Energy-Performance Trade-offs on Energy-Constrained Devices with Multi-Component DVFS

Rizwana Begum, David Werner and Mark Hempstead
Drexel University
{rb639,daw77,mhempstead}@drexel.edu

Guru Prasad, Jerry Antony Ajay and Geoffrey Challen
University at Buffalo
{gurupras,jerryant,challen}@buffalo.edu

***Abstract*—Battery lifetime continues to be a top complaint about smartphones. Dynamic voltage and frequency scaling (DVFS) has existed for mobile device CPUs for some time, and can be used to dynamically trade off energy for performance. To make more energy-performance tradeoffs possible, DVFS is beginning to be applied to memory as well.**

We present the first characterization of the behavior and optimal frequency settings of workloads running both under *energy constraints* and on systems with *both* CPU and memory DVFS, an environment representative of next-generation mobile devices. Our results show that continuously using the optimal frequency settings results in a large number of frequency transitions which end up hurting performance. However, by permitting a small loss in performance, transition overhead can be reduced and end-to-end performance and energy consumption improved. We introduce the idea of *inefficiency* as a way of constraining task energy consumption relative to the most energy-efficient settings, and characterize the performance of multiple workloads running under different inefficiency settings. Overall our results have multiple implications for next-generation mobile devices exposing multiple energy-performance tradeoffs.

I. Introduction

All modern computing devices—from smartphones to datacenters—must manage energy consumption. Energy-performance tradeoffs on mobile devices have existed for some time, such as dynamic voltage and frequency scaling (DVFS) for CPUs and the choice between more (Wifi) and less (mobile data) energy-efficient network interfaces. But as smartphone users continue to report battery lifetime as both their top concern and a growing problem [32], smartphone designs are providing even more energy-performance tradeoffs, such as the heterogeneous cores provided by ARM's `big.LITTLE` [16] architecture. Still other hardware energy-performance tradeoffs are on the horizon, arising from capabilities such as memory frequency scaling [8] and nanosecond-speed DVFS emerging in next-generation hardware designs [21].

We envision a next-generation smartphone capable of both CPU and memory DVFS. While the addition of memory DVFS can be used to improve energy-constrained performance, the larger frequency state space compared to CPU DVFS alone also provides more incorrect settings that waste energy or degrade performance. To better understand these systems, we characterize how the most performant CPU and memory frequency settings change for multiple workloads under various energy constraints.

Our work represents two advances over previous efforts. First, while previous works have explored energy minimizations using DVFS under performance constraints focusing on reducing slack, we are the first to study the potential DVFS settings under an energy constraint. Specifying performance constraints for servers is appropriate, since they are both wall-powered and have terms of service that must be met. Therefore, they do not have to and cannot afford to sacrifice too much performance. However, for mobile systems it is more critical to save energy as battery lifetime is the major concern. Therefore, we argue that optimizing performance under given *energy constraint* is fitting for mobile systems. We introduce a new metric *inefficiency* that can be used to specify energy constraints and it is both application and device independent—unlike existing metrics.

Second, we are the first to characterize optimal frequency settings for systems providing both CPU and memory DVFS. We find that closely tracking the optimal settings during execution produces many transitions and large frequency transition overhead. However, by accepting a certain amount of performance loss, the number of transitions and the corresponding overhead can be reduced. We characterize the relationship between the amount of performance loss and the rate of tuning for several benchmarks, and introduce the concepts of *performance clusters* and *stable regions* to aid the process.

We make following four contributions:

1) We introduce a new metric, *inefficiency*, that allows the system to express the amount of extra energy that can be used to improve performance.
2) We study the energy-performance trade-offs of systems that are capable of both CPU and memory DVFS for multiple applications. We show that poor frequency selection can both hurt performance and energy consumption.
3) We characterize the optimal frequency settings for multiple applications and inefficiency budgets. We introduce *performance clusters* and *stable regions* to reduce tuning overhead when a small degradation in performance is allowed.
4) We study the implications of using performance clusters on energy management algorithms.

We use the `gem5` simulator, the Android smartphone platform and Linux kernel, and an empirical power model to (1) measure the inefficiency of several applications for a wide range of frequency settings, (2) compute performance clusters, and (3) study how they evolve. We are currently constructing a complete system to study tuning algorithms that can build on our insights to adaptively choose frequency settings at runtime.

The rest of our paper is structured as follows. Section II introduces the inefficiency metric, while Section III describes our system, energy model, and experimental methodology. Section IV studies the impact of CPU and memory frequency scaling on the performance and energy consumption of multiple applications, while Section V characterizes the best frequency settings for various phases of the applications. As tracking the best settings is expensive, Section VI introduces *performance clusters*, and *stable regions* and studies their characteristics. Section VII presents implications of using performance clusters on energy-management algorithms, and Section VIII concludes.

II. Inefficiency

Extending battery life is critical for mobile systems, and therefore energy management algorithms for mobile systems should optimize performance under *energy constraints*. While several researchers have proposed algorithms that work under energy constraints [35], [38], these approaches require that the constraints are expressed in terms of absolute energy. For example, rate-limiting approaches [35] take the maximum energy that can be consumed in a given time period as an input. Once the application consumes its limit, it is paused until the next time period begins.

Unfortunately, in practice it is difficult to choose absolute energy constraints appropriately for a diverse group of applications without understanding their inherent energy needs. Energy consumption varies across applications, devices, and operating conditions, making it impractical to choose an absolute energy budget. Also, absolute energy constraints may slow down applications to the point that total energy consumption *increases* at the same time that performance is degraded.

Other metrics that incorporate energy take the form of $Energy * Delay^n$. We argue that while the energy-delay product can be used as a *measure* to gauge energy-performance trade-offs, it is not a suitable *constraint* to specify how much energy can be used to improve performance. A effective constraint should be (1) relative to the applications inherent energy needs and (2) independent of applications and devices. Because it uses absolute energy, the energy-delay product meets neither of these criteria.

We propose a new metric called *inefficiency*, which constrains how much extra energy an application can use to improve performance. Energy efficiency is defined as the work done per unit energy. Therefore, the application is said to be most efficient when it consumes the minimum energy possible on the given device. The application becomes inefficient as it starts consuming more than the minimum energy it requires. We define the ratio of application's energy consumption (E) and the minimum energy the application could have consumed (E_{min}) on the same device as inefficiency: $I = \frac{E}{E_{min}}$. An *inefficiency* of 1 represents an application's most efficient execution, while 1.5 indicate the the application consumed 50% more energy that its most efficient execution. Inefficiency is independent of workloads and devices and avoids the problems inherent to absolute energy constraints.

Four questions arise when using inefficiency to establish energy constraints for real systems:

1) What are the bounds of inefficiency?
2) How is the inefficiency budget set for a given application?
3) How is inefficiency computed?
4) How should systems stay within the inefficiency budget?

We continue by addressing these questions.

A. Inefficiency Bounds and Inefficiency Budget

Devices will operate between an inefficiency of 1 and I_{max} which represents the unbounded energy constraint allowing the application to consume unbounded energy to deliver the best performance. I_{max} depends upon applications and devices. We argue that absolute value of I_{max} is irrelevant because, when energy is unconstrained, algorithms can burn unbounded energy and only focus on delivering the best performance. The inefficiency budget matters the most when application has bounded energy constraints and it can be set by the user or the applications. The OS can also set the inefficiency budget based on application's priority allowing the higher priority applications to burn more energy than lower priority applications. While higher inefficiency values represent looser energy constraints, this does not guarantee higher performance. It is the responsibility of the energy management algorithms to provide best performance under a given inefficiency budget.

B. Computing Inefficiency

Once the system specifies an inefficiency budget, the energy management algorithms must tune the system to stay within the inefficiency budget while delivering the best performance. To compute inefficiency, we need both the energy (E) consumed by the application and the minimum energy (E_{min}) that application could have consumed. Computing E is straight forward; Intel Sandy bridge architecture [18] already provides performance counters capable of measuring energy consumption at runtime and the research community has tools and models to estimate the absolute energy of applications [4], [6], [25], [29], [37].

Computing E_{min} is challenging because of the inter-component dependencies. We propose two methods for computing E_{min}:

- **Brute force search:** E_{min} can be estimated using the power models (or tools) for a given workload at all possible system settings. The minimum of all these estimations is E_{min}. While the overhead of this approach is high, it could be improved with a lookup table.
- **Predicting and learning:** The overhead of the E_{min} computation can be further reduced by predicting E_{min} based on previous observations and learning continuously. A variety of learning based approaches [24] have been proposed in the past to estimate various metrics and application phases which can be applied to E_{min} estimation as well.

We are working towards designing efficient energy prediction models for CPU, memory and network components. Our models consider cross-component interactions on performance and energy consumption. In this work we demonstrate how to use inefficiency, deferring predicting and optimizing E_{min} to future work.

C. Managing Inefficiency

Future energy management algorithms need to tune system settings to keep the system within specified inefficiency budget and deliver the best performance. Techniques that use predictors such as instructions-per-cycle (IPC) to decide when to use DVFS or migrate threads can be extended to operate under given inefficiency budget [1], [19], [20], [34]. Efforts that have

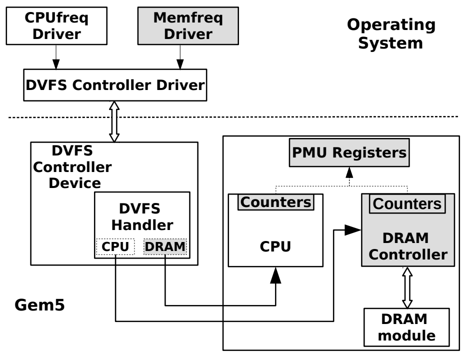


Fig. 1: **System Block Diagram**: Blocks that are newly added or significantly modified from Gem5 origin implementation are shaded.

tried to optimize memory energy consumption can be adapted to use inefficiency as a constraint to their system [8], [10], [11], [12], [23], [27], [36], [39]. While most of the existing multi-component energy management approaches work under performance constraints, some have potential to be modified to work under energy constraints and thus inefficiency [3], [9], [7], [13], [14], [26], [33]. We leave building some of these algorithms into a system as future work. In this paper, we characterize the optimal performance point under different inefficiency constraints and illustrate that the stability of these points have implications for future algorithms.

III. System and Methodology

Energy management algorithms must tune the underlying hardware components to keep the system within the given inefficiency budget. Hardware components provide multiple knobs that can be tuned to trade-off performance for energy savings. For example, the energy consumed by the CPU can be managed by tuning its frequency and voltage. Recent research [8], [11] has shown that DRAM frequency scaling also provides performance and energy trade-offs.

In this work, we scale frequency and voltage for the CPU and scale only frequency for memory. Dynamic Frequency Scaling (DFS) for memory has emerged as a means to trade-off performance for energy savings. As no current hardware systems support memory frequency scaling, we resort to Gem5 [2], a cycle-accurate full system simulator to perform our studies.

A. System Overview

Current Gem5 versions provide the infrastructure necessary to change CPU frequency and voltage; we extended Gem5 DVFS to incorporate memory frequency scaling. As shown in Figure 1, Gem5 provides a DVFS controller device that provides interface to control frequency by the OS at runtime. We developed a memory frequency governor similar to existing Linux CPU frequency governors. The blocks that we added or significantly modified from Gem5's original implementation are shaded in Figure 1.

B. Energy Models

We developed energy models for the CPU and DRAM for our studies. Gem5 comes with the energy models for various DRAM chipsets. The DRAMPower [6] model is integrated into Gem5 and computes the memory energy consumption periodically during the benchmark execution. However, Gem5 lacks a model for CPU energy consumption. We developed a processor power model based on empirical measurements of a PandaBoard [31] evaluation board. The board includes a OMAP4430 chipset with a Cortex A9 processor; this chipset is used in the mobile platform we want to emulate, the Samsung Nexus S. We ran microbenchmarks designed to stress the Pandaboard to its full utilization and measured power consumed using an Agilent 34411A multimeter. Because of the limitations of the platform, we could only measure peak dynamic power. Therefore to model different voltage levels we scaled it quadratically with voltage and linear with frequency $(P \propto V^2 f)$. Our peak dynamic power agrees with the numbers reported by previous work [5] and the datasheets.

We split the power consumption into three categories: dynamic power, background power, and leakage power. Background power is consumed by idle units when the processor is not computing, but unlike leakage power, background power scales with clock frequency. We measure background power by calculating the difference between the CPU power consumption in its power on idle state and deep sleep mode (not clocked). Because background power is clocked, it is scaled in a similar manner to dynamic power. Leakage power comprises up to 30% of microprocessor peak power consumption [15] and is linearly proportional to supply voltage [30].

C. Experimental Methodology

Our simulation infrastructure is based on Android 4.1.1 "Jelly Bean" run on the Gem5 full system simulator. We model a Cortex-A9 processor, single core, out-of-order CPU with an issue width of 8, L1 cache size of 64 KB with access latency of 2 core cycles and a unified L2 cache of size 2 MB with hit latency of 12 core cycles. The CPU and caches operate under the same clock domain. For our purposes, we have configured the CPU clock domain frequency to have a range of 100–1000 MHZ with highest voltage being 1.25V.

For the memory system, we simulated a LPDDR3 single channel, one rank memory access using an open-page policy. Timing and current parameters for LPDDR3 are configured as specified in micron data sheet [28]. Memory clock domain is configured with a frequency range of 200MHz to 800MHz. As mentioned earlier, we did not scale memory voltage. The power supplies—VDD and VDD2—for LPDDR3 are fixed at 1.8V and 1.2V respectively.

We first simulated 12 integer and 9 floating point SPEC CPU2006 benchmarks [17], with each benchmark either running to completion or up to 2 billion instructions. We booted the system and then changed CPU and memory frequency using userspace frequency governors before starting the benchmark. We ran 70 simulations for each benchmark, with a combination of 10 CPU and 7 memory frequency steps using step size of 100MHz. To study the finer details of workload phases, we then ran a total of 496 simulations with a finer step granularity of 30MHz for CPU and 40MHz for memory for selected benchmarks that have interesting and unique phases. Due to limited resources and time, running simulations for all benchmarks with finer frequency steps was difficult as it

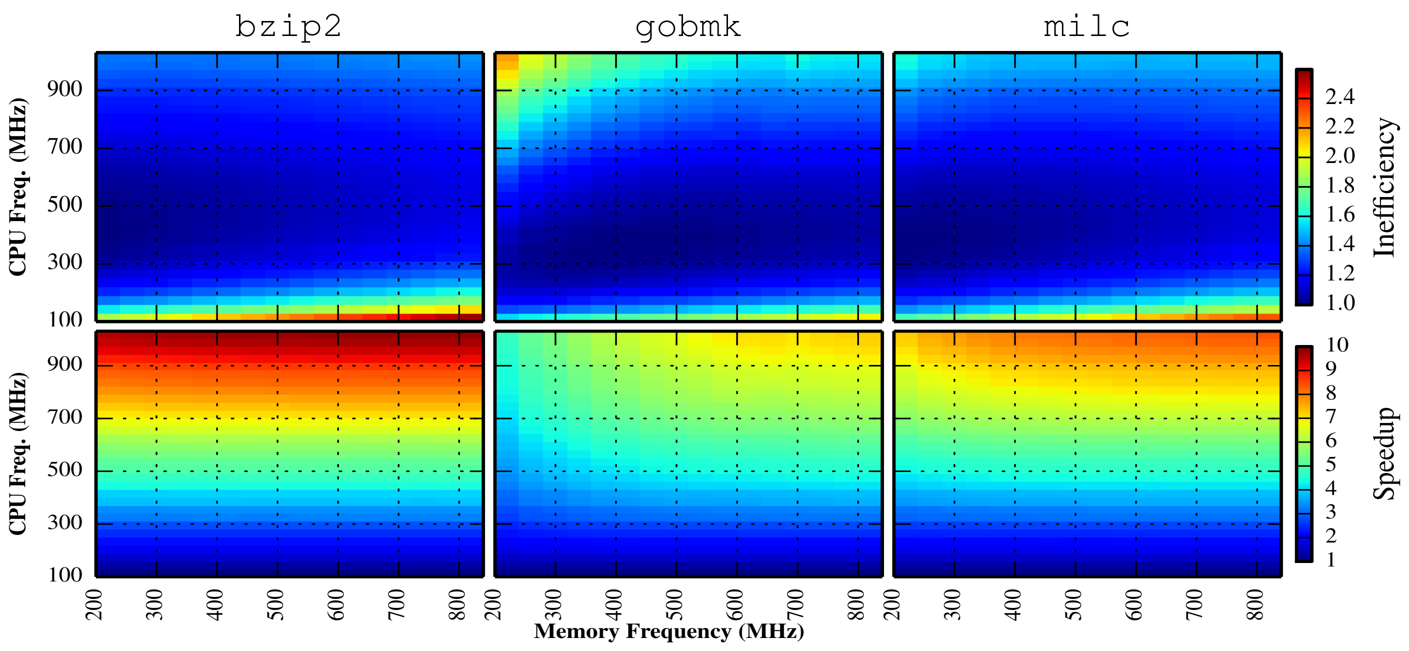


Fig. 2: **Inefficiency vs. Speedup For Multiple Applications:** In general, performance improves with increasing inefficiency budgets. A poorly designed algorithm may select bad frequency settings which could waste energy and degrade performance simultaneously.

would have resulted in more than 10,000 simulations, where each simulation would take anywhere between 4 to 12 hours.

We collected samples of a fixed amount of work so that each sample would represent the same work even across different frequencies. In gem5, we collectd performance and energy consumption data every 10 million user mode instructions. Gem5 provides a mechanism to distinguish between user mode and kernel mode instructions. We used this feature to remove periodic OS traffic and enable a fair comparison across simulations of different CPU and memory frequencies. We used the collected performance and energy data to study the impact of workload dynamics on the stability of CPU and memory frequency settings delivering best performance under a given inefficiency budget. Note that, all our studies are performed using *measured* performance and power data from the simulations, we do not *predict* performance or energy.

Although individual energy-performance trade-offs of DVFS for CPU and DFS for memory have been studied in the past, the trade-off resulting from the cross-component interaction of these two components has not been characterized. CoScale [9] did point out that interplay of performance and energy consumption of these two components is complex and did present a heuristic that attempts to pick the optimal point. However, it did not measure and characterize the larger space of all system level performance and energy trade-offs of various CPU and memory frequency settings.

IV. Inefficiency vs. Speedup

Scaling individual components—CPU and memory—using DVFS has been studied in the past to make power performance trade-offs. To the best of our knowledge, the prior work has not studied the system level energy-performance trade-offs of combined CPU and memory DVFS. We take a first step and explore these trade-offs and show that incorrect frequency settings may burn extra energy without improving performance.

We performed offline analysis of the data collected from our simulations to study the inefficiency-performance trends for various benchmarks. With a brute force search, we found E_{min} and computed inefficiency at all frequency settings. We express performance in terms of *speedup*, the ratio of execution time for a given configuration to the longest execution time.

Figure 2 plots the speedup and inefficiency for three workloads operating with various CPU and memory frequencies. As the figure shows, the ability of a workload to trade-off energy and performance using CPU and memory frequency, depends on its mix of CPU and memory instructions. For CPU intensive workloads like *bzip2*, speedup varies with only CPU frequency, and memory frequency has no impact on speedup. For workloads that have balanced CPU and memory intensive phases like *gobmk*, speedup varies with both CPU and memory frequency. The *milc* benchmark has some memory intensive phases, however it is more CPU intensive and therefore its performance is more dependent upon CPU frequency than memory frequency. We make three major observations:

Running slower doesn't mean that system is running efficiently. At the lowest frequencies, 100MHz and 200MHz for CPU and memory respectively, *gobmk* takes the longest to execute. These settings slow down the application so much that its overall energy consumption increases, thereby resulting in 1.55 inefficiency for *gobmk*. Algorithms that choose these frequency settings spend 55% more energy without any performance improvement.

Higher inefficiency doesn't always result in higher performance: gobmk is fastest at 1000MHz for CPU and 800MHz for memory frequency. It runs at inefficiency of 1.65 at these frequency settings. Allowing *gobmk* to run at higher

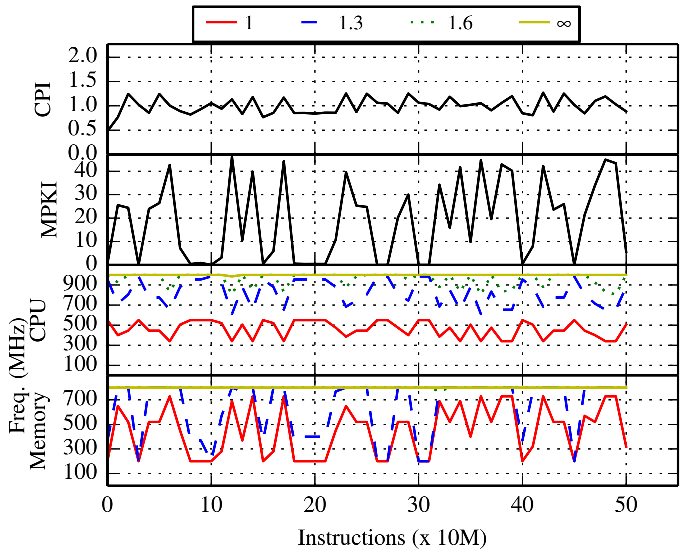


Fig. 3: **Optimal Performance Point for Gobmk Across Inefficiencies:** At low inefficiency budgets, the optimal frequency settings follow CPI of the application, and select high memory frequencies for memory intensive phases with high CPI. Higher inefficiency budgets allow the system to run always at the maximum CPU and memory frequencies.

inefficiency of say 2.2, doesn't improve performance. In fact, any algorithms that force the application to consume all of the given energy budget may degrade application performance. For example, *gobmk* runs 1.5x slower if it is forced to run at budget of 2.2 at 1000MHz and 200MHz of CPU and memory frequencies respectively.

Smart algorithms should search for optimal points ***under*** *the inefficiency constraint and* ***not*** *just* ***at*** *the inefficiency constraint.* Algorithms forcing the system to run exactly at given budget might end up wasting energy or, even worse, degrading performance. A smart algorithm should a) use no more than given inefficiency budget b) should use only as much inefficiency budget as needed c) and deliver the best performance.

Consequently, like other constraints used by algorithms such as performance, power and absolute energy, $inefficiency$ also allows energy management algorithms to waste system energy. We suggest that, even though $inefficiency$ doesn't completely eliminate the problem of wasting energy, it mitigates the problem. For example, rate limiting approaches waste energy as energy budget is specified for a given amount of time interval and doesn't require a specific amount of work to be done within that budget. However, inefficiency mandates the underlying algorithms to complete the given amount of work under the constraint.

V. Performance Under An Inefficiency Budget

In this section we study the characteristics of the best performing CPU and memory frequency settings, *optimal settings*, across different inefficiency constraints and how they change during application execution. To find the optimal settings, we wrote a simple algorithm that first filters all possible frequency settings under given inefficiency budget. It then finds the CPU and memory frequency settings that result in highest speedup. In cases where multiple settings result in similar speedup (within 0.5%), to filter out simulation noise, the algorithm selects the settings with highest CPU (first) and memory frequency as this setting is bound to have highest performance among the other possibilities.

Figure 3 plots the optimal settings for Gobmk for all benchmark samples (each of length 10 million instructions) across multiple inefficiency constraints. At low inefficiencies, the optimal settings follow the trends in CPI (cycles per instruction) and MPKI (misses per thousand instructions). Regions of higher CPI correspond to memory intensive phases, as the SPEC benchmarks don't have any IO or interrupt based portions. For phases that are CPU intensive with (lower CPI), the optimal settings have higher CPU frequency and lower memory frequency. At low inefficiency constraints, due to the limited energy budget, a careful allocation of energy across components becomes critical to achieve optimal performance. Higher inefficiencies allow the algorithms to select higher frequency settings in order to achieve greater speedup. We define unconstrained inefficiency (labeled ∞) as the scenario in which the algorithm always chooses the highest frequency settings as these settings always deliver the highest performance. There are two key problems associated with tracking the optimal settings:

It is expensive. Running the tuning algorithm at the end of every sample to track optimal settings comes at a cost: 1) searching and discovering the optimal settings 2) real hardware has transition latency overhead for both the CPU and the memory frequency. Reducing the frequency at which tuning algorithms need to re-tune is critical to reduce the cost of tuning overhead on application performance.

Limited energy performance trade-off options. Choosing the optimal settings for every sample may hinder some energy-performance trade-off that could have been made if performance was not so tightly bounded (to only highest performance). For example, *bzip2* is CPU bound and therefore its performance at memory frequency of 200MHz is within 3% of performance at a memory frequency of 800MHz while CPU is running at 1000MHz. By sacrificing that 3% of performance, the system could have consumed 1/4 the memory background energy staying well under the given inefficiency budget.

We believe that, if the user is willing to sacrifice some performance under given inefficiency budget, algorithms would be able to make better trade-offs between the cost of frequent tuning and performance.

VI. Performance Clusters

Tracking the best performance settings for a given inefficiency budget is expensive. In this section, we study how we can amortize the cost by trading-off some performance. We define the concept of *performance clusters*. All frequency settings (CPU and memory frequency pairs) that have performance within a performance degradation threshold (*cluster threshold*) compared to the performance of the optimal settings for a given inefficiency budget form the *performance cluster* for that inefficiency constraint. We define the term *stable regions* as regions in which at least one pair of CPU and memory frequency settings is common among all samples in the region.

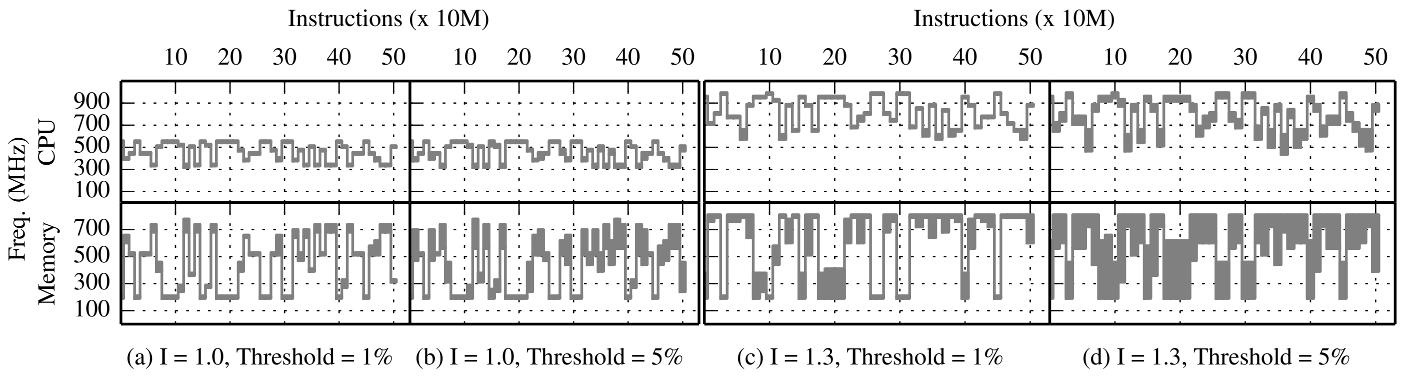


Fig. 4: **Performance Clusters for *gobmk*.** With increase in cluster threshold, the number of available frequency settings increase, eventually leading to fewer transitions. Increase in cluster size with inefficiency budget is a function of workload.

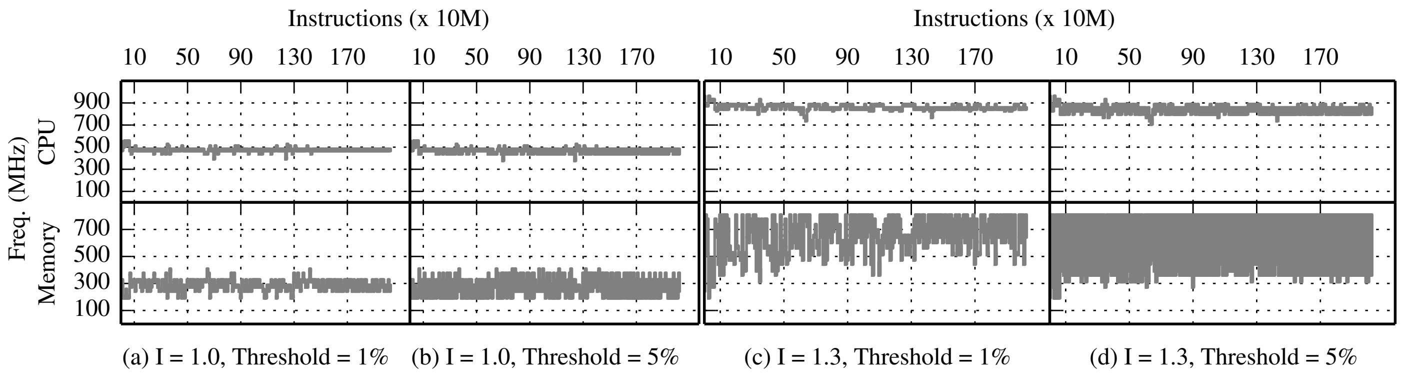


Fig. 5: **Performance Clusters of *milc*.** *Milc* is CPU intensive to a large extent with some memory intensive phases. At higher thresholds, while CPU frequency is tightly bound, performance clusters cover a wide range of memory settings due to small performance difference across these frequencies.

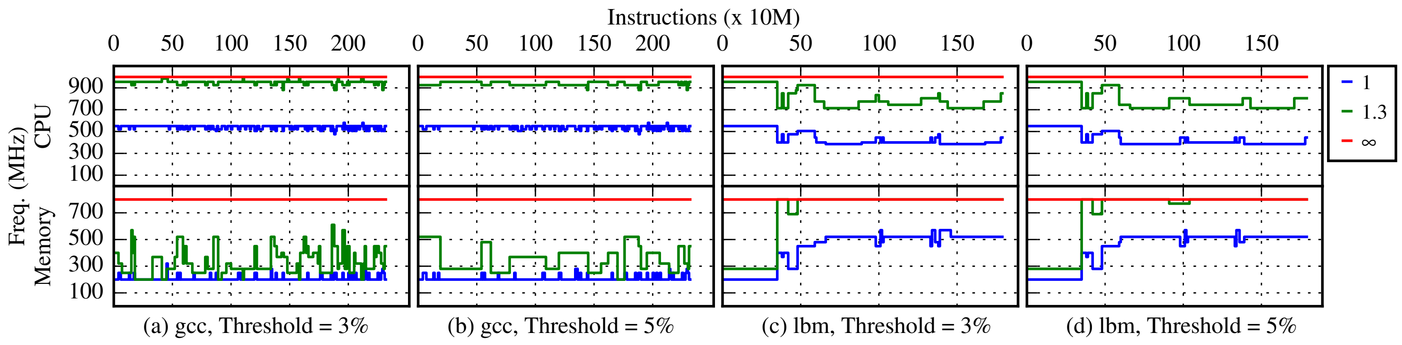


Fig. 7: **Stable Regions of *gcc* and *lbm* for Inefficiency Budget of 1.3:** Increase in cluster threshold increases the length of the stable regions, which eventually leads to less transitions. Higher inefficiency budgets allow system to run unconstrained throughout.

In this section, we first study the trends in performance clusters for multiple applications. Then we characterize the stable regions and explore the implications of using stable regions on energy-performance trade-offs for multiple inefficiencies and cluster thresholds. In the end, we study the sensitivity of performance clusters to number of frequency settings available in the system.

A. Performance Clusters

We search for the performance clusters using an algorithm that is similar to the approach we used to find the optimal settings. We first filter the settings that fall within a given inefficiency budget, and then search for the optimal settings in the first pass. In the second pass, we find all of the settings that have a speedup within the specified *cluster threshold* of the optimal performance.

Figures 4, 5 plot the performance clusters during the execution of the benchmarks *gobmk* and *milc*. We plot inefficiency budgets of 1 and 1.3 and cluster thresholds of 1% and 5%. For our benchmarks, we observed that the maximum achievable inefficiency is anywhere from 1.5 to 2. We chose inefficiency budgets of 1 and 1.3 to cover low and mid inefficiency budgets. Cluster thresholds of 1% and 5% allow us to model the two extremes of tolerable performance degradation bounds. A cluster threshold of less than 1% may limit the ability to tune less often. While cluster thresholds greater than 5% are probably not realistic as user is already compromising performance by setting low inefficiency budgets to save energy.

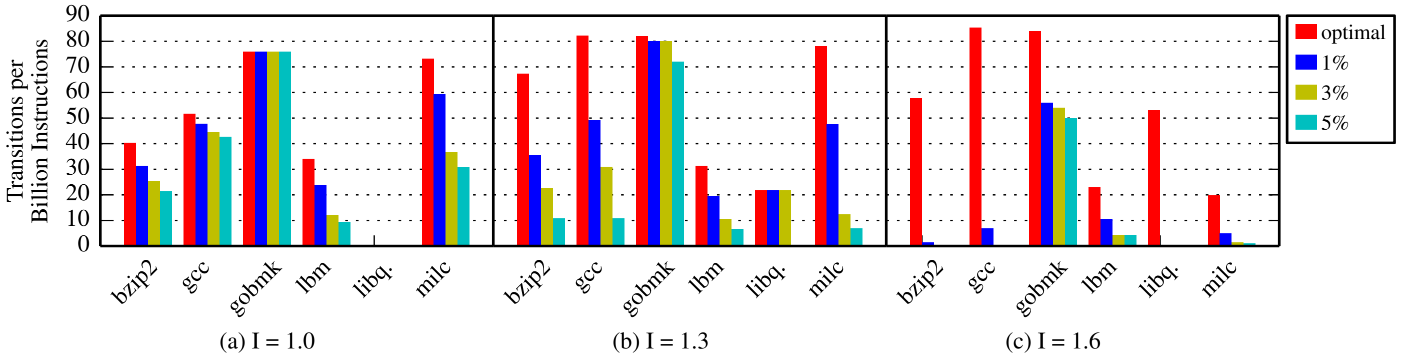


Fig. 8: **Number of Transitions with Varying Inefficiency Budgets and Cluster Thresholds:** The number of frequency transitions decrease with increase in cluster threshold. The amount of change varies with benchmark and inefficiency budget.

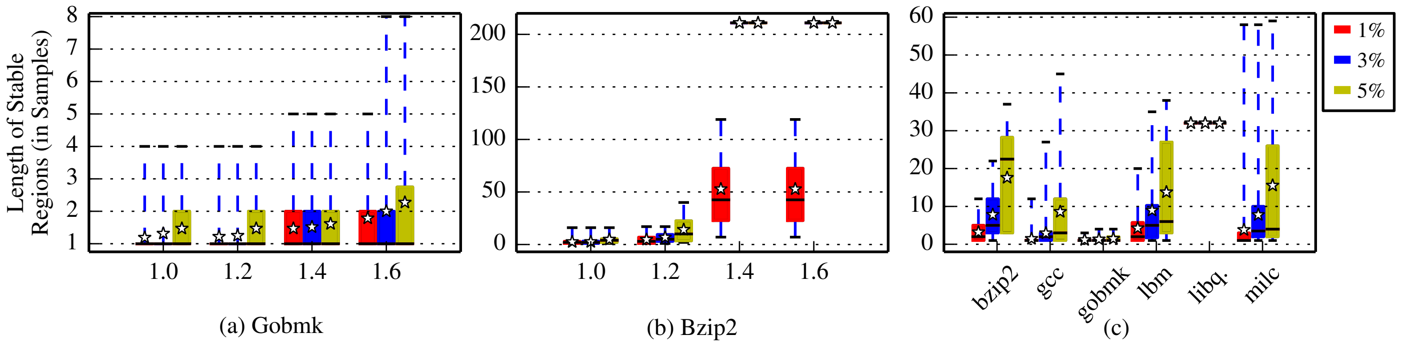


Fig. 9: **Distribution of Length of Stable Regions:** The average length of stable regions increases with cluster threshold.

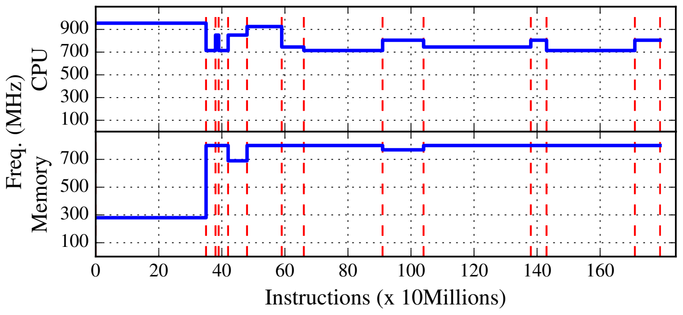


Fig. 6: **Stable Regions and Transitions for *Lbm* with Threshold of 5% and Inefficiency Budget of 1.3:** Solid lines represent the stable regions and vertical dashed lines mark the transitions made by *lbm*.

Figures 4(c), 4(d) plot the performance clusters for *gobmk* for inefficiency budget of 1.3 and cluster thresholds of 1% and 5% respectively. As we saw in Figure 3, the optimal settings for *gobmk* change every sample (of length 10 million instructions) and follows application phases (CPI). Figure 4(c) shows that by allowing just 1% performance degradation, the number of settings available to choose from increase. For example, for sample 11, the optimal settings were at 1000MHz CPU and 500MHz memory. With 1% cluster threshold, the range of available frequencies increases to 970MHz-1000MHz for CPU and 420MHz-580MHz for memory. With a 5% cluster threshold, the range of available frequencies increases further as shown in Figure 4(d). With an increase in number of available settings, the probability of finding common settings in two consecutive samples increases, allowing the system to stay at one frequency setting for a longer time. For example, the optimal settings changed between samples 24 and 25, however with cluster threshold of 5% CPU and memory frequency can stay fixed at 750MHz and 800MHz respectively. The higher the cluster threshold is, the higher the length of the stable regions would be.

Figures 4(a), 4(c) plot the performance clusters for *gobmk* for two different inefficiency budgets of 1.0 and 1.3 for cluster threshold of 1%. Not all of the stable regions increase in length with increasing inefficiency but instead depends on the workload. If consecutive samples of a workload have a small difference in performance but differ significantly in energy consumption then only at higher inefficiency budgets will the system find common settings for these consecutive samples. This is because, the performance clusters of higher inefficiencies can include settings operating at lower inefficiencies as long as their performance degradation is within the cluster threshold. For example, the memory frequency oscillates for samples 32-39 for *gobmk* at inefficiency budget of 1.0, while the system could stay fixed at 800MHz memory at inefficiency of 1.3. However, for workload phases that result in high performance difference in consecutive samples at given pair of frequency settings, higher inefficiency budgets might not help as there might not be any common frequency pairs that have performance within set cluster threshold (for example samples 3-5 in Figures 4(a), 4(c)). Figure 5 shows that *milc* has similar trends as *gobmk*.

An interesting observation from the performance clusters is that algorithms like CoScale [9] that search for the best performing settings every interval starting from the maximum frequency settings are not optimal. Algorithms can reduce the overhead of optimal settings search by starting search from the settings selected for the previous interval as application phases are often stable for multiple sample intervals.

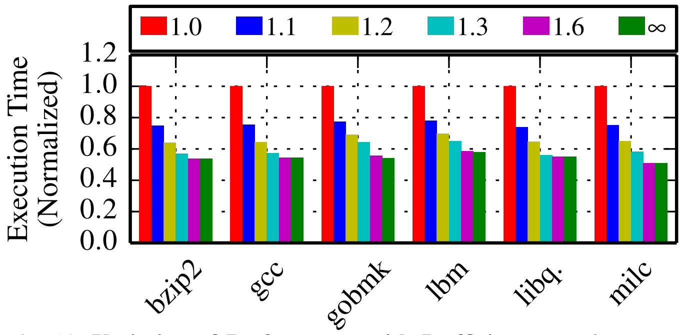


Fig. 10: **Variation of Performance with Inefficiency:** Performance improves with increase in inefficiency budget, but the amount of improvement varies across benchmarks.

B. *Stable Regions*

So far, we have made observations by looking at CPU and memory frequencies separately and finding (visually) where they *both* stay stable. However, when either of the memory or CPU performance clusters move, the system needs to make a transition. Looking at plots of individual components does not provide a clear picture of the stable regions of the entire CPU and memory system. Figures 4 and 5 plot the performance clusters and not the stable regions.

We wrote an algorithm to find all of the stable regions for a given application. It starts by computing performance clusters for a given sample and moves ahead sample by sample. For every sample it computes available settings by finding the common settings between the current sample performance cluster and the available settings until the previous sample. When the algorithm finds no more common samples, it marks the end of the stable region. If more than one frequency pair exists in the available settings for this region, the algorithm chooses the setting with highest CPU (first) and memory frequency as optimal settings for this region. Figure 6 shows the CPU and memory frequency settings selected for stable regions of benchmark *lbm*. It also has markers indicating the end of each stable region. In this figure, note that for every stable region (between any two markers) the frequency of both CPU and memory stay constant.

Our algorithm is not practical for real systems, it knows the characteristics of the future samples and their performance clusters in the beginning of a stable region.We are currently designing algorithms that are capable of tuning the system while running the application as future work. In Section VII, we propose ways in which length of stable regions and the available settings for a given region can be predicted for energy management algorithms in real systems.

Figure 7 plots stable regions for benchmarks *gcc* and *lbm* for multiple inefficiency budgets and cluster thresholds. With increase in cluster threshold from 3% to 5% there is a significant drop in the number of transitions made by *gcc* at lower inefficiency budgets. At higher inefficiency budgets, algorithms can choose the highest available frequency settings and therefore, even if a higher cluster threshold is allowed, we don't observe any changes in the selection of settings. The relative number of transitions made by *lbm* decreases with an increase in cluster threshold, however, the absolute number of transitions compared to other benchmarks does not decrease significantly as it doesn't have too many transition to start with at 3%. Like our previous observation, the number of transitions also decreases with increase in inefficiency for these two benchmarks showing that there is a high number of consecutive samples that have similar performance but different inefficiency at same CPU and memory frequency settings. A decrease in the number of transitions is a result of an increase in the length of stable regions.

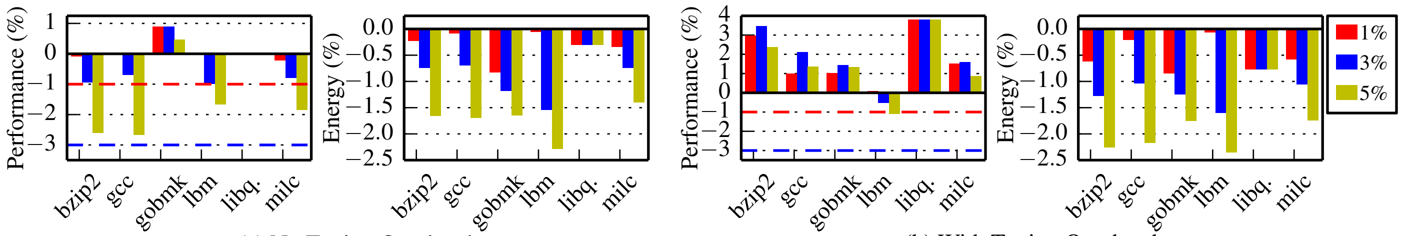
Figure 8 summarizes the number of transitions per billion instructions for multiple cluster thresholds and inefficiency budgets across benchmarks. As the figure shows, tracking the optimal frequency settings results in highest number of transitions. A common observation is that the number of transitions required decreases with an increase in cluster threshold. For *bzip2*, increase in inefficiency from 1.0 to 1.3 increases the number of transitions needed to track the optimal settings. The number of available settings increase with inefficiency, giving more choices for the optimal frequency settings. At an inefficiency budget of 1.6, the average length of a stable region increases drastically as shown in Figure 9(b), which requires much less transitions with 1% cluster threshold and no transitions with higher cluster thresholds of 3% and 5%. Note that there is only one point on the box plot for 3% and 5% cluster thresholds at inefficiency of 1.6, because the benchmark is covered entirely by only one region. However, *gobmk* has rapidly changing phases and therefore, with an increase in either inefficiency or cluster thresholds, there is not much of an increase that we observe in stable region lengths as shown in Figure 9(a). Therefore the number of transition per billion instructions decrease only slightly with increase in cluster threshold and inefficiency budget for *gobmk*. Figure 9(c) summarizes the distribution of stable region lengths observed across benchmarks for multiple cluster thresholds at inefficiency budget of 1.3.

C. *Energy-Performance Trade-offs*

In this subsection we analyze the energy-performance trade-offs made by our ideal algorithm. We then add tuning cost of our algorithm and compare the energy performance trade-offs across multiple applications. We study multiple cluster thresholds and an inefficiency budget of 1.3.

First, we demonstrate that our tuning algorithm was successful in selecting the right settings and thereby keeping the system under the specified inefficiency budget and summarize the total performance achieved. We ran a set of simulations and verified that all applications ran under the given inefficiency budget for all the inefficiency budgets. Figure 10 shows that the higher the inefficiency budget is, the lower the execution time is, making smooth energy-performance trade-offs.

Figure 11 plots the total performance degradation and energy savings for multiple cluster thresholds with and without tuning overhead for inefficiency budget of 1.3. Both performance degradation and energy savings are computed relative to the performance and energy of the application running at optimal settings. Figures 11(a) and 11(b) show that our algorithm is successful in selecting the settings that don't degrade performance more than specified cluster threshold. The figure also shows that with an increase in cluster threshold, energy consumption decreases because lower frequency settings can be chosen at higher cluster thresholds. Figure 11(b) shows that



(a) No Tuning Overhead (b) With Tuning Overhead

Fig. 11: **Energy-performance Trade-offs for Inefficiency Budget of 1.3, Multiple Cluster Thresholds:** Performance degradation is always within the cluster threshold. Allowing small degradation in performance reduces energy consumption, which decreases further when tuning overhead is included.

performance (and energy) may improve when tuning overhead is included due to decrease in frequency transitions. We assume tuning overhead of 500us and 30uJ, which includes computing inefficiencies, searching for the optimal setting and transition the hardware to new settings [9]. We assumed that a space of 100 settings is searched for every transition. *gobmk* is the only benchmark that shows a performance improvement from the optimal settings when performance is allowed to degrade, which is unexpected. We are investigating its root cause.

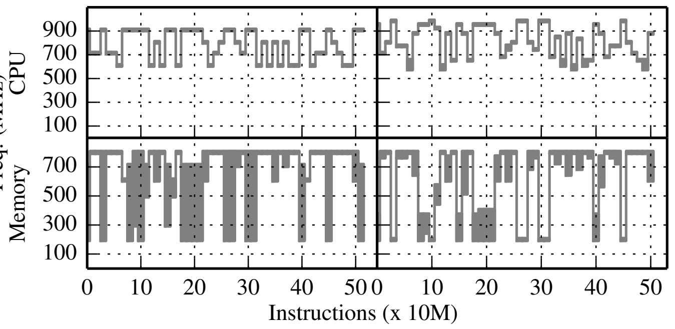
We summarize our observations from this section here:

1) With an increase in cluster threshold, the range of available frequencies increases, which increases the probability of finding common settings in consecutive samples. This results in longer stable regions.
2) The increase in the length of stable regions with an increase in inefficiency depends on the workload.
3) The number of transitions required is dictated by the average length of the stable region. The longer the stable regions, the lower the number of transitions that the system need to make.
4) Allowing a higher degradation in performance may, in fact, result in improved performance when tuning overhead of algorithms is included due to reduction in number of frequency transitions in the system. Consequently energy savings also increase.

D. Sensitivity of Performance Clusters to Frequency Step Size

In this section we study the sensitivity of the performance clusters to number of frequency steps or frequency step sizes available in a given system. We computed performance clusters offline and analyzed the difference between clusters with coarse frequency steps and a clusters with more frequency steps.

Figure 12 plots performance clusters for *gobmk* at inefficiency of 1.3 and cluster threshold of 1%. We chose 1% for our sensitivity analysis, as the trends in performance clusters are more explicit at low cluster thresholds. Figure 12(a) plots the clusters collected with a 100MHz frequency step for both the CPU and the memory, which is a total of 70 possible settings. The clusters in Figure 12(b) are collected with 30MHz steps for CPU frequency and 40MHz steps for memory frequency, for a total of 496 settings. We observed that the average cluster length either remains the same or decreases with increase in number of steps. With increase in number of frequency steps, there are more choices available to make better energy-performance trade-offs. Therefore average number of samples



(a) 70 settings (b) 496 settings

Fig. 12: **Performance Clusters at Two Different Frequency Steps**: Figure (a) plots performance clusters collected using 100MHz of frequency step for both CPU and memory. Figure (b) plots performance clusters collected using frequency steps of 30MHz for CPU and 40MHz for memory.

for which one setting can be chosen decreases. For example, with 70 frequency settings sample 7 through sample 10 can always run at CPU frequency of 900MHz and memory frequency of 300MHz. With 496 frequency settings, sample 7 runs at one setting, sample 8-9 runs at another setting and sample 10 runs at a different setting due to the availability of more (and better) choices. In our system, we observed only a small improvement in performance (<1%) with higher number of frequency steps when tuning is free as optimal settings in both cases were off by only a few MHz. It is the balance between the tuning overhead and the energy-performance savings that is critical in deciding the correct size of the search space.

VII. Algorithm Implications

Higher cluster thresholds indeed result in lower transition overheads by reducing the number of transitions required. One may wonder, however, if the thresholds have an impact on the overhead of energy management algorithms. In other words, how may higher thresholds reduce the overheads of searching for the optimal settings or cluster of settings? We propose that at higher cluster thresholds, algorithms can choose to not run their search at the end of every interval. As shown in Section VI, higher cluster thresholds result in longer stable regions. Smart algorithms can leverage these long stable regions by tuning less often during these time intervals. We propose two ways in which this can be achieved.

1) *Learning:* Algorithms can use learning based approaches to predict when to run again. Isci. et. al [19] propose simple ways in which algorithms can detect how long the current

application phase is going to be stable and only choose to tune at the end of predicted phase for CPU performance. Similar approaches could be developed that extend this methodology to detect stable regions of clusters containing both memory and CPU settings.

2) *Offline Analysis:* Another approach that can be taken to reduce the number of tuning events is offline analysis of the applications. An application can be profiled once offline to identify regions in which the performance cluster is stable. The profiled information of the stable region lengths, positions, and available settings can then be used at run time to enable the system to predict how long it can go without tuning. Algorithms can also extend the usage of the profiled information to new applications that may have phases that match with already profiled data. Previous work has already proposed using offline analysis methods to detect application phases [22], which would be directly applicable here in our system.

VIII. Conclusion

In this work, we introduced the *inefficiency* metric that can be used to express amount of battery life that the user is willing to sacrifice to improve performance. We used DVFS for the CPU and DFS for the memory as a means to trade-off performance and save energy consumption. We demonstrated that, while individual performance-energy trade-offs of single components are intuitive, the interplay of just these two components on the energy and performance of applications is complex. Consequently, we characterized the optimal CPU and memory frequency settings across applications for multiple inefficiency budgets. We demonstrated that if the user is willing to sacrifice minimal performance under a given inefficiency budget, frequent tuning of the system can be avoided and the overhead of energy management algorithms can be mitigated.

As future work, we are working towards developing predictive models for performance and energy that consider cross-component interactions. We are designing algorithms that use these models for tuning systems at runtime. Eventually, we plan on designing a full-system that is capable of tuning multiple components simultaneously while executing applications.

References

[1] A. Bhattacharjee and M. Martonosi, "Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors," in International Symposium on Computer Architecture (ISCA), June 2009.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," SIGARCH Comput. Archit. News, vol. 39, no. 2, pp. 1–7, Aug. 2011. [Online]. Available: http://doi.acm.org/10.1145/2024716.2024718

[3] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2008, pp. 318–329.

[4] D. Brooks, V. Tiwari, and M. Martonosi, Wattch: a framework for architectural-level power analysis and optimizations. ACM, 2000, vol. 28, no. 2.

[5] G. Challen and M. Hempstead, "The case for power-agile computing," in Proc. 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII), May 2011.

[6] K. Chandrasekar, C. Weis, Y. L. S. . G. M. J. O. N. B. A. N. Wehnand, and . K. Goossens, "DRAMPower: Open-source DRAM Power & Energy Estimation Tool," http://www.drampower.info.

[7] M. Chen, X. Wang, and X. Li, "Coordinating processor and main memory for efficientserver power control," in Proceedings of the international conference on Supercomputing. ACM, 2011, pp. 130–140.

[8] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, "Memory power management via dynamic voltage/frequency scaling," in Proceedings of the 8th ACM international conference on Autonomic computing. ACM, 2011, pp. 31–40.

[9] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini, "Coscale: Coordinating cpu and memory system dvfs in server systems," in The 45th Annual IEEE/ACM International Symposium on Microarchitecture, 2012, 2012.

[10] ——, "Multiscale: memory system dvfs with multiple memory controllers," in Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design. ACM, 2012, pp. 297–302.

[11] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini, "Memscale: active low-power modes for main memory," ACM SIGPLAN Notices, vol. 46, no. 3, pp. 225–238, 2011.

[12] B. Diniz, D. Guedes, W. Meira Jr, and R. Bianchini, "Limiting the power consumption of main memory," in ACM SIGARCH Computer Architecture News, vol. 35, no. 2. ACM, 2007, pp. 290–301.

[13] X. Fan, C. S. Ellis, and A. R. Lebeck, "The synergy between power-aware memory systems and processor voltage scaling," in Power-Aware Computer Systems. Springer, 2005, pp. 164–179.

[14] W. Felter, K. Rajamani, T. Keller, and C. Rusu, "A performance-conserving approach for reducing peak power consumption in server systems," in Proceedings of the 19th annual international conference on Supercomputing. ACM, 2005, pp. 293–302.

[15] M. Floyd, M. Allen-Ware, K. Rajamani, B. Brock, C. Lefurgy, A. Drake, L. Pesantez, T. Gloekler, J. Tierno, P. Bose, and A. Buyuktosunoglu, "Introducing the adaptive energy management features of the power7 chip," Micro, IEEE, vol. 31, no. 2, pp. 60 –75, march-april 2011.

[16] P. Greenhalgh, "Big.little processing with arm cortex-a15 & cortex-a7: Improving energy efficiency in high-performance mobile platforms," in white paper, ARM Ltd., September 2011.

[17] J. L. Henning, "SPEC CPU2006 benchmark descriptions," ACM SIGARCH Computer Architecture News, vol. 34, no. 4, pp. 1–17, 2006.

[18] Intel, "Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide," 2009.

[19] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi, "Phases: Duration Predictions and Applications to DVFS," IEEE Micro, 2005.

[20] ——, "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget," in International Symposium on Microarchitecture (MICRO), December 2006.

[21] W. Kim, D. Brooks, and G.-Y. Wei, "A fully-integrated 3-level dc-dc converter for nanosecond-scale dvfs," Solid-State Circuits, IEEE Journal of, vol. 47, no. 1, pp. 206–219, 2012.

[22] J. Lau, E. Perelman, and B. Calder, "Selecting software phase markers with code structure analysis," in Code Generation and Optimization, 2006. CGO 2006. International Symposium on, March 2006, pp. 12 pp.–.

[23] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis, "Power aware page allocation," ACM SIGPLAN Notices, vol. 35, no. 11, pp. 105–116, 2000.

[24] J. Li, X. Ma, K. Singh, M. Schulz, B. R. de Supinski, and S. A. McKee, "Machine learning based online performance prediction for runtime parallelization and task scheduling," in Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on. IEEE, 2009, pp. 89–100.

[25] S. Li, J. H. Ahn, R. D. Strong, J. B. . Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on. IEEE, 2009, pp. 469–480.

[26] X. Li, R. Gupta, S. V. Adve, and Y. Zhou, "Cross-component energy management: Joint adaptation of processor and memory," ACM Transactions on Architecture and Code Optimization (TACO), vol. 4, no. 3, p. 14, 2007.

[27] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile dram," in Proceedings of the 39th International Symposium on Computer Architecture. IEEE Press, 2012, pp. 37–48.

[28] Micron, "16Gb:x8,LPDDR3 SDRAM, 2014."

[29] ——, "Calculating Memory System Power for LPDDR2, May 2013."

[30] S. Narendra, V. De, S. Borkar, D. Antoniadis, and A. P. Chandrakasan, "Full-chip sub-threshold leakage power prediction model for sub-0.18 μm cmos," in Proc. ISLPED, Aug 2002.

[31] Pandaboard, http://pandaboard.org/content/platform.

[32] R. Punzalan, " Smartphone Battery Life a Critical Factor for Customer Satisfaction ," http://www.brighthand.com/default.asp?newsID=18721.

[33] R. Raghavendra, P. Ranganathan, V. Talwar, Z. Wang, and X. Zhu, "No power struggles: Coordinated multi-level power management for the data center," in ACM SIGARCH Computer Architecture News, vol. 36, no. 1. ACM, 2008, pp. 48–59.

[34] K. K. Rangan, G.-Y. Wei, and D. Brooks, "Thread Motion: Fine-Grained Power Management for Multi-Core Systems," in International Symposium on Computer Architecture (ISCA), June 2009.

[35] S. M. Rumble, R. Stutsman, P. Levis, D. Mazieres, and N. Zeldovich, "Apprehending joule thieves with cinder," in Proceedings of the 1st Annual ACM workshop on networking, systems, and applications for mobile handhelds (MobiSys'10), August 2009.

[36] K. Sudan, N. Chatterjee, D. Nellans, M. Awasthi, R. Balasubramonian, and A. Davis, "Micro-pages: increasing dram efficiency with locality-aware data placement," in ACM Sigplan Notices, vol. 45, no. 3. ACM, 2010, pp. 219–230.

[37] S. J. Wilton and N. P. Jouppi, "Cacti: An enhanced cache access and cycle time model," Solid-State Circuits, IEEE Journal of, vol. 31, no. 5, pp. 677–688, 1996.

[38] H. Zeng, X. Fan, C. S. Ellis, A. Lebeck, and A. Vahdat, "ECOSystem: Managing Energy as a First Class Operating System Resource," in Proc. Architectural Support

for Programming Languages and Operating Systems (ASPLOS), San Jose, CA, October 2002.

[39] H. Zheng, J. Lin, Z. Zhang, and Z. Zhu, "Decoupled dimm: building high-bandwidth memory system using low-speed dram devices," ACM SIGARCH Computer Architecture News, vol. 37, no. 3, pp. 255–266, 2009.