DVFS Performance Prediction for Managed Multithreaded Applications

Shoaib Akram*, Jennifer B. Sartor*† and Lieven Eeckhout*
*Ghent University, Belgium
†Vrije Universiteit Brussel, Belgium
Email: {Shoaib.Akram, Jennifer.Sartor, Lieven.Eeckhout}@UGent.be

***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 save energy while meeting performance goals within a user-specified bound. Our proposed energy manager delivers average energy savings of 13% and 19% for a user-specified slowdown of 5% and 10% for memory-intensive Java benchmarks. Accurate performance predictors are key to achieving high performance while keeping energy consumption low for managed language applications using DVFS.

I. Introduction

Today, more than ever before, improving the energy-efficiency of computer systems is of prime importance. In particular, we want our devices to deliver high performance without needlessly wasting energy. For this reason, in recent times, a number of power and energy management techniques, including Dynamic Voltage and Frequency Scaling (DVFS), have made their way into modern devices. DVFS enables simultaneously changing a processor's voltage and frequency to manage power and energy. The effective use of DVFS requires accurate performance prediction models. 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 [9], [16], [26], [31], [34], [42].

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. Figure 1 illustrates this quantitatively for a set of multithreaded Java applications from the DaCapo suite [4], for which we predict performance at a target frequency (shown on the horizontal axis) based on a run at the baseline frequency of 1 GHz. M+CRIT, which is a multithreaded extension of the state-of-the-art CRIT [31], leads to high inaccuracy with an average absolute prediction error of 27% at the 4 GHz target frequency. M+CRIT uses CRIT to predict each thread's performance at the target frequency, and then predicts the total execution time as the execution time of the slowest, most critical thread. Unfortunately, this naive algorithm leads to severe inaccuracies with managed multithreaded applications.

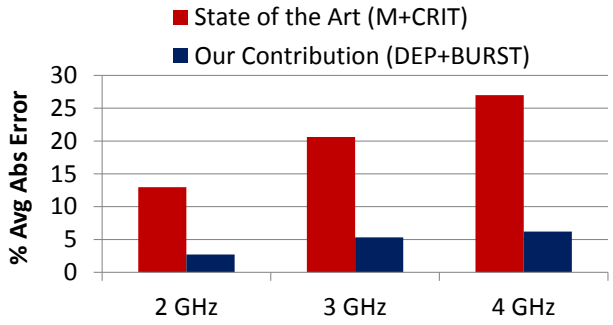


Fig. 1: Average absolute error for state-of-the-art DVFS prediction compared to our predictor DEP+BURST.

The reason for this inaccuracy is multifold. 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 total execution time at the target frequency.

Managed applications, which run on top of a virtual machine, exhibit even more inter-thread dependencies as compared to native applications. Service threads, such as those that perform garbage collection and just-in-time compilation, run alongside the application threads [7], [33]. 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 for managed applications results

from 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 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 handles synchronization and inter-thread dependencies by decomposing the execution time of a multithreaded application into epochs based on the synchronization activity of the application. 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 the 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 frequency settings.

Figure 1 summarizes the key results. DEP+BURST is substantially more accurate than M+CRIT. At the 4 GHz target frequency, DEP+BURST achieves an average absolute error of 6% whereas M+CRIT experiences a 27% error. We observe a similarly low prediction error when predicting performance at 1 GHz based on a baseline run at 4 GHz, with an average absolute error of 8% (not shown in Figure 1).

We integrate DEP+BURST into an energy management framework for managed applications. We use the energy manager to explore potential energy savings when a slack in performance is tolerable compared to running at the highest frequency. We demonstrate that using DEP+BURST as a DVFS predictor, our energy manager is able to dynamically select DVFS settings that result in energy savings in return for a slowdown of the application (compared to the highest frequency setting) close to a user's expectations. On average, for a slowdown threshold of 5% and 10%, our energy management framework delivers energy savings of 13% and 19% for our set of memory-intensive applications. Having an accurate performance predictor for DVFS is key to maintaining good performance, especially for multithreaded managed applications, while reducing energy consumption.

II. Background and Motivation

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

A. 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 [16], [26], 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 [16], [26], [34], 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 [39].

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. [31]. 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.

To the best of our knowledge, there exists no prior work that proposes a DVFS performance predictor for multithreaded applications, let alone managed language workloads.

B. 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. Different threads of a multithreaded application use synchronization primitives to coordinate access to shared variables. The most common examples of synchronization include critical sections and barriers. Synchronization leads to inter-thread dependencies. For example, with a barrier, no thread is allowed to continue past the barrier as long as all threads have not yet reached the barrier. The slowest, most critical thread will therefore determine the barrier execution time at the target frequency. Similarly, with of 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. In other words, scaling the frequency of one thread 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.

C. Straightforward Extensions of Prior Work

Before presenting our DVFS performance predictor for managed multithreaded applications 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.

III. The DEP+BURST Model

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

A. 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 show how we deal with store bursts.

B. 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 [18] 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

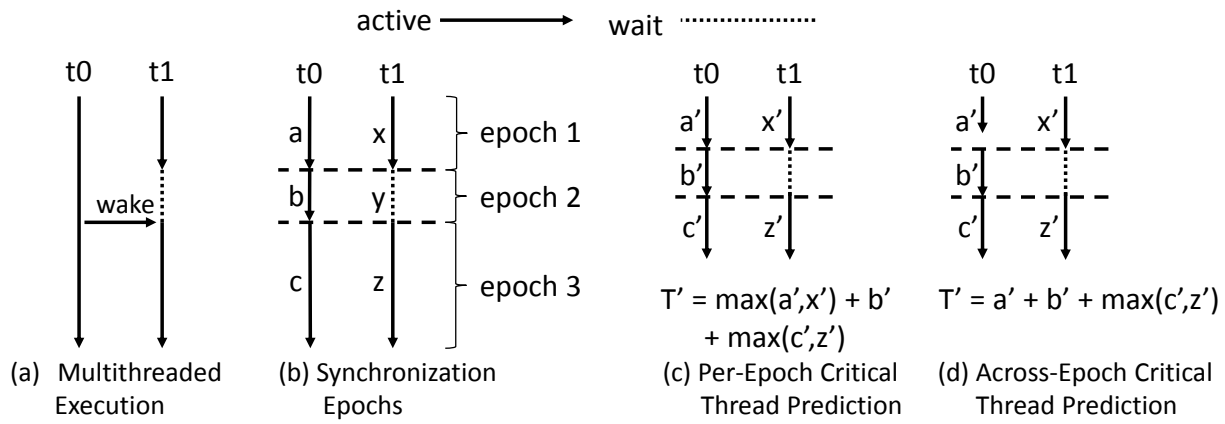


Fig. 2: Showing how DEP breaks up a multithreaded application (a) into synchronization epochs (b) while running at the base frequency. DEP then estimates per-thread epoch durations at the target frequency, calculates the critical thread per epoch (c), and accounts for changes in the critical thread across epochs (d).

the kernel spin locks using the futex interface. Intercepting futex calls incurs limited overhead (less than 1%) [14].

To understand why futex-based decomposition is necessary, consider the example of a multithreaded execution in Figure 2(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 2(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 II). 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 because 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 2(b) shows how our predictor decomposes the execution shown in Figure 2(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 by breaking down execution into epochs, we not only cover inter-thread dependencies between application threads. 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, DEP automatically accounts for the extra interactions between managed language application and service threads.

C. 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 2(c) and Figure 2(d) where a', b' and c' represents 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 2(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 estimation 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 2(d). In the

Algorithm 1: Algorithm for across-epoch CTP.

input : A synchronization epoch S (I time units)
input : Initial delta-counters (δ_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 | α_t = `computeEstimatedTimeUsingCRIT()`
3 | $e_t = \alpha_t$ - δ_t
4 **end**
5 I' = Largest e_t
6 **for** *each active thread t in S* **do**
7 | δ_t = (I' - α_t) + δ_t
8 **end**
9 $\delta_{stall-tid} = 0$

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 contents of 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. α_t represents the estimated duration of a thread t at the target frequency. δ_t is the difference between the estimated duration of thread t and the estimated duration of the critical thread; δ_t of the critical thread is zero. The first step in Algorithm 1 is to compute the estimated duration, α_t, of each thread using CRIT (line 2). Next, we calculate the 'effective' execution time (e_t) of each thread by subtracting δ_t from α_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 δ_t is accumulated across epochs, with a term representing the difference between I' and α_t added during each epoch until the thread stalls (line 7). We reset δ_t of a stalling thread to zero (line 9).

D. 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 caches 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 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.

E. 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. Importantly, our proposal requires no changes to the managed runtime or the JVM.

We also require some extra counters and state to accurately perform DVFS performance prediction with DEP+BURST. We use CRIT [31] 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. Note that as of today, no implementation of CRIT exists on real hardware.

In order to additionally track store bursts, we introduce a new performance counter per core in hardware. This performance counter tracks the time the store queue is full. The additional logic required to track the time the store queue is full is simple, and requires no changes to the design of the store queue. Once all the entries in the store queue are occupied, a signal is generated. This signal is monitored by the performance counter hardware to account for the time the store queue is full.

Finally, to account for critical threads across epochs, we require one counter per thread. However, 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 I: 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.

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

TABLE II: Simulated system parameters.

IV. Experimental Methodology

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

Simulator. We use Sniper [8] 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 [36] 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 [2] to evaluate the seven multithreaded Java benchmarks from the DaCapo suite [4] 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 [43]), that eliminates needless allocation. Finally, we use an updated version of pmd, called pmd-scale (described in [14]) 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 [14]. For the remaining multithreaded benchmarks, we perform evaluation with four application threads. Table I 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 [36]), 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.

We follow common practice in Java performance evaluation by using replay compilation [6], [19] 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], [22]. 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 [5] along with the default nursery settings.

Processor architecture. We consider a quad-core processor configured after the Intel Haswell processor i7-4770K, see Table II. 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 [29] for modeling power consumed by the processor. For DVFS support, we use the Sniper/McPAT integration described in [20] 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 VI). We use the voltage settings similar to Intel's i7-4770K (22 nm Haswell) [11]; see Table II for a subset of settings. When reporting power numbers, we include both static and dynamic power. We model the DVFS transition latency as a fixed cost of 2 μs.

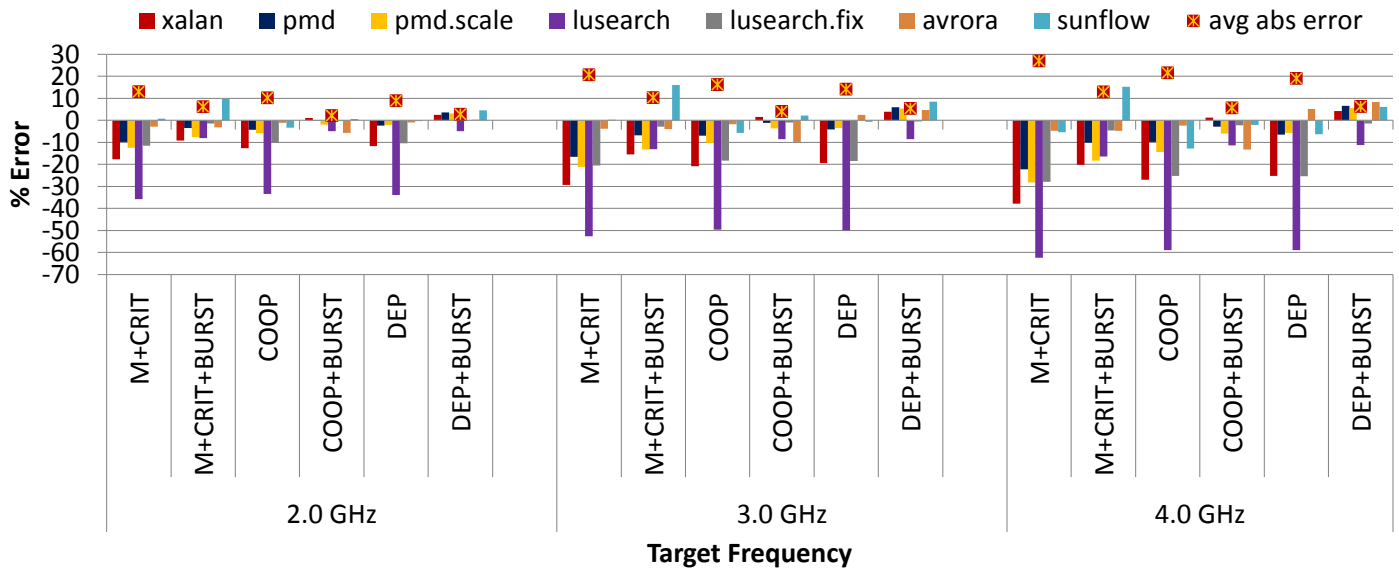
V. Model Evaluation

We now evaluate the accuracy of DEP+BURST and compare against M+CRIT and COOP. For all models we evaluate with and without BURST, teasing apart the contribution of taking store bursts into account.

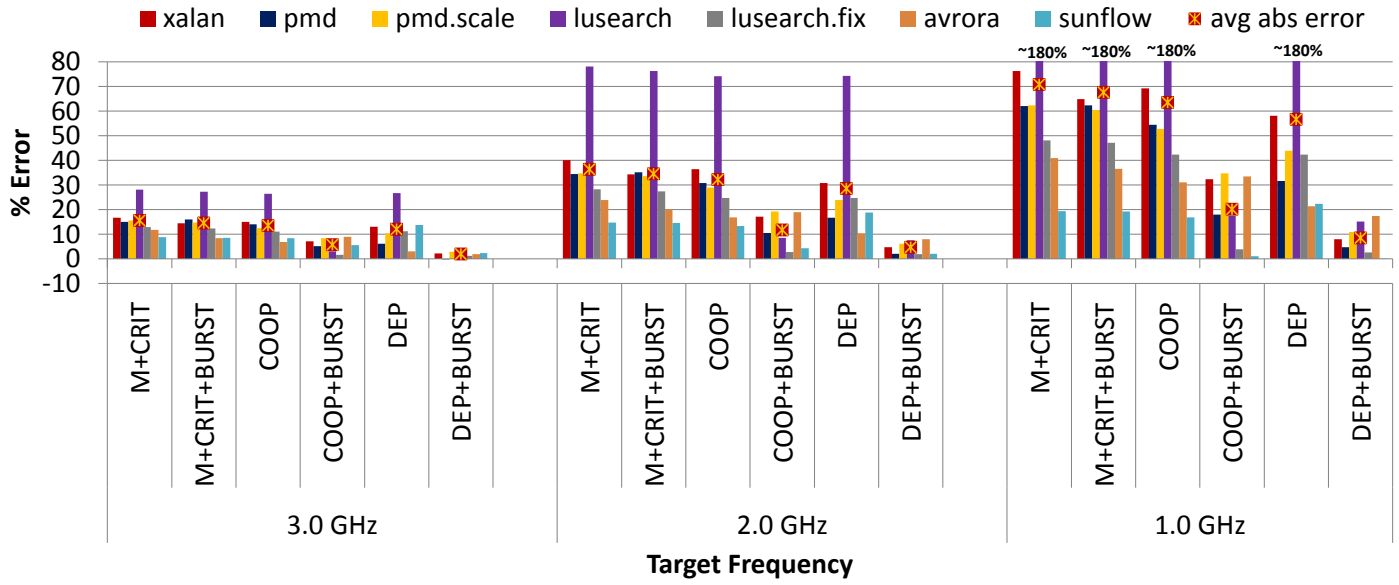
A. Prediction Accuracy

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 our case study



(a) Low-to-high prediction from a baseline at 1.0 GHz



(b) High-to-low prediction from a baseline at 4.0 GHz

Fig. 3: Per-benchmark prediction errors for M+CRIT, COOP and DEP, both with and without BURST: (a) prediction at higher frequency from a baseline of 1 GHz, and (b) prediction at lower frequency from a baseline of 4 GHz. *DEP+BURST outperforms all other predictors with an average absolute estimation error of below 10%, both when predicting from 1 GHz to 4 GHz and vice-versa.*

— 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 3(a) quantifies the prediction error for all benchmarks (including the average absolute error) 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. Figure 3(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.

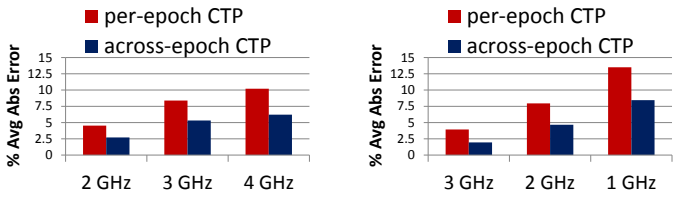
Modeling store bursts brings the error down substantially, especially for the memory-intensive benchmarks. In fact, all three models, M+CRIT, COOP and DEP, benefit from BURST modeling. It is interesting to note that DEP benefits most from BURST, and COOP benefits more than M+CRIT. Because DEP more accurately identifies critical threads, adding modeling of store bursts, which affect the critical thread, improves accuracy more substantially. Likewise, COOP identifies critical threads more accurately than M+CRIT, and hence benefits more from store burst modeling than M+CRIT.

This leads to the overall conclusion that DEP+BURST is the most accurate DVFS performance predictor, with an average absolute error of 6% when predicting from 1 GHz to 4 GHz, and an average absolute error of 8% when predicting from 4 GHz to 1 GHz. This result shows that 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.

B. Per-Epoch vs. Across-Epochs Critical Thread Prediction

As argued in Section III, 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



(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. *Across-epoch CTP decreases the average absolute error by 4% when predicting from 1 GHz to 4 GHz, and by 6% when predicting from 4 Ghz to 1 GHz.*

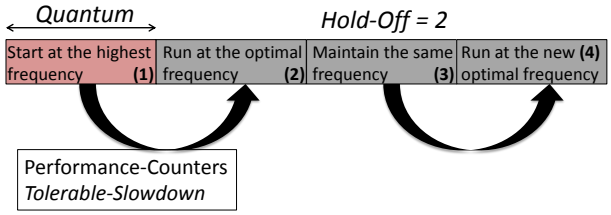


Fig. 5: Example illustrating how the energy manager works using DVFS performance prediction. The input parameters of the energy manager are shown in italics.

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.

VI. Case Study: Energy Minimization with Performance Guarantees

Having described and evaluated the DEP+BURST DVFS performance predictor, we now use it in a case study involving an energy manager that leverages a performance predictor to save energy without slowing down the application more than a user-specified threshold.

It is well-known that it is possible to reduce processor 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 [7], [33]. 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.

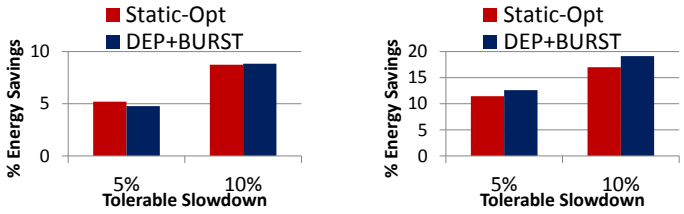
A. Energy Management

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 5. 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 computes the DVFS-related performance counters as described in Section III. 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 use a Hold-Off of one in our experiments.

The key idea we use to guarantee that the application does not experience a slowdown more than the specified threshold compared to running at the highest frequency 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.

B. Evaluation

Figure 6 reports the slowdown experienced by each benchmark and the corresponding reduction in energy for user-specified slowdown thresholds of 5% and 10%. We observe substantial energy savings 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 10% threshold. As expected, the energy savings are not as significant for the compute-intensive workloads.



(a) Compute-Bound **(b)** Memory-Bound

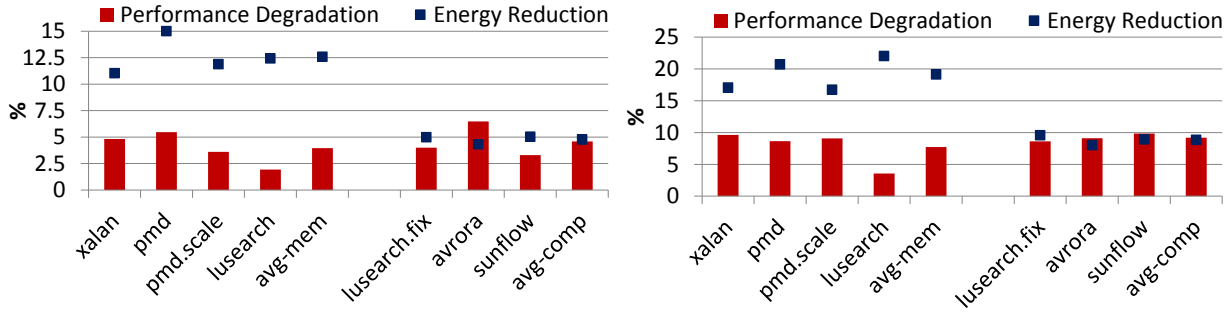
Fig. 7: Energy savings achieved by our energy manager compared to the static optimal (Static-Opt). *Our energy manager provides 2% more energy savings compared to Static-Opt for the memory-intensive benchmarks.*

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.

Finally, to further analyze the robustness and importance of dynamically adjusting frequency, we compare our dynamic energy manager (using DEP+BURST) against a static-optimal frequency setting. Static-optimal is determined by running the application multiple times 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. Figure 7 compares the energy saved by our dynamic energy manager to the savings achieved by the static-optimal method. The energy savings achieved by our manager are on par with static-optimal for the compute-intensive applications, while being slightly higher for the memory-intensive applications (by 2.1% on average for the 10% threshold). The reason why our energy manager outperforms static-optimal for the memory-intensive applications is because it is able to dynamically adjust the frequency in response to varying execution phase behavior, which static-optimal, by definition, is unable to do.

VII. Related Work

This work is the first, to the best of our knowledge, to propose a DVFS performance predictor for multithreaded managed applications. In Section II, we discussed existing DVFS predictors for sequential applications. Here, we discuss three



(a) Tolerable Slowdown = 5%

(b) Tolerable Slowdown = 10%

Fig. 6: Per-benchmark energy savings 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. *For slowdown thresholds of 5% and 10%, our energy manager provides average energy savings of 13% and 19% for the memory-intensive benchmarks.*

areas of related work: performance and power prediction, scheduling of multithreaded applications in power-constrained environments, and energy management of multithreaded applications on multicore hardware.

A. DVFS Performance and Power Prediction

Performance and power prediction is either done using analytical models or using regression models. Section II already discussed previously proposed analytical DVFS performance predictors in great detail [9], [17], [26], [31], [34], [37], [42]. These papers introduce new hardware performance counters specifically for the purpose of predicting the performance impact of DVFS. Su et al. [38] 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 by offline training to predict the power and performance impact of frequency and architectural changes [10], [28], [39]. To build a regression model, these works leverage existing hardware performance counters to measure various microarchitectural events.

Deng el al. [12] 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 in managing the energy consumption of multithreaded applications [21], [27]. However, we leave this for future work.

B. 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 [3], [13], [23], [24], [40]. 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 [40]. 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 [23]. 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. [13], 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. Van Craeynest et al. [41] propose a fair scheduler for multithreaded applications that provides a fair share of the big, out-of-order cores in a heterogeneous multicore processor to each thread. Akram et al. [1] propose a GC-criticality-aware scheduler for managed language applications on heterogeneous multicores.

C. Energy Management

Prior work has proposed frameworks to manage power, energy and thermals through DVFS, hardware adaptation and heterogeneity for multithreaded applications [13], [30], [32]. 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. [35] 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 [7], [15], [25], [36].

VIII. Conclusions

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 multithreaded 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 considering 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. Our comprehensive analysis illustrates the importance of identifying synchronization epochs, predicting critical threads across epochs, and modeling store bursts. We demonstrate the usefulness of DEP+BURST by integrating it into an energy manager that achieves significant energy savings for memory-intensive applications within user-specified performance constraints. In particular, for a user-specified slowdown of 5% and 10%, the energy manager is able to save 13% and 19% in energy consumption on average for a number of memory-intensive benchmarks. Accurate performance prediction for multithreaded applications is critical to effectively use DVFS and achieve good performance while minimizing energy consumption.

IX. Acknowledgements

This research is funded through the European Research Council under the European Community's Seventh Framework Programme (FP7/2007-2013)/ERC grant agreement no. 259295, as well as EU FP7 Adept project number 610490. The experiments were run on computing infrastructure at the ExaScience Lab, Leuven, Belgium.

References

[1] S. Akram, J. B. Sartor, K. Van 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)*.

[2] 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*, 39(1):211–238, 2000.

[3] 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)*, pages 290–301, 2009.

[4] 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)*, pages 169–190, 2006.

[5] 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)*, pages 22–32, 2008.

[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*, 51(8).

[7] 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)*, pages 225–236, 2012.

[8] 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)*, 11(3):28:1–28:25, 2014.

[9] 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)*, pages 4–9 Vol.1, 2004.

[10] 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)*, pages 250–259, 2008.

[11] 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)*, http://www.prism.gatech.edu/ gtg417r/micro47/, 2014.

[12] 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)*, pages 143–154, 2012.

[13] 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)*, pages 511–522, 2013.

[14] 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)*, pages 355–372, 2013.

[15] 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)*, pages 319–332, 2011.

[16] S. Eyerman and L. Eeckhout. A counter architecture for online DVFS profitability estimation. *IEEE Transactions on Computers (TC)*, 59(11):1576–1583, Nov. 2010.

[17] 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)*, pages 175–184, 2006.

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

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

[20] 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)*, pages 3–12, 2012.

[21] 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)*, pages 38–43, 2007.

[22] 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)*, pages 69–80, 2004.

[23] 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)*, pages 223–234, 2012.

[24] 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)*, pages 154–165, 2013.

[25] M. Kambadur and M. A. Kim. An experimental survey of energy management across the stack. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 329–344, 2014.

[26] 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)*, pages 287–296, 2010.

[27] 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)*, pages 123–134, 2008.

[28] 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)*, pages 185–194, 2006.

[29] 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)*, pages 469–480, 2009.

[30] 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)*, pages 449–460, 2011.

[31] 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)*, pages 155–165, 2012.

[32] 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)*, pages 169–180, 2007.

[33] 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)*, pages 387–398, 2000.

[34] 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)*, pages 1–8, 2011.

[35] 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)*, pages 281–296, 2012.

[36] 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)*, pages 15–26, 2014.

[37] 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)*, pages 1–8, 2011.

[38] 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)*, pages 205–210, 2014.

[39] 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)*, pages 445–457, 2014.

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

[41] 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)*, pages 177–188, 2013.

[42] 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)*, pages 282–293, 2005.

[43] 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)*, pages 307–324, 2011.