are blocked, using a dual voltage supply technique. Bottleneck Identification and Scheduling (BIS) by Joao et al. [15] accelerates synchronization primitives (locks, barriers, pipes) with large amounts of contention by migrating them temporarily to a faster core in a heterogeneous multicore. The methods used by both Turbo Boost and Booster to identify threads that need to be accelerated are a subset of the methods used by BIS, which means that the BIS results in Section 7.2 are an upper bound for the results for Turbo Boost and Booster. While BIS optimizes bottlenecks, we identify the thread(s) most critical to overall performance. Optimizing bottlenecks does not necessarily imply improved overall performance, because they also accelerate non-critical threads. In Section 7.2, we showed that our dynamic algorithm results in a higher speedup than BIS for barrier-bound applications.

[5]http://www.intel.com/technology/turboboost

10. CONCLUSIONS

We present a technique to analyze parallel program thread imbalance due to synchronization, and use that to guide online optimizations to speed up multi-threaded applications. We introduce a novel, intuitive criticality metric that is independent of synchronization primitives, which takes into account both a thread's active running time and the number of threads waiting on it. We also design criticality stacks that break down execution time visually based on each thread's criticality, facilitating detailed analysis of parallel imbalance.

We present a simple hardware design that takes a very small amount of power, while being off the processor's critical path, to compute criticality stacks during execution. We validate the accuracy and utility of criticality stacks by demonstrating that our low-overhead online calculation approach indeed finds the thread most critical to performance, improving over a previously proposed metric based on cache misses.

We then present three use cases of criticality stacks to illustrate their broad applicability to (1) optimize software code, (2) dynamically accelerate the critical thread to improve performance, even doubling over the best-performing previous work, and (3) target optimizations of parallel programs to reduce energy consumption. From these case studies, we report that (1) after optimizing the code of one benchmark based on criticality imbalance, we achieve an average speedup of 1.9×; (2) our dynamic algorithm reacts to application phase changes, achieving an average speedup of 4.4%, and up to 17%; (3) by accelerating the most critical thread, we also reduce the total energy consumption by 3% on average, and up to 12.6% (while at the same time improving performance). Overall, we conclude that criticality stacks are instrumental to understanding and improving parallel performance and energy.

11. REFERENCES

[1] G. M. Amdahl. Validity of the single-processor approach to achieving large-scale computing capabilities. In *Proceedings of the American Federation of Information Processing Societies Conference (AFIPS)*, pages 483–485, 1967.

[2] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's law through EPI throttling. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 298–309, June 2005.

[3] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 290–301, June 2009.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the International*

Conference on Parallel Architectures and Compilation Techniques (PACT), pages 72–81, Oct. 2008.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *Computer Architecture News*, 39:1–7, May 2011.

[6] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 83–94, June 2000.

[7] 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 *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 240–249, Oct. 2008.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, Oct. 2009.

[9] G. Chen and P. Stenström. Critical lock analysis: Diagnosing critical section bottlenecks in multithreaded applications. In *Proceedings of Supercomputing: the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pages 71:1–71:11, Nov. 2012.

[10] S. Eyerman, K. Du Bois, and L. Eeckhout. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *Proceedings of the International Symposium on Performance Analysis of Software and Systems (ISPASS)*, pages 145–155, Apr. 2012.

[11] S. Eyerman and L. Eeckhout. Modeling critical sections in Amdahl's law and its implications for multicore design. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 362–370, June 2010.

[12] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 74–85, June 2001.

[13] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 289–300, June 1993.

[14] J. Hollingsworth. An online computation of critical path profiling. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 11–20, May 1996.

[15] J. Joao, M. Suleman, O. Mutlu, and Y. Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–234, Mar. 2012.

[16] N. B. Lakshminarayana, J. Lee, and H. Kim. Age based scheduling for asymmetric multiprocessors. In *Proceedings of Supercomputing: the International Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 199–210, Nov. 2009.

[17] J. Li, J. Martinez, and M. Huang. The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 14–23, Feb. 2004.

[18] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 469–480, Dec. 2009.

[19] T. Li, A. Lebeck, and D. Sorin. Quantifying instruction criticality for shared memory multiprocessors. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 128–137, June 2003.

[20] T. Li, A. R. Lebeck, and D. J. Sorin. Spin detection hardware for improved management of multithreaded systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 17:508–521, June 2006.

[21] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. Irwin. Exploiting barriers to optimize power consumption of CMPs. In *Proceedings of the International Symposium on Parallel and Distributed Processing*, page 5a, Apr. 2005.

[22] J. F. Martinez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 18–29, Oct. 2002.

[23] T. Miller, X. Pan, R. Thomas, N. Sedaghati, and R. Teodorescu. Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips. In *18th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Feb. 2012.

[24] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and A. Ayguade. Performance, power efficiency and scalability of asymmetric cluster chip multiprocessors. *IEEE Computer Architecture Letters*, 5(1):14–17, Jan. 2006.

[25] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 294–305, Dec. 2001.

[26] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 5–17, Oct. 2002.

[27] A. G. Saidi, N. L. Binkert, S. K. Reinhardt, and T. Mudge. End-to-end performance forecasting: finding bottlenecks before they happen. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 361–370, June 2009.

[28] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 253–264, Mar. 2009.

[29] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 269–280, Jan. 2010.

[30] C. Tian, V. Nagarajan, R. Gupta, and S. Tallam. Dynamic recognition of synchronization operations for improved data race detection. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 143–154, July 2008.

[31] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 185–195, Jan. 2001.

[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 24–36, June 1995.