# DEP+BURST: Online DVFS Performance Prediction for Energy-Efficient Managed Language Execution

Shoaib Akram, Jennifer B. Sartor, and Lieven Eeckhout

**Abstract**—Making modern computer systems energy-efficient is of paramount importance. Dynamic Voltage and Frequency Scaling (DVFS) is widely used to manage the energy and power consumption in modern processors; however, for DVFS to be effective, we need the ability to accurately predict the performance impact of scaling a processor's voltage and frequency. No accurate performance predictors exist for multithreaded applications, let alone managed language applications.
In this work, we propose DEP+BURST, a new performance predictor for managed multithreaded applications that takes into account synchronization, inter-thread dependencies, and store bursts, which frequently occur in managed language workloads. Our predictor lowers the performance estimation error from 27% for a state-of-the-art predictor to 6% on average, for a set of multithreaded Java applications when the frequency is scaled from 1 to 4 GHz. We also novelly propose an energy management framework that uses DEP+BURST to reduce energy consumption. We first target reducing the processor's energy consumption by lowering its frequency and hence its power consumption, while staying within a user-specified maximum slowdown threshold. For a slowdown of 5% and 10%, our energy manager reduces on average 13% and 19% of energy consumed by the memory-intensive benchmarks. We then use the energy manager to optimize total system energy, achieving an average reduction of 15.6% for a set of Java benchmarks. Accurate performance predictors are key to achieving high performance while keeping energy consumption low for managed language applications using DVFS.

**Index Terms**—Dynamic voltage and frequency scaling, multithreaded performance estimation, managed runtimes, dynamic energy management.

## 1 Introduction

In modern times, improving the energy-efficiency of computer systems is of prime importance. One way to manage the processor's power and energy consumption is using Dynamic Voltage and Frequency Scaling (DVFS). DVFS allows one to simultaneously change a processor's voltage and frequency. To effectively utilize DVFS, we need the ability to predict its performance impact on applications at run-time. Accurate DVFS performance prediction enables different opportunities for reducing the energy consumed by our applications. In particular, two possibilities include reducing the energy consumption while honoring a user-specified performance constraint, or running applications at the frequency that minimizes total energy consumption.

During the last decade, significant progress has been made in understanding and predicting the performance impact of DVFS for native sequential applications written in C and C++, see for example [11], [19], [29], [35], [38], [46]. However, prior work lacks a DVFS predictor for multithreaded applications, especially those written in managed languages, such as Java.

Existing DVFS predictors for sequential applications view a processor core as either executing instructions or waiting for memory accesses to return. The time spent executing instructions scales with frequency, whereas the time spent waiting for memory does not. Although this view suffices for sequential applications, it is not sufficient for multithreaded applications. For one, synchronization activity in multithreaded applications leads to inter-thread dependencies. Consequently, speeding up or slowing down one thread using DVFS impacts the execution of dependent threads, leading to complex interactions which affect overall application performance. A DVFS predictor for multithreaded applications therefore needs to take into account synchronization when predicting the total execution time at the target frequency.

Managed applications, which run on top of a virtual machine, exhibit even more inter-thread dependencies than native applications. Service threads, such as those that perform garbage collection and just-in-time compilation, run alongside the application threads [9], [37]. Application and service threads need to synchronize from time to time, leading to increased synchronization activity, which further complicates DVFS performance prediction.

An additional complication is that managed applications issue bursts of store operations. These occur for two reasons: due to garbage collection activities that move memory around, and due to the zero-initialization upon fresh allocation that many managed

- *S. Akram is with Ghent University, Belgium. E-mail: Shoaib.Akram@UGent.be.*
- *J. B. Sartor is with Ghent University and Vrije Universiteit Brussel, Belgium. E-mail: Jennifer.Sartor@UGent.be.*
- *L. Eeckhout is with Ghent University, Belgium. E-mail: Lieven.Eeckhout@UGent.be.*

*This article is an extension of 'DVFS Performance Prediction for Managed Multithreaded Applications', by Shoaib Akram, Jennifer B. Sartor and Lieven Eeckhout, presented at the 2016 International Symposium on Performance Analysis of Systems and Software (ISPASS). The new contributions include:*
*(1) An exploration of the scalability of our proposed DVFS predictor.*
*(2) An extension to our energy manager to still achieve energy savings with a coarser frequency step setting.*
*(3) The results of a sensitivity study of different parameters of the energy manager including quantum and hold-off.*
*(4) A new case study using our energy manager to optimize total system energy.*
*(5) New discussions, clarifications, and revisions of many parts of the conference version of the paper.*

languages, such as Java, require to provide memory safety. Current predictors ignore store operations assuming they are not on the critical path. We find that ignoring store operations leads to incorrect DVFS performance prediction for managed applications.

In this paper, we propose DEP+BURST, a novel DVFS performance predictor for managed multithreaded applications. DEP+BURST consists of two key components, DEP and BURST. DEP decomposes the execution time of a multithreaded application into epochs based on the synchronization activity of both the application and service threads. We predict the duration of epochs at a different frequency, and aggregate the predicted epochs while taking into account inter-thread dependencies to predict the total execution time at the target frequency. A crucial component of DEP is its ability to predict critical threads across epochs. BURST identifies store operations that are on the application's critical path, and predicts their impact on performance across different frequency settings. Based on a run at the baseline frequency of 1 GHz, DEP+BURST achieves an average absolute error of 6% when predicting performance at a 4 GHz target frequency, for a set of multithreaded Java applications from the DaCapo suite [5]. DEP+BURST's error is a significant decrease from the 27% error achieved by M+CRIT, a multithreaded extension of the state-of-the-art CRIT [35] performance predictor.

We integrate DEP+BURST into an energy management framework for managed applications. We first use the energy manager to reduce the processor's energy consumption by tolerating a slowdown in performance compared to running at the highest frequency. On average, for a user-specified slowdown threshold of 5% and 10%, our energy management framework reduces energy consumption by 13% and 19%, respectively, for a set of memory-intensive applications. In a second use case, the energy manager optimizes for total system energy consumption, including that of the processor plus DRAM. On average, our energy manager reduces total energy consumption by 15.6% through dynamically responding to application's phase behavior to pick a frequency per time quantum that lowers total system energy. For each use case, we compare against an oracle scheme that explores all possible frequency settings, and for a number of benchmarks, we outperform this scheme by exploiting dynamic phase behavior. We make the following contributions:

1) We identify that inter-thread dependencies and store bursts need to be taken into account to have an accurate performance predictor for multithreaded managed applications.
2) We introduce a performance predictor, DEP+BURST, that significantly lowers the error of accurately predicting performance when scaling the frequency.
3) We perform two case studies with an energy manager that 1) targets reducing the processor's energy consumption by slowing down a program not more than a user-specified slowdown threshold, and 2) optimizes total system energy, taking memory's energy consumption into account.
4) We perform experiments exploring the scalability of DEP+BURST, the ramifications of having a coarser frequency step setting, and the execution time overhead of running the proposed DVFS predictor.

Having an accurate performance predictor for DVFS is crucial to maintaining good performance, especially for multithreaded managed applications, while reducing energy consumption.

## 2 Background and Motivation

In this section, we first provide background on the state-of-the-art predictor for sequential applications. We then describe the challenges introduced by multithreading and managed languages. Finally, we discuss naive extensions to the state-of-the-art predictor to predict the performance of multithreaded managed applications.

### 2.1 DVFS Performance Predictors for Sequential Applications

The impact of changing the frequency on application performance is easily understood by dividing execution time into 'scaling' and 'non-scaling' components. The scaling component scales in proportion to frequency; the non-scaling component remains constant when changing frequency. This simple division of execution time into scaling and non-scaling components works because changing the processor's frequency does not alter DRAM service time, whereas an increase or decrease in processor frequency has a proportional impact on the rate at which instructions execute in the core pipeline. The key challenge for accurately predicting the performance impact of DVFS is due to the out-of-order nature of modern processor pipelines in which memory requests are resolved while executing and retiring other instructions. Three DVFS performance predictors have been proposed over the past few years for sequential applications, with progressively improved accuracy. We now briefly discuss these three predictors.

**Stall Time.** The simplest, and least accurate, of the three models is the *stall time* model [19], [29], which estimates the non-scaling component by measuring the time the pipeline is unable to commit instructions. The non-scaling component is underestimated because it does not account for the fact that instructions may commit underneath a memory access. The simplicity of this model implies that it is easy to deploy on real hardware using existing hardware performance counters.

**Leading Loads.** Proposed by three different groups around the same time [19], [29], [38], the *leading loads* model computes the non-scaling component by accounting for the full latency of the leading load miss in a burst of load misses. Modern out-of-order pipelines are able to exploit memory-level parallelism and handle independent long-latency load misses simultaneously. The leading loads model assumes that each long-latency load miss incurs roughly the same latency, and hence, for a cluster of long-latency load misses, the miss latency of the leading load is a good approximation for the non-scaling component. Recent work shows that the leading loads model can be deployed on real hardware by using performance counters available on modern processors [43].

**CRIT.** A fundamental limitation of the leading loads model is that it does not take into account that long-latency load misses may incur variable latency, for a variety of reasons, including memory scheduling, bank conflicts, open page policy, etc. This leads to prediction inaccuracy for the leading loads model, which is overcome by CRIT, the state-of-the-art DVFS predictor proposed by Miftakhutdinov et al. [35]. CRIT identifies the critical path through a cluster of long-latency load misses to model a realistic, variable-latency memory system. CRIT includes an algorithm to identify dependent long-latency load misses and uses their accumulated latency as an approximation for the non-scaling component. We will use CRIT as our DVFS performance predictor for an individual thread. Note that as of today, no implementation of CRIT exists on real hardware.

### 2.2 Challenges in DVFS Performance Prediction for Managed Multithreaded Applications

There are three major challenges for predicting the performance impact of DVFS for multithreaded managed applications.

**Inter-thread dependencies due to multithreading.** To protect shared variables, different threads of a multithreaded application use synchronization primitives. Common examples of synchronization include critical sections and barriers. Synchronization leads to inter-thread dependencies. No thread is allowed to continue past the barrier until all threads reach the barrier. The slowest thread determines the barrier execution time at the target frequency. With a critical section, the progress of a thread waiting for a lock will depend on how fast the thread currently holding the lock is progressing at the target frequency. Scaling the frequency of one thread in a multithreaded application impacts the execution of other dependent threads, affecting overall performance in a non-trivial way.

**Interaction between application and service threads.** A managed language execution engine, such as the Java Virtual Machine (JVM), consists of application threads and service threads. The most important service threads include garbage collection and just-in-time compilation. Application and service threads interact with each other. For instance, a stop-the-world garbage collector suspends the application for a short duration to traverse the heap, and reclaim memory being used by objects that are no longer referenced. To estimate the total execution time at a different frequency, a DVFS predictor thus needs to take the interaction between application threads and service threads into account.

**Store bursts.** To provide memory safety, the Java programming language requires that a region of memory is zero-initialized upon fresh allocation. The process of zero-initialization leads to a burst of store operations that fill up the processor's pipeline. Another source of store bursts is the copying of objects during garbage collection. Ignoring store operations completely, as prior DVFS predictors do, leads to incorrect predictions for managed language workloads.

### 2.3 Straightforward Extensions of Prior Work

Before describing our new predictor in the next section, we first present two straightforward extensions of prior work to deal with multiple threads and, in the second case, service threads. We will quantitatively compare DEP+BURST against these naive extensions in the results section, and detail why these models are insufficient.

**M+CRIT.** We call the first predictor M+CRIT (short for multithreaded CRIT), which is generally applicable to any multithreaded application. M+CRIT uses the intuition that the execution time of a multithreaded application is determined by the critical (slowest) thread. We first use CRIT to identify each thread's scaling and non-scaling components at the base frequency. We then predict each thread's execution time at the target frequency. The thread with the longest predicted execution time is the critical thread. The execution time of the critical thread is also the total execution time of the application at the target frequency.

**COOP.** We term the second predictor COOP (short for cooperative), which is specific to Java applications. A typical Java application with a stop-the-world garbage collector goes through an 'application' phase, followed by a 'collector' phase. COOP intercepts the communication between the application and collector threads using signals from the JVM. Using these signals, COOP is able to distinguish application and collector phases. Once these individual phases are identified, COOP then uses M+CRIT to predict the execution time of the individual phases and aggregates the predictions to obtain a prediction for the total execution time.

## 3 The DEP+BURST Model

We now discuss our new DVFS performance predictor for managed multithreaded applications in detail.

### 3.1 Overview

Our proposed DVFS predictor estimates the performance of a managed multithreaded application in two steps. In the first step, the predictor decomposes execution time into epochs based on synchronization activity in the application to account for inter-thread dependencies and the interaction between the application and service threads. In the second step, the predictor estimates the execution time of each active thread at a target frequency, taking into account which thread is critical and adjusting for dependencies with other epochs. Our model, which we call DEP, estimates the epoch execution time at the target frequency, and aggregates epochs to predict the total application execution time. To additionally take into account store bursts, we modify the second step to adjust the calculation of the scaling and non-scaling portions per thread within an epoch. When accounting for store bursts, we call our full model DEP+BURST. In the following sections, we first describe DEP, and then BURST.

### 3.2 Identifying Synchronization Epochs

First, we describe how DEP decomposes execution time into *synchronization epochs*. A synchronization epoch consists of a variable number of threads running in parallel. Two events mark the beginning of a new synchronization epoch: a thread is scheduled out by the OS and put to sleep, or a sleeping (or newly spawned) thread is scheduled onto a core. In multithreaded applications, threads typically go to sleep when access to a critical section is not available, or threads sleep while waiting at a barrier for other threads to join.

We identify synchronization epochs by intercepting the futex_wait and futex_wake system calls. Multithreading libraries such as pthreads use futexes, or fast kernel space mutexes [21] for handling locking. In the uncontended case, the application acquires the lock using atomic instructions without entering the kernel. Only in the contended case does the application invoke the kernel spin locks using the futex interface. Intercepting futex calls incurs limited overhead (less than 1%) [17].

To understand why futex-based decomposition is necessary, consider the example of a multithreaded execution in Figure 1(a). Two threads t0 and t1 from the same application are running in parallel. When t1 attempts to enter a critical section, t0 is already executing the critical section, which leads to t1 being scheduled out and made to wait for t0 to finish executing the critical section. When t0 is done executing the critical section, t1 is woken up.

An intuitive way to estimate the execution time of the example in Figure 1(a) is to first identify the non-scaling component of t0 and t1 when running at the base frequency, and subtract those from the total execution time to obtain the scaling components. This is what M+CRIT does. Using these per-thread components, it is straightforward to estimate the execution time of individual threads at the target frequency (see Section 2). Then the estimated execution time of the slowest thread serves as an estimate of total execution time. However, this leads to an incorrect estimation of execution time. Non-scaling component of execution time is

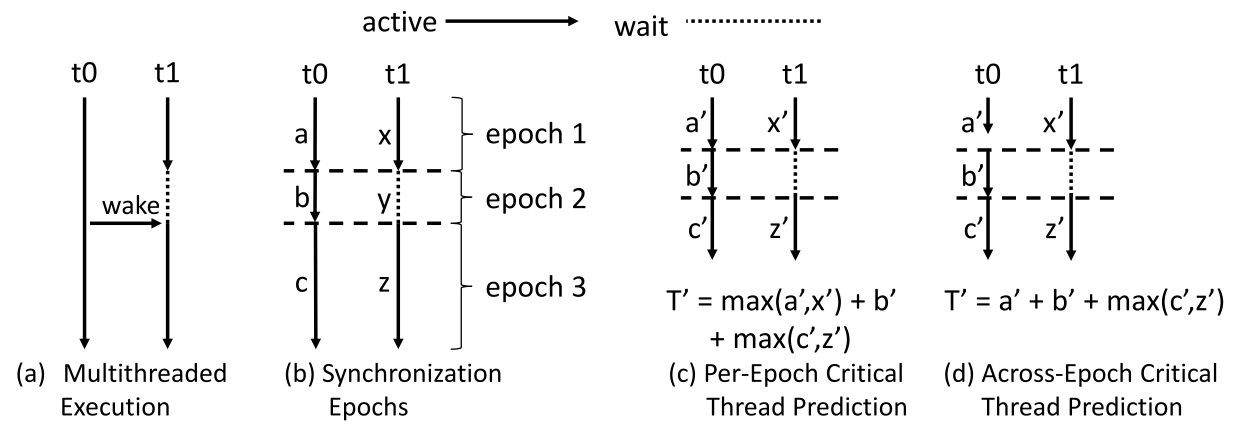

active
wait
t0
t1
wake
a
x
b
y
c
z
epoch 1
epoch 2
epoch 3
a'
x'
b'
c'
z'
T' = max(a',x') + b' + max(c',z')
T' = a' + b' + max(c',z')
(a) Multithreaded Execution
(b) Synchronization Epochs
(c) Per-Epoch Critical Thread Prediction
(d) Across-Epoch Critical Thread Prediction

Fig. 1: Showing how DEP breaks up a multithreaded application (a) into synchronization epochs (b) while running at the base frequency. DEP then estimates per-thread epoch durations at the target frequency, calculates the critical thread per epoch (c), and accounts for changes in the critical thread across epochs (d).

actively accumulated in a counter only during the time a thread is active. During the time t1 is waiting, its non-scaling component depends on the activity taking place in the core where the thread holding the lock is running (t0). In the simple approach, the time t1 is waiting gets incorrectly attributed to the scaling component. Accurately estimating the execution time requires taking the dependency between t0 and t1 into account.

Figure 1(b) shows how our predictor decomposes the execution shown in Figure 1(a) into three epochs. a and x represent the duration of the first epoch, for threads t0 and t1, respectively. While these values are equal at the base frequency, we label them differently per thread because these values could be different when estimating time at the target frequency. b represents the duration of time that t0 is active during the second epoch when running at the base frequency. Similarly, c and z represent the duration of the third epoch. By decomposing execution time into epochs, DEP is able to model the dependency between t0 and t1 by analyzing b and predicting the new duration of b at a different frequency, which affects when both threads begin the third epoch at the new frequency.

It should be noted that the synchronization incurred by service threads, namely between garbage collection threads, *and* the coordination between application and garbage collection threads is also communicated through futex calls. Therefore, by breaking down execution into epochs, we not only model the inter-thread dependencies between the application threads, but also account for the extra interactions between managed language application and service threads.

## 3.3 Predicting Performance at a Target Frequency

We now discuss how DEP estimates the duration of an epoch at a target frequency. During an epoch, DEP uses CRIT to accumulate the non-scaling component of each active thread in a counter. At the end of an epoch, both the scaling and non-scaling components are known. This provides DEP with a prediction of the duration of each thread at the target frequency. This is shown in Figure 1(c) and Figure 1(d) where a', b' and c' represent the estimated duration of t0's first, second and third epoch, respectively, at the target frequency. Similarly, x' and z' is the estimated duration of t1's first and third epoch at the target frequency. The next goal is to predict the execution time of an epoch from these individual estimates of all the active threads.

**Per-epoch Critical Thread Prediction (CTP).** An intuitive approach is to take the duration of the thread that runs the longest in the epoch, i.e., the critical thread, as the duration of the epoch at the target frequency. This approach is shown in Figure 1(c). This approach is simple to implement and does not require any bookkeeping across epochs. This technique does model the dependency between threads t0 and t1 in our running example and predicts when the third epoch would begin for both threads in the target frequency. However, using per-epoch critical thread prediction does not result in an accurate estimate of total execution time.

**Across-epoch Critical Thread Prediction (CTP).** We add across-epoch critical thread prediction to our DEP model to make it more accurate. This is shown in Figure 1(d). In the figure, a' is estimated to be shorter than x'. But if x' is taken as the duration of the first epoch, this leads to an incorrect estimation of the duration of the three epochs i.e., x' + b' + max(c',z'). The correct duration is a' + b' + max(c',z'), because thread t0 would just continue running after a' time units. In effect, part of x' gets overlapped with b' at the target frequency. Therefore, during each epoch, we need to store extra state to be able to identify the identity and duration of the critical thread to take that into account across epochs. Following the current example, we store the delta, x' - a', in a separate counter at the end of the first epoch. We also speculatively estimate the total execution time at the end of first epoch to be x'. In the second epoch, we subtract the delta-counter from b'. This way, at the end of the second epoch, we correctly estimate the total execution time to be a' + b'.

**Algorithm.** Our algorithm for performing across-epoch critical thread prediction is shown in Algorithm 1. First, we introduce the terminology used in Algorithm 1. $\alpha_t$ represents the estimated duration of a thread t at the target frequency. $\delta_t$ is the difference between the estimated duration of thread t and the estimated duration of the critical thread; $\delta_t$ of the critical thread is zero. The first step in Algorithm 1 is to compute the estimated duration, $\alpha_t$, of each thread using CRIT (line 2). Next, we calculate the 'effective' execution time ($e_t$) of each thread by subtracting $\delta_t$ from $\alpha_t$ (line 3). The thread with the largest $e_t$ is the critical thread, and the corresponding $e_t$ is the duration of the epoch ($I'$) (line 5). Note that $\delta_t$ is accumulated across epochs, with a term representing the difference between $I'$ and $\alpha_t$ added during each epoch until the thread stalls (line 7). We reset $\delta_t$ of a stalling thread

**ALGORITHM 1:** Algorithm for across-epoch CTP.

```
input : A synchronization epoch S (I time units)
input : Initial delta-counters ($\delta_t$) of all threads
input : Identity of the stalled thread if any (stall_tid)
output: Estimated duration (I' time units) of S at target frequency
1 for each active thread t in S do
2     $\alpha_t$ = computeEstimatedTimeUsingCRIT()
3     $e_t = \alpha_t - \delta_t$
4 end
5 I' = Largest $e_t$
6 for each active thread t in S do
7     $\delta_t = (I' - \alpha_t) + \delta_t$
8 end
9 $\delta_{stall-tid} = 0$
```

to zero (line 9).

## 3.4 Modeling Store Bursts

Store bursts occur more frequently in managed language workloads than in native applications. In Java in particular, store bursts originate from two main sources: (1) zero-initialization to provide memory safety, and (2) copying of objects during garbage collection. A DVFS model for Java applications should incorporate the impact of store bursts.

CRIT assumes that store instructions are not on the application's critical path. This is true for a few isolated store requests that miss in the L1 cache because the store queue provides modern processors with the ability to execute loads in the presence of outstanding stores (through load bypassing and store-to-load-forwarding). Furthermore, it contains committed stores until they are retired by the memory hierarchy, freeing up space in the ROB or active list. Normally, the work done underneath a store miss scales with frequency. However, a fully-occupied store queue stalls the processor pipeline. Store bursts fill up the store queue before eventually stalling the pipeline.

In typical out-of-order pipelines, an entry is allocated in the ROB and the store queue at the time the store instruction is issued. When a store commits from the head of the ROB, the entry is no longer maintained in the ROB, but the entry is maintained in the store queue until the outstanding request is finally retired. Commit stalls when the store queue is full and the next instruction to commit is a store.

To account for store bursts, we accumulate the amount of time the store queue is full in a counter when running at the base frequency. For each active thread during an epoch, we add the counter's contents to the non-scaling component measured by CRIT. When modeling the impact of store bursts, we add BURST next to the model name. Thus, our proposed model that takes both inter-thread dependencies and store bursts into account is called DEP+BURST.

## 3.5 Implementation Details

Now, we discuss implementation issues when porting DEP+BURST to real hardware. First, the OS is the best place to identify synchronization epochs, for instance, as a kernel module. The OS is aware of thread creation, deletion, and other events regarding thread scheduling including the futex_wait and futex_wake system calls.

Multiple threads time-sharing a single core is a common practice to consolidate resources. In such a case, the OS periodically schedules out the currently executing thread out of the core, and schedules one of the waiting threads in. Whenever that happens, we start a new epoch. As a result, time-multiplexing cores among threads is seamlessly handled by DEP.

We require extra counters to implement DEP+BURST on real hardware. We use CRIT [35] within an epoch to divide a thread's execution into scaling and non-scaling portions, so our model requires the same bookkeeping information as CRIT.

Tracking store bursts requires simple additional logic in the store queue that generates a signal once all its entries are occupied. The performance counter hardware monitors this signal to account for the time the store queue is full.

To account for critical threads across epochs, we require one counter per thread. This counter can be maintained in software inside the kernel module that intercepts the futex calls.

| Benchmark | Type [M/C] | Heap size [MB] | Execution time (ms) | GC time (ms) |
|---|---|---|---|---|
| xalan | M | 108 | 1,400 | 270 |
| pmd | M | 98 | 1,345 | 230 |
| pmd.scale | M | 98 | 500 | 80 |
| lusearch | M | 68 | 2,600 | 285 |
| lusearch.fix | C | 68 | 1,249 | 42 |
| avrora | C | 98 | 1,782 | 5 |
| sunflow | C | 108 | 4,900 | 82 |

TABLE 1: Our benchmarks from the DaCapo suite, including a classification of their type, heap size, execution time and GC time at 1 GHz. M represents a memory-intensive benchmark, and C represents a compute-intensive benchmark.

## 4 Experimental Methodology

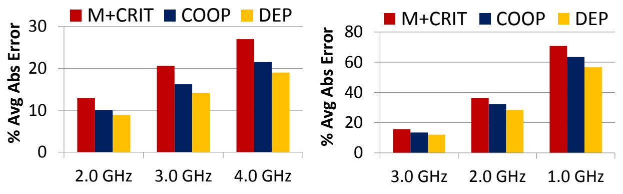
Before evaluating the accuracy of DEP+BURST, we first describe our experimental setup.

**Simulator.** We use Sniper [10] version 6.0, a parallel, high-speed and cycle-level x86 simulator for multicore systems; we use the most detailed cycle-level core model available. Sniper was further extended [40] to run a managed language runtime environment including dynamic compilation, and emulation of frequently used system calls.

**Java Virtual Machine and benchmarks.** We use Jikes RVM 3.1.2 [3] to evaluate the seven multithreaded Java benchmarks from the DaCapo suite [5] that we can get to work on our infrastructure. We use five benchmarks from the DaCapo-9.12-bach benchmark suite (avrora, lusearch, pmd, sunflow, xalan). We also use an updated version of lusearch, called lusearch-fix (described in [47]), that eliminates needless allocation. Finally, we use an updated version of pmd, called pmd-scale (described in [17]) that eliminates the scaling bottleneck due to a large input file. All benchmarks we use in this work are multithreaded. The avrora benchmark uses a fixed number (six) of application threads, but has limited parallelism [17]. For the remaining multithreaded benchmarks, we evaluate using four application threads. Table 1 lists our benchmarks, a classification of whether they are memory or compute-intensive, the heap size we use in our experiments (reflecting moderate, reasonable heap pressure [40]), and their running time when using Sniper with each core running at 1 GHz. We classify the benchmarks based on the intensity of garbage collection. An application that spends more than 10% of its execution time in garbage collection is considered a memory-intensive benchmark. lusearch.fix, avrora, and sunflow are compute-intensive, and the remaining five benchmarks are memory-intensive.

| Component | Parameters |
|---|---|
| Processor | 4 cores, 1.0 GHz to 4.0 GHz<br>4-issue, out-of-order, 128-entry ROB<br>Outstanding loads/stores = 48/32 |
| Cache hierarchy | L1-I/L1-D/L2, Shared L3 (1.5 GHz)<br>Capacity: 32 KB / 32 KB / 256 KB / 4 MB<br>Latency : 2 / 2 / 11 / 40 cycles<br>Set-associativity: 4 / 8 / 16<br>64 B lines, LRU replacement |
| Coherence protocol | MESI |
| DRAM | FR-FCFS, 12 GB/s, 45 ns latency |
| DVFS states<br>(GHz,Vdd) | (1, 0.737); (1.5, 0.791); (2, 0.845);<br>(2.5, 0.899); (3, 0.958); (3.5, 1.012);<br>(4, 1.07) |

TABLE 2: Simulated system parameters.



(a) Low-to-high prediction (b) High-to-low prediction

Fig. 2: Showing the average absolute prediction error of M+CRIT, COOP and DEP: (a) prediction at higher frequency from a baseline of 1 GHz, and (b) prediction at lower frequency from a baseline of 4 GHz. *DEP outperforms all other predictors both when predicting from 1 GHz to 4 GHz and vice-versa.*

We follow common practice in Java performance evaluation by using replay compilation [6], [22] to eliminate non-determinism introduced by the just-in-time compiler. During profiling runs, the optimization level of each method is recorded for the run with the lowest execution time. The JIT compiler then uses this optimization plan in our measured runs, optimizing to the correct level the first time it sees each method [6], [26]. To eliminate the perturbation of the compiler, we measure results during the second invocation, which represents application steady-state behavior. We run each application four times, and report averages in the graphs. We use the default stop-the-world generational Immix garbage collector in JikesRVM [7] along with the default nursery settings.

**Processor architecture.** We consider a quad-core processor configured after the Intel Haswell processor i7-4770K, see Table 2. Each core is a superscalar out-of-order core with private L1 and L2 caches, while sharing the L3 cache. We vary the cores' frequency between 1 and 4 GHz.

**Power and energy modeling.** We use McPAT version 1.0 [32] for modeling power consumed by the processor. For DVFS support, we use the Sniper/McPAT integration described in [23] while considering a 22 nm technology node. We use a frequency step setting of 125 MHz when dynamically adjusting the frequency to save energy (Section 6). We use the voltage and frequency settings for a 22 nm technology node, closely following Intel's i7-4770K (Haswell) [13]; see Table 2 for a subset of settings[1]. When reporting power numbers, we include both static and dynamic power. We model the DVFS transition latency as a fixed cost of 2 $\mu$s [25].

## 5 Model Evaluation

We now evaluate the accuracy of DEP+BURST. We first compare the accuracy of DEP against M+CRIT and COOP to understand the impact of taking inter-thread dependencies into account. We then evaluate DEP with and without BURST, teasing apart the contribution of taking store bursts into account.

### 5.1 Prediction Accuracy

Evaluating the accuracy of a DVFS performance predictor is done as follows. We run the application at both the baseline and target frequency. We predict the execution time at the target frequency based on the run at the baseline frequency, and we compare the predicted execution time against the measured execution time. We quantify prediction accuracy as the relative prediction error (estimated - actual) / actual. A negative error thus implies an underestimation of the execution time or a performance overestimation. The reverse applies for a positive error.

Evaluating a DVFS performance predictor requires choosing a baseline and target frequency. When used as part of an energy management framework — as we will explore in Section 6 — it is important that we are able to accurately predict performance both at higher and lower frequencies. We hence consider two scenarios: one in which we consider a low base frequency and predict performance at higher frequencies, and one in which we consider a high base frequency and predict performance at lower frequencies. Figure 2(a) shows the average absolute error of M+CRIT, COOP, and DEP for three target frequencies when the base frequency is set at 1 GHz, i.e., predicting performance at a higher frequency than the baseline frequency[2]. Figure 2(b) shows similar data for target frequencies smaller than the base frequency set to 4 GHz.

M+CRIT has the worst prediction error of all models. The average absolute error is 27% when predicting from 1 GHz to 4 GHz, and 70% when predicting from 4 GHz to 1 GHz. Clearly, not taking into account synchronization, inter-thread dependencies and store bursts leads to highly inaccurate DVFS performance prediction for managed multithreaded applications.

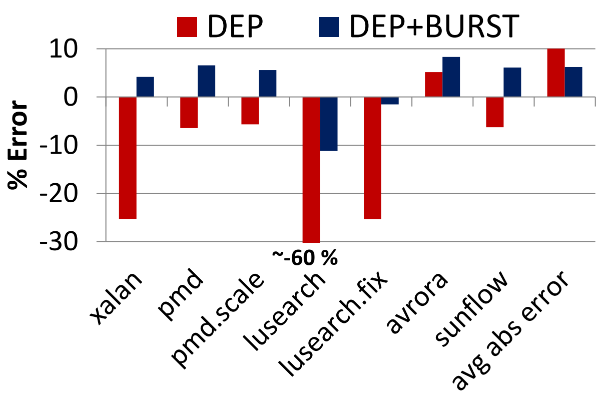
Taking into account the interaction of application and managed language service threads, as COOP does, slightly improves accuracy over M+CRIT. However, the prediction error is still significant with average absolute prediction errors for COOP of 22% and 63% for the base 1 and 4 GHz scenarios, respectively.

Taking all synchronization activity into account, as DEP does, further improves accuracy, with an average absolute error of 19% and 57% for the base 1 and 4 GHz scenarios, respectively. The conclusion from this result is that managed multithreaded applications require accurate modeling of inter-thread dependencies both through coarse-grained synchronization between application phases and garbage collection phases, as well as through fine-grained synchronization between application threads and between garbage collection threads. Unfortunately, although the prediction error is decreased compared to M+CRIT and COOP, DEP's error is still high.

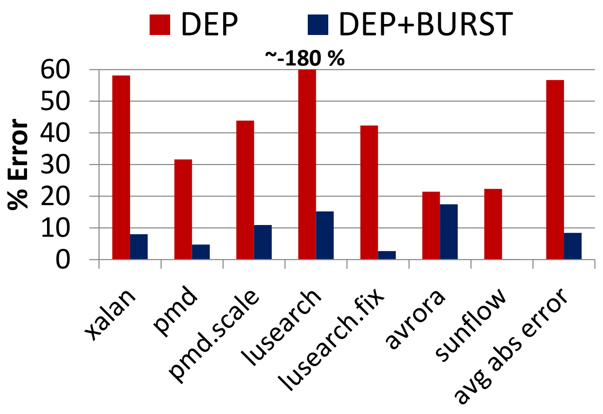
Next, we compare the prediction error of DEP to DEP+BURST in Figure 3(a) which shows the errors for all benchmarks for a target frequency of 4 GHz when the base frequency is set at 1 GHz. Figure 3(b) shows similar data for a target frequency of 1 GHz when the base frequency is 4 GHz. Modeling store bursts brings the error down substantially, especially for the memory-intensive benchmarks. DEP+BURST has an average absolute error of 6% when predicting from 1 GHz to 4 GHz, and an average

1. The remaining settings can be found using the linear relationship between core voltage (v) and frequency (f), v = 0.11 * f + 0.63 [13].

2. An earlier version of this paper includes per-benchmark results [2].



(a) Low-to-high prediction



(b) High-to-low prediction

Fig. 3: Per-benchmark prediction errors for DEP and DEP+BURST: (a) prediction at 4 GHz from a baseline of 1 GHz, and (b) prediction at 1 GHz from a baseline of 4 GHz. *DEP+BURST substantially outperforms DEP both when predicting from 1 GHz to 4 GHz and vice-versa.*

absolute error of 8% when predicting from 4 GHz to 1 GHz. Modeling both synchronization and inter-thread dependencies as well as store bursts is critical for DVFS performance prediction of managed multithreaded applications.

Prediction errors tend to increase for target frequencies that are 'further away' from the base frequency, due to accumulating errors, which is especially noticeable for memory-intensive applications. Further, when predicting the execution time in the high-to-low scenario, an error in incorrectly estimating the scaling component multiplies as the target frequency increases. This leads to increased inaccuracy in identifying the critical thread in an epoch. When predicting low-to-high, the scaling component is divided by a factor, making the error less prominent.

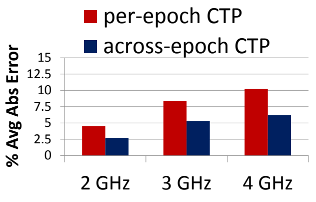
**Explaining lusearch and avrora.** From the results in Figure 3, we note a higher estimation error for two benchmarks: avrora and lusearch. Our analysis indicates that each of the two benchmarks stresses a different component of DEP+BURST. avrora has the largest number of epochs among all of our benchmarks, pointing to a large number of inter-thread dependencies, thus stressing the DEP component. On the other hand, lusearch allocates the most memory. Its error is high because it is overly sensitive to the approximation we make to model store bursts.

## 5.2 Per-Epoch vs. Across-Epochs CTP

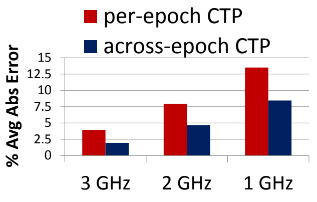
As argued in Section 3, it is important to accurately predict the critical thread at each point during the execution. We described two approaches to this problem, namely per-epoch critical thread prediction (CTP) and across-epoch CTP. We now quantify the importance of across-epoch CTP. Figure 4 reports the prediction error for DEP+BURST with across-epoch CTP versus per-epoch CTP. Across-epoch CTP brings down the average absolute error by a significant margin compared to per-epoch CTP: by 4% (from 10% to 6% average absolute error) at 4 GHz with a 1 GHz base frequency, and by 6% (from 14% to 8% average absolute error) at 1 GHz with a 4 GHz base frequency. This result confirms that being able to accurately predict the critical thread at all points during the execution time, and carry this dependence across epochs, is a key component of DEP+BURST.

## 5.3 Scalability

We have shown the accuracy of DEP+BURST with four application and two GC threads. We now experiment with different thread



(a) 1.0 GHz to 4.0 GHz



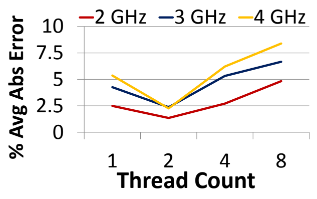
(b) 4.0 GHz to 1.0 GHz

Fig. 4: Comparing per-epoch versus across-epoch critical thread prediction. *Detecting critical threads across epochs leads to a more accurate predictor for multithreaded managed applications.*

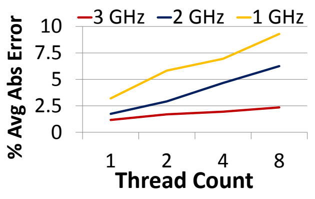
counts to explore our predictor's scalability. Increasing the number of application threads stresses the predictor in different ways. More application threads lead to more inter-thread dependencies. Increasing the thread count also increases the rate that store bursts are issued, because there is also more memory allocation. This is because thread-local storage increases the total amount of memory allocated when running benchmarks with different numbers of threads. Prior work also shows that the amount of work that GC performs increases with more application threads [17].

To understand the scaling behavior of our proposed model, we show the accuracy of DEP+BURST with 1, 2, 4 and 8 application threads. Because of the presence of service threads such as the garbage collector, managed environments are multithreaded even with one application thread. When running the benchmarks with one application thread, we use a single garbage collector thread. We use two garbage collector threads for experiments with more than one application thread. Prior work by Du Bois et al. [17] reports that Jikes' generational Immix garbage collector does not scale beyond two threads. We use a single core per application thread in our experiments. We set the last-level cache to have 1 MB/core and the memory bandwidth is set to 3 GB/s/core. Our modeled processors reflect many commercial designs in the market today.

Figure 5 shows the average absolute error of our predictor with different thread counts. The average error with one application thread is 5.4% when predicting from 1 GHz to the highest target frequency of 4 GHz, and 3.2% when predicting from 4 GHz to 1 GHz. Thus, our DEP+BURST predictor is also accurate

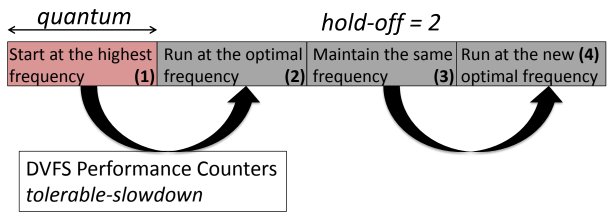


(a) 1.0 GHz to 4.0 GHz



(b) 4.0 GHz to 1.0 GHz

Fig. 5: The accuracy of DEP+BURST for different thread counts. *DEP+BURST's error increases only slightly as the number of threads is increased from one to eight.*



quantum
hold-off = 2
Start at the highest frequency (1)
Run at the optimal frequency (2)
Maintain the same frequency (3)
Run at the new (4) optimal frequency
DVFS Performance Counters
tolerable-slowdown

Fig. 6: Example illustrating the energy manager using DVFS performance prediction. The input parameters of the energy manager are shown in italics.

for single-threaded managed applications. As the thread count increases, the average absolute error goes up. However, for up to eight threads, the increase is not dramatic. When predicting from 1 GHz to higher frequencies, the average error with eight application threads is below 10% for any target frequency. The average error with more than one application thread is slightly higher when predicting from 4 GHz to 1 GHz (9.3% with eight threads). Note that the frequency range that we explore, 1 GHz to 4 GHz, represents a very wide frequency spectrum. Memory behavior is likely to change across such a wide frequency spectrum, making it harder to predict performance accurately, especially as the number of threads is increased.

## 6 Case Studies

Having described and evaluated the DEP+BURST DVFS performance predictor, we now use it in two case studies involving an energy manager. In the first case study, the energy manager leverages a performance predictor to reduce the processor's energy consumption without slowing down the application more than a user-specified threshold. In the second case study, the energy manager uses analytical performance and energy models to optimize the full system energy consumed by an application.

### 6.1 Case Study 1: Energy Minimization under Performance Constraints

It is well-known that it is possible to reduce the processor's energy consumption by lowering the frequency. The intuition is that lowering the frequency reduces power consumption, leading to a more energy-efficient execution. Lowering the frequency reduces energy consumption as long as the reduction in power consumption is not offset by an increase in execution time. This is typically the case for memory-intensive applications for which lowering the frequency incurs a small performance degradation. For compute-intensive applications on the other hand, the reduction in power consumption may be offset by an increase in execution time, leading to a (close to) net energy-neutral operation. In other words, different applications exhibit different sensitivities to scaling the processor's frequency. Moreover, compute- and memory-intensive phases may occur within a single application; this is especially the case for managed language workloads for which garbage collection is typically memory-intensive [9], [37]. Hence, this calls for an energy management approach that dynamically determines when and to what extent to scale the frequency to minimize energy consumption while not exceeding a user-specified slack in performance.

#### 6.1.1 *Energy Manager*

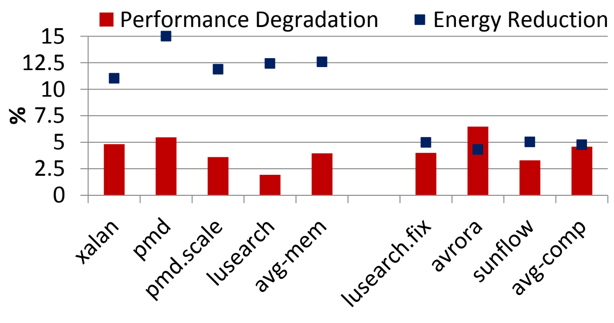
To demonstrate the importance of having an accurate DVFS performance predictor for multithreaded managed applications, we design an energy manager that minimizes energy consumption while guaranteeing performance within a user-specified threshold compared to running at the highest frequency. The high-level design is shown in Figure 6. The figure shows how the manager works for the first four intervals of the application. We always start the application at the highest frequency (4 GHz for our modeled processor). During this interval, the performance predictor reads the DVFS-related performance counters as described in Section 3. At the end of the first interval, the manager estimates performance at all of the DVFS states. The tolerable-slowdown is a user-specified parameter that the manager uses to identify all of the DVFS states that satisfy the performance constraint, i.e., performance is slowed down by no more than tolerable-slowdown, as a percentage compared to running at the highest frequency. Of all the states that satisfy the performance constraint, the manager then chooses the state with the minimum energy consumption (lowest frequency) for the next quantum. The hold-off parameter represents the number of intervals to wait before changing the frequency again. In the example shown in the figure, hold-off is set to two. Therefore, the third interval also runs at the same frequency as the second interval. In case the application has no phase behavior, using a large hold-off prevents needless profiling. The scheduling quantum is also an adjustable parameter, and is set to 5 ms in our experiments. We set the hold-off parameter to one unless otherwise specified.

The key idea we use to guarantee that the application does not experience a slowdown more than the specified threshold is that, if each interval experiences a slowdown of x%, then the entire application experiences a slowdown of x% compared to always running the application at the highest frequency. To fulfill this requirement during each interval, we need to estimate the slowdown that the application experiences compared to running at the highest frequency, even when running at a slower frequency. We solve this problem in two steps. The energy manager first estimates the execution time at the highest frequency, before predicting execution time at the target frequency in the second step and its relative slowdown compared to running at the highest frequency. The manager finally chooses the minimum frequency setting that does not slow down the interval more than the user-specified threshold.

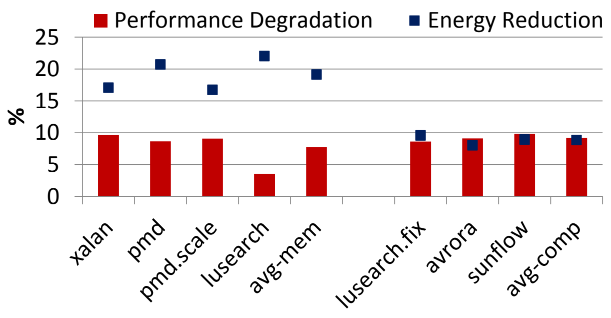
In the following sections, we explore the opportunity to reduce energy consumption with our proposed energy manager in detail.

#### 6.1.2 *Evaluation*

Figure 7 reports the slowdown experienced by each benchmark and the corresponding reduction in energy consumption for user-specified slowdown thresholds of 5% and 10%. We observe substantial reductions in energy consumption for the memory-intensive benchmarks, by 13% on average (and up to 15%) for the 5% threshold, and by 19% on average (and up to 22%) for the



(a) Tolerable Slowdown = 5%



(b) Tolerable Slowdown = 10%

Fig. 7: Per-benchmark reductions in energy consumption using DEP+BURST in our energy manager for a slowdown threshold of (a) 5% and (b) 10%. Memory-intensive benchmarks are to the left while compute-intensive are to the right. *Using the DEP+BURST predictor as part of our energy manager leads to a significant reduction in energy consumption for the memory-intensive benchmarks with only a slight performance degradation.*

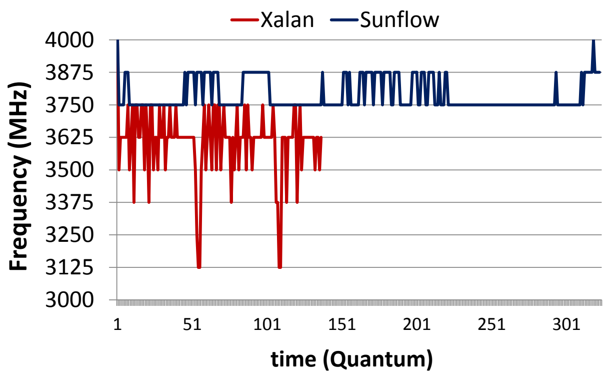


Fig. 8: Per-quantum frequency settings chosen by the energy manager for xalan and sunflow for a slowdown threshold of 5%. *There is a larger variation in the processor's frequency during the execution of the memory-intensive benchmarks, compared to the compute-intensive benchmarks.*

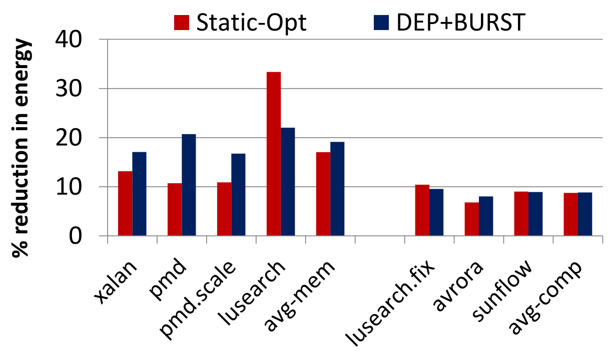


Fig. 9: Reduction in energy consumption achieved by our energy manager compared to the static optimal (Static-Opt) for a slowdown threshold of 10%. *For six out of seven benchmarks, our energy manager reduces the energy consumption about the same as, or more than, that of Static-Opt.*

10% threshold. As expected, the reduction in energy consumption is not as significant for the compute-intensive workloads.

It is interesting to note that the obtained performance is close to the user-specified performance target, i.e., the execution slowdown is around 5% and 10% for most benchmarks for the 5% and 10% thresholds, respectively. The benchmarks for which we observe an exception are avrora and lusearch, with a slight overshoot for avrora at the 5% threshold, and an undershoot for lusearch at both the 5% and 10% thresholds. The reason is the inaccuracy of the DVFS performance predictor: lusearch and avrora experience the largest prediction errors, as shown in Figure 3. This result re-emphasizes the importance of accurate DVFS performance prediction for effectively managing energy consumption and performance when running managed multithreaded applications. Nevertheless, since lusearch stresses the memory subsystem the most, we observe a large reduction in its energy consumption despite slowing its execution less than the user-specified slowdown threshold.

Figure 8 shows the frequency settings chosen by our energy manager for xalan and sunflow for a slowdown threshold of 5%. The frequency settings chosen by our energy manger for the memory-intensive xalan cover a wider range compared to the compute-intensive sunflow. We observe similar trends in other benchmarks.

**Comparison to static-optimal.** To further analyze the robustness and importance of dynamically adjusting frequency, we compare our dynamic energy manager (using DEP+BURST) against the optimal frequency setting obtained statically. Static-optimal (Static-Opt) is determined by running the application multiple times offline, and selecting the optimal frequency that minimizes energy consumption across the entire run; because this static frequency is obtained while using the same input data set, we can consider the static-optimal frequency as an oracle setting. Note that Static-Opt is not a practical approach and is shown here for purposes of comparison only. Figure 9 compares the reduction in energy consumption by our dynamic energy manager to the reduction achieved by Static-Opt for a slowdown threshold of 10%. Our energy manager leads to larger reductions in energy consumed by all of the memory-intensive benchmarks with the exception of lusearch. For lusearch, DEP+BURST exhibits a larger error compared to the other benchmarks, which is the reason our energy manager misses the full potential for reducing the energy consumption. The overall reduction is 2% on average and up to 10%. The reason why our energy manager outperforms Static-Opt for xalan, pmd and pmd.scale is because it is able to dynamically adjust the frequency in response to varying execution phase behavior, which Static-Opt, by definition, is unable to do. The reduction on average is on par with static-optimal for the compute-intensive applications.

### 6.1.3 Frequency Step Setting

The granularity at which you can change the frequency, or the step setting, is an important factor in meeting performance targets yet striving for energy efficiency. In the previous results, we assumed a frequency step setting of 125 MHz with a total of 25 DVFS settings between 1 GHz and 4 GHz. A coarser frequency step setting makes it difficult to meet the user-specified slowdown thresholds. For instance, assume that slowing down one phase of a benchmark by 5% requires a frequency setting of 3.8 GHz. If the machine only offers a frequency step setting of 500 MHz, our energy manager will run that benchmark's phase at 4 GHz, since running at 3.5 GHz slows down the phase by more than 5%. This leads to a missed opportunity to save energy.

We add extra accounting to our energy manager to keep track of these missed opportunities per phase so that in a later phase, the application can run at a lower frequency while still meeting the user-specified slowdown target over the entire run, thus reducing energy consumption. More specifically, during each profiling quantum, our energy manager stores the difference between the execution time of running at the ideal frequency that would get closest to the slowdown threshold and the execution time given the best-available frequency setting (less than the highest frequency and does not violate the slowdown threshold). In our example above, this would be the execution time difference when running the benchmark's phase at the desired 3.8 GHz versus the energy manager's chosen 4 GHz. We call this difference $\sigma_{\text{excess}}$, which is shown in Equation 1. $T_{desired}$ is the execution time running at some ideal frequency if we did have fine-grained step settings, and $T_{estimated}$ represents the estimated execution time at the best-available frequency setting. Note that $T_{desired}$ is just the time quantum's duration plus the user-specified slowdown. $\sigma_{\text{excess}}$ is multiplied by the hold-off to account for there being no change in frequency for hold-off quantums.

$$\begin{aligned} T_{desired} &= quantum * (1 + tolerable\_slowdown) \\ \sigma_{\text{excess}} &= (T_{desired} - T_{estimated}) * hold\_off \end{aligned} \tag{1}$$

During a subsequent profiling quantum, the manager adds this previously calculated $\sigma_{\text{excess}}$ to the execution time we want to achieve in this phase. For example, because we did not slow down the previous phase at all, even though we had a target of 5%, we can slow down the current phase by more than 5%. However, over the entire run, we expect to still meet the user's specified slowdown threshold, while reducing more energy consumption.

Figure 10 presents the per-benchmark slowdown with and without modeling $\sigma_{\text{excess}}$ for two systems with a different number of DVFS states. One system provides a frequency step setting of 125 MHz (25 DVFS states), and the other system provides a frequency step setting of 500 MHz (7 DVFS states), which is more limiting. The slowdown threshold is set to 5%; hold-off is one; and the quantum length is 5 ms. With a frequency step setting of 125 MHz, the slowdown is 3.7% on average both with and without modeling $\sigma_{\text{excess}}$. However, with a 500 MHz frequency step setting, the average slowdown without modeling $\sigma_{\text{excess}}$ (w/o-excess-500) is 0.7%, which is much lower than the 5% target. For several benchmarks, a 5% slowdown in execution time compared to running at 4 GHz is achieved by running at a frequency somewhere between 3.5 GHz and 4 GHz. However, running at 3.5 GHz is likely to slow down the execution more than 5% for these benchmarks. Therefore, the energy manager runs these benchmarks at the highest frequency during most of the execution. The average slowdown increases to 3.2% when modeling $\sigma_{\text{excess}}$ (excess-500). In fact, all except one benchmark (avrora) experience a slowdown similar to excess-125.

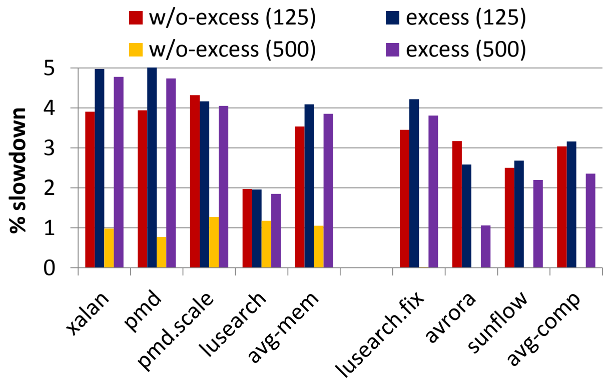


Fig. 10: Per-benchmark slowdown for different frequency step settings with and without modeling excess-time. *Modeling excess-time results in an average slowdown close to the user's expectations, regardless of the available frequency step setting.*

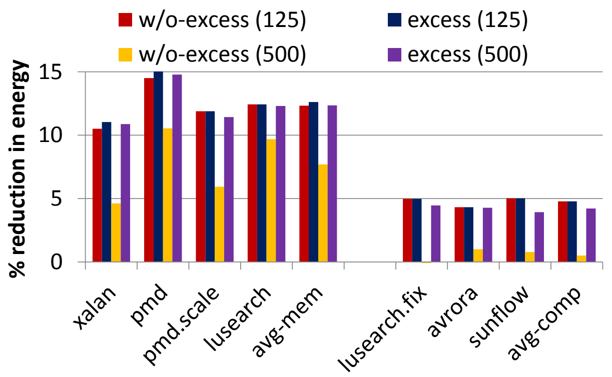


Fig. 11: Per-benchmark reduction in energy consumption for different frequency step settings with and without modeling excess-time. *Modeling excess-time results in an average energy reduction for both the compute-intensive and the memory-intensive benchmarks regardless of the frequency step setting.*

Figure 11 shows the reduction in energy consumption with and without modeling $\sigma_{\text{excess}}$. For both the compute-intensive and the memory-intensive benchmarks, the energy reduction of w/o-excess-125 and excess-125 is almost the same. However, w/o-excess-500 achieves only 7.7% reduction in energy consumption for the memory-intensive benchmarks compared to 12.5% provided by excess-500. For the memory-intensive benchmarks, modeling $\sigma_{\text{excess}}$ helps get closer to the user-specified slowdown threshold, and thus achieves a higher reduction in energy consumption. For the compute-intensive benchmarks, w/o-excess-500 barely provides any reduction in energy consumption at all. We conclude that modeling $\sigma_{\text{excess}}$ is especially important with coarser frequency step settings, and that it makes our energy manager robust to whatever DVFS granularity is provided by the processor.

### 6.1.4 Varying Hold-off and Length of Profiling Quantum

In all previous experiments, we use a hold-off of one and a quantum length of 5 ms. In this section, we vary these parameters to see if it is possible to meet the user's execution time requirement while running the model less often. A large hold-off would translate to the energy manager running the model less often.

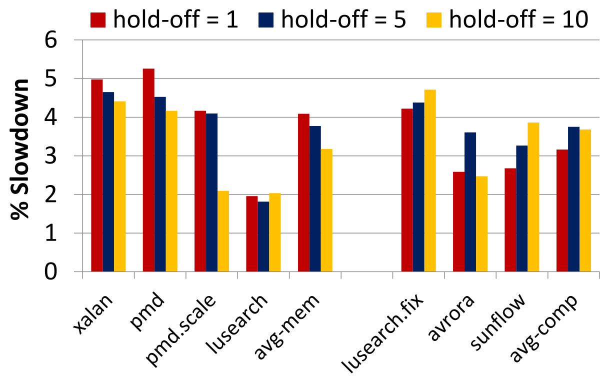


Fig. 12: Per-benchmark slowdown for different values of hold-off. *On average, a hold-off of 5 is a good compromise between running the model less often, and being close to the user-specified slowdown threshold.*

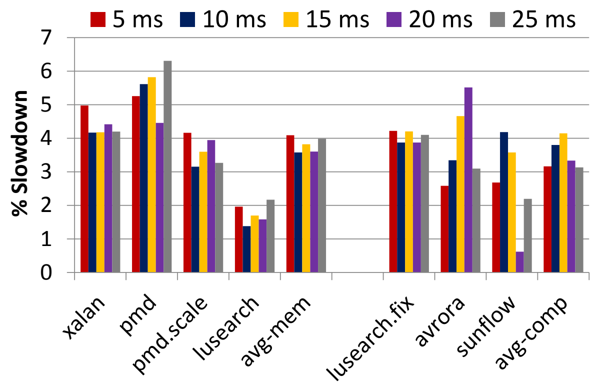


Fig. 13: Per-benchmark slowdown for different quantum lengths. *The smaller quantum length of 5 ms incurs less overhead for the model while still achieving a slowdown close to the user-specified threshold.*

Similarly, if the quantum is small, then the overhead of running the model is small.

In Figure 12, we vary the hold-off and keep the quantum length fixed at 5 ms. We show results with a hold-off of 1, 5, and 10. The tolerable slowdown is set to 5%. Using a large hold-off results in a slowdown further away from the user-specified threshold for three benchmarks, namely xalan, pmd and pmd.scale. Others either show no trend when increasing the hold-off or a slowdown slightly closer to the user's expectations. The average slowdown with a hold-off of 5, which runs the model less often, is 3.8%, compared to an average slowdown of 3.7% with a hold-off of one.

We show the impact of varying the quantum length on each benchmark's slowdown in Figure 13. We show results for five different quantum lengths, each a multiple of five. Because we are investigating the sensitivity to quantum length, we keep the hold-off at one. As before, the slowdown threshold is 5%. Each benchmark is affected differently, and we observe no general trend. For instance, avrora and pmd generally see increasing performance slowdowns as the quantum is increased, and sometimes their slowdown exceeds the user-specified 5%. lusearch.fix is not sensitive to quantum length, implying little or no phase behavior. We conclude that using a quantum length of 5 ms and a hold-off of 5 leads to performance that is, on average, close to the user's expectation. Using these parameters, the DEP+BURST model is active during only 20% of the benchmark's execution time.

To estimate the overhead of running DEP+BURST, we first note the number of epochs during the time we run DEP+BURST. We then use the latency of reading DVFS-related performance counters per epoch from prior work [14]. Our analysis show that the overhead of running DEP+BURST is less than 1% of the execution time of our benchmarks on average.

## 6.2 Case Study 2: Minimizing Full System Energy

To demonstrate the robustness of our energy manager to different optimization targets, we perform another case study, this time optimizing total system energy, i.e., the sum of the energy consumed by both the processor and DRAM. The optimal system energy is not obtained by running the processor at the lowest frequency, since as the processor frequency is lowered, the energy consumed by DRAM becomes the dominant factor. To optimize total system energy, the energy manager estimates the energy consumption at all the available DVFS states at the end of each profiling quantum, using the predictions of DEP+BURST for estimating the execution time, $T'$. The optimal frequency is the one which results in the lowest energy consumption. To estimate the total energy at a target DVFS setting $(v', f')$, when running at a base DVFS setting $(v, f)$, we perform the following steps:

1) We scale the static power of the processor (processor-p-static) by a factor $v/v'$ to estimate the processor-p-static at $(v', f')$ [8].
2) We collect the estimated execution time, $T'$, at $f'$ from DEP+BURST. We multiply $T'$ by the estimated processor-p-static to get the estimated static energy of the processor (processor-e-static).
3) We estimate the dynamic energy of the processor (processor-e-dynamic) as the sum of the dynamic energy of individual cores. The dynamic energy of each core at $(v', f')$ is estimated by multiplying the energy consumed by the core at $(v, f)$ with the factor, $(v/v')^2$, similar to [19].
4) The static energy of DRAM (dram-e-static) at $(v', f')$ is estimated as $T'$ multiplied by the static power consumed by the DRAM at $(v, f)$.
5) For the number of DRAM requests seen in the previous quantum, we obtain the dynamic energy consumed by DRAM (dram-e-dynamic) from McPAT. Since a change in execution time does not impact the number of DRAM requests (only the request rate), we use the value obtained from McPAT as an estimate of the dynamic energy consumed by DRAM at the target frequency.
6) Finally, the total estimated energy at $(v', f')$ is the sum of the estimated processor-e-static, processor-e-dynamic, dram-e-static and dram-e-dynamic.

Figure 14 shows the per-benchmark reduction in energy consumption obtained from different executions: running at the lowest frequency (1 GHz); running each benchmark multiple times offline, each time statically setting the frequency, and choosing the optimal energy consumption (Static-Opt); and dynamically adjusting the frequency at the end of each quantum using the above steps (Dynamic). Our baseline is the energy consumption obtained by running the entire benchmark at 4 GHz. We simulate a DDR3 DRAM main memory based on specifications from Micron [34].

First, some benchmarks experience an increase in energy from running at 1 GHz. Running at 1 GHz increases the execution time,

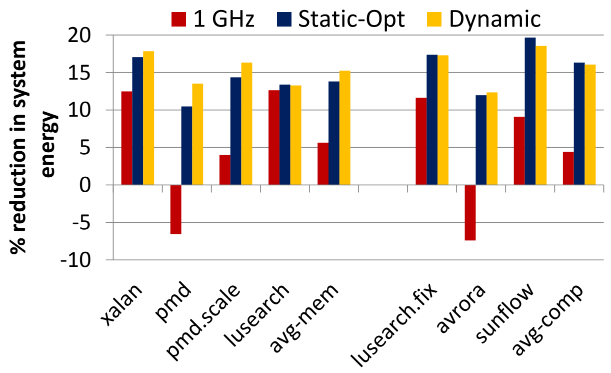


Fig. 14: Per-benchmark reduction in energy consumption when the energy manager optimizes for total system energy. *Our energy manager achieves reduction in energy consumption that is comparable to or better than the optimal reduction in energy consumption obtained statically.*

which in turn increases the DRAM static energy. On average, running at 1 GHz reduces the energy consumption by only 5%. On the other hand, Static-Opt provides a higher reduction in energy consumption for all benchmarks. The average energy reduction is 15%, and the maximum reduction is 20% for sunflow. Since this case study does not impose a performance constraint, even the compute-intensive benchmarks benefit greatly from DVFS because reducing the processor voltage and frequency results in a quadratic drop in the dynamic energy consumed by the processor (at the expense of performance).

Next, we observe that our dynamic energy manager delivers a reduction in energy consumption on par with or better than Static-Opt. The average reduction in energy consumption is 15.6% with a maximum reduction of 18.5% for sunflow. For three benchmarks, including xalan, pmd, and pmd.scale, our proposed energy manager achieves a higher reduction in energy consumption than Static-Opt. Unlike Static-Opt, our dynamic manager is able to exploit phase behavior. Having an accurate performance predictor is necessary to optimize the full system energy consumption of multithreaded managed language applications.

In this section, we considered only the energy consumed by the processor and the DRAM. Other components such as the cooling unit, motherboard etc., also contribute to the system energy. Our energy manager can be easily extended to take into account any of these non-scaling components of system energy. It should be noted that if the increase in execution time due to lowering the processor's frequency leads to an increase in the total system energy - because the energy consumed by the other components offsets the reduction in the processor's dynamic energy consumption - our energy manager will run the processor at the highest frequency.

## 7 Related Work

In this section, we discuss three areas of related work.

### 7.1 DVFS Performance and Power Prediction

Performance and power prediction is either done using analytical models or using regression models. Section 2 already discussed previously proposed analytical DVFS performance predictors in great detail [11], [20], [29], [35], [38], [41], [46]. These papers introduce new hardware performance counters specifically for the purpose of predicting the performance impact of DVFS. Su et al. [42] have recently shown how to implement the Leading Loads DVFS predictor on real AMD CPUs. In contrast, other works propose regression models that are built using offline training to predict the power and performance impact of frequency and architectural changes [12], [31], [43]. To build a regression model, these works leverage existing hardware performance counters to measure various microarchitectural events.

Deng el al. [15] propose an algorithm to manage DVFS for both the processor and the memory while honoring a user-specified slowdown threshold. However, this and many other works on DVFS power management do not consider multithreaded applications.

In this work, we investigate predicting the performance impact of chip-wide DVFS settings. Prior work investigates the potential of per-core DVFS to manage the energy consumption of multi-threaded applications [24], [30]. However, we leave this for future work.

### 7.2 Scheduling Multithreaded Applications

Recently, there is increased interest in scheduling multithreaded applications on multicore hardware to optimize performance and energy. The main focus to date is in identifying and accelerating bottlenecks in multithreaded code, such as serial sections, critical sections, and lagging threads [4], [16], [27], [28], [44]. Accelerated Critical Sections (ACS) is a technique that leverages support from the ISA, compiler, and the large cores on a single-ISA heterogeneous multicore to accelerate critical sections [44]. Unlike accelerating only critical sections, Bottleneck Identification and Scheduling (BIS) also targets other bottlenecks that occur during the execution of a multithreaded application such as serial sections, lagging threads, and slow pipeline stages [27]. The above works use ISA and compiler support to delimit bottlenecks in software, and use this information during execution to accelerate bottlenecks. On the other hand, Criticality Stacks, proposed by Du Bois et al. [16], identify critical threads in multithreaded applications by monitoring synchronization behavior.

Finally, when running multithreaded applications on heterogeneous multicore processors, an important goal is to prevent one or more threads from lagging behind other threads. To this end, Van Craeynest et al. [45] propose a fair scheduler for multithreaded applications that provides a fair share of the big, out-of-order cores to each thread in a heterogeneous multicore processor. Akram et al. [1] propose a GC-criticality-aware scheduler for managed language applications on heterogeneous multicores.

### 7.3 Energy Management

Prior work has proposed frameworks to manage power, energy and thermals through DVFS, hardware adaptation and heterogeneity for multithreaded applications [16], [33], [36]. Although managed code is now ubiquitous and used in many application domains and run on a variety of hardware substrates, relatively few works have looked into the energy management of managed applications. Sartor et al. [39] explored the potential of DVFS for managed applications, teasing apart the performance impact of scaling the frequency of application and service threads in isolation. However, their work does not propose an analytical model to quantify the performance impact. Other works that shed light on different aspects of managed applications relating to energy consumption include [9], [18], [40].

## 8 Conclusions and Future Work

Accurate performance predictors are key to making effective use of dynamic voltage and frequency scaling (DVFS) to reduce energy consumption in modern processors. Multithreaded managed applications are ubiquitous yet prior work lacks accurate DVFS performance predictors for these applications. In this work, we propose DEP+BURST, a novel performance prediction model to accurately predict the performance impact of DVFS for multi-threaded managed applications. DEP decomposes execution time into epochs based on synchronization activity. This allows DEP to accurately capture inter-thread dependencies, and take the critical threads into account across epochs. BURST identifies critical store bursts and predicts their impact on overall performance as the frequency is scaled.

Our experimental results with multithreaded Java applications on a simulated quad-core processor report an average absolute error of 6% when predicting from 1 GHz to 4 GHz, and 8% when predicting from 4 GHz to 1 GHz using DEP+BURST, which is a substantial improvement over prior work. We demonstrate the usefulness of DEP+BURST by integrating it into an energy manager that 1) reduces the processor's energy by sacrificing a user-specified amount of performance, and 2) optimizes total system energy. For 1), with a user-specified slowdown of 5% and 10%, the energy manager is able to reduce energy consumption by 13% and 19% on average for a number of memory-intensive benchmarks. We show that our energy manager is robust to coarser frequency step settings, and incurs negligible execution time overhead. Finally, for 2), our energy manager demonstrates 15.6% total system energy consumption reduction on average.

This work is the first to propose a DVFS performance predictor for managed multithreaded applications. Several directions for future work are possible. Fine-grained dependencies between threads, such as those resulting from shared critical sections, could change at the target frequency. When this happens, DEP mispredicts the execution time at the target frequency. Efficiently dealing with mispredictions could improve the accuracy in meeting the user-specified slowdown thresholds, further reducing energy consumption. Another avenue for future work would be to explore per-core DVFS, as opposed to our current implementation that changes the frequency setting of all cores running a multithreaded application. DEP needs modifications to predict the performance impact of per-core DVFS. What is even more challenging is identifying the threads whose frequency change would result in the largest reduction in energy consumption. Finally, investigating the performance impact of DVFS when there is contention for either bandwidth or shared cache capacity, is a direction for future work.

## References

[1] S. Akram, J. B. Sartor, K. V. Craeynest, W. Heirman, and L. Eeckhout, "Boosting the priority of garbage: Scheduling collection on heterogeneous multicore processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 13, no. 1, pp. 4:1–4:25, Mar. 2016.

[2] S. Akram, J. B. Sartor, and L. Eeckhout, "DVFS performance prediction for managed multithreaded applications," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS*, 2016, pp. 12–23.

[3] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño virtual machine," *IBM Systems Journal*, vol. 39, no. 1, pp. 211–238, 2000.

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

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, F. D., S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006, pp. 169–190.

[6] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffman, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century," *Communications of the ACM*, vol. 51, no. 8, pp. 83–89, 2008.

[7] S. M. Blackburn and K. S. McKinley, "Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008, pp. 22–32.

[8] J. Butts and G. Sohi, "A static power model for architects," in *Proceedings of the 33rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2000, pp. 191–201.

[9] T. Cao, S. M. Blackburn, T. Gao, and K. S. McKinley, "The yin and yang of power and performance for asymmetric hardware and managed software," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2012, pp. 225–236.

[10] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, pp. 28:1–28:25, 2014.

[11] K. Choi, R. Soma, and M. Pedram, "Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times," in *Proceedings of the Design, Automation and Test in Europe (DATE)*, 2004, pp. 4–9 Vol.1.

[12] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz, "Prediction models for multi-dimensional power-performance optimization on many cores," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008, pp. 250–259.

[13] K. Czechowski, V. W. Lee, and J. Choi, "Measuring the power/energy of modern hardware," in *Tutorial at the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, ser. http://www.prism.gatech.edu/ gtg417r/micro47/, 2014.

[14] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 353–364.

[15] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "CoScale: Coordinating CPU and memory system DVFS in server systems," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 143–154.

[16] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, "Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013, pp. 511–522.

[17] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout, "Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013, pp. 355–372.

[18] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley, "Looking back on the language and hardware revolutions: Measured power, performance, and scaling," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 319–332.

[19] S. Eyerman and L. Eeckhout, "A counter architecture for online DVFS profitability estimation," *IEEE Transactions on Computers (TC)*, vol. 59, no. 11, pp. 1576–1583, Nov. 2010.

[20] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A performance counter architecture for computing accurate CPI components," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 175–184.

[21] H. Franke, R. Russell, and M. Kirkwood, "Fuss, futexes and furwocks: Fast userlevel locking in linux," in *Ottawa Linux Symposium*, 2002, pp. 479–495.

[22] J. Ha, M. Gustafsson, S. Blackburn, and K. S. McKinley, "Microarchitectural characterization of production JVMs and Java workloads," in *IBM CAS Workshop*, 2008.

[23] W. Heirman, S. Sarkar, T. E. Carlson, I. Hur, and L. Eeckhout, "Power-aware multi-core simulation for early design stage hardware/software co-optimization," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012, pp. 3–12.

[24] S. Herbert and D. Marculescu, "Analysis of dynamic voltage/frequency scaling in chip-multiprocessors," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2007, pp. 38–43.

[25] A. Hoban, "Designing realtime solutions on embedded intel architecture processors," 2010.

[26] X. Huang, Z. Wang, S. Blackburn, K. S. McKinley, J. E. B. Moss, and P. Cheng, "The garbage collection advantage: Improving mutator locality." in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2004, pp. 69–80.

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

[28] J. A. Joao, M. A. Suleman, O. Mutlu, and Y. N. Patt, "Utility-based acceleration of multithreaded applications on asymmetric cmps," in *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, 2013, pp. 154–165.

[29] G. Keramidas, V. Spiliopoulos, and S. Kaxiras, "Interval-based models for run-time DVFS orchestration in superscalar processors," in *Proceedings of the 7th ACM International Conference on Computing Frontiers (CF)*, 2010, pp. 287–296.

[30] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks, "System level analysis of fast, per-core DVFS using on-chip switching regulators," in *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2008, pp. 123–134.

[31] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006, pp. 185–194.

[32] S. Li, J. H. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009, pp. 469–480.

[33] K. Ma, X. Li, M. Chen, and X. Wang, "Scalable power control for many-core architectures running multi-threaded applications," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011, pp. 449–460.

[34] Micron, "Tn-41-01: Calculating memory system power for ddr3," 2007.

[35] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt, "Predicting performance impact of DVFS for realistic memory systems," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012, pp. 155–165.

[36] S. Park, W. Jiang, Y. Zhou, and S. Adve, "Managing energy-performance tradeoffs for multithreaded applications on multiprocessor architectures," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2007, pp. 169–180.

[37] R. Radhakrishnan, N. Vijaykrishnan, L. John, and A. Sivasubramaniam, "Architectural issues in Java runtime systems," in *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture (HPCA)*, 2000, pp. 387–398.

[38] B. Rountree, D. Lowenthal, M. Schulz, and B. de Supinski, "Practical performance prediction under dynamic voltage frequency scaling," in *Proceedings of the International Green Computing Conference and Workshops (IGCC)*, 2011, pp. 1–8.

[39] J. B. Sartor and L. Eeckhout, "Exploring multi-threaded Java application performance on multicore hardware," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012, pp. 281–296.

[40] J. B. Sartor, W. Heirman, S. M. Blackburn, L. Eeckhout, and K. S. McKinley, "Cooperative cache scrubbing," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014, pp. 15–26.

[41] V. Spiliopoulos, S. Kaxiras, and G. Keramidas, "Green governors: A framework for continuously adaptive DVFS," in *Proceedings of the International Green Computing Conference and Workshops (IGCC)*, 2011, pp. 1–8.

[42] B. Su, J. L. Greathouse, J. Gu, M. Boyer, L. Shen, and Z. Wang, "Implementing a leading loads performance predictor on commodity processors," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2014, pp. 205–210.

[43] B. Su, J. Gu, L. Shen, W. Huang, J. Greathouse, and Z. Wang, "PPEP: Online performance, power, and energy prediction framework and DVFS space exploration," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014, pp. 445–457.

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

[45] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout, "Fairness-aware scheduling on single-ISA heterogeneous multi-cores," in *Proceedings of the international conference on Parallel architectures and compilation techniques (PACT)*, 2013, pp. 177–188.

[46] Q. Wu, V. Reddi, Y. Wu, J. Lee, D. Connors, D. Brooks, M. Martonosi, and D. Clark, "A dynamic compilation framework for controlling microprocessor energy and performance," in *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2005, pp. 282–293.

[47] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley, "Why nothing matters: The impact of zeroing," in *Proceedings of the ACM Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011, pp. 307–324.

**Shoaib Akram** is a fourth year PhD student at Ghent University, Belgium. He received his M.S. in Electrical and Computer Engineering from the University of Illinois at Urbana Champaign in 2009. His research interests include performance modeling and evaluation, run-time scheduling, and energy-efficient computer systems in general.

**Jennifer B. Sartor** is a professor at Vrije Universiteit Brussel, Belgium. She also has research collaborations with Ghent University. She received her PhD in Computer Science from The University of Texas at Austin in 2010. Her research interests are in managed languages, optimizing performance with the language runtime environment, memory management and memory efficiency.

**Lieven Eeckhout** is Professor at Ghent University, Belgium. He received his PhD in Computer Science and Engineering from Ghent University in 2002. His research interests are in the area of computer architecture, with a specific interest in performance analysis, evaluation and modeling. He is the current editor-in-chief of IEEE Micro. His research is funded by the European Research Council under the European Communitys Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 259295, as well as by the European Commission under the Seventh Framework Programme, Grant Agreement no. 610490.