Algorithms for CPU and DRAM DVFS Under Inefficiency Constraints

Rizwana Begum*, Mark Hempstead†, Guru Prasad Srinivasa‡ and Geoffrey Challen‡
*School of Electrical and Computer Engineering, Drexel University, Philadelphia, PA, USA. Email: rb639@drexel.edu
†School of Electrical and Computer Engineering, Tufts University, Medford, MA, USA. Email: mark.hempstead@tufts.edu
‡School of Computer Science and Engineering, University at Buffalo, Buffalo, NY, USA. Email: {gurupras, challen}@buffalo.edu

***Abstract*—Dynamic voltage and frequency scaling (DVFS) of both the core and DRAM provides opportunities to trade-off performance in order to save energy. Previous approaches to core and DRAM power management using DVFS used performance, specifically acceptable performance loss, as a constraint. We present energy management algorithms that coordinate core and DRAM frequency scaling under a specified energy budget. Approaches that work under performance constraints, as we will show, are not directly applicable to systems operating under energy constraints, as it is difficult to calculate the correct performance bounds in real-time to stay under an energy budget.**

Setting arbitrary energy budgets for a diverse set of applications can be harmful to application performance. We use the previously introduced concept of *Inefficiency*—the additional amount of energy above the minimum required energy that can be used to improve performance—to provide a dynamic energy constraint to our system. We introduce new power management algorithms that search the power and performance space to find the best performing point under this constraint. We demonstrate the efficacy of our algorithms using CPU DVFS and DRAM frequency scaling. We show that our algorithms have 24% lower tuning cost and save up to 5% energy with a little performance loss compared to a state-of-the-art performance constrained system.

I. Introduction

New algorithms and metrics are needed for systems with multiple tunable components in order to efficiently search a multidimensional energy and performance space. Dynamic Voltage and Frequency Scaling (DVFS) has been explored in the past for compute cores to make energy-performance trade-offs [1], [2]. Recently, frequency scaling for main memory, specifically for DRAMs has shown promise in trading energy savings for performance [3]. Scaling frequency of both cores and DRAMs simultaneously demands a coordinated approach that considers the cross-component effect of frequency scaling. CoScale is one such approach, proposed for server systems, that effectively searches for core and DRAM frequencies while staying under a configurable performance loss constraint [4].

Setting performance constraints for server systems is apt, as these systems are wall powered and have QoS requirements to meet. Setting power constraints is also reasonable in order to reduce the operational cost of the servers. However, devices that operate on a battery need to operate under energy constraints and not performance or power constraints in order to prolong battery life. While performance constraints do not consider the availability of limited energy resources, power constraints only capture instantaneous current draw rather than the total energy consumption of an application [5]. Energy management approaches that work under performance constraints, as we will show, can not be directly used for systems operating under energy constraints. One approach, that could be used for mobile devices, is to specify a performance bound as a proxy for energy savings; this is difficult, as the necessary performance bound for a target energy reduction varies across applications and devices. In this paper, we focus on exploring the CPU and DRAM frequency space under energy constraints. Approaches proposed in the past, that use energy constraints to manage energy, use metrics of absolute energy or rate of energy consumption (power) as a constraint [6], [7], [8]. Choosing the right absolute value for an energy budget in Joules is hard as different applications and devices have different ranges of energy consumption. We use a relative energy constraint, *Inefficiency* that specifies the additional amount of energy that can be used to improve performance [9].

We introduced *Inefficiency* in our previous work in the context of studying application characteristics and their energy-performance trade-offs using offline analysis [9]. However, computing *Inefficiency* can be computationally expensive and in our previous work, we neither computed *Inefficiency* at run-time nor did we use it to tune the system dynamically. In this paper, we present a holistic approach that uses inefficiency as a constraint, profiles the application at runtime and, using the energy models that we developed, selects best frequency settings to stay under given inefficiency budget. Our cross-component performance and energy models consider the impact of scaling frequency of one component on the other, and have average error of less than 4% across SPEC benchmarks. We profile the application periodically and feed the profiled statistics to our energy management algorithms which search for the best frequency settings to stay under given inefficiency budget. Doing an exhaustive search for the best frequency settings comes at a high cost, though it succeeds in delivering the best performance. We present new relative and adaptive algorithms that reduce the cost of a search with lower performance loss. We compare our algorithms with CoScale's search algorithm [4] and show that:

- Our algorithms have 24% lower tuning cost than CoScale on average across all SPEC benchmarks.
- The cost of our tuning algorithms is a function of application characteristics and not the inefficiency constraint, while CoScale's tuning cost increases with the value of the performance bound.
- Our system stays within the specified inefficiency budget and saves up to 5% energy with a little performance loss compared to the system using CoScale's search algorithm.

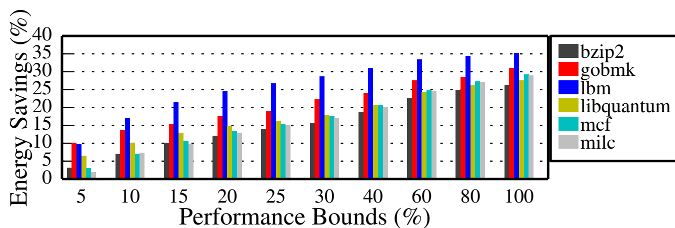


Fig. 1: **Energy Savings vs. Performance Bounds:** Energy savings vary across applications for a given performance bound.

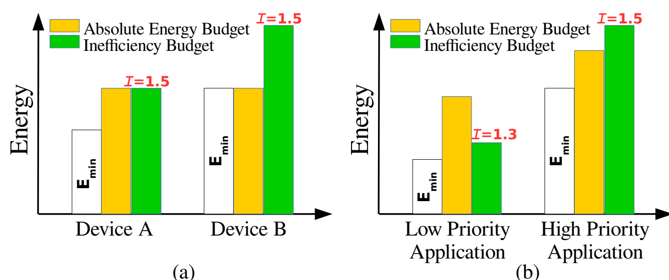


Fig. 2: **Inefficiency vs. Absolute Energy Constraints:** Inefficiency is both application and device independent.

II. Performance vs. Energy Constraints

Previously, DVFS has been used to save energy by either taking advantage of slack in the system or by degrading performance within a performance constraint [2], [4], [10]. Mobile systems are battery powered and have limited energy resources. Therefore, it is imperative that these systems are aware of the amount of energy they have and need so the system can manage energy as a resource. Choosing the acceptable performance loss bound in order to achieve a specific energy savings target is difficult, as the relationship between performance and energy varies across applications and devices. Figure 1 plots the energy savings achieved by CoScale for multiple applications across various performance bounds. The algorithm lowers frequency of the core and DRAM to save energy but aims to never reduce performance more than the specified bound. A performance bound of 10% allows the execution time to increase by 10% compared to the execution time at maximum frequencies, while 100% performance bound allows the execution time to be doubled in order to save energy. As shown in the figure, the energy savings achieved for a specified performance bound vary across applications. For example, a performance bound of 10% saves anywhere from 5% to 15% energy. This makes energy management difficult because if the system wanted to save a certain amount of energy, say 20%, it would require a performance bound of 15%-50% depending upon the application. Determining the performance bound for a target energy savings is a daunting task and requires oracle knowledge of the applications and devices.

Energy management approaches that operate under energy constraints have been proposed in the past—primarily for single components [7], [8]. Most of these approaches use an absolute energy budget or rate of energy consumption as constraints. Both, energy and rate of energy consumption are application and device dependent. Therefore they need to be recomputed for every combination of application and device which makes it impractical to use these metrics as user or OS configurable constraints.

A. *Inefficiency*

We define *Inefficiency* as the additional amount of energy that the application can use to improve performance [9]. It is the ratio of application's energy consumption (E) and the minimum energy the application could have consumed (E_{min}) on the same device: $I = \frac{E}{E_{min}}$

Inefficiency is application independent: Minimum energy that the applications consume on a given device, E_{min}, varies across applications. Therefore specifying a fixed absolute energy budget across applications may allow some applications to run with their maximum performance while make others run the slowest. If a fixed absolute energy budget is used, then low priority background applications with lower E_{min} could run with higher performance compared to the gaming applications that have higher E_{min} requirements. Inefficiency considers the inherent energy needs of the application by considering E_{min}. Therefore, a fixed inefficiency budget gives similar room to all the applications to improve performance, removing the need to recompute the inefficiency budget across applications, and thereby making inefficiency an application independent metric.

Inefficiency is device agnostic: E_{min} of an application varies across devices, based on the power-performance profile of the device. Figure 2(a) illustrates that using a similar inefficiency budget across devices (A and B) results in different energy consumption while leaving similar room for the application to improve performance. While an absolute energy budget needs to be recomputed to provide similar energy resources to the application across devices, inefficiency doesn't.

Inefficiency as a system resource: The application and device agnostic nature of inefficiency makes it a powerful tool that can be used by the OS to prioritize energy needs across applications and divide energy resources across multiple components of a device. A question that arises while using inefficiency to specify energy constraints in a real system is: *How is the inefficiency budget for an application chosen?* Inefficiency can take any value between 1 and I_{max}, where I_{max} is application and device dependent. The inefficiency budget can either be set by the user or the application to meet a specific system goal. A user may be willing to use more energy on an application either because more performance is required or the device is in an environment where it can readily be recharged. As an example, if the user is willing to use 50% more energy for a given application, their specification can be met by setting an inefficiency budget of 1.5 for that application. On the other hand, the OS can also map the inefficiency budget to application priorities, as illustrated in Figure 2(b), allowing higher priority applications to run at higher inefficiency and achieve higher performance, while lower priority applications can be given lower inefficiency budgets, such as the value of 1.3 used in the example illustration.

In this paper, we propose a system to demonstrate the use of inefficiency as an energy constraint and show how a system can stay within a given inefficiency budget while delivering best performance. We leave exploring the other aspects of inefficiency, including the system-level interfaces, hardware

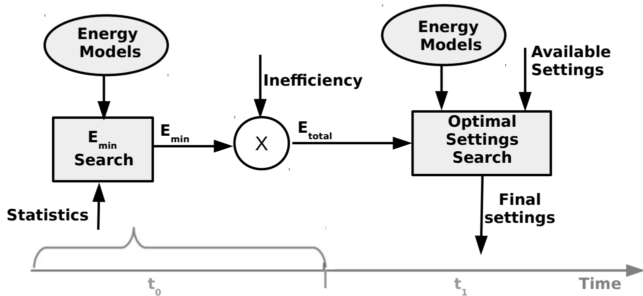


Fig. 3: **Energy Management Approach:** Searches for E_{min} and best settings for a given inefficiency budget using energy models.

implementations, and choice of system-wide inefficiency selection to future work.

III. Energy Management System Design

In this section we present the design of our system that takes inefficiency as an input and dynamically—during the application execution—chooses the core and DRAM frequencies that deliver optimal performance and stay under the specified inefficiency budget. Figure 3 summarizes the approach that our system takes to choose the best settings. The application statistics are collected periodically and are passed to the E_{min} search algorithm. The algorithm finds the minimum energy, E_{min}, that the given application phase could have consumed using the energy models. The total energy budget, E, is then computed using the inefficiency and estimated E_{min} and is passed to the algorithm that searches for the best settings. The algorithm filters the settings that fall under given budget and chooses the settings that deliver the best performance. In the end, the system transitions to the selected frequency settings. Two important aspects of our system are: 1) the performance and energy estimation models 2) the algorithms that search for E_{min} and the best frequency settings for the core and DRAM.

A. Performance and Energy Models

Our energy management system needs predictive models that can estimate the performance and energy of an application under a new set of frequency settings based on statistics gathered from the current frequency settings. Our cross-component models consider the impact of scaling one component's frequency on both the performance and energy of the other component. Our performance model estimates the execution time at different frequency settings using statistics (*stats*) gathered from the application while it executes at the current frequency setting. The energy model uses the estimated execution time from the performance model and the *stats* to predict energy consumption at target frequency settings.

Performance Model: The performance model works by computing the time spent in three *states* for each component, *Busy*, *Idle* and *Waiting*. In *Busy state*, CPU executes instructions and DRAM serves read, write requests or performs mandatory refreshes. *Idle* is the *state* when a component is stalled on other components, while in *Waiting state* the component is idle due to no work in the system. We also distinguish why the CPU is *Idle*; is it waiting on the cache (CPU frequency domain) or waiting on the DRAM (DRAM frequency domain)? We do not classify the DRAM *Idle* time further as the DRAM is idle only when it receives no requests to process from the CPU (CPU frequency domain). We then scale the time for each *state* to estimate the total amount of time it will take the system to complete the work at target frequency settings.

In our model, the *Idle* time of one component depends on the settings of the second component. The time that CPU is *Idle* waiting for the cache is scaled linearly with the CPU frequency as the caches in our system operate under the CPU frequency domain [10]. The CPU *Idle* time waiting for the DRAM is scaled with memory frequency. The performance of the DRAM doesn't scale linearly with frequency, as it consists of both synchronous (read, write) and asynchronous (activate, precharge and refresh) operations. We added hardware performance counters to measure scalable and non-scalable time of the DRAM, and use the ratio to scale the time spent in DRAM frequency dependent *states* as shown in the below equation:

$$C_{MI} = (C_{MI} * memScalableFraction * currMemFreq / newMemFreq) + C_{MI} * (1 - memScalableFraction)$$

where C_{MI} denotes the idle time of CPU waiting for the memory and *memScalableFraction* is the ratio of *memScalableTime* and the total time over which *stats* are collected. The *Busy* time of each component scales with its own frequency. However, part of the *Busy* time that overlaps with the other component is constrained by the slowest component.

Our model is able to estimate and scale the time spent in each of the three *states* in a light-weight manner using a total of 14 *stats*—also shared by the energy model. The *stats* used by our models are measured using hardware performance counters, which are used widely in commercial chips [11], [12]. ARM microprocessors can already monitor more than 50 performance events [11] and Intel processors include counters that estimate CPU energy [12]. Table I lists the *stats* used by our performance and energy models. Out of the 14 *stats*, 7 *stats* are needed by the memory power model as dictated by the Micron power models [13], and hardware performance counters used for measuring these are already included in modern systems [11], [12]. Only 7 new *stats* are introduced:

- **cpuBusyTime** – Represents the time the CPU spends executing instructions.
- **cpuIdleForDRAMTime, cpuIdleForCacheTime** – Time when the CPU pipeline is stalled waiting for the DRAM or Cache respectively.
- **cpuQuiesceTime** – Time when the CPU is idle due to lack of instructions to execute. Currently in our framework, we always observed zero *cpuQuiesceTime*, as there is no network or Input/Output operations in our workloads.

	Stat	Comments
CPU	cpuBusyTime, cpuIdleForCacheTime, cpuIdleForDRAMTime, cpuQuiesceTime	Energy, New
DRAM	memReads, memWrites, memActivates, memPrecharges, memRefreshes, memPrechargeTime, memActiveTime	Energy
	memBusyTime, memPrechargeIdleOverlapTime, memActiveIdleOverlapTime	New

TABLE I: **Stats Used in Our Models:** Counters labeled 'Energy' are used in the energy models in addition to performance model. The counters that are newly introduced are labeled with 'New'.

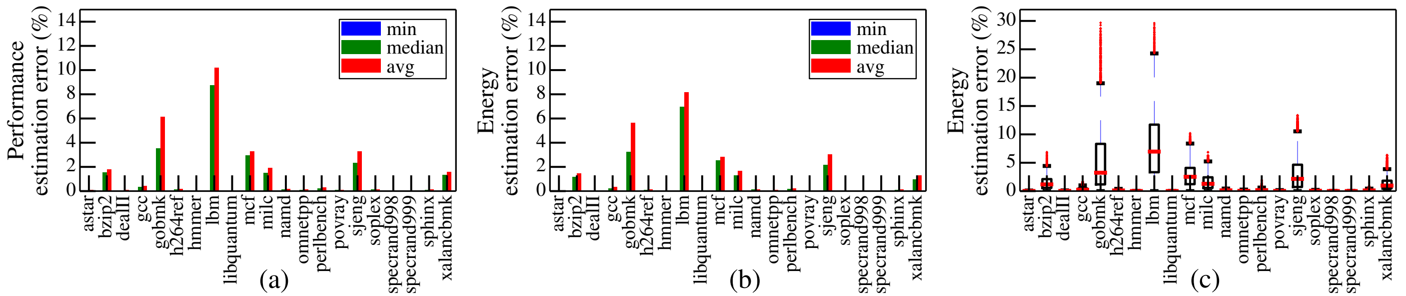


Fig. 4: **Model Estimation Error for SPEC2006:** Average error is less than 4%; max error is less than 10% except *gobmk* and *lbm*.

However, incorporating this *stat* into our model makes it possible to add additional components to the system.

- **memBusyTime** – Time when the DRAM is busy either reading or writing. It also includes the time taken by those refreshes that halt the processing of requests, as these refreshes are mandatory and cannot be eliminated.
- **memPrechargeIdleOverlapTime, memActiveIdleOverlapTime** – Time when the memory is idle and all of its rows are precharged or at least one of its rows is open respectively. We classify idle time of memory into active and precharge time as the background power of DRAM varies across its active and precharge states.

Once the performance model estimates execution time by scaling each of the *states*, the execution time along with scaled *stats* are passed on to the energy model to compute energy consumption at target frequencies.

CPU Energy Model: The CPU energy model predicts the energy consumption of the CPU at target frequency settings using measured *stats* and estimated execution time. The model can by summarized by the following equation:

$$cpuEnergy = P_{dyn} * cpuBusyTime + P_{bkgnd} * cpuClockedTime + P_{leak} * cpuTotalTime$$

While *cpuBusyTime* and *cpuTotalTime* are measured *stats* and estimated execution time respectively, *cpuClockedTime* is the difference between *cpuTotalTime* and *cpuQuiesceTime*—the time when CPU in our system is clock gated as it has no instructions to execute.

Our processor power model is based on empirical measurements of a PandaBoard [14] consisting of OMAP4430 chipset with Cortex A9 processor. We measure peak dynamic power (P_{dyn}) of 190mW and scale it quadratic with voltage and linear with frequency ($P \propto V^2 f$). We measured peak background power (P_{bkgnd}) of 208mW 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 (P_{leak}) comprises up to 30% of peak power consumption [15] and is linearly proportional to supply voltage [16]. Our peak power measurements agree with the numbers reported by [17] and the datasheets.

Memory Energy Model: We designed an extended version of the Micron Power Calculators [13] that incorporates frequency scaling. Similar to the CPU energy model, we divide the memory energy into dynamic, leakage and background energy. Our *stats* measure the events, specifically activates, precharges, reads, writes that result in dynamic energy consumption. Leakage energy is a result of refreshes that are performed periodically to retain the saved data. Memory background power varies based on memory state, active or precharge measured by *memActiveTime* and *memPrechargeTime* *stats* respectively. We take the timing and power numbers from Micron spec sheets [18] and use our *stats* to estimate energy consumption of DRAM.

Model Estimation Error: We measure our model error by comparing the predicted energy with the measured (using our models) energy consumption after running the application at the actual frequency. We simulate 12 integer and 9 floating point SPEC CPU2006 benchmarks [19] using a cycle accurate full system simulator, Gem5 [20]. We sample hardware performance counters every 10 million userspace instructions. We distinguish between user and kernel mode instructions to ensure uniform samples across frequencies. We utilize Gem5's capabilities to separate the user mode and kernel mode hardware performance counters. We ran 70 simulations for each benchmark, for a combination of 10 CPU and 7 memory frequency steps. For each of the 70 settings, we estimate energy using the collected *stats* and use it as ground truth. We predict performance and energy for the other 69 settings and compare the predicted energy and performance with the ground truth performance and energy data of the target frequencies.

We summarize the amount of error observed in performance and energy estimation in Figure 4 by plotting minimum, median and average absolute error for each simulated benchmark. As shown in Figure 4(a) average error of our performance model across benchmarks is less than 4% except for *gobmk*(6%) and *lbm*(10%). More than 50% of the predictions have less than 3% error, showing that our model is more than 97% accurate. Figure 4(b) shows that, in general, the trend in energy estimation error follows the error in performance estimation. However, not all of the performance estimation error translates to energy estimation error. Error in the energy estimate depends on the ratio of background and dynamic energy of the application which in-turn depend on the estimated execution time and the amount of work done respectively.

To understand the entire distribution of error we plotted box plots for the error and analyzed benchmarks with large error distributions. Figure 4(c) shows that the maximum energy estimation error barring the outliers is less than 10% for all of the SPEC benchmarks except for *gobmk* (18%) and *lbm*(24%). The large error observed for *gobmk* and *lbm* occurs rarely in our simulations, only when jumping from one frequency to frequencies vary far away.

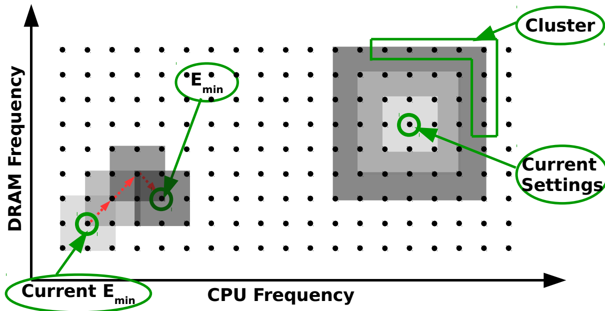


Fig. 5: **Relative Search:** Uses a hill climbing algorithm for E_{min} search, and gradual increase in search steps for *cluster* search.

B. Algorithms

In this section, we present algorithms for searching E_{min} and the best frequency settings under given inefficiency budget. These algorithms form an essential part of our energy management system as shown in Figure 3. We classify our algorithms into two categories 1) ***search*** and 2) ***select*** algorithms. ***Search*** algorithms search for E_{min} and find all possible settings that fall under given inefficiency budget. We call the set of possible settings that fall under given inefficiency budget a *cluster*. ***Select*** algorithms select the optimal frequency settings from the *cluster*. We present two search algorithms, *Exhaustive search* and *Relative search*. E_{min} is a function of application characteristics including the mix of, and dependencies between, CPU and memory instructions. Therefore, E_{min} varies across applications and application phases, consequently the optimal settings (CPU and memory frequency) also move. An *Exhaustive search* for E_{min} finds the optimal settings, however there is a high cost to perform the computation. The *Relative search* algorithm reduces the tuning cost with a very little loss in application performance. We propose two selection algorithms, *Best performing selection* and *Adaptive selection*. *Best performing selection* selects the settings that deliver the best performance however, it requires frequent tuning in order to adapt to application phases and not lose performance. On the other hand, *Adaptive selection* further reduces tuning overhead by not tuning in the long stable phases of the application and aims to not lose more performance than a specified *threshold*. Each algorithm is described in detail below.

Exhaustive Search (ex): The algorithm estimates energy consumption at each combination of core and DRAM frequency. After it populates the entire surface describing all of core and DRAM frequencies and their energy consumption, it finds the settings that consume least energy, E_{min}. Note that, because of leakage energy and the application's dynamic power, the lowest frequency point is not where E_{min} exists. Low frequency settings result in longer execution times which increase the leakage energy, sometimes more than the decrease in the dynamic energy, thereby shifting E_{min} to higher than minimum frequencies. Once E_{min} is found, using the already populated energy surface, the *cluster* of settings that fall under given inefficiency budget are filtered. The *cluster* is then passed on to the selection algorithms that choose the optimal settings.

Relative Search (rl): Estimating energy at each core and DRAM frequency combination is expensive. Our models take 10us to estimate energy at one pair of core and DRAM frequencies, as measured on a 1GHz Cortex-A9 processor.

```
1  Initialize best_settings with the first setting in the cluster
2  current_settings = settings selected during the previous tuning
3  for settings in cluster
4     if settings.performance > best_settings.performance
5        best_settings = settings
6     if settings == current_settings
7        current_settings_found = True
8  if current_settings_found is True
9     performance_loss = difference(best_settings,current_settings)
10    if performance_loss is within threshold
11       final_settings = current_settings
12       skip_tuning_intervals++
13        relative_search_steps++
14    else
15       final_settings = best_settings
16       skip_tuning_intervals = 0
17  else
18     final_settings = best_settings
19     skip_tuning_intervals = 0
```

Fig. 6: **Adaptive Selection:** An efficient algorithm to select optimal frequency settings under given inefficiency budget.

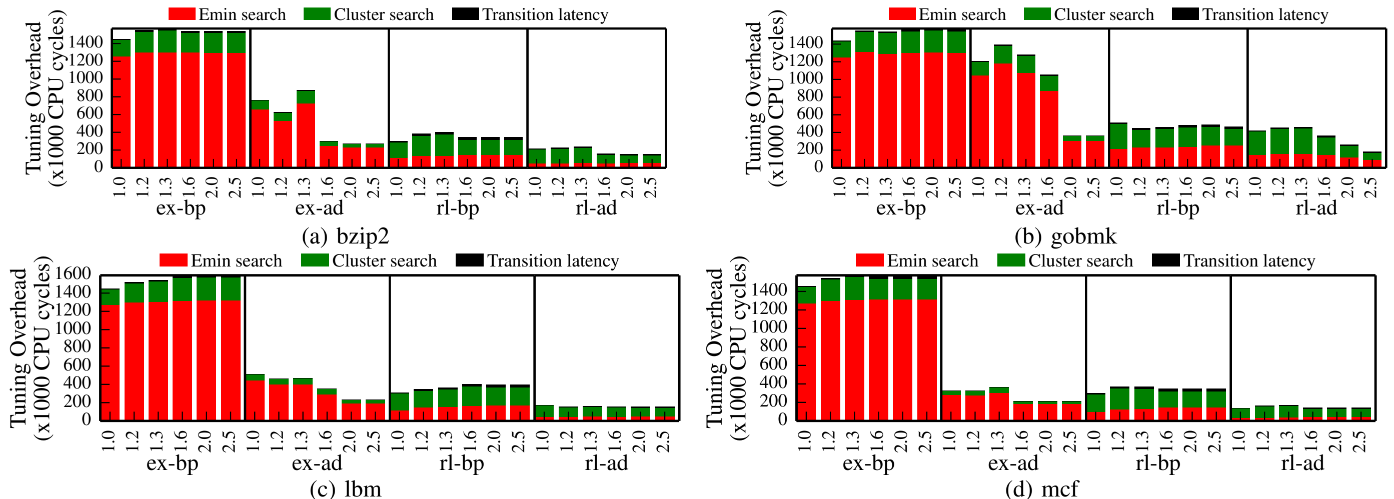
Based on the observation that for some applications E_{min} and the best settings move slowly, we propose *relative search* algorithm that works as shown in Figure 5.

We implement a hill climbing algorithm for E_{min} search that starts from the E_{min} settings found in the previous tuning interval. It estimates energy at all combinations of core and DRAM frequencies that are one step away and finds the settings with the lowest energy. The algorithm moves to the newly found settings and repeats the search until no other settings in the region have lower energy than the current E_{min} point. We also propose a relative search for finding the *cluster* of frequencies that fall under given inefficiency budget. The search starts from the best settings found in the previous interval and increases the search space by one step each time, until it finds at least one setting that has inefficiency within given budget. Note that, a relative search might get stuck in a local minima and not find the point that delivers best performance as the search stops as soon as it finds any point that falls within given budget in the interest of reducing tuning overhead. To address this shortcoming, the algorithm gradually increases it's search steps and moves to the best performing point in subsequent tunings (with the help of feedback from adaptive select, described in detail below), thereby results in loss of application performance (less than 3% for most of the SPEC benchmarks) compared to *exhaustive search*.

Best Performing Selection (bp): The algorithm selects the frequency settings that result in least execution time among all of the settings in the given *cluster*.

Adaptive Selection (ad): This algorithm is designed to detect application phases and skip tuning if the application has longer stable phases. The algorithm is also capable of detecting "possible" stuck in local minima due to *relative search* and helps by giving feedback to the *relative search* algorithm to exit the local minima region.

Pseudo code for the adaptive selection algorithm is presented in Figure 6. It starts by searching for the settings that deliver best performance in the given *cluster* (lines 3-5). It also determines if the current operating frequencies of the system fall within given inefficiency budget (lines 6-7). If so, the algorithm computes the performance loss of running at current settings compared to the best performing settings (lines 8-9). If the performance loss is within given *threshold*, then the algorithm directs the system to continue running at current



(a) bzip2 (b) gobmk (c) lbm (d) mcf

Fig. 7: **Tuning Overhead:** *rl-ad* significantly improves tuning overhead compared to *ex-bp* with <3% loss in application performance.

frequencies (line 11). The algorithm takes the selection of the same settings as a hint of stable phase, and therefore directs the system to skip tuning for the next interval. The selection of same settings in subsequent tunings results in a linear increase in tuning intervals to be skipped (line 12). A selection of same settings could also be the result of being stuck in a local minima, therefore it increments the *relative search* steps by 1 to expand the search space for next tuning in order to exit the local minima region (line 13). If the algorithm doesn't find the current settings in the *cluster* or if the performance loss at current settings is greater than the specified *threshold*, the algorithm directs the system to run at best performing settings and resets the number of tuning intervals to be skipped to zero (line 14-19).

IV. Evaluation

Algorithms presented in Section III-B perform two computations: 1) search for E_{min} and 2) find the *cluster* to select optimal settings, therefore result in higher tuning overhead compared to the algorithms operating under performance constraints, which only do the second computation. We introduced *relative search* and *adaptive selection* algorithms to reduce the tuning cost. In this section, we evaluate our algorithms and compare their tuning cost, application performance, and energy to the results of CoScale's search algorithm [4].

A. Experimental Methodology

Memory frequency scaling is not supported in current hardware systems, therefore we resort to `Gem5`—a cycle accurate full system simulator [20]—with default core configuration provided by `Gem5` in revision 10585, that reflects ARM Cortex-A15 processor with L1 cache of size 64 KB, 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. CPU clock domain frequency is configured to have a range of 100–1000 MHZ in steps of 30MHz with highest voltage being 1.25V. For the memory system, we simulate a LPDDR3 single channel, one rank memory using an open-page access policy. Timing and current parameters for LPDDR3 are configured as specified in data sheets from Micron [18]. We extended `Gem5` to support memory frequency scaling, and developed governors for the Linux kernel to support dynamic frequency scaling of DRAM. The memory clock domain is configured with a frequency range of 200MHz to 800MHz in steps of 100MHz. We do not scale memory voltage. The power supplies—VDD and VDD2—for LPDDR3 are fixed at 1.8V and 1.2V respectively [18].

We simulate 21 SPEC CPU2006 benchmarks [19]. We tune every ≈10M cycles with the goal of keeping the amount of work the same between two consecutive tunings. The system presented in CoScale tunes every scheduling quantum, i.e., every 5ms. To do a fair comparison, we run CoScale's algorithm at the same tuning interval as our system.

B. Results

We experiment with all four combinations of our search and select algorithms, namely 1) Exhaustive search-Best Performing selection(*ex-bp*) 2) Exhaustive search-Adaptive selection (*ex-ad*) 3) Relative search-Best Performing selection(*rl-bp*) and 4) Relative search-Adaptive selection(rl-ad). We compare results of these 4 combinations to CoScale's algorithm. Comparing the results of our algorithm to that of CoScale's is difficult because our algorithm works under energy constraints while CoScale works under a performance constraint. Therefore, we first need to determine the performance bound for each application for a given inefficiency budget. We do so by computing the difference between application's execution time at maximum frequencies and the execution time using *ex-bp* algorithm—that results in best performance for a given inefficiency budget—without including its tuning overhead. We then run CoScale simulations with the estimated performance bounds.

Tuning Overhead: Figure 7 plots tuning cost of our algorithms for *bzip2*, *gobmk*, *lbm* and *mcf* benchmarks for multiple inefficiencies. *ex-bp* has the highest tuning cost among all algorithms as it searches the entire space of frequency settings for E_{min} and the *cluster*. The E_{min} search itself takes up to 84% of the total cost on average, as it involves estimating energy at all settings before finding minimum of all the settings. The cost of the *Cluster* search is low as it uses the already populated

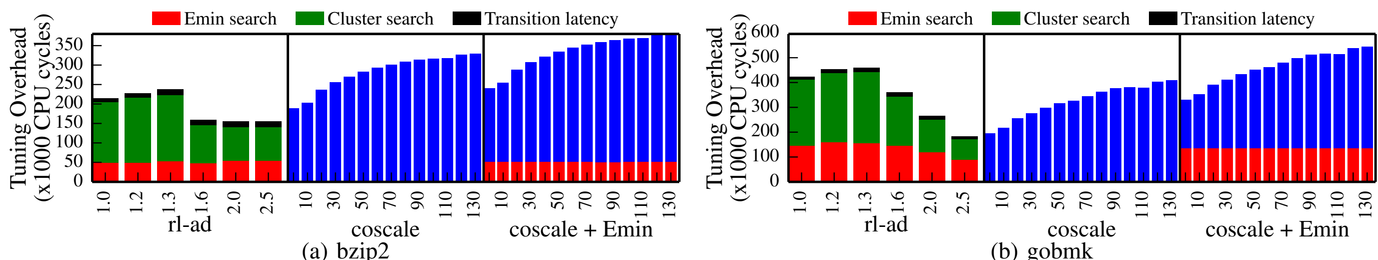


Fig. 8: **Tuning Overhead Compared to CoScale:** *rl-ad* has 24% lower tuning overhead than CoScale across all SPEC benchmarks. While for *bzip2*, *rl-ad* wins over CoScale, for *gobmk*, it performs better when compared fairly by including E_{min} search cost to CoScale.

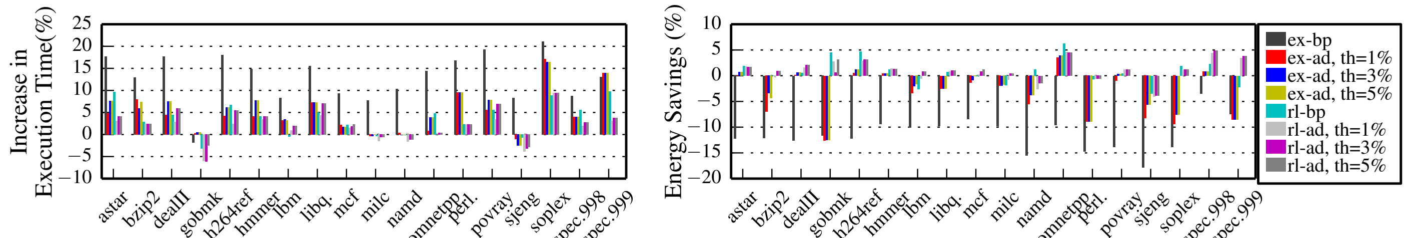


Fig. 9: **Performance and Energy Compared to CoScale For *Inefficiency* of 1.3:** *rl-ad* results in performance loss of less than 5% for most of the benchmarks with an average of 2.8% across all SPEC benchmarks. Energy savings are up to 5% with an average of 1.5%.

energy surface during the E_{min} search. When the exhaustive search is combined with adaptive select, *ex-ad*, the average tuning cost decreases as adaptive select skips tuning if the application has long stable phases. The decrease is a function of the application. As shown in the figure, the reduction in tuning cost of *ex-ad* compared to *ex-bp* is higher for *bzip2*, *lbm* and *mcf* than for *gobmk* as these benchmarks have long stable phases, while *gobmk* has rapidly changing CPU and memory intensive phases. Relative search (*rl-bp*) significantly improves tuning cost, 32% for *bzip2* and 50% for *gobmk* compared to *ex-ad*. However, it performs similar to *ex-ad* for *lbm* (with only 2% improvement in tuning cost), and has 20% higher tuning cost for *mcf*. The higher tuning cost of best performing select dominates the savings achieved by relative search increasing the overall average tuning cost for *mcf*. Finally, relative search combined with adaptive select (*rl-ad*) results in the lowest tuning overhead among all the algorithms. *rl-ad* improves tuning cost by as much as 46% on average compared to *rl-bp* by both selectively tuning and reducing the search space.

Next, we compare the tuning cost of our best algorithm, *rl-ad*, with that of CoScale's algorithm—designed to select settings that result in lowest energy consumption under performance constraints. Figure 8 plots tuning cost of *rl-ad* and CoScale for *bzip2* and *gobmk* benchmarks. As shown in the Figure, tuning cost of *rl-ad* for *bzip2* is lower than CoScale by 32%, while for *gobmk* the average cost of *rl-ad* is higher than CoScale by 10%. Note that while our algorithms do two computations: 1) finding E_{min} and 2) searching the *cluster*, CoScale, as published, only performs the later: searching for possible settings that don't violate the set performance constraints. Adaptive selection reduces the average tuning cost of the algorithms per application by selectively skipping tuning. For applications that have rapidly changing phases, adaptive selection decides to run the algorithm quite often in order to bound the performance loss to set threshold, thereby results in higher tuning overhead than CoScale for *gobmk*. However, the E_{min} search itself takes 40% of the *rl-ad* tuning overhead for *gobmk*. Because we had to perform an E_{min} search to properly use CoScale by determining the performance bounds, it would be fair to compare our tuning cost with that of CoScale with E_{min} search cost included. We add the average E_{min} search cost to the tuning overhead of CoScale and plot it for *bzip2* and *gobmk* in Figure 8. As we can see from the figure, *rl-ad* has lower average tuning cost for *gobmk* compared to CoScale+E_{min} cost by 22%. Overall our *rl-ad* algorithm has on average 24.3% lower tuning cost compared to just CoScale's algorithm across all SPEC benchmarks. Another important observation from Figure 8 is that tuning cost of CoScale increases for higher performance bounds as the algorithm always starts its search from the highest frequencies. As application phases don't change sharply, by starting from the current settings, tuning cost of our algorithms is independent of inefficiency constraint and is only a function of application phases.

Performance and Energy: We plot the increase in execution time of the applications compared to CoScale for an inefficiency budget of 1.3 in Figure 9(a). As CoScale chooses the settings that perform within bounds, it never exceeds the set performance loss. In the interest of reducing the tuning cost, *relative search* stops as soon as it finds any setting under given inefficiency budget and *adaptive select* skips tuning if performance loss is within specified *threshold*, thereby loosing performance. We observe that for most of the benchmarks the performance loss for *rl-ad* is less than 5% and in a very few cases goes up to 7% with an average of 2.8%. Tuning overhead is included; therefore, *ex-bp* results in highest increase in execution time due to it's high tuning cost. Figure 9(b) plots energy savings compared to CoScale. *rl-ad* saves up to 5% energy with an average of 1.5% across all SPEC benchmarks. We observed similar savings for other inefficiency budgets.

Achieved Inefficiency: Figure 10 plots inefficiency achieved by the applications using our algorithms for the inefficiency budget of 1.2. As shown in Figure 10(a), applications always stay within the specified inefficiency budget using our algorithms (As shown by negative difference in inefficiency), as the optimal frequencies are chosen from the *cluster* which contains only those settings that fall within given inefficiency budget.

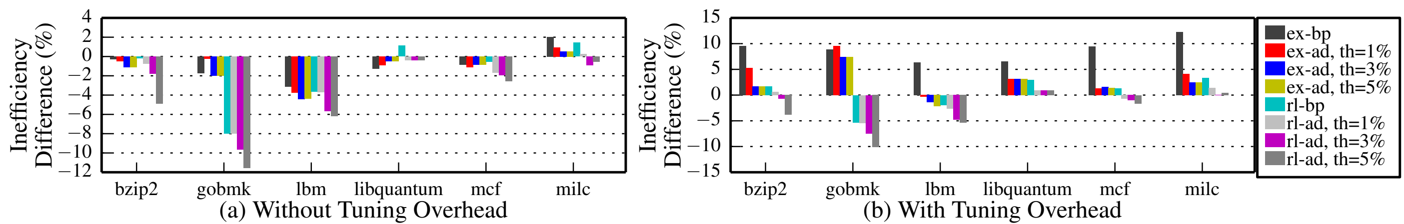


Fig. 10: **Inefficiency Difference (%) for I = 1.2:** *rl-ad* always stays within specified inefficiency budget.

Figure 10(b) plots combined inefficiency of applications and algorithms. As we don't include the cost of tuning while selecting the frequency settings, *ex-bp* results in higher (upto 10%) than specified inefficiency budget due to its high tuning overhead, while *rl-ad* always stays within specified inefficiency budget (illustrated by inefficiency difference of less than zero).

V. Conclusion

Core DVFS when combined with DRAM frequency scaling provides greater opportunities for energy savings by trading off performance. Managing the frequency of both components simultaneously requires a coordinated approach that considers the cross-component affects of scaling one component's frequency on the other. We demonstrated how systems working under performance constraints can't be directly used for systems operating under energy constraints and presented a holistic approach that works under an energy constraint, *Inefficiency* and selects the optimal frequencies staying under the specified inefficiency budget.

Computing inefficiency is expensive and results in high tuning cost if done with exhaustive search. Therefore we presented relative and adaptive algorithms that reduce the tuning cost with very little loss in performance. Our system is successful in staying within specified inefficiency budget and our algorithms have 24% lower tuning cost and save up to 5% energy across all SPEC benchmarks compared to the search algorithm proposed by CoScale.

VI. Acknowledgement

This material is based on work partially supported by NSF Awards CSR-1409014 and CSR-1409367. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] R. Z. Ayoub, U. Ogras, E. Gorbatov, Y. Jin, T. Kam, P. Diefenbaugh, and T. Rosing, "Os-level power minimization under tight performance constraints in general purpose systems," in *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*. IEEE Press, 2011, pp. 321–326.

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

[3] 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.

[4] 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.

[5] 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.

[6] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat, "Currentcy: a unifying abstraction for expressing energy management policies," in *Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2003, pp. 4–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247340.1247344

[7] 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.

[8] 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.

[9] R. Begum, D. Werner, M. Hempstead, G. Prasad, and G. Challen, "Energy-performance trade-offs on energy-constrained devices with multi-component dvfs," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 34–43.

[10] N. C. Nachiappan, P. Yedlapalli, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "Domain knowledge based energy management in handhelds," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015, pp. 150–160.

[11] ARM, "Cortex-A9 Technical Reference Manual, Revision 4." 2012.

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

[13] Micron, "Calculating Memory System Power for LPDDR2, May 2013."

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

[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] 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.

[17] 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.

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

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

[20] 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