Thread Progress Equalization: Dynamically Adaptive Power-Constrained Performance Optimization of Multi-Threaded Applications

Yatish Turakhia, Guangshuo Liu, Siddharth Garg, and Diana Marculescu, *Fellow, IEEE*

Abstract—Dynamically adaptive multi-core architectures have been proposed as an effective solution to optimize performance for peak power constrained processors. In processors, the micro-architectural parameters or voltage/frequency of each core can be changed at run-time, thus providing a range of power/performance operating points for each core. In this paper, we propose Thread Progress Equalization (TPEq), a run-time mechanism for power constrained performance maximization of multithreaded applications running on dynamically adaptive multicore processors. Compared to existing approaches, TPEq (i) identifies and addresses two primary sources of inter-thread heterogeneity in multithreaded applications, (ii) determines the optimal core configurations in polynomial time with respect to the number of cores and configurations, and (iii) requires no modifications in the user-level source code. Our experimental evaluations demonstrate that TPEq outperforms state-of-the-art run-time power/performance optimization techniques proposed in literature for dynamically adaptive multicores by up to 23 percent.

Index Terms—Multi-threaded applications, thread progress, power-constrained performance maximization

1 Introduction

Technology scaling has enabled greater integration because of reduced transistor dimensions. Microprocessor designers have exploited the greater transistor budget to provision an increasing number of processing cores on the chip, effectively using thread-level parallelism to increase performance. However, the power consumption per transistor has not been scaling commensurately with transistor dimensions [1]. This problem is compounded by the so-called "power wall," a hard limit on the maximum power that a chip can draw. A critical challenge, in this context, is to devise techniques that maximize performance within a power budget. One solution to this problem is fine-grained, *dynamic adaptation* at run-time. Broadly speaking, dynamic adaptation refers to the ability to dynamically distribute the available power budget amongst the cores on a multicore processor.

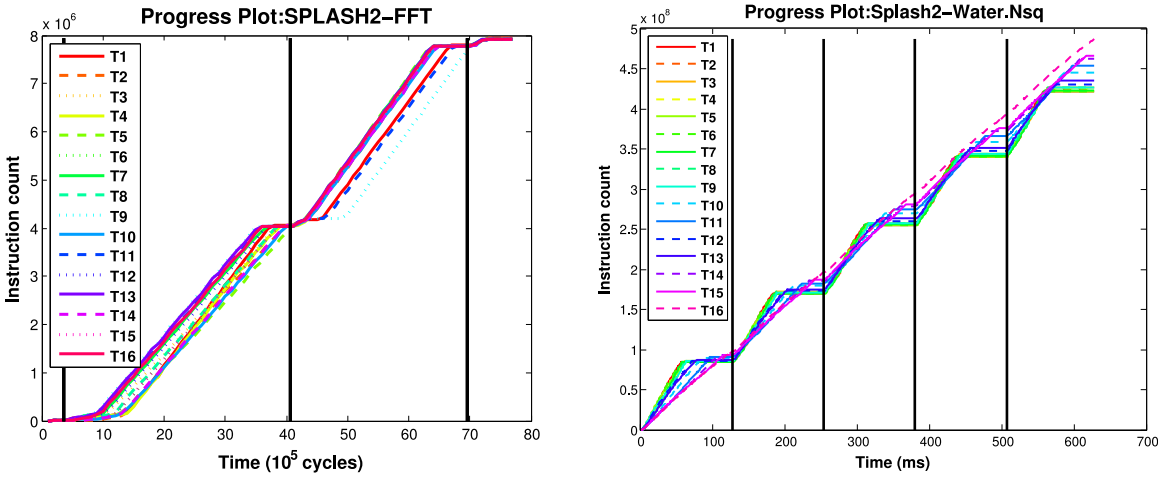
The traditional approach for fine-grained dynamic adaptation is based on dynamic voltage and frequency scaling (DVFS), in which the voltage and frequency of each core (or group thereof) can be adjusted dynamically, providing a range of power/performance operating points. Moreover, recent work has advocated the use of micro-architectural adaptation, in which the micro-architectural configuration of each core to be adjusted dynamically (issue width, reorder buffer size and cache capacity etc.), which is particularly effective in the context of "dark silicon" era [1] where the power budget is constrained but transistors are abundant. Although the techniques proposed in this paper are described in the context of micro-architectural adaptation, they are equally applicable for DVFS and we provide experimental results for both dynamic adaptation techniques.

To perform fine-grained dynamic adaptation, the operating system has to solve a challenging global optimization problem, i.e., how to determine the configuration of each core so as to maximize performance within the power budget. The problem is challenging for three reasons: (i) the solution must scale efficiently to multicore systems that have tens or even hundreds of cores; (ii) there is a complex relationship between core configurations and the corresponding power/performance of the thread running on the core (this is particularly true for micro-architectural adaptation); and, (iii) for multithreaded applications, there is no direct performance metric to maximize, i.e., it is unclear how speeding up a single thread will affect the performance of the application as a whole. These are the challenges that we address in this paper.

For sequential (i.e., single-threaded) applications, instructions per second (IPS) is a clear, measurable indicator of performance. Moreover, for multiprogrammed workloads, the IPS summed over all threads indicates net throughput, and is a commonly used performance metric [2], [3], [4]. However, for multithreaded applications, the sum of IPS metric can be a poor indicator of performance. For example, a thread that is spinning on a lock or waiting at a barrier might execute user-mode synchronization instructions, but these do *not* correspond to useful work.

- *Y. Turakhia is with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305. E-mail: yatisht@stanford.edu.*
- *G. Liu is with Alphabet's Nest Labs, Palo Alto, CA 94304. E-mail: Gsliu@cmu.edu.*
- *S. Garg is with the Department of Electrical and Computer Engineering, New York University, New York, NY 10012. E-mail: sg175@nyu.edu.*
- *D. Marculescu is with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213. E-mail: dianam@cmu.edu.*

Manuscript received 14 Mar. 2016; revised 7 Aug. 2016; accepted 10 Sept. 2016. Date of publication 12 Sept. 2016; date of current version 17 Mar. 2017. Recommended for acceptance by P. Eles.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2016.2608951



(a) IPC heterogeneity only. (b) IPC and instruction count heterogeneity.

Fig. 1. Progress plots for the FFT and Water.Nsquared (SPLASH-2) benchmarks with 16 threads on a 16-core architecture. The solid vertical lines indicate barriers. Slope corresponds to IPS of that thread and hence the flat regions indicate time periods when the thread is stalled waiting for lagging threads to arrive.

The problem is heightened by the fact that programmers exploit parallelism in different ways—for example, using data-level parallelism with barrier synchronization, or task-level parallelism with local/global task queues and static/dynamic load balancing.

Key Contributions. In this paper, we propose Thread Progress Equalization (TPEq), a run-time mechanism to maximize performance within a power budget for multithreaded applications running on multicore processors with per-core dynamic adaptation. The design of TPEq is motivated by multithreaded applications that make frequent use of barrier synchronization, but also generalizes, as we later discuss, to other models of parallelism.

We start with the observation that, to best utilize the available power budget, all threads that are expected to synchronize on a barrier should arrive at the barrier at the same time. If this is not the case, *early* threads (threads that arrive at a barrier earlier than others) can be slowed down and the power saved by doing so can be allocated to speed-up *lagging* threads (threads that arrive at a barrier later than others). In this context, a natural question is why threads arrive at barriers at different times.

Empirically, we have observed two fundamental reasons for differences in the times at which threads arrive at barriers. First, even if each thread executes exactly the same sequence of instructions, threads can have different instructions per cycle (IPC) counts. For example, the sequence of data accesses that one thread makes can have less spatial locality than another thread's accesses, resulting in more cache misses and lower IPC for the first thread. We refer to this as *IPC heterogeneity*. Second, each thread might execute a different number of instructions until it reaches a barrier. This is because the threads need not be inherently load balanced and depending on the input data, each thread can follow a different control flow path until it arrives at the barrier. We refer to this as *instruction count heterogeneity*.

Fig. 1 shows an example of two benchmark applications, FFT and Water.Nsq (SPLASH-2 [5]), executing on a homogeneous multicore processor. FFT exhibits IPC heterogeneity but no instruction count heterogeneity, i.e., each thread executes exactly the same number of instructions between barriers. Water.Nsq exhibits both IPC heterogeneity, evident from different slopes of threads in progress plot, *and* instruction count heterogeneity. Note that over the entire length of the application, thread T16 executes more than $1.15\times$ the number of instructions compared to thread T7.

The goal of TPEq is to dynamically optimize the configuration of each core/thread such that each thread reaches the barrier at the same time by simultaneously accounting for *both* IPC and instruction count heterogeneity. The design of TPEq is based on two components that operate synergistically:

- *TPEq Optimizer:* Given an oracle that can predict (i) the IPC and power consumption of each thread for every core configuration, and (ii) the total number of instructions the thread must execute until the next barrier, we propose an efficient *polynomial-time* algorithm that *optimally*determines the core configuration for each thread to maximize application performance under power constraints.
- *TPEq Predictors:* As input to the TPEq Optimizer, we implement accurate run-time predictors for (a) IPC and power consumption of a thread for different core configurations, and (b) the number of instructions each thread executes between barriers.

TPEq is evaluated in the context of the Flicker [6] architecture, a recently proposed multicore processor design that supports dynamic adaptation of the micro-architectural parameters of each core. We compare TPEq to a number of existing techniques for power/performance optimization of multithreaded applications.

Distinguishing Features of TPEq. Compared to existing state-of-the-art approaches, TPEq has the following distinguishing features: (i) TPEq holistically accounts for both IPC and instruction count heterogeneity, while a number of other approaches only address one or the other; (ii) TPEq enables fine-grained adaptation for multicore processors where each core has multiple configurations; (iii) the TPEq optimizer provides optimal solutions in polynomial time, as opposed to other fine-grained optimization techniques that

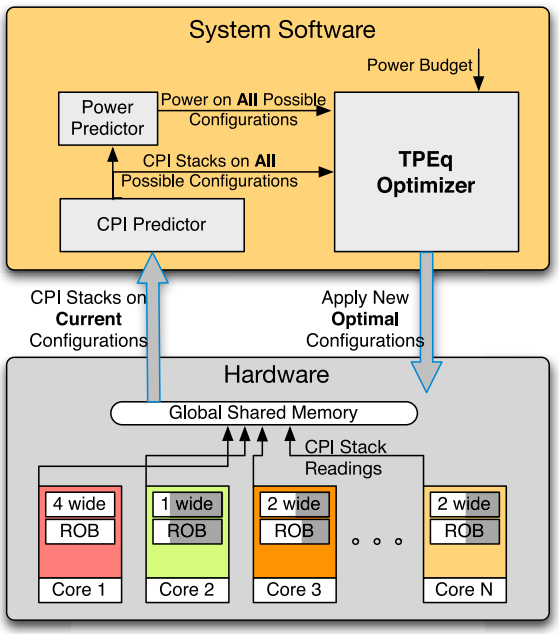


Fig. 2. Overview of TPEq approach on a dynamically adaptive multicore processor.

solve NP-hard problems, such as maximizing instruction throughput under power constraint [2], [6], [7], [8], and can only achieve sub-optimal results in practice; (iv) TPEq requires no software annotations or programmer specified progress metrics; and (v) TPEq generalizes to multithreaded applications that exploit different models of parallelization, including barrier synchronization, pipeline parallel and thread pool models with dynamic load balancing.

2 TPEq Design and Implementation

Fig. 2 shows an overview of the design of TPEq. The hardware platform consists of a dynamically adaptive multicore architecture where, for example, each core can have a variable re-order buffer (ROB) size and the fetch width. In general, we will assume that each of the N cores can be set in one of M different *configurations* as described later in Section 3. In its current implementation, TPEq assumes that the number of threads equals the number of cores, and a static mapping of threads to cores [9]. We believe TPEq can be extended to the case where there are more threads than cores [10], but leave that as a topic for future work.

The TPEq run-time system consists of two components. The TPEq predictors monitor on-chip performance counters and predict the future application characteristics. The predictions are passed on to the TPEq optimizer, which determines the optimal configuration of each core so as to maximize overall system performance within a power budget. We now describe the design and implementation of TPEq.

2.1 TPEq Optimizer

The TPEq optimizer is at the heart of TPEq approach. Although, in practice, the optimizer takes inputs from the TPEq predictor, we will discuss the optimizer in the context of an oracle that provides the optimizer with perfect information and relax this assumption later.

To understand how the optimizer works, assume that we begin at the time instant when N threads exit a barrier and start making progress towards the next barrier. The optimal configuration for each core/thread needs to be decided for the interval between these two successive barriers. Assume that an oracle provides access to the following information:

- The number of instructions each thread executes until it enters the next barrier is in ratio $w(1) : w(2) : \ldots : w(N)$. Note that $w(1), w(2), \ldots, w(N)$ can be absolute instruction counts, but we only require the number of instructions each thread executes relative to other threads.
- The CPI of thread i $(1 \leq i \leq N)$ when it executes on a core with configuration j $(1 \leq j \leq M)$ is $CPI(i, j)$, and the corresponding power dissipation is $P(i, j)$. We assume, for now, that for a given core configuration, the CPI and power dissipation of each thread do not change with time, at least until it reaches the next barrier. This assumption is relaxed later.

Under the assumptions above, TPEq tries to assign a configuration to each core/thread so as to stay within power budget P_{budget}, while minimizing the time taken by the most lagging thread barrier to reach the next barrier. A key contribution of our work is an algorithm that *optimally* solves this problem in $\mathcal{O}(MN \log N)$ time.

The algorithm works as follows: TPEq starts by setting all cores to the configuration that consumes the least power and determines the identity of the *most lagging* thread for this setting, i.e., the thread that would reach the barrier last. For thread i, the number of clock cycles required to reach the barrier when executing on configuration j would be $w(i)CPI(i, j)$. We define the *progress* of this thread as

$$progress(i) = \frac{1}{w(i)CPI(i, j)}, \tag{1}$$

to capture the intuition that larger values of "progress" are better.

The configuration of the most lagging thread is then moved up to the next level,[1] and the new most lagging thread is determined. The core configuration for this new most lagging thread is now moved up by one level, and so on. This continues until there is no core whose configuration can be increased to the next level without violating the power budget. The resulting core configurations are optimal in terms of total execution time and are then updated in hardware. Algorithm 1 is a formal description of this optimization procedure.

Intuitively, the TPEq configuration C is optimal as a better configuration C^* within the same power budget cannot exist which has all threads running faster than with the TPEq configuration. If so, TPEq can still speed up at least one thread within the power budget. Thus, there must be a thread i that runs faster in C than in C^*. The progress of thread i in C^* is clearly an upper bound on the progress of C^*. Moreover, since TPEq must have sped up thread i at some point, the progress of thread i in C^* is also a lower

1. Without loss of generality, the configurations are, by convention, sorted in ascending order of power consumption. Also, we limit the search to *Pareto optimal* configurations, by simply discarding ones where increasing power does not lead to increased performance.

bound on the progress of C. Hence, the minimum thread progress in C is at least as large as that in C*.

A formal proof of optimality for this algorithm is provided below.

Algorithm 1. TPEq Optimization Procedure

```
1  P_tot ← 0;
   // Init. all threads to lowest core config.
2  for i ∈ [1, N] do
3     c(i) ← 1;
4     P_tot ← P_tot + P(i, c(i));
5     progress(i) ← 1/(w(i)CPI(i,c(i)));
6  end
7  while P_tot ≤ P_budget do
      // Determine lagging thread l
8     l ← arg min_{i∈[1,N], c(i)<M} {1/(w(i)CPI(i,c(i)))};
      // If no such thread exists
9     if l = ∅ then
10       break;
11    end
      // Increase core configuration of lagging thread
12    c(l) ← c(l) + 1;
      // Update progress and power
13    progress(l) ← 1/(w(l)CPI(l,c(l)));
14    P_tot ← P_tot − P(l, c(l) − 1) + P(l, c(l));
15 end
   // Return optimal core configurations
16 return c;
```

Proof of Optimality (by Contradiction). Let $C = <c(1)\ c(2)\ \ldots\ c(N)>$ be the TPEq configuration vector of cores for N threads, such that $c(1)$ corresponds to the core configuration of thread 1, $c(2)$ corresponds to the core configuration of thread 2, and so on. Let P_{tot} be the total power consumption with configuration vector C, such that $P_{tot} \leq P_{budget}$. Let $progress(i, c(i))$ denote the progress of thread i with core configuration $c(i)$ and $min\_progress(C)$ denote the progress of the most lagging thread with configuration vector C. Since only Pareto optimal configurations are considered, $progress(i, c(i)) > progress(i, c^*(i)) \Rightarrow P(i, c(i)) > P(i, c^*(i))$. Now assume a better configuration vector $C^* = <c^*(1)\ c^*(2)\ \ldots\ c^*(N)>$ with total power P^*_{tot} within same power budget exists, i.e., $min\_progress(C^*) > min\_progress(C)$ and $P^*_{tot} \leq P_{budget}$.

First, consider a case in which TPEq does not assign a configuration to any thread that is larger than the optimal configuration, i.e., $c(i) \leq c^*(i)\ \forall i \in [1, N]$. This also implies that $P_{tot} \leq P^*_{tot}$. Without loss of generality, assume that first K threads have strictly larger core configurations in the optimal assignment, i.e., $c(i) < c^*(i)\ \forall i \in [1, K]$ and the remaining threads have same configurations as TPEq $c(i) = c^*(i)\ \forall i \in [K+1, N]$. If the most lagging thread l for configuration C was in the first K threads, the Algorithm 1 would not terminate as it is possible to increase core configuration of thread l to $c^*(l)$ and remain within total power P^*_{tot} (and therefore, P_{budget}). If $l \in [K+1, N]$, $c(l) = c^*(l)$ and therefore, $min\_progress(C) \geq min\_progress(C^*)$. This is a contradiction.

Next, consider a case in which TPEq assigns a configuration to a thread j that is larger than the optimal configuration, i.e., $c(j) > c^*(j)$. This implies $progress(j, c(j)) > progress(j, c^*(j))$. But since TPEq only accelerates the most lagging thread in each iteration and since TPEq assigned thread j to $c(j)$, which is larger than $c^*(j)$, $min\_progress(C) \geq progress(j, c(j))$. This implies $min\_progress(C) \geq progress(j, c^*(j)) \geq min\_progress(C^*)$. Again, a contradiction. Therefore, TPEq configuration is optimal.

Min-Heap Based Implementation. A naive implementation of the TPEq optimization algorithm (Algorithm 1) would use a linear array to store the progress metric of each thread (line 12), resulting in a $\mathcal{O}(MN^2)$ time complexity. However, note that in each iteration of Algorithm 1, we only update the progress of the currently slowest thread, i.e., least progress. Based on this observation, we propose an improved implementation of Algorithm 1 with $\mathcal{O}(MN \log N)$ that stores the progress metric of each thread in a *min-heap* data-structure. A min-heap is a binary tree where the data in each node is less than or equal to its children.

In the proposed implementation, setting up the min-heap data structure takes $\mathcal{O}(\log N)$ time (line 4), determining the currently most lagging thread takes $\mathcal{O}(1)$ time (line 8), and updating the progress metric of the lagging thread and reinserting it back into the heap takes $\mathcal{O}(\log N)$ time (line 12). Finally, the outermost *while* loop iterates at most MN, resulting in a time complexity of $\mathcal{O}(MN \log N)$.

Epoch Length. In practice, the TPEq optimization routine is called once every *epoch* in order to address fast variations in thread characteristics. The epoch length ($\mathcal{E}$, measured in number of clock cycles) is configurable. The epoch length should be short enough to quickly adapt to CPI and power variations, but is practically limited by the computational overhead of the optimization procedure. In this context, the polynomial time complexity of the TPEq optimizer, which counts for less than 1 percent run-time overhead for a 1 ms epoch, enables the use of relatively fine-grained temporal adaptation that would be otherwise impractical.

Note that since epochs are not necessarily synchronized with barriers, in practice we need a slightly updated progress metric from the one used in Algorithm 1. Therefore, the progress of a thread is measured in terms of its *predicted* progress by the end of the current epoch:

$$progress(i) = \frac{instrCount(i)}{w(i)} + \frac{\mathcal{E}}{w(i)CPI(i, c(i))}, \qquad (2)$$

where the first term represents progress made so far and the second term represents predicted progress in the next epoch.

2.2 TPEq Predictors

In the previous section we assumed that the TPEq optimizer has oracular knowledge of the relative instruction counts of the threads. In practice, the TPEq predictors determine these values at run-time for each thread immediately after a synchronization related stall. TPEq also requires predictions for CPI and power consumption of each thread for every core configuration once every *epoch*, i.e., in synchrony with the TPEq optimization procedure.

2.2.1 Relative Instruction Count Prediction

We start by describing the relative instruction count predictor. The instruction count predictor predicts the number of

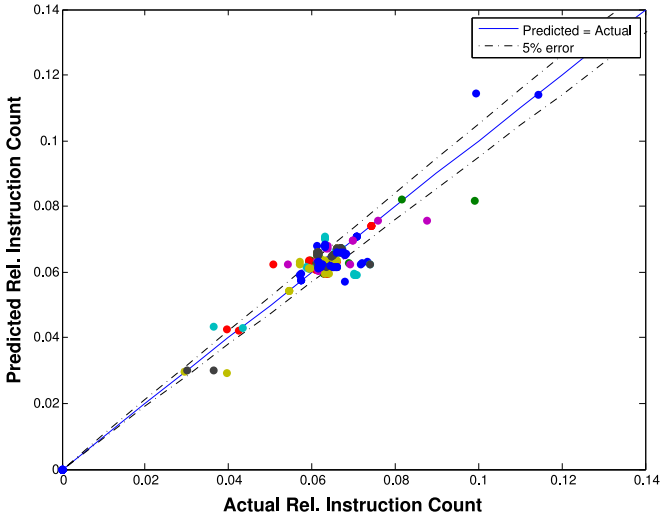


Fig. 3. Scatter plot of predicted and actual relative instruction counts. All benchmarks used 16 threads but only threads 1 to 8 are shown for better clarity. Out of 360 points in the plot, only 82 points (23 percent) had error larger than 5 percent.

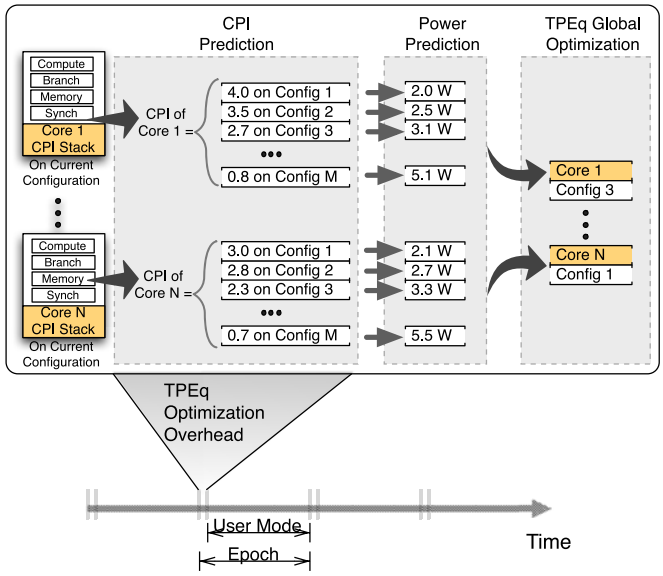


Fig. 4. CPI and power prediction overview. A detailed discussion of the TPEq predictors is in Section 2.2.2.

instructions each thread executes relative to other threads. Our predictor is based on the observation that *the number of instructions each thread executes between barriers, relative to other threads, remains roughly the same*. This motivates the use of a history based predictor to predict relative instruction counts.

Intuitively, we note that the difference in relative instruction counts of several multithreaded workloads arise as a result of imbalance in the amount of computing for a thread, which persists across several barriers. Singh et al. [11] were perhaps the first to qualitatively observe the locality in the data distribution in threads across successive barriers in many of our benchmark algorithms and provide insights into this characteristic. They noted that successive barriers correspond to very small "time-steps" in the physical world, and that the characteristics of the physical world change slowly with time. Hence, the amount of work to be performed by a thread in one time-step, is a good predictor for the amount of work in the next time-step. In the progress plot for Water.Nsq (see Fig. 1b), for instance, the number of water molecules per thread remain nearly constant across barriers. Consequently, thread T16 (T7), with most (least) number of water molecules, always executes the most (least) instructions in any inter-barrier interval.

Quantitatively, we have verified this trend over all barrier synchronization based benchmarks in the SPLASH-2, PARSEC and Phoenix benchmarks suites (see Table 4 for more details) that we experimented with. In particular, Fig. 3 shows the a scatter plot of relative instruction counts in barrier phase $t+1$ versus the relative instruction counts of threads in barrier phase t across all benchmarks with instruction count heterogeneity (coded in different colors). The mean absolute relative error using a last-value-predictor for relative instruction counts was found to be only 4.2 percent. Liu et al. [12] have observed similar locality behaviour across the outermost loops of the SpecOMP parallel applications, and use last-value prediction to perform voltage/frequency scaling for each thread. However, there are significant differences between their work and ours and these are discussed in Section 5.

Implementation Details. The TPEq relative instruction count predictor keeps a running count of the number of user mode instructions executed by a thread. The relative instruction count, $w(i)$, of each thread is updated with its running count at the end of *any* synchronization related stall. This technique is simple, requires no synchronization between threads to detect barriers and avoids the need for the user to indicate when barriers occur.

In our scheme, if an application has only barrier related synchronization, all threads will update as they exit the barriers. In addition, the weight in any inter-barrier interval will automatically correspond to the average number of instructions per barrier executed by the thread so far. In fact, TPEq does not distinguish barrier related stalls from other synchronization stalls such as ones due to critical sections. For one, barriers can be implemented using other synchronization primitives as well, including locks [13], which we would like to capture. As well, taking into account *all* synchronization related stalls introduces certain advantages that will be discussed in later sections.

Any synchronization related stall detection mechanism can be used to determine when these stalls occur. Hardware based thread progress metrics have been proposed [14] to detect threads that are spinning on locks or waiting at barriers. These solutions are the most general but suffer from false positives and true negatives, resulting in incorrect optimization decisions. Alternatively, software based solutions can be implemented, either using programmer or compiler inserted annotations, or by modifying the threading library and OS synchronization primitives [15]. We adopt the latter approach. As in [15], we detect scenarios in which the threads are stalled due to synchronization and update our predictors with current relative instruction counts while exiting the stall.

2.2.2 CPI and Power Prediction

We now describe the CPI and power predictors that we use in TPEq which, as shown in Fig. 4, are called once every epoch.

Let $CPI_t(i,j)$ be the CPI of thread i on core configuration j in epoch t. The goal of the CPI predictor is to determine $CPI_{t+1}(i,j)$ for all $j \in [1,M]$. Duesterwald et al. [16] have shown that for predicting the CPI in the next epoch assuming the same core configuration, i.e., characterizing temporal variability in CPI, last-value predictors perform on a par with exponentially-weighted mean, table-based and cross-metric predictors. The accuracy of last-value predictors improves for shorter prediction epochs. We choose to use a last-value predictor in TPEq because of its simplicity, and because we are able to afford relatively short epoch lengths. The last-value predictor simply implements:

$$CPI_{t+1}(i,j) = CPI_t(i,j), \tag{3}$$

To predict $CPI_{t+1}(i,k)$ for all $k \neq j$ given $CPI_{t+1}(i,j)$, we need an approach that predicts the performance on one core type given performance on another core type. For this, TPEq uses CPI stack information measured using hardware counters broken down into four components: compute CPI (base CPI in the absence of miss events), memory CPI (cycles lost due to misses in the memory hierarchy), branch CPI (cycles lost due to branch misprediction) and synchronization CPI (cycles lost due to stalls on synchronization instructions).

With these measurements on configuration j, we predict the CPI on configuration k using a linear predictor as follows:

$$\begin{aligned} CPI_t(i,k) = {} & \alpha_{jk}^{comp} CPI_t^{comp}(i,j) + \alpha_{jk}^{mem} CPI_t^{mem}(i,j) \\ & + \alpha_{jk}^{branch} CPI_t^{branch}(i,j) \\ & + \alpha_{jk}^{synch} CPI_t^{synch}(i,j). \end{aligned} \tag{4}$$

The pairwise α_{jk}^{*} parameters, one for every pair of core configurations, are learned offline using training data obtained from a set of representative benchmarks and stored for online use. Note that the learned parameters are not benchmark specific and depend only on the core configurations.

A similar linear predictor that utilizes CPI components was proposed by Lukefahr et al. [17], although only for big-little core configurations. Another CPI predictor is PIE [18], which makes use of information collected using hardware counters including the total CPI, CPI of memory instructions, misses per instruction (MPI), and data dependencies between instructions. However, PIE has been proposed for CPI prediction between small in-order and large out-of-order cores, while TPEq also requires predictions between different out-of-order core configurations and also faces the challenge of predicting over future epoch. Furthermore, since training the TPEq predictor is automated and data-driven, it is easy to deploy for a large number of core configurations.

We note that existing processors such as the Intel Pentium 4 [19] and the IBM POWER5 [20] have built-in hardware support for performance counters that measure CPI components. In addition, Eyerman et al. [21] have proposed a performance counter architecture that further improves upon the accuracy of these commercial implementations with similar hardware complexity. Their approach provides very accurate estimates of the CPI stack components with only 2 percent average absolute error.

The TPEq power predictor uses the predicted CPI values for each core configuration (as described above) to predict their power consumption. This is based on previous work which indicates that that CPI (or IPC) is highly correlated with power consumption [22], [23]; for instance, Bircher and John report on average only 3 percent error when compared to measured CPU power [23]. Indeed, we empirically verified that incorporating more fine-grained data like the individual CPI stack components did not improve the accuracy of power prediction significantly. However, we did observe that moving from a simple linear predictor to a quadratic model did improve accuracy. Thus, the TPEq power predictor predicts the power consumption for different core types as follows:

$$P(i,j) = \beta_{0,j} + \frac{\beta_{1,j}}{CPI(i,j)} + \frac{\beta_{2,j}}{CPI(i,j)^2}, \tag{5}$$

where $\beta_{0,j}$, $\beta_{1,j}$, and $\beta_{2,j}$ are fixed parameters that are learned for each core type offline and stored for online use.

2.3 Implementation Details

Hardware. The primary hardware overhead that TPEq introduces is the hardware required to track the CPI stack components. As mentioned before, existing commercially available processors such as the Intel Pentium 4 and IBM POWER5 already have hardware performance counters to measure CPI stack components. Based on the design proposed by Eyerman et al. [21], we estimate the hardware overhead for TPEq as follows: (i) one global 32-bit register and counter per CPI stack component (five registers/counters in all), (ii) a shared front-end miss table (sFMT) with as many rows as number of outstanding branches supported, an ROB ID and local branch misprediction penalty counter per-row, and a shared I-cache/TLB miss counter, (iii) a back-end miss counter for D-cache/D-TLB misses; and (iv) a long latency functional unit counter. The counters in (ii), (iii) and (iv) are all local counters and only need to count up to the maximum miss penalties for their respective events.

Software. The TPEq optimization (Algorithm 1) and prediction routines (Equation (4) and (5)) are implemented in software. The TPEq prediction and optimization procedures are invoked by the OS in every epoch using an interrupt. The CPI stack values on each core are stored to shared memory, after which one core, designated as the leader, reads these values, performs CPI and power predictions and determines the optimal core configurations. All other cores are stalled in this period. Finally, the configuration of each core is updated based on the optimal configurations and control is passed back to user code. In the empirical results section, we quantify all the execution time overheads of the TPEq procedures.

2.4 Comparative Analysis of TPEq

To provide more insight into our proposed approach, we compare TPEq qualitatively to three state-of-the-art approaches for maximizing the performance of multi-threaded applications.

Criticality Stacks (CS). Criticality Stacks [15] is a recently proposed metric for thread criticality that measures the amount of time in which a thread is active (not stalled due to

TABLE 1
Microarchitectural Adaptation Configurations

Configuration	Dispatch width	ROB size	Integer ALUs
1	1	16	1
2	2	32	3
3	2	64	3
4	4	64	6
5	4	128	6

Number of cores: 16, Number of threads: 16 (1 thread/core)
Frequency: 3.5 GHz, Voltage: 1.00 V, 22 nm Technology Node
L1-I cache: 128 KB, write-back, four-way, four-cycle
L1-D cache: 128 KB, write-back, eight-way, four-cycle
L2 cache: private 256 KB, write-back, eight-way, eight-cycle
L3 cache: 8 MB shared/4 cores, write-back, 16-way, 30-cycle
Cache coherence: directory-based MSI protocol
Floating point units: 2, Complex ALUs: 1

synchronization) in each epoch divided by the number of other threads active in the same epoch. Intuitively, the most critical thread is one that is active while all others are stalled.

TPEq incorporates a notion of criticality similar to that of CS through the weights $w(i)$. Threads that spend more (less) time stalled will have lower (higher) $w(i)$ values for TPEq, and similarly, lower (higher) criticality values in CS. The numbers will not be identical though, since the time spent in active state is weighted differently in the two approaches.

Most importantly, CS is a coarse-grained optimization techniques, in that it only accelerates the "most-lagging" thread. In contrast, the TPEq optimizer performs fine-grained optimization based on the progress metric and weight of every thread, and is therefore able to best utilize the available power budget.

We note that CS is itself a generalization of Age-based Scheduling [24] (AGETS), in which the thread which has executed the least number of instructions relative to other threads is sped up on a faster core. Although we also implemented and experimented with AGETS [24], we found that CS outperformed AGETS across all the benchmarks we studied, so we do not report any data for AGETS in this paper.

MaxBIPS. Maximizing sum-IPS [2] is a commonly used (and intuitive) objective for applications where the threads are independent—multiprogrammed workloads, for example, or multithreaded benchmarks with dynamically load balanced task queues and task stealing [3]. Like TPEq (and unlike CS), MaxBIPS can be used for fine-grained optimization of core configurations. However, the primary problem with MaxBIPS in the context of multi-threaded benchmarks is that it has no notion of thread synchronization and does not take thread criticality into account.

Bottleneck Identification and Scheduling. Bottleneck Identification and Scheduling [25] (BIS) annotates critical sections in the code and uses these annotations to determine and accelerate bottlenecks, i.e., performance critical threads, at run-time. As opposed to the previously discussed techniques, BIS does require access to source code, and, at least for a set of benchmarks evaluated, CS performs at least as well as BIS [15].

Nonetheless, we believe techniques such as BIS, and its updated version UBA [26], are orthogonal to and can be used in conjunction with TPEq. For example, threads that have BIS-based criticality greater than a threshold can be assigned to the highest core configuration, while the remaining $N-1$ cores configurations can be optimized using TPEq. We leave this as future work.

TABLE 2
Maximum IPC and Power Observed for Different Configurations Using Swaptions

	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5
IPC	0.65	1.06	1.13	1.27	1.36
Power (W)	3.93	5.43	5.56	6.51	6.69

TABLE 3
Parameters for Predicting Power for Different Core Types Learned with Offline Training

	Conf. 1	Conf. 2	Conf. 3	Conf. 4	Conf. 5
β_0	1.16	2.03	2.14	2.69	2.80
β_1	4.05	4.65	4.55	4.34	4.17
β_2	−0.44	−1.17	−1.16	−0.96	−0.86

3 Experimental Setup

Our empirical evaluation of TPEq is based on the Sniper [27] multicore simulator for x86 processors. We augment Sniper with our TPEq code and our patch that enables dynamic adaption of hardware parameters, including front-end pipeline width and reorder buffer (ROB) size, and scripts for the other state-of-the art techniques we compare against. For power estimation, we use McPAT [28], which is seamlessly integrated with Sniper.

We model a processor with 16 cores and an 80 W power budget. The relevant core/uncore micro-architectural parameters are shown in Table 1. Each core can pick from one of five different configurations which are also listed in Table 1. We note that in our experiments, the issue queue and load-store queue are scaled automatically with ROB size, since in Sniper all three are governed by a single parameter "*window size*"). Table 2 shows the maximum observed IPC and power values over all epochs for different static core configurations using Swaptions benchmark. Table 3 shows the learned parameters for different core types for predicting power using Equation (5). Finally, in all experiments, the epoch length is set to 1 ms (3.5 million clock cycles at the baseline clock frequency of 3.5 GHz).

The workloads used in our experiments are multi-threaded applications from the PARSEC [29], SPLASH-2 [5] and Phoenix [30] benchmark suites. We have included 18 out of 22 benchmarks in SPLASH-2 and PARSEC combined, excluding only the ones which we had compilation or run-time issues with Sniper.

Table 4 shows the subset of the benchmarks that extensively make use of barrier synchronization for parallelization. These are the benchmarks for which we expect TPEq to perform the best, since it is designed keep barrier synchronization based parallelism in mind. These benchmarks are further classified as: (i) homogeneous benchmarks: threads execute the same number of instructions between barriers; (ii) heterogeneous: threads execute the same number of instructions between barriers relative to each other.

TABLE 4
Barrier Synchronization Based Benchmarks Classified as Either Homogeneous or Heterogeneous

Category	Workload	Benchmark Suite
Homogeneous (HO)	Blackscholes (C)	PARSEC
	Canneal (M)	PARSEC
	FFT (M)	SPLASH-2
	Ocean.cont (M)	SPLASH-2
	Radix (M)	SPLASH-2
	Streamcluster (C)	PARSEC
	Swaptions (C)	PARSEC
Heterogeneous (HT)	Barnes (C)	SPLASH-2
	Fluidanimate (C)	PARSEC
	LU.cont (C)	SPLASH-2
	LU.ncont (C)	SPLASH-2
	Water.nsq (C)	SPLASH-2
	Water.sp (C)	SPLASH-2
	Bodytrack (C)	PARSEC
	Kmeans (C)	Phoenix

Additional labels in paranthesis indicate whether the application is memory-bound (M) or CPU-bound (C).

The applications were also classified as memory-bound (M) if the misses per kilo instructions (mpki) for last-level cache (LLC) was greater than 1, or as CPU-bound (C) otherwise.

Table 5 shows the remaining benchmarks that use other types of parallelism. We classify these as follows: (i) thread pool: a number of independent tasks are organized in a shared or distributed queues and a thread requests a new task from the task queue after it completes the previous task; (ii) pipeline parallel: groups of threads executing different stages in a software pipeline on an incoming stream of data, with task queues between pipeline stages and (iii) mapreduce: different threads independently executing "map" functions on incoming data before synchronizing on the reduce thread.

We note that Bodytrack from PARSEC and kmeans from Phoenix that are both classified as barrier-based, actually use mixed modes of parallelism: barriers across iterations, but thread pool and mapreduce parallelism within barriers, respectively. Other mapreduce benchmarks used in this paper have a single "reduce" operation towards the end of execution.

Although TPEq is not designed keeping the characteristics of the benchmarks in Table 5 in mind, we nonetheless also compare CS and MaxBIPS with TPEq on these benchmark applications. In fact, for the thread pool and mapreduce benchmarks, we expect MaxBIPS to perform the best. However, we find that TPEq is, in fact, competitive with, and in some cases outperforms, MaxBIPS for these benchmarks as well.

Sixteen parallel threads were used for each benchmark except Dedup and Ferret, which allow only 14 parallel threads (we note, however, Dedup and Ferret are not barrier synchronization based benchmarks, which are the main focus of this work). For 16 parallel threads, the PARSEC benchmarks also launch a 17th "initialization" thread that executes by itself on a core at the highest power and performance configuration. For a fair comparison, we report execution times starting from the time when parallel threads are first launched to the end of program execution.

TABLE 5
Benchmarks Using Alternative Approaches to Parallelization

Category	Workload	Benchmark Suite
Thread pool (TP)	Cholesky (C)	SPLASH-2
	Radiosity (C)	PARSEC
Pipeline Parallel (PP)	Dedup (M)	PARSEC
	Ferret (M)	PARSEC
MapReduce (MR)	Histogram (M)	Phoenix
	Linear regression (M)	Phoenix
	Matrix multiply (M)	Phoenix
	String match (C)	Phoenix
	Word count (M)	Phoenix

Labels in paranthesis indicate whether the application is memory-bound (M) or CPU-bound (C).

4 Experimental Evaluation

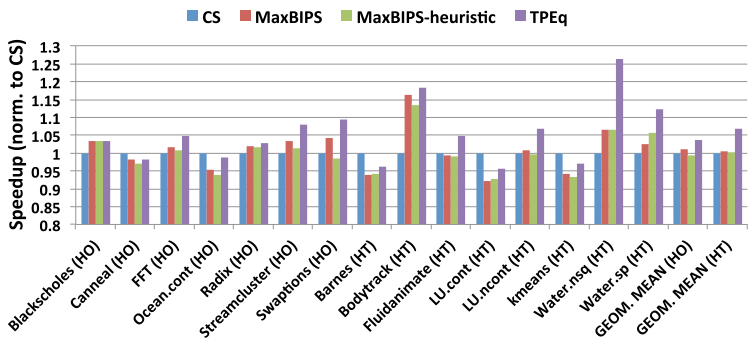
We have compared TPEq with state-of-the-art techniques discussed in Section 2.4. We briefly describe our implementation of these techniques.

Criticality Stacks (CS) [15]: Our CS implementation is faithful to the one reported in [15]. In every epoch, the thread with the highest criticality and above a threshold of 1.2 is accelerated on the fastest core configuration, and all the other configurations are set to the highest homogeneous configuration that consumes the remaining power budget. This thread is accelerated until its criticality value becomes less than 0.8 or another more critical thread above the threshold is found, in which case the new critical thread is accelerated. In *addition*, for a fair comparison, we ensure that if there is any residual power at this point, the remaining threads are accelerated to highest possible configurations in the order of decreasing thread criticality. This is the baseline approach over which we will compare TPEq. Although one can potentially devise more elaborate heuristics that look at the next most critical thread(s), we are not aware of any principled way to use CS for fine-grained optimization as enabled by TPEq.

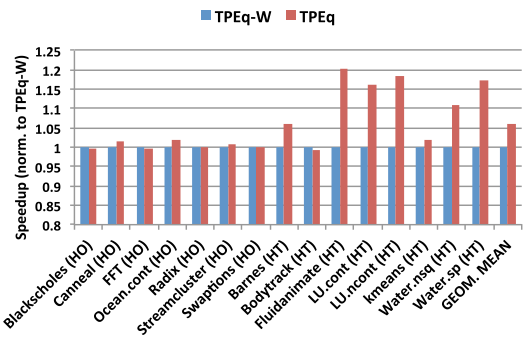
MaxBIPS [2]: MaxBIPS uses the same epoch length and predictors (power and performance) as TPEq. The sum-IPS optimization is performed using an off-the-shelf ILP solver in Matlab [31], and the solutions are fed back to Sniper. Since running an ILP solver would not be a practical solution in a real implementation, we also implemented *MaxBIPS-heuristic*, a polynomial time heuristic solver for the MaxBIPS objective function.

4.1 Power and Performance Prediction

The TPEq predictors were trained offline on a small subset of five randomly chosen benchmarks (out of 24) with different input sets as used in the rest of the experiments. In terms of CPI prediction, we observe a mean absolute error of 13.7 percent over more than 100,000 samples collected over all benchmarks. Of this, 5.7 percent can be attributed directly to temporal errors from last-value prediction. The rest of the error comes from predicting the CPI on one core configuration based on measurements on another core configuration. The PIE prediction mechanism [18] reports similar errors of 9-13 percent with only two (big and little) core configurations, while we have five. Further PIE only predicts CPI on a different core configuration for the current epoch, while we predict CPI for the *next* epoch.



(a) TPEq compared to CS and MaxBIPS



(b) Benefits of relative instruction count prediction (weights).

Fig. 5. (a) Speed-up of TPEq and MaxBIPS using the execution time of CS as baseline. Also shown is MAXBIPS-heuristic. (b) Speed-up of TPEq with respect to TPEq without relative instruction count prediction (TPEq-W), i.e., where all the weights are set to one.

The mean absolute error in power prediction is 4.43 percent, which is competitive with the errors reported in the state-of-the-art [22], [23], [32]. It is important to note that although the CPI predictions are used for predicting power, the power prediction error is lower for two reasons: (i) some positive and negative error terms from CPI estimation of individual cores cancel out in total power and (ii) power also has a constant static component. Because of the inaccuracy in power prediction, we observe that the power consumption occasionally exceeds the 80 W budget, but the average overshoot is only 3 W for both TPEq and MaxBIPS, and slightly higher for CS. Furthermore, power overshoots are short-lived and we observed only *one* instance in all our experiments where the overshoot exceeds 3 W for more than three successive epochs. In this (rare) event, a throttling mechanism kicks in to reduce power consumption. Prior work on proactive dynamic power management makes similar observations about overshoots [2], [33]. Although the performance sensitivity of TPEq to prediction accuracy was not evaluated, since it requires an oracular knowledge of both CPI and power for an exponential number of cases (i.e., for every configuration of every epoch), TPEq performance should improve if prediction accuracy improves.

We also note that TPEq power model implicitly depends on the behavior of the inter thread interactions on the shared last level cache (LLC). Note that our power model depends on the CPI per-core (Equation (5)), which itself depends on CPI_t^{mem} (Equation (4)), i.e., the number of clock cycles spent per load-store instruction in the memory subsystem. To illustrate this further, we conducted two experiments executing FFT with 4 threads: first, in which the 4 threads were scheduled on cores of different LLC banks (Table 1 microarchitecture has 4 LLC banks) and second, in which they were scheduled on cores sharing a single LLC bank. We observed that in doing so, while the uncore power component (LLC + memory interconnect) increased from 3.4 to 3.8 W due to higher miss rate, the core component of power decreased from 52.1 to 48.5 W due to lower IPC, thus more than compensating for the increase in uncore power. We also noted that for PARSEC benchmarks, the number of threads/cores sharing the last-level cache has little or no impact on the cache miss rate of shared cache, as also observed by Zhang et al. [9]. However, explicitly modeling the LLC cache miss rate might further improve the accuracy of TPEq power model.

4.2 Results on Barrier Synchronization Based Benchmarks

Fig. 5a compares the execution time of TPEq to the competing state-of-the-art techniques, MaxBIPS and CS for the benchmarks in Table 4. Also shown are the mean speed-ups (with CS as baseline) separately for the homogeneous (HO) and heterogeneous (HT) benchmarks.

Several observations can be made: first, we observe that TPEq is the best performing technique for all but one benchmark (out of 15). TPEq is up to 23 percent faster than CS and up to 15 percent faster than MaxBIPS. On average, TPEq is 5 and 10 percent faster than CS for homogeneous and heterogeneous benchmarks, respectively. The speed-up of TPEq over CS is greater for heterogeneous benchmarks since these benchmarks feature *both* IPC and instruction count heterogeneity, and provide greater opportunities for fine-grained optimization of core configurations. It is instructive to note that the performance improvements of TPEq are over and above techniques that are already very competitive: CS has been shown to improve over both AGETS and BIS (all coarse-grained optimization techniques since they only speed-up the most critical thread), while MaxBIPS is the only general, fine-grained technique that we are aware of. In addition, our results are over a wide range of barrier synchronization benchmarks over three benchmark suites without any benchmark sub-setting.

Does Relative Instruction Count Prediction Help? Fig. 5b compares TPEq with a version, TPEq-W, in which we do not perform relative instruction count prediction and instead set all the weights, $w(i)$, to one. Effectively, TPEq-W assumes that all threads execute the same number of instructions, and only account for IPC heterogeneity.

Note that although TPEq and TPEq-W are nearly identical for all homogeneous benchmarks (as expected, since all threads execute the same number of instructions), the speed-up of TPEq over TPEq-W is *significant* for heterogeneous benchmarks—15 percent on average and up to 20 percent.

Why Does TPEq Outperform MaxBIPS and CS? To better understand why TPEq outperforms MaxBIPS, Fig. 6 shows the MaxBIPS and TPEq progress plots for the FFT benchmark, focusing on the second barrier phase. Note that, compared to the baseline FFT progress plot in Fig. 1a, both the MaxBIPS and TPEq progress plots have much less heterogeneity in thread progress. In fact, although MaxBIPS is not explicitly meant to equalize IPCs, we observe that it many

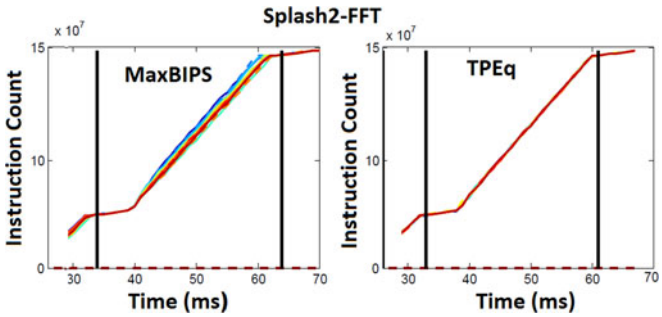


Fig. 6. Progress plots for FFT benchmark.

case, such as this, it speeds up low IPC threads and slows down high IPC threads. Nonetheless, it is not able to equalize as IPCs as effectively as TPEq, as is clear from Fig. 6—the progress plots for all threads are almost perfectly aligned with TPEq, but more spread out for MaxBIPS. Compared to the progress plot in Fig. 1a, TPEq achieves an almost 60 percent *reduction in stall time*.

Next we compare CS with TPEq, this time using Fluidanimate, a heterogeneous benchmark using progress plots shown in Fig. 7. Again, observe that TPEq is more successful in reducing thread stalls (regions where a thread's progress plot is flat) than CS, primarily because TPEq speeds-up or slows-down each thread optimally so they reach barriers at (about) the same time, while CS only speeds-up the most critical thread. In fact, the most critical thread identified by CS is sped-up more than necessary, and end up stalling on the next barrier. Compared to the baseline, in which all threads are executed on identical cores within same power budget, TPEq reduces total stall time by as much as 50 percent, while CS only results in less than 20 percent reduction in stall time.

Further insight can be obtained from Fig. 8, which shows the time spent by each thread in each configuration for CS and TPEq. Although it is clear that both CS and TPEq identify Thread 11 as most critical (assigning it to higher power/performance configurations), TPEq assigns each thread (including Thread 11) to a great range of configurations since it is able to perform fine-grained optimization. In fact, configuration 4 is not utilized by CS at all, while this is not the case for TPEq.

4.3 Results on Remaining Benchmarks

Fig. 9 shows the speed-up of TPEq and MaxBIPS normalized to CS for the benchmarks in Table 5 that do not use barrier synchronization. We reiterate that TPEq is best suited for barrier synchronization based parallel programs.

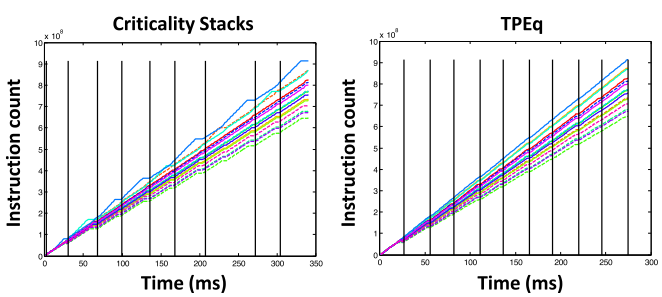


Fig. 7. Progress plots for Fluidanimate using CS and TPEq.

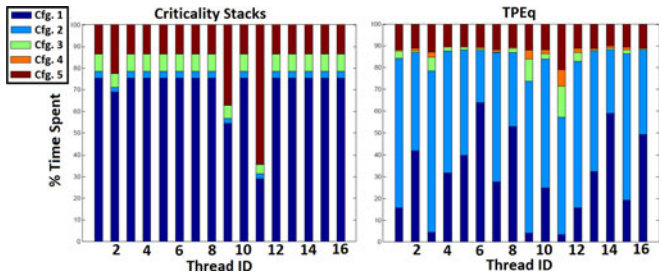


Fig. 8. Time spent by each thread in different configurations for CS and TPEq.

However, since TPEq does not explicitly look for barriers—adaptation happens at regularly sized epochs and threads asynchronously update their weights—it can be used with any parallel program.

We observe in Fig. 9 that even for these benchmarks TPEq has higher average performance than CS. In addition, it is competitive with MaxBIPS on average and on a per-benchmark basis. The improvement with respect to CS can be explained, in part, because TPEq (and MaxBIPS) both perform fine-grained optimization while CS is coarse-grained. On the other hand, the competitiveness of TPEq with MaxBIPS is more surprising since MaxBIPS should be the ideal objective at least for the thread pool and mapreduce benchmarks. We make the following observations to help explain the results: (a) for thread-pool and mapreduce benchmarks, we observed TPEq-W performs as well as TPEq and therefore TPEq is primarily equalizing instruction counts in these settings, in other words, acting as a load balancer; and (b) for pipeline benchmarks we note that the TPEq weights track thread criticality (to some extent), since the least (most) critical threads frequently (rarely) stall on full/empty queues. We have observed that for mapreduce benchmarks where TPEq performed worse, namely matrix multiply and string match, the IPC heterogeneity was also high and maintaining equal instruction counts among threads (employed by TPEq) had lower instruction throughput than simply assigning same core configuration to threads (employed by CS). Likewise, in pipeline benchmark ferret, CS only accelerates the most critical stage (with least stalls) of the pipeline, and this works better than TPEq, which attempts to maintain same relative instruction counts among stages. We, therefore, note that more work needs to be done on generalizing TPEq for other modes of parallelism beyond barrier synchronization.

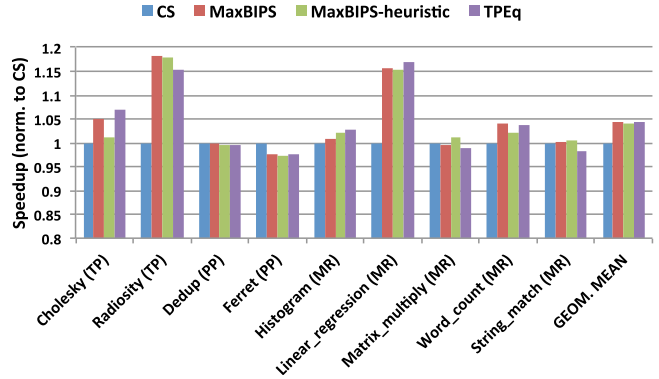


Fig. 9. Execution time on benchmarks in Table 5.

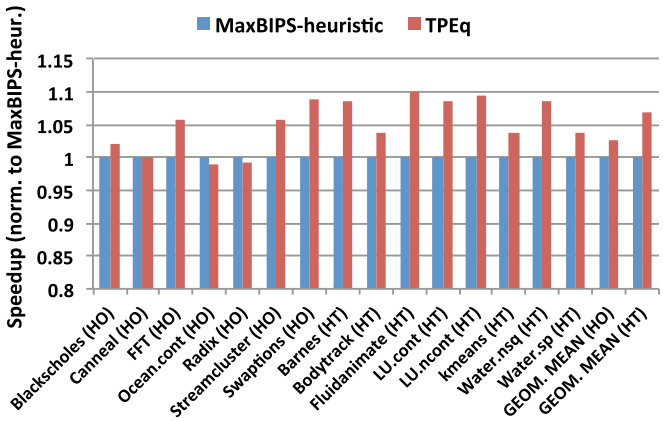


Fig. 10. Performance results for DVFS based dynamic adaptation comparing MaxBIPS with TPEq.

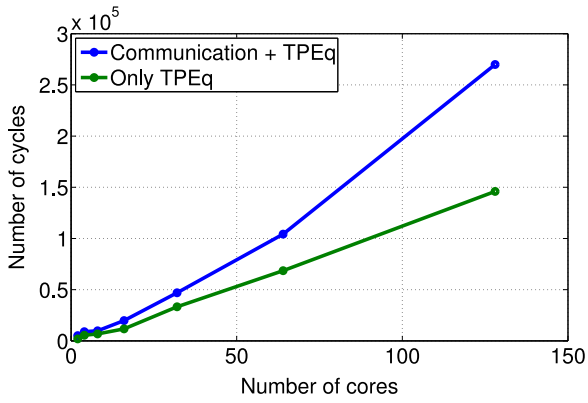


Fig. 11. TPEq run-time overhead.

4.4 DVFS Results

TPEq can be easily modified for DVFS based dynamic adaptation. For DVFS, the power and CPI predictors are trained for every voltage-frequency configuration (as opposed to every micro-architectural configuration) using the model described in Equations (4) and (5) in Section 2.2.2. In addition, the second term of the progress metric in Equation (2) is modified to $\frac{\mathcal{E} freq(i)}{w(i)CPI(i)}$, where $\mathcal{E}$ is now measured in seconds (as opposed to clock cycles) and $freq(i)$ is the frequency of thread i. We performed DVFS experiments with five voltage-frequency levels ranging between {0.8 V, 2.5 GHz} and {1 V, 3.5 GHz} and compare the performance of TPEq with MaxBIPS in Fig. 10. The average performance improvement over MaxBIPS is (6.9 percent) is slightly better than the improvements over MaxBIPS obtained for micro-architectural adaptation, in part because of more accurate CPI prediction.

We note that implementing TPEq in Linux on a real platform supporting core-level DVFS, such as the Intel Haswell processors [34], would require modifying a *governor* for the CPUFreq driver. For instance, the default *ondemand governor* [35] uses CPU utilization as a metric for switching the CPU frequency every 1 usec. Currently, this governor neither considers a power constraint nor inter thread interactions (i.e., thread criticality/progress). It is possible to modify this governor to use performance counters for TPEq prediction (progress, CPI and power) and set the voltage/frequency level based on TPEq optimization. However, we could not find any utility that allowed for core-level power measurement which is necessary for training TPEq models and further evaluating its performance. All current power measurement tools, including Intel's RAPL utility [36], only allow for package-level power measurement using software models, which are not necessarily more accurate than simulation.

4.5 TPEq Algorithm Runtime Overhead

The TPEq optimizer needs CPI stack information from all cores which happens implicitly via reads and writes to/from a shared address space, along with required synchronization between threads (as discussed in Section 2.3). For 16-cores, global communication takes roughly 10 K cycles and the TPEq prediction and optimization procedures (including prediction overhead) take another 10 K cycles. Together, the overhead amounts to 0.58 percent for a 1 ms epoch length, assuming conservatively, that all other cores are stalled while the leader executes the TPEq routine. The corresponding overhead for Criticality Stacks and MaxBIPS-heuristic was found to be 0.48 and 0.54 percent respectively. We also conduct a sensitivity analysis for TPEq overhead with increasing core/thread counts which is shown in Fig. 11. The close-to-linear scalability of TPEq optimization can be seen and is consistent with the complexity analysis of the algorithm which is $\mathcal{O}(MN \log N)$, with N being the number of cores. We observe that the overhead of global communication grows faster than that of TPEq optimization. For many core systems with 100 s of cores, hardware based communication and optimization support may be necessary.

5 Related Work

Dynamic power and resource management of multi-core processors is an issue of critical importance. Kumar et al. [37] proposed the notion of single-ISA heterogeneous architectures to maximize power efficiency while addressing temporal and spatial application variations. Their focus was primarily on multiprogrammed workloads. A number of papers have proposed scalable thread scheduling and mapping techniques for such workloads [38], [39], [40], [41], [42]. Others have focused on leveraging asymmetry to increase the performance of multithreaded applications by identifying and accelerating critical sections [15], [24], [25], [26], [43]. A more recent work by Craeynest et al. [44] proposed a technique for fairness-aware equal progress scheduling on heterogeneous multi-cores having a static configuration with two core types (big and small). Their proposed technique dynamically monitors the progress of each thread based on a slowdown metric, and schedules the slowest threads on big cores. The challenge in extending their approach is that they assume each core has a static configuration, i.e., the chip's TDP constraints are always met regardless of scheduling decisions. In contrast, in our setting, each core's configuration changes dynamically and unless the configurations are carefully chosen, the chip TDP constraint will not be met (for example, this can happen if all cores are set to the highest performance configuration). TPEq explicitly addresses the TDP constraint in its optimization procedure.

The work on DVFS based dynamic adaptation of multi-core processors has made use of the sum-IPS/Watt [3] or

MaxBIPS [2] objectives, and different optimization algorithms including distributed optimization [45], [46] and control theory [33], [47]. Cochran et al. [48] present a machine learning based approach based on offline workload characterization (and online prediction) but perform DVFS adaptation at a coarse time granularity of 100 billion uops (once in several seconds). TPEq, in contrast, adapts configuration every 1 ms, allowing it to maximize performance at a finer granularity. Recently, Godycki et al. [49] have proposed reconfigurable power distribution networks to enable fast, fine-grained, per-core voltage scaling and use this to *reactively* (as opposed to TPEq's proactive approach) slow down stalled threads and redistribute power to working threads. Also, unlike TPEq, this technique requires programmer inserted hints to determine the remaining work for each thread, and uses a heuristic approach to decide the voltage level of each core.

Prior work has also focused on the efficient implementation and utilization synchronization barriers. Wei et al. [50] have recently proposed tree-based barrier to reduce the synchronization time on NoC-based processors. Jea et al. [51] have proposed using barrier synchronization register to reduce inter-task communication overhead in barrier synchronization. Diamos et al. [52] have proposed using convergence barrier to synchronize threads diverging on a control flow on GPUs. TPEq, in contrast to above, is designed to equalize progress of threads synchronizing on a barrier on adaptive multicores and could work with with in synchrony with the aforementioned barrier implementations.

Barrier synchronization has also been well studied in the context of different programming models. Roth et al. [53] found that dynamic load balancing and using spin barrier instead of *pthread* barrier could significantly improve the performance of PARSEC benchmarks. Chen et al. [54] and Carlos et al. [55] have shown that load balancing on multiple devices could significantly improve the synchronization overhead for molecular dynamics (MD) simulation on CUDA and OpenCL implementations, respectively. Liu et al. [12] were perhaps the first to observe locality across barriers in the context of OpenMP workloads. Their work is most similar in spirit to TPEq, since they also use *last-value predictor* at the barriers to set the frequency of cores so as to save energy without compromising performance. However, TPEq is different from this technique on several counts. For one, [12] assumes that the slow-down of each thread is directly proportional to frequency, while the TPEq optimizer is more general and works for complex power-performance relationships that arise from micro-architectural adaptation and not just a simplified linear slow-down model. Second, TPEq does not require any explicit knowledge of a barrier event and is transparent to the programmer, while [12] requires programmer annotations. Thus TPEq generalizes easily to a broader set of barrier synchronization based benchmarks, and is not restricted to applications where barriers follow an easily discernible template (i.e., outermost *for* loops in OpenMP as studied by [12]). Finally, TPEq performs fine-grained adaptation (in time) at the granularity of an epoch, while [12] only changes frequency once every barrier phase.

6 Future Work

TPEq currently supports only single thread per core. When multiple threads are co-scheduled on the same core, the run-time decision would involve not only the optimal micro-architectural configuration (as TPEq does), but also which thread should be scheduled in each scheduling quanta. However, the goal remains the same—equalizing thread progress. Prior work has shown how thread progress metrics can be useful in making these thread scheduling decisions [56]; for example, by scheduling the thread that has made the least progress. We believe the TPEq progress metric might in fact help better estimate which thread has made the least progress, but a detailed analysis of TPEq in this context is future work. Note that if the cores support multiple threads with simultaneous multi-threading (SMT), the TPEq predictors would need to be extended to account for shared resources.

We will also extend TPEq to support DVFS islands in our future work. We believe that extending TPEq to DVFS islands would require TPEq optimization procedure to be combined with prior work on thread migration [57]. This is because with static thread-to-core mapping that TPEq currently assumes, it is not possible to ensure that the difference in thread arrival times at the barrier is minimized (e.g., when fastest and slowest thread share the same DVFS island), which is the primary objective of TPEq optimizer (Algorithm 1).

7 Conclusion

We proposed *Thread Progress Equalization*, a run-time mechanism to maximize performance under a power constraint for multithreaded applications running on multicores with support for fine grained dynamic adaption of core configurations. Compared with existing approaches, TPEq addresses all sources of inter-thread heterogeneity and determines in polynomial time the optimal configuration for each core so as to minimize execution time within a power budget. Experimental results show that TPEq outperforms state-of-the-art techniques on average in the context of both micro-architecturally adaptive multicores and DVFS, and while incurring modest execution time and hardware overheads.

References

[1] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proc. 38th Annu. Int. Symp. Comput. Archit.*, 2011, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2000064.2000108

[2] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *Proc. 39th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2006, pp. 347–358.

[3] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in *Proc. Int. Symp. Low Power Electron. Des.*, 2007, pp. 38–43.

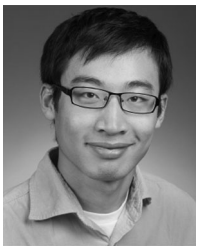
[4] G. Liu, J. Park, and D. Marculescu, "Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems," in *Proc. IEEE 31st Int. Conf. Comput. Des.*, 2013, pp. 54–61.

[5] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Proc. 22nd Annu. Int. Symp. Comput. Archit.*, 1995, pp. 24–36.

[6] P. Petrica and A. M. Izraelevitz, "Flicker: A dynamically adaptive architecture for power limited multicore systems," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 13–23.
[7] K. Meng, R. Joseph, R. P. Dick, and L. Shang, "Multi-optimization power management for chip multiprocessors," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 177–186.
[8] J. Sartori and R. Kumar, "Three scalable approaches to improving many-core throughput for a given peak power budget," in *Proc. Int. Conf. High Performance Comput.*, 2009, pp. 89–98.
[9] E. Z. Zhang, Y. Jiang, and X. Shen, "Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?" *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 203–212, 2010.
[10] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan, "Thread reinforcer: Dynamically determining number of threads via OS level monitoring," in *Proc. IEEE Int. Symp. Workload Characterization*, 2011, pp. 116–125.
[11] J. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy, "Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity," *J. Parallel Distrib. Comput.*, vol. 27, no. 2, pp. 118–141, 1995.
[12] C. Liu, A. Sivasubramaniam, M. Kandemir and M. J. Irwin, "Exploiting barriers to optimize power consumption of CMPs," in *Proc. 19th IEEE Int. Parallel Distrib. Process. Symp.*, 2005, pp. 5a–5a.
[13] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.
[14] T. Li, A. R. Lebeck, and D. J. Sorin, "Spin detection hardware for improved management of multithreaded systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 6, pp. 508–521, Jun. 2006.
[15] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, "Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 511–522.
[16] E. Duesterwald, J. Torrellas, and S. Dwarkadas, "Characterizing and predicting program behavior and its variability," in *Proc. 12th Int. Conf. Parallel Archit. Compilation Techn.*, 2003, pp. 220–231.
[17] A. Lukefahr, et al., "Composite cores: Pushing heterogeneity into a core," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2012, pp. 317–328.
[18] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer, "Scheduling heterogeneous multi-cores through performance impact estimation (PIE)," in *Proc. 39th Annu. Int. Symp. Comput. Archit.*, 2012, pp. 213–224.
[19] B. Sprunt, "Pentium 4 performance-monitoring features," *IEEE Micro*, vol. 22, no. 4, pp. 72–82, Jul./Aug. 2002.
[20] Q. Liang and IBM, "Performance monitor counter data analysis using counter analyzer," 2009. [Online]. Available: www.ibm.com/developerworks/aix/library/au-counteranalyzer/
[21] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate CPI components," in *Proc. 12th Int. Conf. Archit. Support Programming Languages Operating Syst.*, 2006, pp. 175–184.
[22] G. Contreras and M. Martonosi, "Power prediction for Intel xscale processors using performance monitoring unit events," in *Proc. Int. Symp. Low Power Electron. Des.*, 2005, pp. 221–226.
[23] W. L. Bircher and L. K. John, "Complete system power estimation using processor performance events," *IEEE Trans. Comput.*, vol. 61, no. 4, pp. 563–577, Apr. 2012.
[24] N. B. Lakshminarayana, J. Lee, and H. Kim, "Age based scheduling for asymmetric multiprocessors," in *Proc. Conf. High Performance Comput. Netw. Storage Anal.*, 2009, Art. no. 25.
[25] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *Proc. 17th Int. Conf. Archit. Support Programming Languages Operating Syst.*, 2012, pp. 223–234.
[26] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Utility-based acceleration of multithreaded applications on asymmetric CMPs," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 154–165.
[27] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2011, Art. no. 52.
[28] 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 *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 469–480.
[29] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Archit. Compilation Techn.*, 2008, pp. 72–81.
[30] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for multi-core and multiprocessor systems," in *Proc. IEEE 13th Int. Symp. High Performance Comput. Archit.*, 2007, pp. 13–24.
[31] MATLAB, *Version 8.1.0.604 (R2013a)*. Natick, MA, USA: The MathWorks Inc., 2013.
[32] K. Singh, M. Bhadauria, and S. A. McKee, "Real time power estimation and thread scheduling via performance counters," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 2, pp. 46–55, 2009.
[33] K. Ma, X. Li, M. Chen, and X. Wang, "Scalable power control for many-core architectures running multi-threaded applications," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 3, pp. 449–460, 2011.
[34] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An energy efficiency feature survey of the Intel Haswell processor," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshop*, 2015, pp. 896–904.
[35] V. Pallipadi and A. Starikovskiy, "The ondemand governor," in *Proc. Linux Symp.*, 2006, vol. 2, pp. 215–230.
[36] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory power estimation and capping," in *Proc. ACM/IEEE Int. Symp. Low-Power Electron. Des.*, 2010, pp. 189–194.
[37] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction," in *Proc. 36th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2003, pp. 81–92.
[38] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *Proc. ACM/IEEE Conf. Supercomputing*, 2007, Art. no. 53.
[39] R. Teodorescu and J. Torrellas, "Variation-aware application scheduling and power management for chip multiprocessors," *ACM SIGARCH Comput. Archit. News*, vol. 36, no. 3, pp. 363–374, 2008.
[40] D. Shelepov, et al., "HASS: A scheduler for heterogeneous multicore systems," *ACM SIGOPS Operating Syst. Rev.*, vol. 43, no. 2, pp. 66–75, 2009.
[41] J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov, "A comprehensive scheduler for asymmetric multicore systems," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 139–152.
[42] K. K. Rangan, G.-Y. Wei, and D. Brooks, "Thread motion: Fine-grained power management for multi-core systems," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 302–313, 2009.
[43] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 290–301, 2009.
[44] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-ISA heterogeneous multi-cores," in *Proc. 22nd Int. Conf. Parallel Archit. Compilation Techn.*, 2013, pp. 177–187.
[45] T. Ebi, M. Faruque, and J. Henkel, "Tape: Thermal-aware agent-based power econom multi/many-core architectures," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2009, pp. 302–309.
[46] J. Sartori and R. Kumar, "Distributed peak power management for many-core architectures," in *Proc. Des. Autom. Test Europe Conf. Exhibition*, 2009, pp. 1556–1559.
[47] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 199–212, 2011.
[48] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & cap: Adaptive DVFS and thread packing under power caps," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2011, pp. 175–185.
[49] W. Godycki, C. Torng, I. Bukreyev, A. Apsel, and C. Batten, "Enabling realistic fine-grain voltage scaling with reconfigurable power distribution networks," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2014, pp. 381–393.
[50] Z. Wei, P. Liu, R. Sun, and R. Ying, "Tab barrier: Hybrid barrier synchronization for NOC-based processors," in *Proc. IEEE Int. Symp. Circuits Syst.*, 2015, pp. 409–412.
[51] T.-Y. Jea, W. P. Lepera, H. Xue, and Z. Zhang, "Preparing parallel tasks to use a synchronization register," U.S. Patent 9,092,272, Jul. 28, 2015.

[52] G. F. Diamos, et al., "Execution of divergent threads using a convergence barrier," U.S. Patent 20,160,019,066, Jan. 21, 2016.
[53] M. Roth, M. J. Best, C. Mustard, and A. Fedorova, "Deconstructing the overhead in parallel applications," in *Proc. IEEE Int. Symp. Workload Characterization*, 2012, pp. 59–68.
[54] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao, "Dynamic load balancing on single-and multi-GPU systems," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2010, pp. 1–12.
[55] S. Carlos, P. Toharia, J. L. Bosque, and O. D. Robles, "Static multi-device load balancing for opencl," in *Proc. IEEE 10th Int. Symp. Parallel Distrib. Process. Appl.*, 2012, pp. 675–682.
[56] Q. Cai, et al., "Meeting points: using thread criticality to adapt multicore hardware to parallel regions," in *PACT. ACM*, 2008, pp. 240–249.
[57] Q. Cai, J. González, G. Magklis, P. Chaparro, and A. González, "Thread shuffling: Combining DVFS and thread migration to reduce energy consumptions for multi-core systems," in *Proc. Int. Symp. Low Power Electron. Des*, 2011, pp. 379–384.

Yatish Turakhia received the BTech and MTech (microelectronics) degrees in electrical engineering, along with a minor in computer science and engineering, from Indian Institute of Technology, Bombay. He is working toward the PhD degree in the Electrical Engineering Department, Stanford University. He received the NVIDIA graduate fellowship.



Guangshuo Liu received the BS degree in electrical and computer engineering from the University of Michigan-Shanghai Jiao Tong University Joint Institute, Shanghai, China, in 2012, and the MS degree in electrical and computer engineering from Carnegie Mellon University, in 2014. He is currently a software engineer with Alphabet's Nest Labs, Palo Alto, California.

Siddharth Garg received the BTech degree in electrical engineering from the Indian Institute of Technology Madras, and the PhD degree in electrical and computer engineering from Carnegie Mellon University, in 2009. He is currently an assistant professor with New York University, and prior to that, was an assistant professor with the University of Waterloo, from 2010-2014. He received the US National Science Foundation CAREER Award (2015) and his work has been recognized with awards at the IEEE Symposium on Security and Privacy (S&P) 2016, USENIX Security Symposium 2013, at the Semiconductor Research Consortium (SRC) TECHCON in 2010, and the International Symposium on Quality in Electronic Design (ISQED) in 2009. He also received the Angel G. Jordan Award from ECE Department, Carnegie Mellon University for outstanding thesis contributions and service to the community.

Diana Marculescu (S'94-M'98-SM'09-F'15) received the Dipl.Ing. degree in computer science from Polytechnic University of Bucharest, Bucharest, Romania, and the PhD degree in computer engineering from the University of Southern California, Los Angeles, California, in 1991 and 1998, respectively. She is currently in the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania. Her current research interests include energy- and reliability-aware computing, and more recently, in CAD for non-silicon applications, including computational biology and sustainability. She received the National Science Foundation Faculty Career Award from 2000 to 2004, the ACM SIGDA Technical Leadership Award in 2003, the Carnegie Institute of Technology George Tallman Ladd Research Award in 2004, and the Best Paper Award at the IEEE Asia and South Pacific Design Automation Conference in 2005, the Best Paper Award at the IEEE International Conference on Computer Design in 2008, the Best Paper Award at the International Symposium on Quality Electronic Design in 2009, and the Best Paper Award at the IEEE Transactions on Very Large Scale Integration (VLSI) Systems in 2011. She was a distinguished lecturer of the IEEE Circuits and Systems Society from 2004 to 2005 and the chair of the Association for Computing Machinery Special Interest Group on Design Automation from 2005 to 2009. She was the technical program chair of the ACM/IEEE International Workshop on Logic and Synthesis in 2004, the ACM/IEEE International Symposium on Low Power Electronics and Design in 2006, and the general chair of the same symposia in 2003 and 2007, respectively. She also served as the technical program chair of the IEEE/ACM International Symposium on Networks-on-Chip in 2012, the IEEE/ACM International Conference on Computer-Aided Design in 2013, and the general chair of the same conferences in 2015. She has served as an associate editor of the *IEEE Transactions on Very Large Scale Integration Systems*, the *IEEE Transactions on Computers*, and the *ACM Transactions on Design Automation of Electronic Systems*. She was recently selected as an ELATE fellow (2013-2014) and received the Australian Research Council Future Fellowship (2013-2017) and the Marie R. Pistilli Women in EDA Achievement Award (2014). She is a fellow of the IEEE and a distinguished scientist of the ACM.