Appears in the Proceedings of the 35th IEEE International Conference on Computer Design (ICCD), 2017

Using Application-Level Thread Progress Information to Manage Power and Performance

Sabrina M. Neuman, Jason E. Miller, Daniel Sanchez, Srinivas Devadas
MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA
{sneuman, jasonm, sanchez, devadas}@csail.mit.edu

***Abstract*—Power and thermal limitations make it impossible to run all cores on a multicore system at their maximum frequency. Therefore, modern systems require careful power management. These systems must manage complex tradeoffs between energy, power, and frequency, choosing which cores to accelerate to achieve good performance while maintaining energy efficiency or operating under a power budget.**

Navigating these tradeoffs is especially hard with multi-threaded applications, where performance depends on the relative progress of parallel worker threads between synchronization points. Prior work on chip-level power management for multi-threaded applications has largely relied on indirect heuristics and metrics calculated from low-level performance counters to estimate each thread's progress. However, these indirect metrics are often inaccurate. Instead, we propose to gather progress information directly from software itself.

We present ThreadBeats, a simple application-level annotation framework that directly and accurately conveys thread progress information to hardware. We design DVFS controllers that exploit ThreadBeats information for two purposes: (i) improving performance by equalizing thread progress and (ii) minimizing runtime under a power budget constraint. These controllers reduce wait time at barriers by 77% on average and improve energy-delay product under a power budget by 23% over prior work.

I. Introduction

Over the last several years power consumption has become a critical design constraint in nearly every type of computer system, making it necessary to impose a cap on the maximum power a processor can consume. Since the peak power of the processor is considerably higher than the cap, some portion of the processor must be turned off or throttled back to stay under the cap [1] using mechanisms such as dynamic voltage and frequency scaling (DVFS). Current trends indicate that future multicore processors will have fine-grained per-core DVFS [2]–[4]. The challenge then becomes dynamically adjusting each core's voltage and frequency to achieve the best possible application performance without exceeding the power limit.

Selecting the best DVFS settings is particularly challenging for multi-threaded applications. An important class of multi-threaded applications in the scientific, engineering, graphics, and data mining domains is data-parallel apps where multiple threads work together to complete tasks and meet up at periodic synchronization points after each one [5], [6]. In these apps, overall performance for a task is determined by the last thread to complete its work and reach a synchronization point. However, threads can have different computational loads which result in imbalances in performance or energy efficiency. Even when computational load is balanced, contention for shared resources or non-uniform machine architectures can create imbalance. To make matters worse, complex inter-thread interactions mean that adjustments to one thread can impact other threads. As a result, it is challenging to predict which threads should get more resources.

Previous work in power control of multi-threaded applications has relied heavily on low-level metrics (*e.g.,* instructions-per-cycle, cache miss rate) and heuristics to guess which threads should be sped up and which should be slowed down. Early work simply tried to maximize the instantaneous total number of instructions per second across all threads [7]. This type of solution works well for throughput-oriented applications with highly-independent threads, but does nothing to address imbalanced threads that synchronize periodically. Later attempts [8] use metrics like criticality [9] (based on cache misses) to guess which threads are lagging behind and should be accelerated. However, as shown in Section II-B, criticality is actually a poor estimate of relative thread progress.

A more reliable way to measure the individual progress of different threads is to instrument them at the application level [10]–[12]. The application knows how much useful work a thread has actually completed and how much it has left to do before the next synchronization point. Therefore, we propose a new thread progress metric that leverages application information to help energy, power, and performance management schemes better optimize the allocation of resources to different threads. Our technique uses lightweight annotations to gauge when each thread will complete the current task. This information can then be used to adjust the relative frequencies of the corresponding cores to complete the task as quickly as possible with minimal wasted energy.

This paper makes the following key contributions:

- Shows how using application-level thread progress information can be more accurate than indirect metrics such as criticality.
- Demonstrates the use of application-level thread progress information in runtime control algorithms for minimizing wait time at barriers and optimizing performance under a power cap.
- Demonstrates wait time reductions of 77.4% on average, and energy-delay product improvements of 22.9% on average under a power cap compared to prior work.

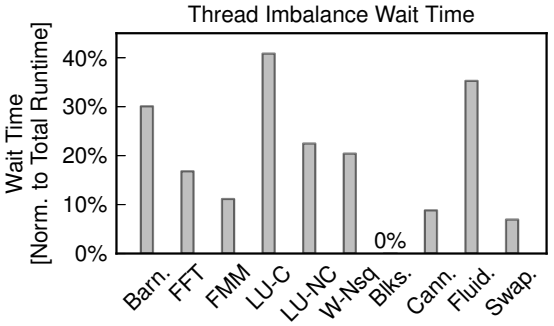

Fig. 1. Wait time for a collection of benchmarks at a fixed-frequency. The wait time makes up a significant portion of total runtime for many of the applications.

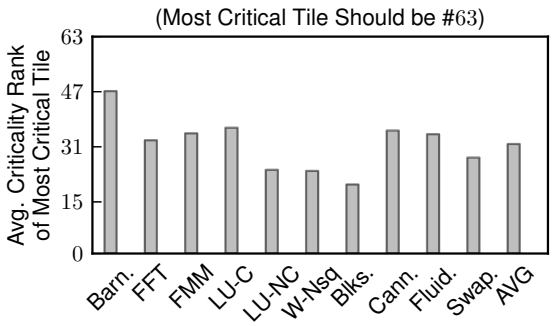

Fig. 2. Average rank that the criticality metric assigns to the truly most critical thread. In this 64-core experiment, that rank should always be #63, but it is instead misidentified as middling in criticality on average.

II. Motivation

Multithreaded applications have varied energy-performance tradeoff spaces. These tradeoffs are especially complicated because overall application performance depends on the relative progress individual threads make between synchronization events (such as barriers). However, it's very difficult for a low-level power controller to determine which threads are behind and which are ahead. Previous work has tried to infer thread progress using low-level metrics like IPC (instructions-per-cycle) or cache misses, but the resulting estimates frequently do not correlate well with actual application-level progress.

A. Thread Imbalance in Multithreaded Apps

Uneven rates of thread progress result in periods of idle waiting time for faster threads, which hit synchronization points early, then wait for slower threads to catch up. Figure 1 shows the fraction of time threads spend waiting at barriers in several benchmarks when all cores are clocked at the same fixed frequency.

To make optimal performance or energy decisions, a management system must be able to anticipate the effects of uneven thread wait time at synchronization points.

B. Indirect Thread "Criticality" Estimates

Previous work in power management has largely relied on indirect metrics to infer thread "criticality" for use in tuning application performance. An example criticality metric is the one presented by Bhattacharjee and Martonosi [9] and used in power controllers such as the work by Ma et al. [8], which infers thread criticality from the weighted amounts of L1 and L2 cache misses incurred by a thread over the last time interval:

$$crit_N = \text{Num}_{\text{L1 Misses}} + \left(\frac{\text{t}_{\text{L1L2 Miss}} \cdot \text{Num}_{\text{L1L2 Misses}}}{\text{t}_{\text{L1 Miss}}} \right) \quad (1)$$

In Equation 1, $\text{Num}_{\text{L1 Misses}}$ is the number of L1 cache misses that hit in the L2 cache. $\text{Num}_{\text{L1L2 Misses}}$ is the number of L1 cache misses that also miss in the L2 cache. The cache miss penalties for the L1 cache and the L2 cache are $\text{t}_{\text{L1 Miss}}$ and $\text{t}_{\text{L1L2 Miss}}$.

Indirect thread criticality metrics like this one are error-prone and can provide poor information for estimating application performance during runtime. This can have disastrous consequences for power controllers making decisions based on their feedback.

To demonstrate the potential inaccuracy of estimated criticality, we ran simulations of several benchmarks on a 64-core processor (using the methodology described in Section V), and collected thread criticality information as well as the actual thread progress using simple software-annotation. At periodic intervals, we examined the rank (from #0 to #63) that the criticality metric assigned to the true critical thread (determined by the actual progress information). The true critical thread should have been consistently ranked as most critical (#63). However, Figure 2 shows that this was not the case. On average, the criticality metric ranked the truly most critical thread in the middle, similar to what we would get from random guessing.

In a multithreaded application with synchronization points, misidentifying the most critical thread can have negative consequences for a controller seeking to manage application performance, because the performance of the whole application is dictated by the performance of the most critical thread.

III. ThreadBeats

The information needed to make good energy-performance tradeoff decisions is available in the software, and we can pass it directly to an energy management system with some simple annotations.

A. Application-Level Thread Progress Information

We propose the use of an application annotation scheme called ThreadBeats to provide application-level thread progress information. Pseudocode 1 shows an example of application instrumentation with ThreadBeats. In software, it is easy to annotate explicit synchronization barriers that bookend regions of substantial parallel work in multithreaded applications. In Pseudocode 1 we denote the barriers with calls to `tb_barrier`. In this work, we refer to a region of annotated parallel work between two annotated barriers as a "barrier-interval."

Pseudocode 1 ThreadBeats application annotation example.

```
1  perThreadWork(arguments) {
2    tb_barrier()       // Annotate barrier for controller
3    // Use arguments and thread index to determine
4    // total loop iterations to be performed (ntimes)
5    ntimes = f(arguments, thread_index)
6    tb_total(ntimes)   // Pass ntimes variable to controller
7    for (i = 0; i < ntimes; i++) { // Work loop
8      some_work()
9      tb_threadbeat()  // Emit ThreadBeat to controller
10   }
11   tb_barrier()       // Annotate barrier for controller
12 }
```

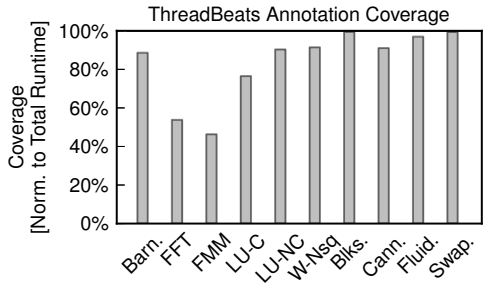

Fig. 3. Application runtime coverage of the ThreadBeats annotation. Time spent within ThreadBeats-annotated code regions covers a majority of the runtime for most of the applications.

Within a barrier-interval, we can annotate the bottom of loops between barriers (`tb_threadbeat` in Pseudocode 1) to emit a ThreadBeat to keep track of the number of loop iterations performed by a thread. For the best predictions of thread progress, the average amount of work performed per ThreadBeat should be nearly constant within a particular barrier-interval within a particular thread. As this amount of work becomes more variable, predictions of thread progress degrade accordingly. It is also best to make predictions at a rate slow enough to observe multiple elapsed ThreadBeats.

In the data-parallel applications addressed by this work, it is possible to determine during runtime the total number of loop iterations to expect between barriers, as a function of the thread index and arguments passed to the thread. This information can be used to track individual thread progress during the runtime of an application. Note that work does not need to be partitioned uniformly. The total number of iterations (the variable `ntimes`, passed to `tb_total`) can be different for different threads and from one barrier-interval to the next.

This style of software annotation is similar to previous work [10]–[12]. Instrumentation is a one-time effort. In this work we chose to annotate the applications by hand for simplicity and convenience, but instrumentation could be also be done automatically by a compiler or profiling tool.

Annotation with ThreadBeats amounted to a $\leq 1\%$ increase in total instructions executed at runtime. The time spent within regions of the code covered by ThreadBeats instrumentation covered a majority of the total runtime of most of the applications, as shown in Figure 3.

B. Performance Prediction Using ThreadBeats

Once an application is instrumented with ThreadBeats, the progress information can be used to make easy per-thread predictions of completion time for each barrier-interval.

We conceptually divide the predicted time a thread will spend until its completion of the barrier-interval into two components: t'_{comp} and t'_{mem}. The time that a thread will spend working on computation, t'_{comp}, can be adjusted by changes in frequency. The time that a thread will spend waiting for memory accesses, t'_{mem}, can not be changed because it depends on the fixed frequency of memory. The full predicted per-thread completion time of a barrier-interval, t', is current time, t, plus the sum of the computation and memory time predictions:

$$t' = t + t'_{\text{comp}} + t'_{\text{mem}} \tag{2}$$

To predict computation time t'_{comp} (within a thread, within a barrier-interval), the idea is that when we make a modeling assumption that the amount of work between ThreadBeats is nearly constant and the total amount of ThreadBeats remaining is known, the amount of computation time remaining in the barrier-interval for that thread can be linearly estimated from the computation time spent so far and the ratio of ThreadBeats remaining versus ThreadBeats seen so far. As mentioned in the previous section, when the assumption about nearly constant work between ThreadBeats is different from the behavior of the application, the predictions made by this model degrade. To incorporate frequency changes into this estimate, we assume that the estimate of computation time remaining will be linearly scaled by the ratio of the operating frequency used so far to the new desired frequency.

At some point in the middle of a barrier-interval, for a thread, the predicted t'_{comp} is proportional to the computation time elapsed so far, adjusted by the ratio of remaining ThreadBeat progress left, Num'_{TB}, to total ThreadBeats seen so far, Num_{TB}, and by the ratio of next proposed frequency, f', and average frequency so far, f (Equation 3). t_{mem} can be determined from hardware performance counters.

$$t'_{\text{comp}} = (t - t_{\text{mem}}) \left(\frac{\text{Num}'_{\text{TB}}}{\text{Num}_{\text{TB}}} \right) \left(\frac{f}{f'} \right) \tag{3}$$

To predict time spent waiting for memory during a barrier-interval, t'_{mem}, we use a simple model of memory behavior where we assume that the amount of memory wait time per ThreadBeat is constant. The total amount of memory time can then be linearly estimated from the memory time spent so far and the ratio of ThreadBeats remaining, Num'_{TB}, versus ThreadBeats seen so far, Num_{TB}. This model will also degrade if the assumption about constant memory wait time per ThreadBeat is not accurate.

The predicted t'_{mem} for a thread is only proportional to the memory time elapsed so far, adjusted by the ratio of remaining ThreadBeat progress left (Equation 4). Again, it is independent of frequency.

$$t'_{\text{mem}} = t_{\text{mem}} \left(\frac{\text{Num}'_{\text{TB}}}{\text{Num}_{\text{TB}}} \right) \tag{4}$$

The maximum of the per-thread completion time predictions, t', at a given set of frequencies will belong to the most critical thread, who will arrive at the end of the barrier-interval last. Managing the relative predicted completion times of the threads, especially the most critical thread, allows energy, power, and performance control systems to make informed tradeoffs and achieve efficient operation.

IV. ThreadBeats Use-Case Examples

In this section, we describe TBSync and TBPower, two example use-cases for employing ThreadBeats in the management of energy, power, and performance.

A. Thread Synchronization Demonstration

Thread imbalance in multithreaded applications causes periods of wait time at the end of barrier-intervals, during which one or more threads have completed their work and are waiting for slower threads to finish (see Figure 1). We show how ThreadBeats can be used to try to minimize the wait time in barrier-intervals in a simple demonstration called TBSync.

The goal of TBSync is to have all tiles complete the barrier-interval at as close to the same time as possible, reducing or eliminating the wait time. At each invocation, TBSync computes the predicted completion time of all tiles assuming they will all be run at some fixed frequency for the rest of the barrier-interval using the equations derived in Section III-B. The minimum of these completion times (representing the fastest-finishing, least-critical thread) becomes the completion time target. TBSync computes and applies the frequencies necessary for all tiles to match that target completion time.

In Equation 5, f_{fixed} is some fixed frequency that the least-critical thread is run at to establish a completion time target. For each of N threads, a speedup ratio, $k_{\text{speedup, } n}$, is calculated by dividing the completion time target by the predicted completion time of thread n at that same fixed frequency. The fixed frequency and the speedup ratios are used to calculate f'_n, the frequency that each thread should run at to try to match the completion time target.

$$f'_n = k_{\text{speedup, } n} \cdot f_{\text{fixed}} \tag{5}$$

Experimental results using TBSync to try to eliminate wait time compared to minimum fixed frequency runs of applications are presented in Section VI-A.

B. Power Controller

For power control, the goal is to maximize performance under a given power budget. To do this, the TBPower controller tries to relatively speed up all threads to match the completion time of the least-critical thread, while scaling all thread frequencies together to stay under the power budget.

At each control interval, the controller first predicts the completion time of the least-critical thread if it were run at the minimum frequency. Then, it calculates the ratio of relative frequency speedup needed for each thread to match that completion time, as described in Section IV-A. Finally, using those frequency speedup ratios and a simple model of energy prediction, the controller solves for a "baseline" frequency for the least-critical thread. The baseline frequency is the highest frequency it can run the least-critical thread, with all other threads run at the speedup ratios up from that baseline frequency, while not violating the imposed power cap.

To make energy predictions, we use a simple model based on recent energy expenditure. For each of N threads, a per-thread estimated linear relationship between the frequency and energy over the last control interval is found, represented by the constant $b_{\text{energy, } n}$. Equation 6 shows how this linear model is then used to predict the energy of thread n for the next interval given the new frequency setting f'_n.

$$E'_n = b_{\text{energy, } n} \cdot f'_n \tag{6}$$

Combining the speedup ratios from Equation 5 and the simple energy model from Equation 6 gives Equation 7, where the total energy for all N threads is the linear combination of thread frequencies represented as speedups over the baseline frequency, f_{baseline}. The total energy is set to E_{budget} to solve for a baseline frequency that does not violate the power budget.

$$E_{\text{budget}} = \sum_{N} b_{\text{energy, } n} \cdot k_{\text{speedup, } n} \cdot f_{\text{baseline}} \tag{7}$$

The threads are then set to frequencies equal to their speedup factors, $k_{\text{speedup, } n}$, multiplied by the baseline frequency, f_{baseline}. Experimental results using TBPower are presented in Section VI-B.

TABLE I
Configuration of the Simulated System

Parameter	Settings
Cores	64, in-order
Max Frequency/Vdd	4GHz/1.57V
Min Frequency/Vdd	0.72GHz/0.8V
Technology/Temperature	32nm/340K
Private L1 I/D-Cache per Core	4-way, 16/32KB
Private L2 Cache per Core	8-way, 512KB
Memory Controllers	8, each 5GB/s
Memory Access Latency	60ns, plus queueing

V. Methodology

In this section we describe the simulation setup, benchmarks, and baselines for comparison used in the evaluation of our ThreadBeats use-cases presented in Section VI.

A. Simulation Setup

We use the Graphite multicore simulator with integrated power modeling from McPAT [13]. Simulation parameters are summarized in Table I.

Graphite/McPAT's default static power values were disproportionately large in relation to the dynamic power values. To address this, we re-scaled the static power component by a constant factor (0.25) so that static power was $\sim$20% of dynamic power at the maximum processor clock frequency

TABLE II
BENCHMARKS EVALUATED

Benchmark	Suite	Problem Size
Barnes	SPLASH-2	64k particles
FFT	SPLASH-2	4M points
FMM	SPLASH-2	64k particles
LU-C	SPLASH-2	2048x2048 matrix
LU-NC	SPLASH-2	1280x1280 matrix
Water-Nsq	SPLASH-2	1728 molecules
Blackscholes	PARSEC	64k options
Canneal	PARSEC	60k, 2k°, 32 steps
Fluidanimate	PARSEC	5 frames, 100k part.
Swaptions	PARSEC	64 swap., 10k sim.

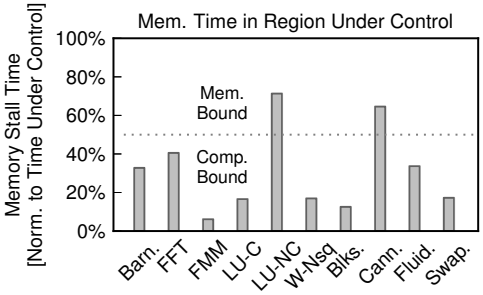

Fig. 4. Relative amount of memory delay time compared to the total runtime of applications. A dashed dividing line is drawn down the middle at 50%. Applications whose proportion of delay time falls under the line can be said to be compute-bound, while those above the line are memory-bound.

and $\sim$50% of dynamic power at the minimum frequency, a range that is representative of real systems [14], [15].

Current processors have only a few DVFS points, but technology is headed towards more fine-grained DVFS [2]–[4]. Some prior work in power control [8] has essentially created intermediate operating points by interpolating (switching back and forth) between adjacent operating points during a control period. To permit the DVFS flexibility expected in future processors, we model a large number (21) of voltage operating levels and we allow the continuous selection of frequency. The system automatically selects the best voltage for that frequency.

We use a control period of 250µs (in keeping with prior work [8]), which is long enough to amortize the cost of DVFS transitions. Future technologies have been proposed that minimize these costs by having gradual frequency transitions.

The ThreadBeats control algorithms presented in Section IV can be implemented in software or a dedicated microcontroller. The relative runtime overhead of performing the control algorithms in software is an average of <2% compared to the total runtime of the benchmark applications. This overhead and the overheads of the baselines for comparison are not included in the evaluation results because the focus of this work is demonstrating the use of the ThreadBeats progress metric with simple use-cases. We leave the design of sophisticated control algorithms to future work.

B. Benchmarks

We use applications from the SPLASH-2 [5] and PARSEC [6] benchmark suites. These applications were selected from among the benchmarks supported by the Graphite simulator because they demonstrate the algorithmic paradigm targeted by our approach—they are multithreaded applications that contain parallel regions bookended by synchronization points. Table II details benchmark information.

LU-NC and Canneal are predominantly memory-bound applications (see Figure 4). They are difficult to manage using DVFS since they are less sensitive to changes in the frequency of computational elements, and they have very low power dissipation on the simulated system configuration under test.

C. Baselines for Comparison

We compare our ThreadBeats use-case examples to several baselines in the evaluation presented in Section VI.

Thread Synchronization Baselines. The baseline for the thread synchronization demonstration is a fixed frequency run of the applications at the minimum frequency setting.

Power Control Baselines. We compare against three power control baselines. FreqPar is a power controller from prior work that uses the thread criticality metric described in [9] to assign frequencies to threads in a multithreaded application. For details, we refer the reader to the FreqPar paper [8]. We should note that FreqPar is intended to manage multiprogrammed workloads, so we only compare against a subset of the FreqPar scheme, the power control loop that determines an "aggregated frequency quota" and the algorithm wherein the *freq*uency quota is *par*titioned among cores in a single application. The power control loop is a simple proportional controller. The single application frequency partitioning is done according to the relative proportions of thread criticality over the last interval:

$$f_N = \frac{crit_N}{\sum_{\text{all tiles}} crit_k} \tag{8}$$

The other two controllers are novel naïve baselines. SamePar is a naïve version of FreqPar that retains the simple proportional power control loop, but always divides the frequency quota equally among threads. SamePower is the naïve version of our power control algorithm, TBPower. Like SamePar, it preserves the energy estimation and power control of our power controller, but under those constraints it always gives equal frequencies to all threads. These naïve baselines are intended to evaluate whether FreqPar and our power controller can add value with their use of heuristics based on either thread progress estimates or direct information—criticality in FreqPar's case, and ThreadBeats for our power controller.

VI. EVALUATION

This section evaluates TBSync and TBPower against the baselines described in Section V-C and previous work.

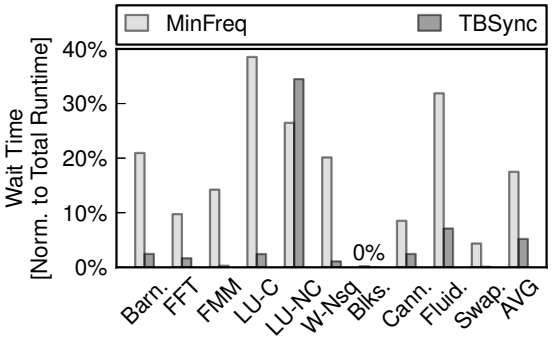

Fig. 5. Wait time at the minimum fixed frequency compared to wait time under TBSync management (smaller is better). Relative wait time across all applications was reduced by an average of 77.4% by TBSync. It was not effective on the memory-bound app LU-NC, where app behavior and modeling error led to a wait time increase.

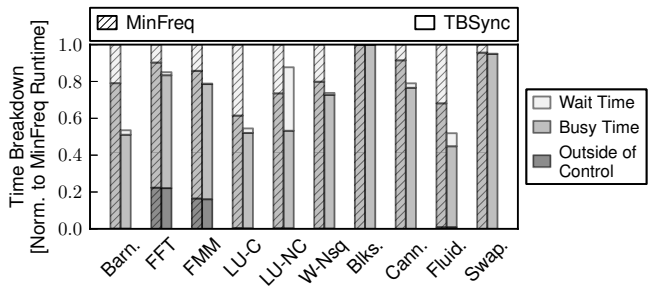

Fig. 6. Time breakdown at the minimum fixed frequency compared to wait time under TBSync management. The wait time data is the same as in Figure 5. Time outside of control is constant. The goal of this demonstration was to leave the busy time unchanged while decreasing the wait time. While TBSync was successful at reducing wait time on all apps except LU-NC, modeling error during runtime led to unintentional reductions in busy time as well, particularly in Barnes, LU-NC, and Fluidanimate.

A. Thread Synchronization Experiments

The goal of thread synchronization is to reduce or eliminate wait time at barriers. We therefore ran the benchmarks twice: once with DVFS fixed at the minimum frequency and once with TBSync enabled (see Section IV-A). Figure 5 shows the results normalized to the runtime of the fixed frequency case. Figure 6 gives a complete time breakdown for the minimum frequency and TBSync experiments for all applications, dividing runtime into wait time, busy time, and any time spent outside of control. Time spent outside of control is constant.

The goal of TBSync was to decrease the wait time while leaving the busy time unchanged. While TBSync was successful in reducing wait time for most of the applications, modeling imperfections did cause it to unintentionally speed up the busy time of some applications, most notably in Barnes, LU-NC, and Fluidanimate (see Figure 6). As seen in Figure 5, TBSync was able to reduce the amount of wait time natively present in all applications at the fixed frequency by an average of 77.4%. The only application in which the wait time increased was LU-NC, where irregular thread activity degraded the accuracy of the time prediction models and led the controller to speed up some non-critical threads. These non-critical threads reached the barrier so prematurely that they had to wait longer for their peers to catch up than in the fixed frequency case. Thus, the wait time of the application increased, even though the total runtime decreased.

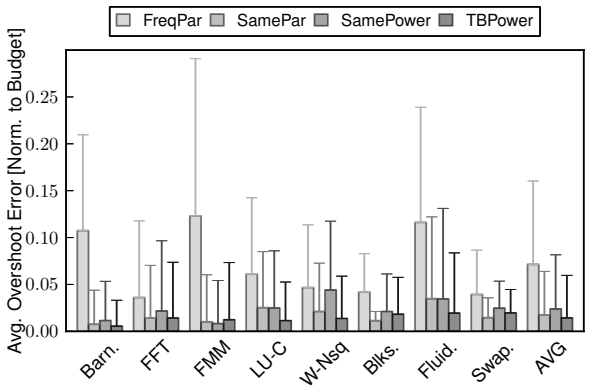

Fig. 7. Average power in excess of budget, or overshoot error, under a power budget of 50% peak power, normalized to the budget power (smaller is better). The bars represent the average excess power over the power budget during runtime. The whiskers are at one standard deviation of instantaneous power above the average excess power during the run. TBPower control gave the lowest amount of overshoot error, an improvement of 80% over FreqPar. In real systems, a large amount of overshoot error exceeding a power budget can lead to processor performance degradation or failure.

B. Power Control Experiments

In this experiment, we compared TBPower to three other controllers (described in Section V-C): a controller from previous work, FreqPar [8], and two novel baselines – SamePar (a naïve baseline for FreqPar), and SamePower (a naïve baseline for TBPower).

We ran all power controllers with a budget of 50% of the peak processor power, determined by running the applications with all tiles at maximum frequency. The two memory-bound benchmarks, LU-NC and Canneal, have very low power consumption on the simulated system configuration and never exceed this budget, even at maximum frequency. Consequently, power control results are not presented for those two benchmarks.

The average power in excess of the power budget, or overshoot error, for the four power controllers under test is presented in Figure 7. Results for the total energy-delay product, runtime, and energy for the controllers are presented in Figures 8, 9, and 10, normalized to SamePower's values.

As seen in Figure 7, the TBPower controller gave 80% less power error on average than FreqPar, and the lowest amount of overshoot error on average overall. FreqPar's high overshoot error is problematic, since large overshoot violations can cause a real processor to exceed its thermal operating limits or the capacity of its power supply, leading to a failure of the system.

Figure 8 shows energy-delay product (EDP) results for all controllers. EDP is an important metric in this scenario because it gives a holistic view of the tradeoffs the controllers make between power and performance. EDP de-emphasizes performance gains that come at the expense of violating

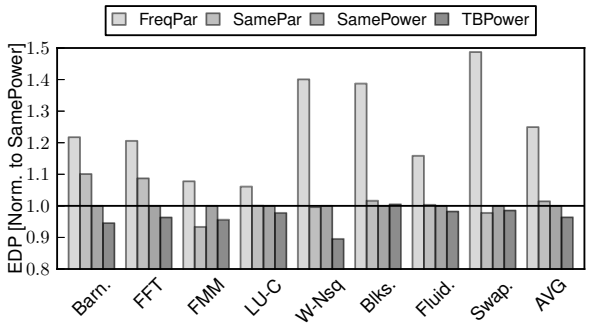

Fig. 8. Energy-delay product (EDP) under a power budget of 50% peak power (smaller is better). EDP gives insight into the balance of power and performance managed by the controllers, de-emphasizing performance gains that come at the cost of violations of the power budget. TBPower control gave an average EDP improvement of 3.6% over its baseline, SamePower, and an average of 22.9% compared to FreqPar.

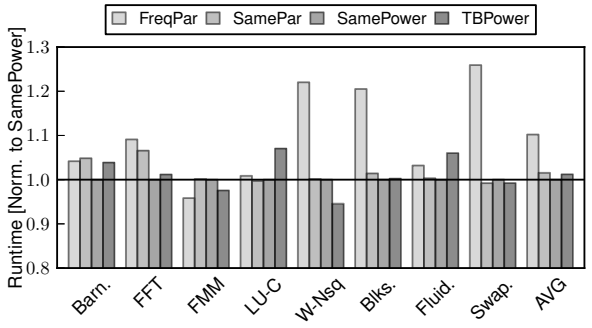

Fig. 9. Runtime under a power budget of 50% peak power (smaller is better). FreqPar achieved a lower runtime on some applications, but it came at the cost of large violations of the power budget, invalidating those results. Even so, TBPower control still gave an average runtime improvement of 8.2% compared to FreqPar.

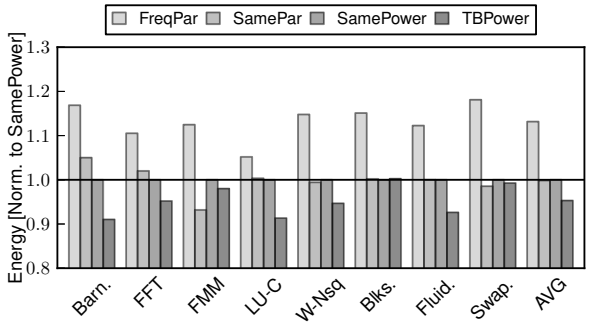

Fig. 10. Energy under a power budget of 50% peak power (smaller is better). TBPower control gave an average energy reduction of 15.8% over FreqPar.

the power budget, contrary to the intended purpose of these controllers. The TBPower controller achieved an average of 22.9% EDP improvement compared to FreqPar, and it showed an overall EDP improvement of 3.6% on average compared to SamePower, the novel naïve baseline controller.

Figures 9 and 10 show the individual components of runtime and energy for all controllers that are represented in the EDP

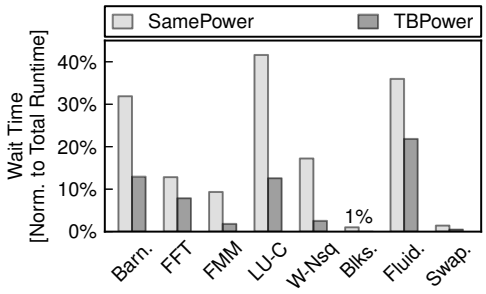

Fig. 11. Wait time under a power budget for TBPower and its baseline, SamePower. For most applications, TBPower reduced wait time, exploiting the power-performance tradeoff opportunity.

results in Figure 8. On applications where FreqPar achieved a lower runtime (*e.g.,* FMM), it did so by using a large amount of energy and violating the power budget (Figure 7). This is poor behavior from a power controller. Interestingly, SamePar, the supposedly "naïve" version of FreqPar, does much better. This highlights the negative impact of using the inaccurate thread criticality metric. For these applications, using the metric is worse than doing nothing. However, TBPower does better by using application-level information. Despite FreqPar's misbehavior, TBPower still gave an average runtime improvement of 8.2% over FreqPar. Because TBPower respected the power budget while achieving good runtime, it managed an average energy reduction of 15.8% compared to FreqPar and an average energy improvement of 4.7% over SamePower.

For most applications, TBPower reduced wait time compared to SamePower (see Figure 11), which gave it the energy-runtime tradeoff flexibility to improve EDP. Only on Blackscholes does SamePower get a slightly lower EDP than TBPower, but the results are very close. As seen in Figure 11, Blackscholes has almost no wait time opportunities to exploit for an EDP improvement. The other applications had more inherent wait time, so TBPower had the flexibility to achieve an EDP improvement.

VII. Related Work

Instruction Count Metrics. There have been a number of techniques proposed for power management using various instructions per cycle (IPC)-based metrics [7], [16]–[18]. However, these low-level performance counter metrics are ill-suited to gauging the application-level progress of threads in multithreaded programs that use synchronization and can exhibit workload imbalance.

Criticality Metrics. Thread criticality metrics attempt to estimate relative thread progress and identify critical threads. In Section II-B we described a thread criticality metric based on cache misses [9]. Various other criticality metrics calculated from indirect progress information have been proposed by previous work. Thread Progress Equalization [19] is recent work addressing thread imbalance that estimates thread progress during an epoch from low-level measurements of cycles per instruction weighted by expected total thread progress during the epoch. However, it estimates the total thread progress by

profiling previous epochs, which means both a startup time before progress can be estimated as well as susceptibility to confusion from applications that exhibit different workload balances from one phase to the next– both issues are neatly circumvented by using application-level information like ThreadBeats. Criticality Stacks [20] calculates thread criticality scores by weighing the amount of time a thread is active while other threads are inactive. This approach is generalized to react to multiple types of underlying causes for thread asynchrony, but it is still an indirect heuristic measurement of progress and it does not address imbalanced workloads.

Application-level Information. Some previous work uses metrics that are informed by application-level hints like the annotation of synchronization primitives, but which stop short of providing complete thread progress information like ThreadBeats. Thrifty Barrier [21] is an early work in synchronization primitive annotation that relies on profiling past barrier wait times to optimize future barrier wait times. Bottleneck Identification and Scheduling [22] and Booster [23] both seek to accelerate threads under high contention for software-annotated synchronization bottlenecks using coarse-grained thread acceleration techniques. Utility-Based Acceleration [24] combines bottleneck acceleration with an instruction-count based thread criticality metric. Meeting Points [10] is an early work that uses application-level thread progress reporting annotation, but is only applicable to parallel loops with balanced workloads. Dynamic Core Boosting [12] is a recent work that uses application-level thread progress reporting annotation that accommodates imbalanced workloads, like ThreadBeats. However, the focus of that work is on accelerating cores in asymmetric chip multiprocessors using coarse-grained boosting techniques, whereas ThreadBeats is focused on tuning fine-grained per-core DVFS in symmetric chip multiprocessors. There has been a wealth of power control work that uses Application Heartbeats [11] software instrumentation, but Application Heartbeats is aimed at marking the progress rate of an entire application, not the relative progress of its individual threads.

VIII. Conclusion

This paper presents a simple application annotation technique, ThreadBeats, that provides direct information about thread progress in multi-threaded applications. We presented two use cases for ThreadBeats: TBSync, which seeks to improve performance by reducing barrier wait time, and TBPower, which seeks to minimize runtime under a power cap. Evaluation results show that TBSync reduces barrier wait times by 77.4% on average and TBPower improves energy-delay product by 22.9% on average over prior work. These results indicate that using an accurate thread progress metric is crucial to improving system efficiency. The use of application-level information, like ThreadBeats provides, will be essential to navigating the increasingly complex power and performance tradeoffs that future machines will exhibit.

References

[1] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ISCA*, 2011.

[2] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core dvfs using on-chip switching regulators," in *HPCA*, 2008.

[3] W. Godycki, C. Torng, I. Bukreyev, A. Apsel, and C. Batten, "Enabling realistic fine-grain voltage scaling with reconfigurable power distribution networks," in *MICRO*, 2014.

[4] R. Jevtić, H.-P. Le, M. Blagojević, S. Bailey, K. Asanović, E. Alon, and B. Nikolić, "Per-core dvfs with switched-capacitor converters for energy efficiency in manycore processors," in *IEEE Trans. VLSI Syst.*, 2015.

[5] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ISCA*, 1995.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *PACT*, 2008.

[7] 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 *MICRO*, 2006.

[8] K. Ma, X. Li, M. Chen, and X. Wang, "Scalable power control for many-core architectures running multi-threaded applications," in *ISCA*, 2011.

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

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

[11] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments," in *ICAC*, 2010.

[12] H. K. Cho and S. Mahlke, "Embracing heterogeneity with dynamic core boosting," in *Intl. Conf. on Comput. Frontiers*, 2014.

[13] G. Kurian, S. M. Neuman, G. Bezerra, A. Giovinazzo, S. Devadas, and J. E. Miller, "Power modeling and other new features in the graphite simulator," in *ISPASS*, 2014.

[14] P. Zajac, M. Janicki, M. Szermer, and A. Napieralski, "Evaluating the impact of scaling on temperature in finfet-technology multicore processors," in *Microelectronics Journal*. Elsevier, 2014.

[15] M. Kulkarni, K. Sheth, and V. D. Agrawal, "Architectural power management for high leakage technologies," in *IEEE Southeastern Symp. on Syst. Theory (SSST)*, 2011.

[16] K. Meng, R. Joseph, R. P. Dick, and L. Shang, "Multi-optimization power management for chip multiprocessors," in *PACT*, 2008.

[17] J. A. Winter, D. H. Albonesi, and C. A. Shoemaker, "Scalable thread scheduling and global power management for heterogeneous many-core architectures," in *PACT*, 2010.

[18] S. S. Jha, W. Heirman, A. Falcón, T. E. Carlson, K. Van Craeynest, J. Tubella, A. González, and L. Eeckhout, "Chrysso: An integrated power manager for constrained many-core processors," in *Intl. Conf. on Comput. Frontiers*, 2015.

[19] Y. Turakhia, G. Liu, S. Garg, and D. Marculescu, "Thread progress equalization: Dynamically adaptive power and performance optimization of multi-threaded applications," *IEEE Trans. Comput.*, 2017.

[20] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, "Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior," in *ISCA*, 2013.

[21] J. Li, J. F. Martínez, and M. C. Huang, "The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors," in *HPCA*, 2004.

[22] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Bottleneck identification and scheduling in multithreaded applications," in *ASPLOS*, 2012.

[23] T. N. Miller, X. Pan, R. Thomas, N. Sedaghati, and R. Teodorescu, "Booster: Reactive core acceleration for mitigating the effects of process variation and application imbalance in low-voltage chips," in *HPCA*, 2012.

[24] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Utility-based acceleration of multithreaded applications on asymmetric cmps," in *ISCA*, 2013.