Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors

Abhishek Bhattacharjee, Margaret Martonosi
Dept. of Electrical Engineering
Princeton University
{abhattac, mrm}@@princeton.edu

ABSTRACT

With the shift towards chip multiprocessors (CMPs), exploiting and managing parallelism has become a central problem in computer systems. Many issues of parallelism management boil down to discerning which running threads or processes are *critical*, or slowest, versus which are non-critical. If one can accurately predict critical threads in a parallel program, then one can respond in a variety of ways. Possibilities include running the critical thread at a faster clock rate, performing load balancing techniques to offload work onto currently non-critical threads, or giving the critical thread more on-chip resources to execute faster.

This paper proposes and evaluates simple but effective thread criticality predictors for parallel applications. We show that accurate predictors can be built using counters that are typically already available on-chip. Our predictor, based on memory hierarchy statistics, identifies thread criticality with an average accuracy of 93% across a range of architectures.

We also demonstrate two applications of our predictor. First, we show how Intel's Threading Building Blocks (TBB) parallel runtime system can benefit from task stealing techniques that use our criticality predictor to reduce load imbalance. Using criticality prediction to guide TBB's task-stealing decisions improves performance by 13-32% for TBB-based PARSEC benchmarks running on a 32-core CMP. As a second application, criticality prediction guides dynamic energy optimizations in barrier-based applications. By running the predicted critical thread at the full clock rate and frequency-scaling non-critical threads, this approach achieves average energy savings of 15% while negligibly degrading performance for SPLASH-2 and PARSEC benchmarks.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Parallel Architectures; C.4 [**Performance of Systems**]: Design Studies

General Terms

Design, Performance

Keywords

Thread Criticality Prediction, Parallel Processing, Intel TBB, DVFS, Caches

1. INTRODUCTION

While chip multiprocessors (CMPs) already dominate computer systems, key research issues remain in exposing and managing the parallelism required to fully exploit them. In particular, good performance for a parallel application requires reducing load imbalance and ensuring that processor resources (functional units, cache space, power/energy budget and others) are used efficiently. As a result, the ability to accurately predict *criticality* in a computation is a fundamental research problem. If the system can accurately gauge the critical, or slowest, threads of a parallel program, this information can be used for a variety of techniques including load rebalancing, energy optimization, and capacity management on constrained resources.

Predicting thread criticality accurately can lead to substantial performance and energy improvements. Consider, for example, the extreme case of a load-imbalanced parallel program in which one thread runs twice as long as the others. If this thread's work could be redistributed among all the other threads, performance would improve by up to 2×. If one instead focuses on energy improvements, then the other threads could be frequency-scaled or power-gated to save energy during their prolonged wait time. While this example is extreme, Section 2 shows that significant real-world opportunities exist in a range of benchmarks. The key challenge, however, is that one must accurately and confidently predict the critical thread, or else large performance and energy degradations can arise from responses to incorrect predictions.

This work focuses on low-overhead and general thread criticality prediction schemes that harness counters and metrics mostly available on-chip. Having tested a range of techniques based on instruction counts and other possibilities, we find that per-core memory hierarchy statistics offer the best accuracy for criticality prediction. Our work shows how to form memory statistics into useful thread criticality predictors (TCPs). We also explore pairing them with confidence estimators that gauge the likelihood of correct predictions and reduce the high-cost responses to incorrect ones.

To demonstrate its generality, we also apply our TCP hardware to two possible uses. First, we study how TCP can assist Intel Threading Building Blocks (TBB) [1] in improving task-stealing decisions for load balancing among threads. By identifying the critical thread in an accurate and lightweight manner, threads with empty task queues can "steal" work from the most critical thread and shorten program runtimes. In the second application study, we focus on barrier-based applications and use TCP to guide dynamic voltage and frequency scaling (DVFS) decisions. Here we show that gauging the *degree* of criticality of different threads can also be useful. By determining which threads are non-critical (and by how much) we can choose to run the most critical thread at a high clock rate, while operating other threads on cores that are frequency-scaled to be more energy efficient.

Overall, this paper both describes the implementation issues for TCP and also evaluates methods for using it. The key contributions of this work are:

1. We demonstrate that accurate predictions of thread criticality can be built out of relatively-accessible on-chip

ISCA'09, June 20–24, 2009, Austin, Texas, USA.

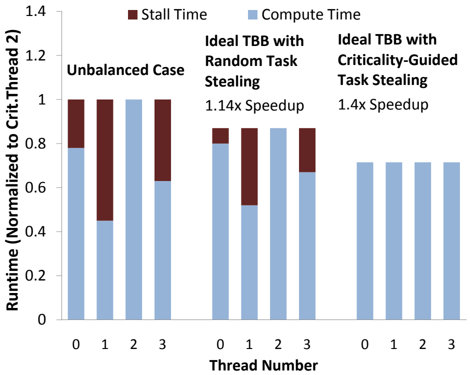

Figure 1: `Blackscholes` uses thread criticality predictors for TBB task stealing, outperforming the random task stealer.

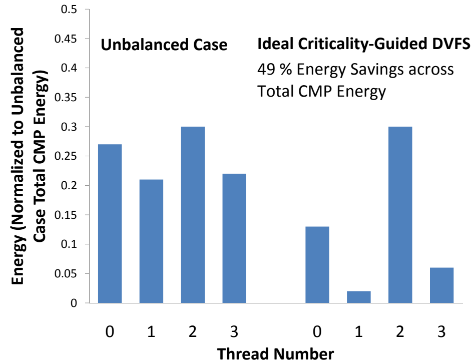

Figure 2: `Blackscholes` uses thread criticality predictors to DVFS threads, saving 49% energy without performance loss.

information such as memory hierarchy statistics. We describe our proposed TCP hardware and then test it on both in-order and out-of-order machines to show its effectiveness across different microarchitectures. Despite TCP's simple design, it attains an average prediction accuracy of 93% on SPLASH-2 [33] and PARSEC [4] workloads.

2. In our first usage scenario, we apply TCP to task-stealing in Intel TBB. Here, our predictions improve the performance of PARSEC benchmarks ported to TBB by 13% to 32% on a 32-core CMP. While our results are demonstrated with TBB, similar approaches could be built for other environments based on dynamic parallelism.

3. In our second usage scenario, TCP techniques guide DVFS settings. In particular, we identify threads that are confidently non-critical and DVFS them to save energy. In this scenario, TCP achieves an average energy savings of 15% for a 4-core CMP running SPLASH-2 and PARSEC workloads. Our approach to barrier energy reductions offers potentially higher payoffs with lower hardware overheads than prior proposals.

Beyond the two application scenarios evaluated in this paper, thread criticality prediction can be applied to many other parallelism management problems. For example, TCP could shape how constrained in-core resources like issue width or execution units are apportioned among parallel tasks. It can also help guide the allocation of more global resources such as bus and interconnect bandwidth.

This paper is structured as follows. Section 2 offers quantitative motivations for TCPs and our specific design objectives. After describing methodology in Section 3, Section 4 evaluates the possible architectural metrics for predicting criticality, quantifying the link between cache misses and thread criticality. Section 5 proposes a base TCP design, and Section 6 elaborates on this for the TBB task stealing application. Section 7 then details and evaluates TCPs for barrier energy optimizations. Finally, Section 8 addresses related work and Section 9 offers our conclusions.

2. OUR MOTIVATION AND OBJECTIVES

2.1 The Case for Criticality Predictors

Fundamentally, criticality prediction is aimed at estimating load imbalance in parallel programs. To explore the potential benefits of approaches using criticality prediction, we characterized the load imbalance for SPLASH-2 and PARSEC applications with the data-sets listed in Table 2 on a cycle-accurate CMP simulator described in Section 3. For example, `LU`, `Volrend`, `Fluidanimate` and `Water-Nsq` have at least one core stalling for 10% of the execution time at 4 cores, worsening to an average of 25% at 32 cores. These stall cycles arise from wait times at barriers and on shared locks, the former dominating in barrier-based applications. Already substantial, we expect load imbalance to worsen as future CMPs expose more performance variations due to technology issues and thermal emergencies [8, 13].

Since imbalance degrades performance and energy efficiency, it is useful to predict thread criticality and react accordingly. Figure 1 shows potential *performance* improvements for barrier 2 of `Blackscholes`. The first four bars plot thread compute and stall times (light and dark portions) normalized to critical thread 2. These show large thread stall times, with a worst-case stall time of 55% of runtime for thread 1. In response, a TBB version of this application can split work into small tasks and insert them into per-core task queues. When one thread runs out of work, it steals tasks; however TBB currently steals from randomly-chosen cores and the middle bars of Figure 1 shows that this only modestly improves performance. In contrast, the right set of bars show the potential of TCP-guided prediction. These bars depict an oracle criticality prediction, in which tasks are stolen from the slowest thread, for a much higher speedup of 1.4×.

As a second approach, Figure 2 sketches the potential *energy* benefits of TCPs. The first four bars plot per-thread energy normalized to the total energy across the cores for unbalanced `Blackscholes` whose per-thread performance has already been given on the left-hand-side of Figure 1. The relative performance of non-critical threads 0, 1, and 3 indicate that they could ideally be slowed down to 0.78, 0.45, and 0.63 of the nominal frequency to eliminate barrier stalls without performance degradation. The second set of bars in Figure 2 shows that as much as 49% of the total initial energy could be saved using this TCP-guided approach. The key challenge here lies, however, in identifying non-critical threads, gauging their relative non-criticality to set clock rates appropriately, and balancing the benefits of dynamic energy savings against the potential extra leakage energy and switching overhead costs. We address these issues in the TCP design detailed in the following sections.

2.2 Goals of our Predictor Design

Before detailing the criticality predictor hardware, we first state the particular goals of our TCP and differentiate them from important related work.

First, our TCP needs to be highly accurate. Since the TCP will be used to either redistribute work among threads for higher performance or to frequency-scale threads for energy-efficiency, we need to minimize TCP inaccuracy, which can give rise to performance and energy degradations.

Second, our TCP design should be low-overhead. If, for example, the TCP relies on complex software analysis, overall system performance may be adversely affected. On the other hand, if the TCP is designed with complex hardware, it may scale poorly for future CMPs. Therefore, we strike a balance by avoiding software-only TCPs that involve high performance overheads and using simple and scalable hardware. We do however allow software-guided policies to be built on top of the hardware if necessary.

Third, the TCP is designed for versatility across a range of

Infrastructure	*ARM-based In-order Simulator*	*GEMS Simulator*	*FPGA-based Full-System CMP Emulator*
System	4-32 core cache-coherent CMP	16 core cache-coherent CMP	4 core cache-coherent CMP
Cores	2-issue, in-order ARM	4-issue, out-of-order Sparc 32 entry inst. window 64 entry ROB	1-issue, in-order 7-stage pipeline
DVFS Settings	Nominal (1.2V, 1GHz) Eff 1 (1.02V, 850 MHz) Eff 2 (0.84V, 700 MHz) Eff 3 (0.66V, 550 MHz)	No DVFS Capability	Nominal (1.2V, 1GHz) Eff 1 (1.02V, 850 MHz) Eff 2 (0.84V, 700 MHz) Eff 3 (0.66V, 550 MHz)
L1 I-Cache (private)	32KB, 4-way, 32 byte lines	32KB, 4-way, 64 byte lines	4KB, 2-way, 32 byte lines
L1 D-Cache (private)	32KB, 4-way, 32 byte lines	32KB, 4-way, 64 byte lines	8KB, 2-way, 32 byte lines
L2 Cache (shared, unified)	4MB NUCA 8-way, 32 byte lines Avg. latency ~40 cc †	4MB NUCA 8-way, 64 byte lines Avg. latency ~ 40 cc	None
Main Memory	Avg. latency ~ 400 cc †	Avg. latency ~ 400cc	Avg. latency ~ 400cc †
Bus / Network	Mesh, dim.-routing	Mesh, dimension-routing	AMBA snooping bus
OS	None	Full Solaris 10	Full Linux 2.6
TCP App. Studied	TCP Accuracy Studies TBB Studies	TCP Accuracy Studies	DVFS Studies

† All latencies are relative to the nominal frequency

Table 1: Simulators and emulator used in our studies.

Benchmark	**Suite**	**Problem Size**	**TCP Application Studied**
LU	SPLASH-2	1024x1024 matrix, 64 x 64 blocks	DVFS
Barnes	SPLASH-2	65, 536 particles	DVFS
Volrend	SPLASH-2	head	DVFS
Ocean	SPLASH-2	514 x 514 grid	DVFS
FFT	SPLASH-2	4,194,304 data points	DVFS
Cholesky	SPLASH-2	tk29.O	DVFS
Radix	SPLASH-2	8, 388, 608 integers	DVFS
Water-Nsq	SPLASH-2	4096 molecules	DVFS
Water-Sp	SPLASH-2	4096 molecules	DVFS
Blackscholes	PARSEC	16,385 (simmedium)	DVFS, TBB
Streamcluster	PARSEC	8192 points per block (simmedium)	DVFS, TBB
Swaptions	PARSEC	32 swaptions, 5000 simulations (simmedium)	TBB
Fluidanimate	PARSEC	5 frames, 100K particles (simmedium)	TBB

Table 2: SPLASH-2 and PARSEC workloads used in our studies.

applications with different parallelization characteristics and applicable to a number of resource management issues. In this work, we address TCP's role in improving TBB performance and minimizing barrier-induced energy waste; however, the TCP has many other uses such as shared last-level cache management [17, 18, 30], SMT priority schemes [9, 25, 31], and memory controller priority schemes [15, 22, 27], just to name a few.

In the context of these goals, it is important to review prior work. While *instruction* criticality has been explored for superscalars [14, 32], interest in *thread* criticality prediction has been a more recent phenomenon. Consider, for example, the *thrifty barrier* [21], in which faster cores transition into low-power modes after they reach a barrier in order to reduce energy waste. The thrifty barrier uses barrier stall time prediction to choose among sleep states with different transition times. While considerable energy savings are achieved, this technique runs all threads at the nominal frequency until the barrier. In contrast, we predict criticality before the barrier, offering more flexibility in response.

In response to the thrifty barrier, Cai et al. propose *meeting points* to gauge thread criticality in order to save energy by frequency scaling non-critical threads [9]. While this does result in even greater energy savings than the thrifty barrier, it is intended for parallel loops. These loops are instrumented with specialized instructions to count the number of iterations completed by each core. The counter values are then periodically broadcast, allowing each core to predict thread progress and DVFS accordingly.

In contrast, our TCP is aimed at a range of parallelization schemes beyond parallel loops. We will show that our approach works well even for benchmarks like `LU`, which include loops with variable iteration times.

3. METHODOLOGY

The following sections present our methodology choices and setups.

3.1 TCP Accuracy Evaluation

Table 1 shows the two software simulators used to evaluate the accuracy of our TCP hardware. As shown, for evaluations on architectures representative of the high-performance domain (out-of-order execution, wide-issue etc.), we use Simics/GEMS [26]. We load both Ruby and Opal to simulate a 16 core cache-coherent CMP running Solaris 10. This configuration enjoys particularly high prediction accuracy because dynamic instruction reordering enables stable predictions. However, in-order microarchitectures, representative of the embedded space, see IPC and other metrics vary more in short periods of time. To compare these, we evaluate TCPs on the ARM CMP simulator using in-order cores.

3.2 TCP-Aided TBB Performance Evaluations

The ARM CMP shown in Table 1 is also used to evaluate the performance benefits of TCP-guided task stealing in Intel's TBB. We choose this infrastructure due to the availability of TBB-ported benchmarks for the ARM ISA [12]. We have not evaluated our TBB improvements on the GEMS infrastructure due to the difficulty of procuring a Sparc port of the TBB benchmarks at the time of writing the paper. Note however, that since the ARM simulator uses in-order cores that are harder to build robust TCPs for, we expect our results to hold across the easier case of out-of-order architectures.

3.3 TCP-Guided DVFS Energy Evaluations

Finally, Table 1 shows the FPGA-based emulator used to assess energy savings from TCP-guided DVFS. Our emulator is extended from [3] to support per-core DVFS with voltage/frequency levels similar to those in [9, 16]. Like our ARM simulator, the cores are in-order, representative of embedded systems. The emulator speed (35× speedup over GEMS/Ruby for a subset of Spec 2006) allows performance and power modeling of CMPs running Linux 2.6. The power models assume a $0.13\mu m$ technology and achieve under 8% error compared to gate-level simulations from Synopsys PrimeTime. Despite the negligible leakage of this technology, we use projections from the International Technology Roadmap for Semiconductors to assume a forward-looking fixed leakage cost of 50% of total energy [10, 24]. Since our DVFS optimization targets dynamic energy, this assumption leads to conservative (pessimistic) energy savings estimates.

3.4 Benchmarks

Table 2 shows the workloads used in our studies. The right-most column shows the TCP experiments that the benchmark is used for (TBB, DVFS, or both). While we use all the benchmarks to assess TCP accuracy, only the four PARSEC workloads that have been ported to TBB are used for our TBB studies [12]. We use the SPLASH-2 benchmarks, `Blackscholes`, and `Streamcluster` for our DVFS studies as they contain barriers.

Since the reference inputs for the SPLASH-2 suite are considered out of date, we use the enhanced inputs provided by Bienia et al. [5]. These inputs are derived using a combination of profiling and timing on real machines and have computational demands similar to the PARSEC inputs.

3.5 Details of Our Approach

Our investigation of TCP design and its uses is accomplished in the following steps.

1. We investigate architectural events that have high correlation with thread criticality. We carry out this analysis using both software simulators for metrics such as instruction count, cache misses, and others.
2. We introduce the TCP hardware details based on the architectural metric correlating most with thread criticality.
3. We then apply our TCP design to TBB's task stealer and assess the resulting performance improvements.
4. Finally, we apply our TCP for energy-efficiency in barrier-based parallel programs.

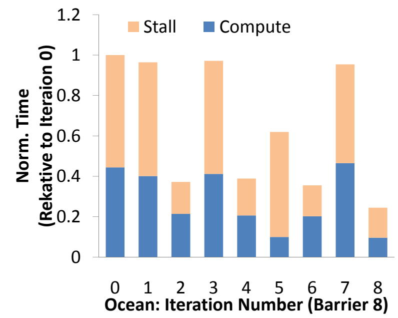

Figure 3: `Ocean's` criticality varies with barrier iteration.

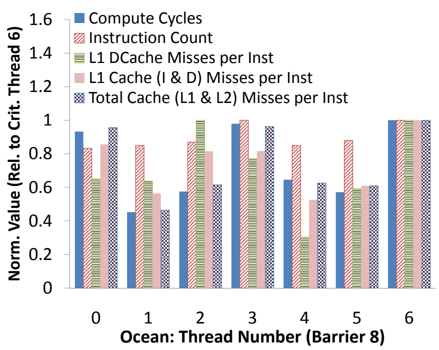

Figure 4: Cache misses track thread criticality most accurately for all threads in barrier 8 of `Ocean`.

4. ARCHITECTURAL METRICS IMPACTING THREAD CRITICALITY

Having motivated TCPs, we now consider simple but effective methods for implementing them. We first study the viability of simple, history-based TCPs local to each core. We then study inter-core TCP metrics such as instruction counts, cache misses, control flow changes and TLB misses.

4.1 History-Based Local TCPs

If thread criticality and barrier stall times were repetitive and predictable, threads could use simple *fully-local* TCPs. The *thrifty barrier* uses this insight for its predictions [21]. While our initial studies on GEMS show promise in this direction, in-order pipelines (not considered in [21]) can show great variation in thread criticality through application runtime. This is because in-order pipelines cannot reorder instructions to hide stall latencies. As an example, Figure 3 shows a snapshot of `Ocean` on the ARM simulator. Each bar indicates the execution time for thread 2 on the first 9 iterations of barrier 8, normalized to iteration 0. The lower portion of the bar corresponds to the compute time while the upper portion represents stall time at the barrier. Clearly, barrier compute and stall times are highly variant.

We therefore conclude that a TCP using thread *comparative* information might fare better on both in-order and out-of-order CPUs, since the criticality of a single thread is intrinsically relative to others.

4.2 Impact of Instruction Count on Criticality

We first consider the correlation between instruction count and thread criticality. Figure 4 plots the behavior of 7 of the 16 threads from the first iteration of barrier 8 of `Ocean` (ARM simulator). To assess the accuracy of a metric in tracking criticality, we normalize thread compute time (the first bar) and the metric against the critical thread (in this case, thread 6) and then compare. In this example, instruction count (the second bar) is a poor indicator of thread criticality, exhibiting little variation. Figure 5 shows this trend

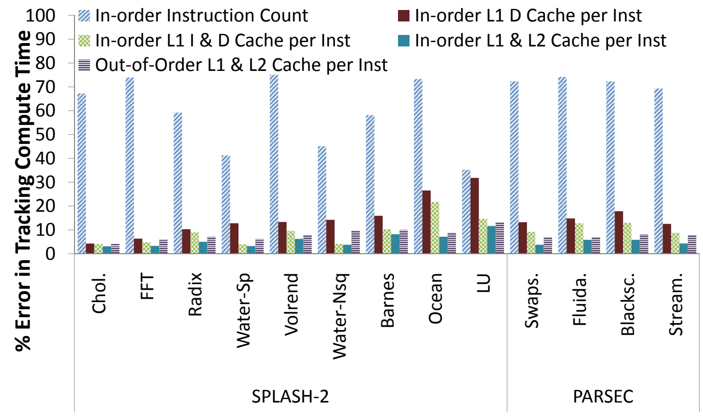

Figure 5: Combined cache misses (I and D L1 misses and L2 misses) tracks thread criticality most accurately.

across other benchmarks by plotting the error between the normalized thread criticality and normalized metric in question. For barrier-based benchmarks, the graphed error is calculated by averaging the error from all barrier iterations. In the absence of barriers (eg. for example, in `Swaptions` and `Fluidanimate`), we split the execution time into 10% windows across which we calculate an average error. This periodic tracking of metric correlation yields a consistently accurate criticality indicator through execution.

Figure 5 shows that although instruction count may track criticality for `LU`, it is inaccurate for other benchmarks. This is because most benchmarks employ threads with similar instruction counts, typical of the *single program, multiple data* (SPMD) style used in parallel programming [21]. Therefore, instruction count is a poor indicator of thread criticality and we need to consider alternate metrics.

4.3 Impact of Cache Misses on Criticality

We find that cache misses intrinsically affect thread criticality. For example, Figure 4 shows that L1 data cache misses per instruction are a much better indicator of thread criticality for `Ocean`. Similarly, Figure 5 shows that this is true for other benchmarks as well, especially for `Cholesky` and `FFT`, which enjoy under 8% error (ARM simulator). Despite significant improvement however, benchmarks such as `Ocean` and `LU` still suffer from over 25% error.

In response, we consider all L1 cache misses (instruction and data). Figure 4 shows that `Ocean's` criticality is much more accurate with this change, especially for threads 0, 2, and 4. Figure 5 also shows that all benchmarks benefit from this approach, especially `Water-Sp`, `Water-Nsq`, and `LU`. This is because instruction cache misses capture the impact of variable instruction counts and control flow changes, which affect these three benchmarks.

To tackle the remaining 22% error for `Ocean`, we integrate L2 cache misses. Since these experience 10× the latency of L1 misses on average in our simulators, L2 misses are scaled commensurately. Figure 4 demonstrates that this metric is indeed most accurate in tracking criticality for `Ocean`. Figure 5 shows improvements for other benchmarks too, particularly for memory-intensive applications like `Ocean`, `Volrend`, `Radix`, and the PARSEC workloads.

Finally, we gauge accuracy across other microarchitectures by testing the data cache misses per instruction metric on the out-of-order GEMS simulator. Figure 5 shows marginal error increases of 3% on GEMS. Benchmarks with larger working sets like `Radix`, `Volrend`, `Ocean` and the PARSEC workloads are least impacted by the microarchitectural change. This bodes well for our results as PARSEC mirrors future memory-heavy workloads. Moreover, our metric is robust to OS effects since GEMS boots a full version of Solaris 10. We therefore conclude from these observations that

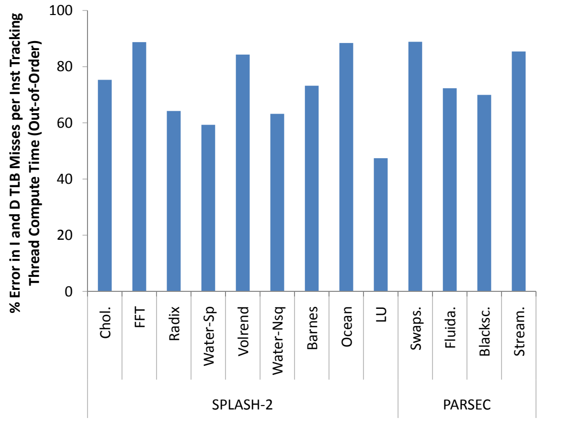

Figure 6: Combined I and D TLB misses have low correlation with thread criticality.

cache misses represent a simple yet accurate means of predicting thread criticality.

4.4 Impact of Control Flow Changes and TLB Misses on Criticality

Apart from cache statistics, one may consider the impact of control flow and branch mispredictions on thread criticality. Fortunately, we find that instruction cache misses, which we have already accounted for, capture much of this behavior. This is unsurprising since more instructions and control flow changes typically lead to greater instruction cache misses.

Finally, Figure 6 considers the impact of TLB misses on thread criticality. Each bar presents the % error between combined instruction and data TLB misses per instruction and criticality on the out-of-order GEMS setup. As shown, TLB misses are a poor indicator of thread criticality; this is because multiple threads usually cooperate to process elements of a larger data set, accessing the same pages. Therefore, they tend to experience similar TLB misses. As a result, our predictor is agnostic to TLB statistics. However, should future workloads require it, a weighted TLB miss contribution can easily be made.

5. BASIC DESIGN OF OUR TCP

Based on our goals and chosen metric, we now present the details of our TCP hardware. Figure 7 details a high-level view of the design. The TCP hardware is located at the shared, unified L2 cache where all cache miss information is centrally available. Our proposed hardware includes *Criticality Counters*, which count L1 and L2 cache misses resulting from each core's references. As cache misses define thread progress, these counters track thread criticalities, with larger ones indicative of slower, poorly cached threads. Since individual L1 cache misses contribute less to thread stall times and criticality than individual L2 misses and beyond, we propose a weighted combination of L1 instruction, L1 data, and L2 cache misses, and others when needed. Currently, our weighted criticality counter values may be expressed by:

$$N(Crit.Count.) = N(L1_{miss}) + \frac{(L1L2_{penalty}) \times N(L1L2_{miss})}{L1_{penalty}} \tag{1}$$

In this equation, $N(Crit.Count.)$ represents the value of the criticality counter, while $N(L1_{miss})$ and $N(L1L2_{miss})$ are equal to the number of L1 misses that hit in the L2 cache and the L1 misses that also miss in the L2 cache. Thus, since L2 misses incur a larger penalty, their weight is proportionately higher.

The counters are controlled by light-weight *Predictor Control Logic*, developed in subsequent sections. An *Interval Bound Register*, which is incremented on every clock cycle, ensures that criticality predictions are based on relatively recent application behavior. This is accomplished by resetting all *Criticality Counters* whenever the *Interval Bound Register* reaches a pre-defined threshold M. We will investigate values for M in subsequent sections.

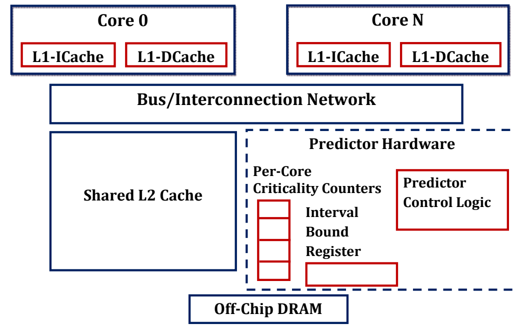

Figure 7: High-level predictor design with per-core *Criticality Counters* placed with the shared L2 cache, *Interval Bound Register*, and *Prediction Control Logic*. Hardware units are not drawn to scale.

The TCP hardware we propose is simple and scalable. First, a simple state machine suffices in incrementing the counters instead of an adder. Unlike meeting points, counter information is in a unified location, eliminating redundancy. More cores are readily accommodated by adding counters. Furthermore, additional network messages are unnecessary as the L2 cache controller may be trivially modified to notify the predictor of relevant cache misses.

Our hardware is also applicable to alternate designs. For example, more cache levels and rare split L2 caches can be handled with trivial hardware and message overhead. For both these cases, the TCP hardware counters would need to be distributed among multiple cache controllers and would need to communicate with centralized prediction control logic. Furthermore, while distributed caches with multiple controllers are uncommon, if necessary we could accommodate messages to our counters on inter-controller coherence traffic. This would, however, still constitute low message overhead.

6. USING TCP TO IMPROVE TBB TASK STEALING

We now showcase the power and generality of TCP by applying it to two uses. We first study TCP-guided task stealing policies in Intel TBB and achieve significant performance improvements.

The Intel TBB library has been designed to promote portable parallel C++ code, with particular support for dynamic, fine-grained parallelism. Given its growing importance, it is a useful platform for studying critical thread prediction, although our techniques will likely apply to other environments as well [6]. We investigate the benefits of TCP-aware TBB in the following steps:

1. We study the TBB scheduler to better understand how our TCP mechanisms may be used to augment it.
2. We present our proposed TCP hardware for TBB task stealing.
3. Finally, we show performance results from our TCP-guided TBB task stealing and compare them with TBB's default random task stealer for the 4 PARSEC benchmarks.

6.1 Introduction to the TBB Task Scheduler

TBB programs express concurrency in parallel *tasks* rather than threads. TBB relies on a dynamic scheduler, hidden from the programmer, to store and distribute parallelism. On initialization, the scheduler is given control of a set of slave worker threads and a master thread (the caller of the initialization). Each thread is associated with its own software task queue, into which tasks are enqueued directly by

```
if (Cache miss from Core P)    {
    Update criticality counter for Core P based on cache miss type
}
if(Steal request from a Core){
    Scan all criticality counters to find the maximum value
    Report core with highest criticality counter value as steal victim
}
if(Message indicating steal from victim Core P unsuccessful){
    Reset criticality counter for Core P
}
if (   (Number of Cycles % Interval Bound)    = =    0    )
    Reset all criticality counters
```

Figure 8: Our TCP-guided task stealing algorithm improves victim selection by choosing the core with the largest *Criticality Counter* value as the steal victim.

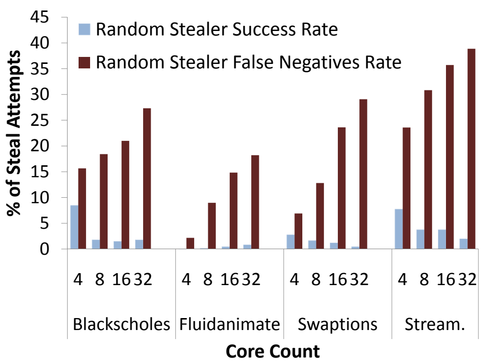

Figure 9: Random stealing prompts a large *false negatives* contribution, leading to poor performance.

the programmer. Task dequeuing is performed implicitly by the runtime system in its main scheduling loop.

If possible, threads dequeue work locally from their task queues. In the absence of local work, task stealing tries to maximize concurrency by keeping all threads busy. The TBB scheduler *randomly* chooses a victim thread, whose task queue is then examined. If a task can be stolen, it is extracted and executed by the stealing thread. If the victim queue is empty, stealing fails and the stealer thread backs off for a pre-determined time before retrying.

While random task stealing is fast and easy to implement, it does not scale well. As the number of potential victims increases, the probability of selecting the best victim decreases. This is particularly true under load imbalance when few threads shoulder most of the work. As such, our TCP design can help by identifying critical threads, which are ideal steal candidates. Subsequent sections detail the performance benefits of such a TCP-guided stealing approach.

6.2 Developing Predictor Hardware to Improve TBB Task Stealing

Figure 8 details our predictor algorithm applied to task stealing. Cache misses are recorded by the *Criticality Counters*. When the TBB scheduler informs the predictor of a steal attempt, the TCP scans its criticality counters for the maximum value and replies to the stealer core that this maximum counter's corresponding core number should be the steal victim. If the steal is unsuccessful, the stealer sends a message to the TCP to reset the victim counter, minimizing further incorrect victim prediction. As before, the *Criticality Counters* are reset every *Interval* so that stealing decisions are based on recent application behavior. As such, this algorithm may be mapped readily to the basic TCP hardware presented in Figure 7, with the *Predictor Control Logic* in charge of scanning for the largest *Criticality Counter*.

Our experiments indicate an *Interval Bound* value of 100K cycles and 14-bit *Criticality Counters* as sufficient to show-

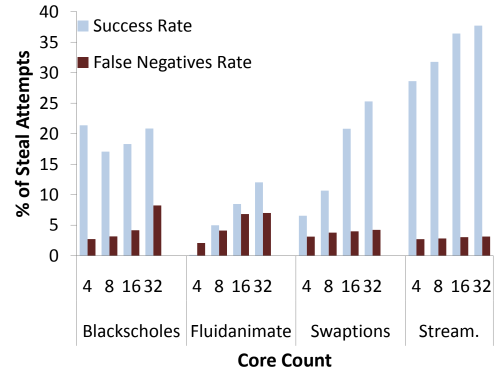

Figure 10: Our thread criticality-guided task stealing cuts the *false negatives* rate under 8% for all benchmarks.

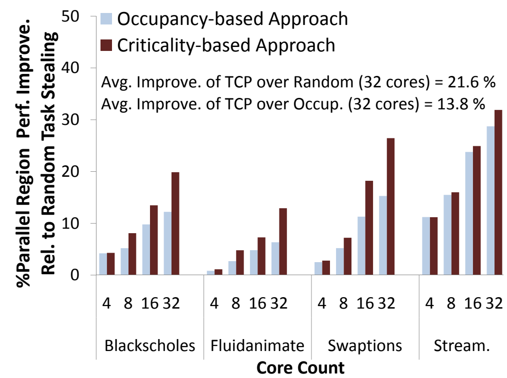

Figure 11: TCP-guided stealing yields up to 32% performance gains against random task stealing and regularly outperforms occupancy-guided stealing.

case our performance benefits. Hence, a 64-core CMP requires merely 114 bytes for the *Criticality Counters* and *Interval Bound Register*, in addition to comparators. Nevertheless, should comparator complexity become an issue, the predictor can be implemented in software. We believe that any performance overhead from this would be trivial due to the simple manipulations of the *Criticality Counters*.

Our experiments also account for the latency overhead in accessing the TCP hardware. Since the TCP is placed with the L2 cache, we assume that a TCP access costs an additional delay equivalent to the L2 latency. This is in contrast with the random task stealer, which makes purely local decisions and is not charged this delay.

6.3 Results

We now study the performance of TCP-guided task stealing across the four PARSEC benchmarks (`Fluidanimate`, `Swaptions`, `Blackscholes`, and `Streamcluster`) ported to TBB. We compare to the random task stealing implemented in TBB version 2.0 (release tbb20_010oss), but our results remain valid for newer TBB releases because random task stealing is still employed. We also compare our results against the occupancy-based TBB task stealer proposed in [12].

6.3.1 TBB with Random Task Stealing

As a baseline, Figure 9 characterizes *random* task stealing with two metrics. The *success rate* (light bar) is the ratio of successful steals to attempted steals. The *false negatives rate* (dark bar) indicates how often a steal is unsuccessful because its chosen target has no work, despite the presence of a stealable task elsewhere in the system. The latter arises from the randomness in choosing a steal victim. Unsurprisingly, random stealing deteriorates (decreasing success rate) as core counts rise. This occurs despite the presence of stealable tasks, indicated by higher *false negatives rate* at greater core counts. Even at low core counts, random task stealing performs poorly for `Blackscholes` and `Streamcluster` (above 15% *false negatives*). This is because just a few threads operate on many longer tasks and random stealing does not successfully select these as steal victims.

6.3.2 TBB with TCP-Guided Task Stealing

Figure 10 shows the improvements of modifying the TBB task stealer to be criticality-aware by plotting the *success rate* and *false negatives rate* of our TCP-guided stealing. Compared to random stealing, *false negatives* fall drastically and remain under 8% for all benchmarks. The benefits are immense especially at higher core counts. For example, the *success rate* for `Streamcluster` improves from 5% to around 37% at 32 cores. We also see that the *success rate* increases with higher core counts, contrary to the random stealing results. This is because increased core counts raise the availability of stealable tasks, which we now exploit through reliable stealing predictions.

6.3.3 TCP-Guided TBB Performance Improvements

While Figure 10 does indicate the benefits of our TCP-based task stealing, we still need to investigate how this translates into raw performance numbers. In order to study performance improvements, we again use the ARM CMP simulator. As previously noted, our TCP-aware TBB task stealer includes the extra delay of accessing the criticality counters as an additional L2 access due to their proximity to the L2 cache. This is in contrast with the random stealing approach which is fully-local and not charged this delay.

Apart from studying the performance improvements of the TCP approach against random task stealing, we also compare our results against the *occupancy-based* task stealer. As detailed in [12], occupancy-based stealing keeps track, in software, of the number of elements in each core's task queue and selects the core with the maximum number as the next stealing victim. Since this results in higher performance than random task stealing, we need to check whether TCP-guided task stealing provides any further benefits.

Figure 11 plots the performance improvements of TCP-guided task stealing (parallel region of the benchmarks only) against TBB's random task stealing in the dark bars. Furthermore, we also plot the performance improvements of the occupancy-based approach, again versus random stealing in the light bars. As shown, criticality-guided stealing offers significant performance improvements over random stealing, particularly at higher core counts (average of 21.6% at 32 cores). Moreover, we also improve upon the occupancy-based stealing results. This is because occupancy-based techniques only count the *number* of tasks in each queue, but do not gauge their relative complexity or expected execution time. In contrast, TCP-based approaches can better account for the relative work in each task, by tracking cache miss behavior. This generally improves performance, especially at higher core counts (13.8% at 32 cores). The only exception is `Streamcluster` where a few threads hold a large number of stealable tasks of similar duration. In such a scenario, thread criticality yields little benefit over simple occupancy statistics. Generally however, the performance benefits of our approach are evident.

7. USING TCP TO SAVE ENERGY IN BARRIER-BASED APPLICATIONS

While Section 6 focused on TCP-guided TBB performance, we now use the predictor to save energy in barrier-based programs. Our analysis is conducted as follows:

1. We describe our prediction algorithm for TCP-guided per-core DVFS and show how this maps to hardware.
2. We then study predictor accuracy on both the in-order and out-of-order simulators to ensure that our TCP design is robust across microarchitectures and applications.

```
if (Cache miss from core P)    {      // Our algorithm is triggered on every cache miss
     Update the criticality counter for core P based on cache miss type

     /*  T is pre-defined and determines when thread criticalities are calculated  */
     if (Core P is currently running at nominal freq. and its criticality counter ≥ T){
           for (every core)    {    // Loop initiating thread criticality calculation

                /*  Err on the side of performance by selecting SST entry of higher freq. */
                Look up core criticality counter and find SST entry nearest to it
                If( matching SST entry indicates target frequency different from current one)
                      Switch_frequency_suggestion = True;       // SST is suggesting DVFS

                /* SST feeds DVFS suggestion to SCT which decides if this yields an actual DVFS */
                if (Switch_frequenc_suggestion = =  True)

                      Update SCT register for the core by incrementing the target
                      frequency's saturating counter and decrementing others
                      if (max counter in Confidence entry is for alternate freq.)
                         Switch Core frequency, update Current DVFS Tag

                }  else                                // If SST does not suggest DVFS, update SCT
                   Update SCT register for the core by incrementing the current
                   frequency's saturating counter and decrementing others
           }
           Reset all criticality counters
     }
if ((Number of Cycles % Interval Bound)  = = 0 )  /* Ensure that counters
     Reset all criticality counters                    are based on recent behavior */
```

Figure 12: Our TCP-guided DVFS algorithm tracks cache miss patterns to set threads to appropriate frequencies.

3. Finally, we present the energy savings from our approach on the FPGA-based emulator.

7.1 Algorithm and Proposed Hardware

The goal of our energy-efficient algorithm is to predict thread criticality and DVFS accordingly so as to minimize barrier wait-time. Though energy is the primary concern, accurate prediction is essential to reduce the performance impact of DVFS, by clock-scaling only non-critical threads. An effective TCP must also use confidence from past behavior to reduce the number of "wrong answers" it provides.

Figure 12 details our prediction algorithm, highlighting the TCP components required for DVFS. On an L1 instruction cache, L1 data cache, or L2 cache miss, the *Criticality Counter* corresponding to the relevant core is updated. If the core's counter value is now above a pre-defined threshold T and it is currently running at the nominal (fastest) frequency f_0, thread criticality calculations commence.

The first step uses the *Switching Suggestion Table* (*SST*) to translate *Criticality Counter* values into thread criticalities and suggest potential frequency switches. The *SST* is row-indexed with the current DVFS setting. Every row holds pre-calculated values corresponding to criticality counts required to switch to potential target frequencies. Each column thus corresponds to a different target frequency. Every core's relative criticality is determined by row-indexing the *SST* with the current DVFS setting and comparing its criticality counter value against the row *SST* entries. The matching *SST* entry's column index then indicates the target frequency. If this is different from the current DVFS setting, the *SST suggests* a frequency switch. If the criticality count falls between two *SST* entries, we err on the side of performance by picking the entry for the higher frequency.

The second step in our algorithm feeds the suggested target frequency from the *SST* to the *Suggestion Confidence Table* (*SCT*). The *SCT* minimizes the impact of fast-changing, spurious program behavior which can lead to criticality mispredictions. The latter can degrade performance either by sufficiently slowing non-critical threads to make them critical or by further slowing down critical threads. To combat this, the *SCT* assesses confidence on the *SST*'s DVFS suggestion and permits switches only for consistently-observed criticality behavior. The *SCT* contains per-core registers maintaining a set of saturating counters, one for each DVFS level. At system initialization, the f_0 counter is set to its

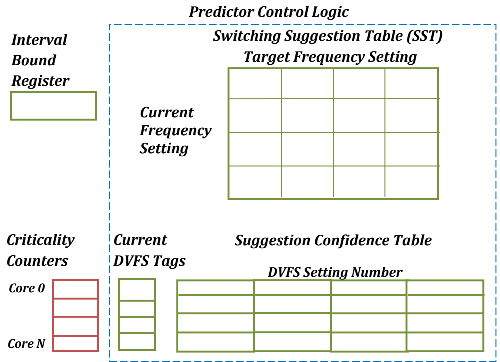

Figure 13: Our TCP hardware includes the *Criticality Counters*, *SST*, *SCT*, *Current DVFS Tags*, and *Interval Bound Register*; for a 16-core CMP with 4 DVFS levels and a threshold T of 1024, we use 71 bytes of storage overhead.

maximum value while the others are zero. Now, when the *SST* suggests a frequency switch for a particular core, that core's *SCT* register is checked. The register's saturating counter for the suggested target frequency is then incremented while the other counters are decremented. An *actual* frequency switch is initiated only if the counter with the largest (most confident) *SCT* value corresponds to a DVFS setting different from the current one.

Our algorithm also uses the *Interval Bound Register* to periodically reset the *Criticality Counters* to ensure that predictions are based on recent thread behavior. For our results, this is set to 100K cycles.

Figure 13 shows the structure of the described hardware in more detail. The predictor control logic includes the *SST*, *SCT*, and *Current DVFS tags*. *SST* entries are preset by scaling a threshold T by the potential DVFS levels. For example, in our DVFS environment with four levels, the *SST* row corresponding to f_0 would hold $0.85T$, $0.70T$, and $0.55T$ as the values required to scale to $0.85f_0$, $0.70f_0$ and $0.55f_0$ respectively. These values are calculated with the insight that higher *Criticality Counter* values imply slower, poorly-cached threads; therefore, if a core holds a criticality count of T while another holds $0.55T$, the latter is non-critical and its frequency can be proportionately scaled down.

Our TCP design is clearly low-overhead and scalable. Based on our chosen TCP parameters, a 16-core CMP requires only 71 bytes of storage aside from comparators. At 64 cores, assuming the same TCP parameters, the required storage increases to merely 215 bytes, making for a scalable design.

Our algorithm requires modest inter-core communication; this however, only occurs when a core needs to DVFS. We have reduced communication requirements by housing the TCP state at the L2 cache controller, where counted events will be observed anyways. As such, this is lower overhead than meeting points, which requires frequent broadcasts. This allows for fast criticality predictions.

Having detailed our hardware structures, we now focus on tuning the structure parameters for optimal predictor accuracy, starting with the threshold T, which determines how often we consider clock-scaling threads.

7.2 Criticality Counter Threshold: How Often to Consider Switching?

Selecting an appropriate threshold parameter T involves balancing predictor speed and accuracy. While a low T provides the opportunity for fine-grained DVFS, it may increase susceptibility to temporal noise. Without good suggestion confidence, this results in too many frequency changes and excessive performance overhead.

We begin by measuring *T's* effect on prediction accuracy using the ARM simulator with 16 cores and DVFS. To isolate the effect of T, we temporarily ignore the *SCT* and DVFS transition overheads; all suggested *SST* frequency

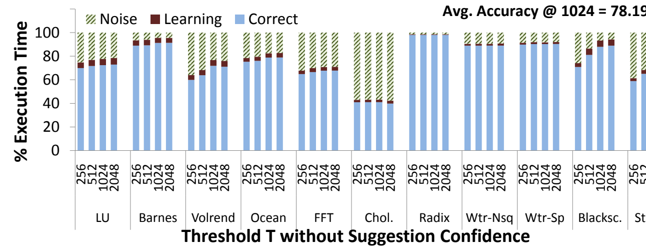

Figure 14: While prediction accuracy typically increases with T, prediction *noise* is a dominant problem.

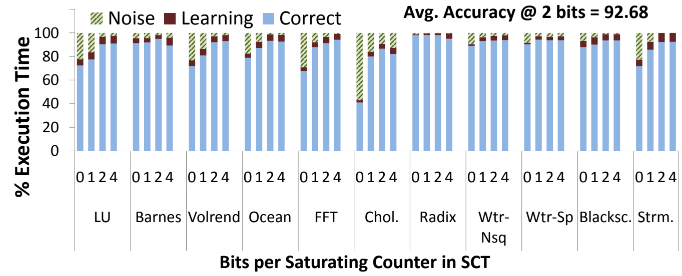

Figure 15: Suggestion confidence mitigates *noise* with an average accuracy of 92.68% with 2 bits.

switches are actually executed without going through the *SCT* or incurring voltage rail switching penalties.

To assess TCP accuracy, we first profile the benchmarks without TCP. We record thread compute and stall times across barriers and then use them to pre-calculate the thread frequencies required to minimize barrier stalling. A highly accurate criticality predictor would DVFS threads into these pre-calculated settings. We then integrate our TCP and study how closely we track the pre-calculated frequencies.

Figure 14 shows the accuracy with which TCP tracks the pre-calculated frequency settings. The lowest bar component represents time spent in the pre-calculated or *correct* state, averaged across all barrier instances. The central bar component shows the *learning time* taken until the correct DVFS state is first reached. Finally, the upper portion shows prediction *noise* or time in spent in erroneous DVFS *after* having arrived at the correct one.

Figure 14 shows how prediction accuracy varies with benchmark. While `Radix` and `Barnes` enjoy above 90% accuracy, `Cholesky` does poorly at 40%. In general, increasing T bolsters prediction accuracy though `Cholesky` actually gradually deteriorates. One reason for this is the *learning time* taken to study application behavior before predicting future thread criticality. While a perfect predictor eliminates *learning time*, the time taken to reach T limits us in practice. Typically, a larger T increases *learning time*. Figure 14 shows this, particularly for `Blackscholes` and `Streamcluster`. Longer *learning times*, however, usually provide more reliable predictions.

Figure 14 also shows that though *learning time* contributes to predictor inaccuracy, prediction *noise* from fast-changing program behavior is far more harmful, causing excessive frequency switching. For example, `Cholesky` and `Streamcluster` suffer particularly from this behavior. This inaccuracy can result in severe performance degradation.

Therefore, TCP *noise* must be eliminated for higher accuracy. We investigate mechanisms for doing this with the *Suggestion Confidence Table* in the following sections. These investigations will assume a T of 1024 since Figure 14 shows that this has the highest average accuracy.

7.3 Suggestion Confidence Table Design

Suggestion confidence can reduce noise-induced mispredictions by requiring that a thread's behavior be consistent

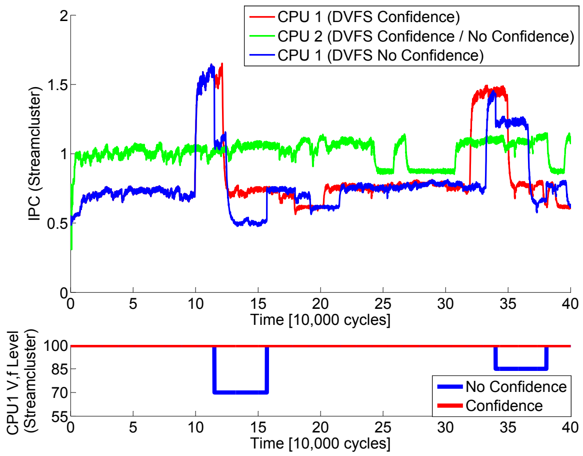

Figure 16: Temporal noise in barrier 4 of `Streamcluster` causes incorrect slowdown of critical thread 1, degrading overall performance; confidence estimation prevents this problem.

over a longer time-frame before making a prediction. Figure 15 demonstrates the benefits of this by varying the bit-width of the counters in each entry of the *Suggestion Confidence Table*. Clearly, even 2 bits of confidence particularly improve `LU`, `Volrend`, `Streamcluster`, and `Cholesky` accuracy.

However, Figure 15 also shows that while confidence estimation mitigates *noise*, *learning time* tends to rise. This is because even correct frequency switches now take longer to predict. This is why the accuracy of `Barnes`, `Cholesky`, and `Radix` degrades as the number of confidence bits rises. From now, we therefore assume 2 bits per *SCT* saturating counter as this presents the best results.

To further investigate how confidence estimation removes erroneous frequency transitions, Figure 16 shows a snapshot of `Streamcluster` on the ARM simulator. The upper subgraph plots IPC profiles for thread 1 and 2 in the presence and absence of confidence estimation. The lower sub-graph plots thread 1's DVFS setting through runtime with and without confidence estimation.

We begin by considering the thread 1 and 2 IPC profiles without confidence estimation. Thread 1 is critical, usually maintaining an IPC lower than thread 2. However, thread 1's IPC does have occasional surges (100k - 120k and 320k - 350k cycles), unrepresentative of its general behavior. Unfortunately, at these points, our TCP *without* confidence estimation would infer that thread 1 is non-critical, scaling down its frequency. Indeed, the lower graph of Figure 16 shows the incorrect prediction around 110k cycles, causing a switch to $0.70f_0$. The resulting thread 1 IPC therefore decreases, degrading performance. Although the predictor eventually reorients at 155k cycles, critical thread 1 is now slowed down. Furthermore, another TCP misprediction is made around 340k cycles, when thread 1's IPC again surges.

Figure 16 shows that with confidence estimation, these incorrect frequency switches are prevented. The upper subgraph shows that thread 1's IPC with confidence estimation again surges at 100K and 320k cycles. This time, however, there are no frequency switches and the resulting thread 1 IPC profile (*Confidence case*) is not degraded.

7.4 Prediction Accuracy in the Presence of Out-of-Order Pipeline and Memory Parallelism

Our goal is to provide general TCPs that work well across a range of microarchitectures. Therefore, we need to evaluate our techniques on both in-order and out-of-order scenarios. Thus far, our results have assumed an in-order pipeline and blocking caches. We now consider the effect of out-of-order pipelines and non-blocking caches.

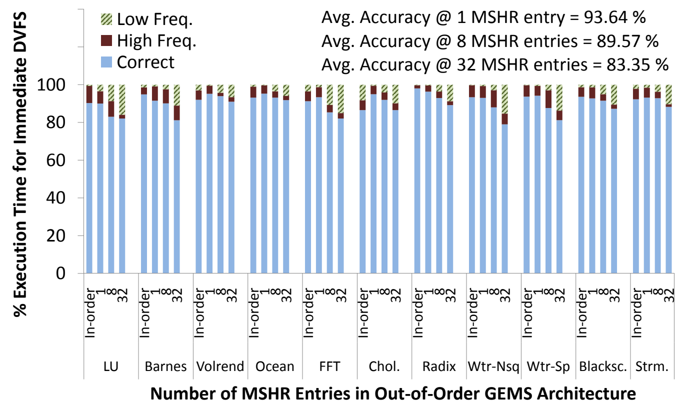

Figure 17: Increased memory parallelism decreases the criticality of a single cache miss.

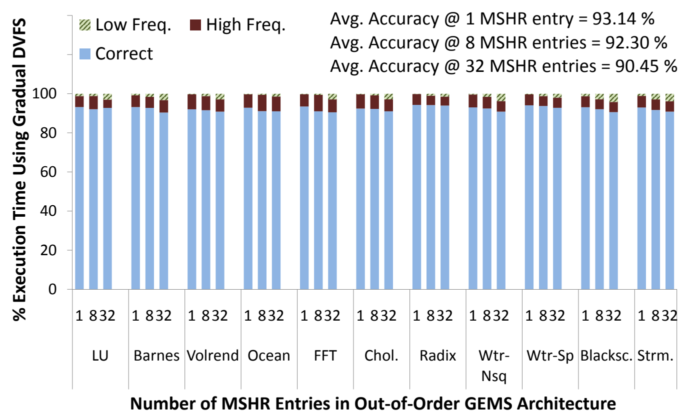

Figure 18: Gradual DVFS demonstrates high accuracy despite memory parallelism.

Figure 17 shows TCP accuracy on both the ARM simulator (first set of bars) and the out-of-order GEMS configurations for varying levels of memory parallelism. The latter is accomplished by changing the number of Miss Status Holding Register (MSHR) entries. Our plot also shows the time spent in the pre-calculated or *correct* DVFS settings (lower bar portion), as well as time spent in frequency settings higher (middle portion of each bar) or lower (upper portion of each bar) than our pre-calculated values.

Figure 17 shows that the out-of-order GEMS pipeline with blocking caches (1 MSHR entry) actually enjoys slightly higher TCP accuracy than the in-order pipeline. In particular, `Cholesky's` accuracy now improves to 93% because of the minimization of *noise* from out-of-order instruction scheduling on program behavior. Unfortunately, the results change when increasing memory parallelism. We still achieve an accuracy of 89.57% at 8 MSHR entries but at 32 entries, this degrades to 83.35%. This is because the criticality of a single cache miss decreases as the cache is able to overlap the miss penalty with subsequent accesses. While benchmarks with large working sets (`Ocean`, `Volrend`, `Blackscholes`, and `Streamcluster`) remain more immune to this, others are susceptible. In fact, Figure 17 shows that these TCP mispredictions are especially problematic as they result in frequency settings lower than those pre-calculated, degrading performance.

Fortunately, an elegant solution to this problem is to favor cautious, *gradual* DVFS. In this scheme, downward transitions can only occur to immediately lower frequencies (eg. f_0 can switch to only $0.85f_0$), while upward transitions can be handled as before (eg.$0.70f_0$ can to either $0.85f_0$ or f_0). Figure 18 reveals that this greatly improves predictor accuracy

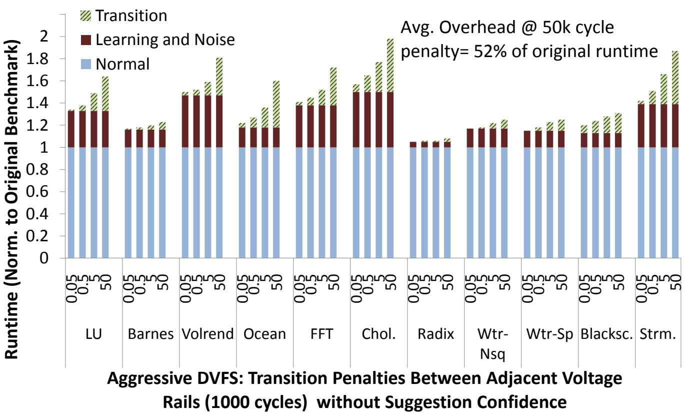

Figure 19: DVFS transition penalties without the *SCT* severely degrade performance.

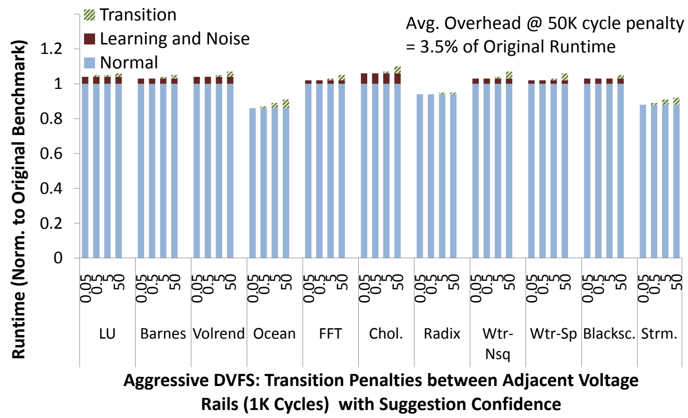

Figure 20: The *SCT* eliminates *noise* and *transition overhead.*

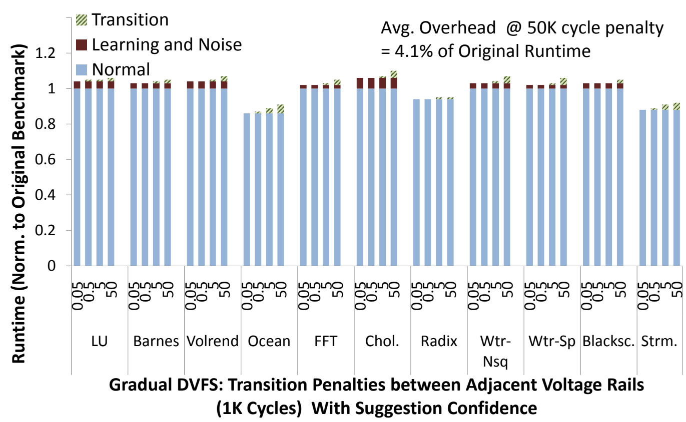

Figure 21: Gradual DVFS with the *SCT* enjoys low runtime overheads.

even at high MSHR counts. While accuracy may decrease at low MSHR counts (eg. `Radix` accuracy decreases by 4%) because of longer *learning time* from gradual frequency scaling, we now see above 90% accuracy across every tested benchmark for 32-entry MSHRs. Moreover, most of the inaccuracy is spent at higher frequencies. While this might reduce energy savings, it ensures good performance for many levels of memory parallelism.

7.5 Results

The previous sections have detailed TCP-driven DVFS hardware and algorithms to mitigate barrier-induced energy waste. We have presented predictor accuracy for a range of microarchitectures and discussed the merits of both aggressive DVFS and gradual, conservative DVFS. We now present performance and energy results.

7.5.1 Impact of TCP-Driven DVFS Transition Overheads on Benchmark Performance

While energy efficiency is the motivation for applying our TCP to DVFS, we want to avoid compromising performance. Therefore, we now consider the impact of DVFS transition time penalties on application performance for both the conventional or *aggressive* DVFS case and the alternate *gradual* DVFS case for increased memory parallelism.

DVFS relies on off-chip switching voltage regulators with transition penalties in the order of microseconds. While preliminary work on fast on-chip voltage regulators for per-core DVFS shows promise [19], our predictor must also handle the larger transition times typical of contemporary systems.

Figure 19 shows how transition penalties affect benchmark runtimes for aggressive DVFS on our emulator *in the absence of the SCT*. Runtimes are normalized to their baseline execution. The lower bar portions indicate baseline runtime while the middle portion indicates *learning time* and *noise* overheads. The upper bar portion accounts for performance degradation due to *transitions*. We vary voltage transitions from 50 cycles (typical of on-chip regulators) to 50K cycles (typical of off-chip regulators). As expected, the lack of suggestion confidence introduces significant *noise*. Even worse however, transition overheads further degrade execution time by 52% on average at 50K cycles. `Cholesky` is particularly affected, almost doubling in runtime.

Fortunately, Figure 20 shows that the *SCT* removes both *noise* and *transition* overheads for aggressive DVFS, even at the high penalties associated with off-chip regulators. Since predictions now have to build confidence before frequency scaling, mispredictions from *noise* and their associated transitions are prevented. Thus, benchmark overheads remain under 10%. Interestingly, we see that the runtimes of `Ocean`, `Radix`, and `Streamcluster` are actually *improved* by 5-10%. This occurs because non-critical threads are slowed down, spreading out their cache misses. Hence, bus/interconnect congestion is lowered enough for the critical thread cache misses to be serviced faster, boosting overall performance.

Figure 21 considers the alternate case of gradual DVFS on our emulator. Again, minimal performance degradation occurs due to our confidence estimation scheme. While some benchmarks see a slightly higher overhead due to the longer *learning time*, `LU` and `Cholesky` see roughly 4% improvement in runtime overhead. This is because TCP mispredictions are minimized, lowering the number of frequency switches and their transition overhead.

Note that we also tested performance on the in-order ARM simulator, with similar results. We therefore conclude that our TCP predictor is robust to DVFS non-idealities as well.

7.5.2 Energy Savings from TCP-Driven DVFS

We now present the energy savings from integrating our TCP in barrier-based programs. We run our selected benchmarks on the FPGA-based emulator assuming a transition penalty of 50K cycles. Figure 22 shows that TCP-driven DVFS saves considerable energy across benchmarks, an average of 15% and 11% for aggressive and gradual DVFS respectively. Benchmarks with more load imbalance generally save more energy; the large imbalance for `LU` and `Volrend` leads to energy savings above 20% for both DVFS modes. Meanwhile, `Radix`, `Ocean`, and `Streamcluster` benefit from their shorter runtimes, which decreases idle energy contributions. Note that while gradual DVFS usually leads to lower energy savings than the aggressive case, this situation is reversed for `Cholesky`. This is because the gradual case minimizes TCP mispredictions, decreasing the number of frequency switches and their transition overheads.

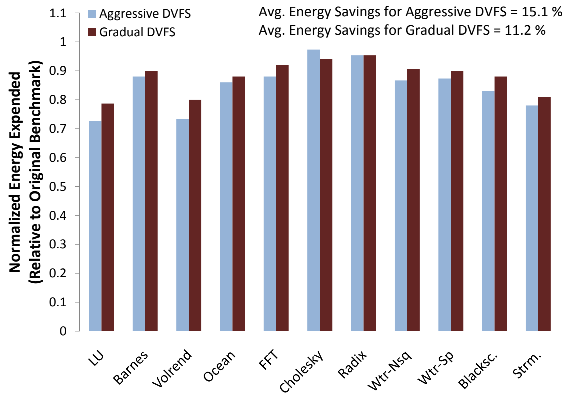

Figure 22: Energy expended by TCP-guided DVFS normalized to no DVFS case (lower values indicate savings).

Our energy savings are actually a conservative estimate for a few reasons. First, our results are based on 4 simple cores. Increasing load imbalance from a greater number of complicated cores will yield considerably higher energy savings. Second, we fix leakage cost to a forward-looking value of 50% of total baseline energy. In reality, lower power from our scheme on the non-critical threads will lead to lower temperatures, leading to leakage savings. Moreover, benchmarks with shorter runtime would lead to even lower leakage power, which we do not account for. Finally, as on-chip regulators become the norm, energy waste from transition penalties will be eliminated.

8. RELATED WORK

While application criticality has been studied, most prior work has explored this in the context of *instructions* [14, 32]. The advent of CMPs however, has pushed the focus on *thread* criticality prediction. We have already detailed the thrifty barrier [21] and meeting points [9] approaches and shown our distinct research goals. Other than these approaches, Liu et al. use past non-critical thread barrier stall times to predict future thread criticality and DVFS accordingly [23]. In contrast to this history-based approach, we predict thread criticality based on current behavior, regardless of barriers. We also use our predictor to improve TBB task stealing, building from the *occupancy-based* approach of Contreras and Martonosi [12]. Our work is distinct, however, in that we use criticality to guide task stealing for performance gains with little hardware overhead.

Apart from these applications, the power of our TCP-hardware lies in its generality and applicability to a range of adaptive resource management schemes. For example, TCPs could be used to guide shared last-level cache management [17, 18, 30], QoS-aware cache designs such as Virtual Private Caches [28], the design of fair memory controllers and their priority schemes [15, 22, 27], the development of SMT priority, throughput and fairness schemes [9, 25, 31], as well as the management of other parallelization libraries such as CAPSULE [29] and CARBON [20] and other work-stealing schemes [2, 7, 11].

9. CONCLUSION

Our overarching goal has been to explore the accuracy and usefulness of simple TCPs based largely on metrics already available on most CMPs. By focusing on the large amounts of run-time variation introduced by the memory hierarchy, our TCPs offer useful accuracy at very low hardware overhead. Furthermore, situating the TCP near the L2 cache controller allows it to collect the necessary inputs with little network or bus overhead.

One of our goals has been to develop TCPs general enough for several applications. To demonstrate this, we implemented a TCP-based TBB task stealer, and a TCP-based DVFS controller for energy savings in barrier-based programs. The TCP-based task stealer offers 12.9% to 31.8% performance improvements on a 32-core CMP. The TCP-based DVFS controller offers an average of 15% energy savings on a 4-core CMP.

Looking beyond the initial applications covered in this paper, the real promise of our work lies in its ability to provide a cost-effective foundation for a large variety of performance and resource management problems in future CMPs. As future CMPs scale to higher core counts, greater complexity, and increased heterogeneity, the need to dynamically apportion system resources among multiple threads will be crucial. Our TCP mechanisms present a first effort in this regard and, we expect it to be valuable for a range of resource management issues in both hardware and software.

10. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. We also thank Gilberto Contreras and Niket Agarwal for their help with the simulation infrastructure and Sibren Isaacman for his help with the emulator. This work was supported in part by the Gigascale Systems Research Center, funded under the Focus Center Research Program, a Semiconductor Research Corporation program. In addition, this work was supported by the National Science Foundation under grant CNS-0720561.

11. REFERENCES

[1] Intel Threading Building Blocks 2.0. 2008.
[2] U. Acar, G. Blelloch, and R. Blumofe. The Data Locality of Work Stealing. *ACM Symp. on Parallel Algorithms and Architectures*, 2000.
[3] A. Bhattacharjee, G. Contreras, and M. Martonosi. Full-System Chip Multiprocessor Power Evaluations using FPGA-Based Emulation. *Intl. Symp. on Low Power Electronics and Design*, 2008.
[4] C. Bienia et al. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2008.
[5] C. Bienia, S. Kumar, and K. Li. PARSEC vs SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip Multiprocessors. *IEEE Intl. Symp. on Workload Characterization*, 2008.
[6] R. Blumofe et al. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 1996.
[7] R. Blumofe and C. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Journal of ACM*, 46(5):720–748, 1999.
[8] S. Borkar et al. Parameter Variations and Impact on Circuits and Microarchitecture. *Design Automation Conference*, 2003.
[9] Q. Cai et al. Meeting Points: Using Thread Criticality to Adapt Multicore Hardware to Parallel Regions. *Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2008.
[10] H. Chang and S. Sapatnekar. Full-Chip Analysis of Leakage Power Under Process Variation, including Spatial Correlations. *Design Automation Conference*, 2005.
[11] D. Chase and Y. Lev. Dynamic Circular Work-Stealing Deque. *ACM Symp. on Parallelism in Algorithms and Architectures*, 2005.
[12] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of Intel Threading Building Blocks. *IEEE Intl. Symp. on Workload Characterization*, 2008.
[13] J. Donald and M. Martonosi. Techniques for Multicore Thermal Management: Classification and New Exploration. *Intl. Symp. on Computer Architecture*, 2006.

[14] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies via Critical-Path Prediction. *Intl. Symp. on High Performance Computer Architecture*, 2001.

[15] E. Ipek et al. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. *Intl. Symp. on Computer Architecture*, 2008.

[16] C. Isci et al. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. *Intl. Symp. on Microarchitecture*, 2006.

[17] A. Jaleel et al. Adaptive Insertion Policies for Managing Shared Caches. *Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[18] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. *Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2004.

[19] W. Kim et al. System Level Analysis of Fast, Per-Core DVFS Using On-Chip Switching Regulators. *Intl. Symp. on High Performance Computer Architecture*, 2008.

[20] S. Kumar, C. Hughes, and A. Nguyen. Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. *Intl. Symp. on Computer Architecture*, 2007.

[21] J. Li, J. Martinez, and M. Huang. The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors. *Intl. Symp. on High Performance Computer Architecture*, 2005.

[22] W. Lin et al. Reducing DRAM Latencies with an Integrated Memory Hierarchy Design. *Intl. Symp. on High Performance Architecture*, 2001.

[23] C. Liu et al. Exploiting Barriers to Optimize Power Consumption of CMPs. *Intl. Symp. on Parallel and Distributed Processing*, 2005.

[24] S. Liu et al. A Probabilistic Technique for Full-Chip Leakage Estimation. *Intl. Symp. on Low Power Electronics and Design*, 2008.

[25] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. *Intl. Symp. on Performance Analysis of Systems and Software*, 2001.

[26] M. Martin et al. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *Comp. Arch. News*, 2005.

[27] K. Nesbit et al. Fair Queuing Memory Systems. *Intl. Symp. Microarchitecture*, 2006.

[28] K. Nesbit, J. Laudon, and J. Smith. Virtual Private Caches. *Intl. Symp. on Computer Architecture*, 2007.

[29] P. Palatin, Y. Lhuillier, and O. Temam. CAPSULE: Hardware-Assisted Parallel Execution of Component-Based Programs. *Intl. Symp. on Microarchitecture*, 2006.

[30] S. Srikantaiah, M. Kandemir, and M. J. Irwin. Adaptive Set Pinning: Managing Shared Caches in CMPs. *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2007.

[31] D. Tullsen et al. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. *Intl. Symp. on Computer Architecture*, 1996.

[32] E. Tune et al. Dynamic Prediction of Critical Path Instructions. *Intl. Symp. on High Performance Computer Architecture*, 2001.

[33] S. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Intl. Symp. on Computer Architecture*, 1995.