Fast Power and Energy Management for Future Many-Core Systems

YANPEI LIU, University of Wisconsin Madison
GUILHERME COX, Rutgers University
QINGYUAN DENG, Facebook Inc.
STARK C. DRAPER, University of Toronto
RICARDO BIANCHINI, Microsoft Research

Future servers will incorporate many active low-power modes for each core and for the main memory subsystem. Though these modes provide flexibility for power and/or energy management via Dynamic Voltage and Frequency Scaling (DVFS), prior work has shown that they must be managed in a coordinated manner. This requirement creates a combinatorial space of possible power mode configurations. As a result, it becomes increasingly challenging to quickly select the configuration that optimizes for both performance and power/energy efficiency.

In this article, we propose a novel queuing model for working with the abundant active low-power modes in many-core systems. Based on the queuing model, we derive two fast algorithms that optimize for performance and efficiency using both CPU and memory DVFS. Our first algorithm, called FastCap, maximizes the performance of applications under a full-system power cap, while promoting fairness across applications. Our second algorithm, called FastEnergy, maximizes the full-system energy savings under predefined application performance loss bounds. Both FastCap and FastEnergy operate online and efficiently, using a small set of performance counters as input. To evaluate them, we simulate both algorithms for a many-core server running different types of workloads. Our results show that FastCap achieves better application performance and fairness than prior power capping techniques for the same power budget, whereas FastEnergy conserves more energy than prior energy management techniques for the same performance constraint. FastCap and FastEnergy together demonstrate the applicability of the queuing model for managing the abundant active low-power modes in many-core systems.

CCS Concepts: • **Computing methodologies** → **Modeling methodologies**; • **Hardware** → **Power and energy**; • **Computer systems organization** → *Architectures;*

Additional Key Words and Phrases: Queuing theory and optimization

ACM Reference Format:
Yanpei Liu, Guilherme Cox, Qingyuan Deng, Stark C. Draper, and Ricardo Bianchini. 2017. Fast power and energy management for future many-core systems. ACM Trans. Model. Perform. Eval. Comput. Syst. 2, 3, Article 17 (September 2017), 31 pages.
DOI: http://dx.doi.org/10.1145/3086504

This work was funded in part by NSF grant CCF-1319755. The work of Yanpei Liu was partially supported by a visitor grant from DIMACS, funded by the National Science Foundation under grant numbers CCF-1144502 and CNS-0721113.
Extension of Conference Paper. This publication is an extension of "FastCap: An Efficient and Fair Algorithm for Power Capping in Many-Core Systems," a published work of Liu et al. at IEEE ISPASS, 2016. The additional contribution is the algorithm generalization for energy conservation. (The methodology was previously designed exclusively for power capping in multi-core systems.) A new algorithm was developed and its effectiveness in energy conservation is supported by theoretical analysis and numerical results.
Authors' addresses: Y. Liu, 1415 Engineering Drive, Madison, WI 53706; G. Cox, 110 Frelinghuysen Road Piscataway, NJ 08854-8019; Q. Deng, 1 Facebook Way, Menlo Park, CA 94025; R. Bianchini, One Microsoft Way Redmond, WA 98052; S. Draper, 10 King's College Road, Toronto, Ontario, M5S 3G4.

DOI: http://dx.doi.org/10.1145/3086504

1. INTRODUCTION

As power and energy consumption become significant concerns for server systems, servers have started to incorporate an increasing number of idle low-power states (such as CPU sleep states) and active low-power modes of execution (such as CPU DVFS). Researchers have also proposed active low-power modes for the main memory subsystem [11, 13], for disk drives [8, 18], and for interconnects [1]. Active low-power modes are often more desirable than idle low-power states, as they enable the component to continue operating in a more efficient fashion, without activation delays [30]. For this reason, we focus on active low-power modes for CPUs and memory, using DVFS.

As suggested by many prior works [13, 28, 26], active low-power modes should be managed in a coordinated manner. A lack of coordination hampers a system's ability to optimize performance, power, and/or energy. For example, consider a scenario in which our goal is to manage low-power modes to maximize the full-system energy savings within a preselected bound on the acceptable application performance loss. Further, suppose that the CPU cores are stalled waiting for the memory a significant fraction of the time. In this situation, the CPU power manager might predict that lowering voltage/frequency will improve energy efficiency while meeting the performance bound. By effecting the requisite state change, the lower core frequency would reduce traffic to the memory subsystem, which in turn could cause its (independent) power manager to lower the memory frequency. After this latter frequency change, the performance of the server as a whole may dip below the CPU power manager's projections, potentially violating the target performance bound. So, at its next opportunity, the CPU manager might start increasing the core frequency, inducing a similar response from the memory subsystem manager. Such interactions between CPU and memory frequencies can cause performance oscillations. These unintended behaviors suggest that it is essential to coordinate power-performance management techniques across system components to ensure that the system is balanced for the best efficiency.

The abundance of active low-power modes provides great flexibility in performance-aware power/energy management via DVFS. However, the need for coordinated management creates a combinatorial space of possible power mode configurations. This problem is especially acute for servers that run many applications (each with a potentially different behavior), since it is unlikely that the power mode selected for a core running one application can be used for a core running another one. Quickly traversing the large space of mode combinations to select a good configuration as applications change behavior is difficult. Existing works often rely on exhaustive search [21, 28, 27] or suboptimal heuristics [7, 13]. Clearly, suboptimal or high time complexity algorithms are unlikely to perform well for future many-core servers.

With these observations in mind, in this article, we propose a novel queuing model for many-core systems with many active low-power modes. Based on the queuing model, we derive two fast algorithms that jointly operate CPU and memory DVFS: (1) FastCap, an optimization framework and search algorithm for performance-aware full-system power capping while promoting fairness across applications, and (2) FastEnergy, an optimization framework and search algorithm for energy conservation within predefined performance loss bounds. These algorithms efficiently select voltage/frequency configurations that are optimal or near optimal for many-core systems. The time complexity of our algorithms (linear in the number of cores) is lower than any of the prior approaches, despite the combinatorial number of possible power mode configurations. Furthermore, FastCap achieves better application performance and fairness than prior power capping techniques for the same power budget, whereas FastEnergy conserves more energy than prior energy management techniques for the same performance constraint. Based on our results, we conclude that queueing models and linear-time

Table I. Comparison of FastCap and FastEnergy with Existing Approaches

Objective	Method	Complexity	Mem. DVFS
Power capping	Exhaustive [21]	$\sim O(F^N)$	No
Power capping	Numeric Opt. [6]	$\sim O(N^4)$	No
Power capping	Heuristics [31]	$\sim O(FN \log N)$	No
Power capping	**FastCap**	$O(N \log M)$	Yes
Energy saving	Exhaustive [28]	$\sim O(F^N)$	No
Energy saving	Heuristics [13]	$O(M + FN^2)$	Yes
Energy saving	**FastEnergy**	$O(NM)$	Yes

N: Number of cores. F: Number of frequency levels per core. M: Number of memory frequency levels. *Both FastCap and FastEnergy scale linearly with the number of cores.*

optimization can be used to efficiently manage the combinatorial explosion of possible power modes in future many-core servers.

The rest of the article is organized as follows. We first summarize some related works in Section 2 and foreshadow the algorithmic complexity of our FastCap and FastEnergy in Table I. Then, in Section 3, we introduce our novel queuing model designed for power/energy management. Based on the queuing model, in Section 4, we introduce FastCap and evaluation results. To study energy conservation, in Section 5, we present the FastEnergy algorithm. Finally, we conclude in Section 6.

2. RELATED WORK

2.1. Power Management

To cap power, as in FastCap, most existing works used optimization methods and/or control-theoretic approaches. We review some of the major works in both categories.

Optimization approaches. The authors in [35] first suggested that global power management is better than a distributed method in which each core manages its own power. They argued that all cores receiving an equal share of the total power budget is preferred over a dynamic power redistribution, due to the complexity of the latter approach. The authors in [21] used exhaustive search over precomputed power and performance information about all possible power mode combinations. Their algorithm's time and storage space complexities grow exponentially with the number of cores. The authors in [37] developed a linear programming method to find the best DVFS settings under power constraints. However, they assumed power is linearly dependent on the core frequency, which is often a poor approximation. The authors in [31] developed a greedy algorithm that starts with maximum speeds for all cores and repeatedly selects the neighboring lower global power mode with the best $\Delta_{power}/\Delta_{perf}$ ratio. The algorithm may traverse the entire space of power mode combinations. The authors in [6] formulated a nonlinear optimization and solved it via the interior-point method. The method usually takes many steps to converge and its average complexity is polynomial in the number of cores.

Control-theoretic approaches. Recently, a power monitoring and control system was built for the entire Facebook datacenter fleet across multiple levels of power hierarchy [40]. The authors in [33] studied power management in multicore CPUs with voltage islands. They proposed a two-tier feedback-based design. They assumed that the power-frequency model (power consumption as a function of the cores' frequencies) is fixed for all islands, which may be inaccurate under changing workload dynamics. The authors in [29] used a method that stabilizes the power consumption by adjusting a frequency quota for all cores. The quota is distributed to core groups according to their power efficiency. A group's quota is further distributed to individual cores according to their thread criticality. In a similar vein, [10] used a control-theoretic

method along with (idle) memory power management via rank activation/deactivation. Unfortunately, rank activation/deactivation is too slow for many applications [30]. The work in [10, 29] required a linear power-frequency model, which may cause under- and overcorrection in the feedback control due to poor accuracy. This may lead to large power fluctuations, though the long-term average power is guaranteed to be under the budget. In fact, a recent survey [12] discussed different power models used in power/energy management. Finally, the authors in [34] considered power allocation as a shared resource contention problem and developed a scheduling algorithm that mitigates the contention for power and the shared memory subsystem at the same time.

2.2. Energy Management

Using CPU DVFS for energy management has been a well-studied area. Bansal et al. [4] studied minimizing energy via CPU speed scaling without violating task deadlines. Later, Wierman et al. [38] showed that a dynamic policy that adapts the server speed in proportion to queue length is robust to bursty traffic and mispredictions of workload parameters. The authors in [36] exploited the instruction-level slack to save processor energy. The idea is to reduce processor frequency when the programs have memory access slack. The authors in [9] proposed a mechanism for maximizing energy efficiency in multicore CPUs by balancing the power among critical and noncritical cores.

In contrast, FastEnergy leverages both CPU and memory DVFS. The closest work to FastEnergy is CoScale [13]. CoScale jointly manages the DVFS in cores and memory to minimize energy consumption while meeting user-defined performance loss bounds. CoScale explores the space of per-core and memory frequency settings in a greedy manner. It repeatedly computes the marginal energy and performance cost (or benefit) of altering each component's (or set of components') power mode by one step. CoScale then greedily selects the next best frequency combination and iterates until a (local) minimum is attained.

Although CoScale is effective in conserving energy and meeting performance bounds, its search heuristic has a high time complexity of $O(M + FN^2)$, where M is the number of memory frequencies, F is the number of possible core frequencies, and N is the number of cores. The *quadratic* dependence on the number of cores is problematic, as core counts may increase at the speed of Moore's law. In contrast, our FastEnergy algorithm is *linear* in the number of cores.

A few other works also considered coordinating CPU and memory power management for energy conservation subject to performance bounds [15, 24]. These earlier works focused on single-core systems, which are easier to manage than many-core systems. Also, they assume only idle low-power states for memory, while we consider active low-power modes for memory (and cores). Different from these works, [5] recently studied coordinating core and DRAM frequencies under a specific energy budget.

To summarize, Table I lists some representative works and their algorithmic time. We compare them with our FastCap and FastEnergy as a preview.

3. MANY-CORE SYSTEM MODEL

We consider a system with N (in-order) cores, K memory banks, and a common memory bus for data transfers, as depicted in Figure 1. (We study out-of-order cores in Section 4.3.2.) Denote by $\mathcal{N}$ the set of cores. We assume each core runs one *application* and we name the collection of N applications a *workload.*

3.1. CPU Performance Model

Every core periodically issues memory access requests (resulting from last-level cache misses and writebacks) independently of the other cores. Though the following description focuses on cache misses for simplicity, we also model writebacks as occupying their

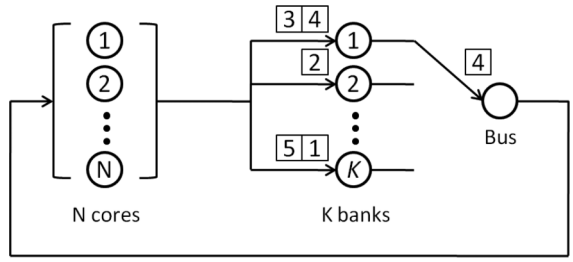


Fig. 1. The many-core queuing model demonstrating the "transfer blocking" property. Memory bank 1 receives requests from cores 4 and 3. The requested data for core 4 has been fetched and is being transferred on the memory bus. At the same time, bank 1 is blocked from processing the request from core 3 until the previous request is successfully transferred to core 4.

Fig. 2. An example workload dynamics with $N = 3$ cores. Variables z_i and c_i are the think time and cache time for core i, respectively. $R(s_b)$ is the response time of the memory. z_i, c_i, and $R(s_b)$ are all average values. The sum $R(s_b) + c_i + z_i$ is the total time for one memory access of core i.

target memory banks and the memory bus. In addition, we assume that writebacks happen in the background, off the critical performance path of the cores.

After issuing a request, the core waits for the memory subsystem to fetch and return the requested cache line before executing future instructions. We denote by $z_i, i \in \mathcal{N}$ the average time core i takes to generate a new request after the previous request completes (i.e., data for the previous request is sent back to core i, see Figure 2). The term z_i is often called the *think time* in the literature on closed queuing networks [19]. Further, to model core DVFS, we assume each core can be voltage and frequency scaled independently of the other cores, as in [23, 41]. This translates to a scaled think time: denote by $\overline{z_i}$ the minimum think time achievable at the maximum core frequency. Thus, the ratio $\overline{z_i}/z_i \in [0, 1]$ is the frequency scaling factor: setting frequency to the maximum yields $z_i = \overline{z_i}$. The minimum think time depends on the application running on the corresponding core and may change over time. Determining the frequency for core i is equivalent to determining the think time z_i. We assume there are F frequency levels for each core.

We assume the shared last-level cache (L2) sits in a separate voltage domain that does not scale with core frequencies. According to our detailed simulations, changing core frequencies does not significantly change the per-core cache miss rate. Thus, for simplicity, we model the average L2 *cache time* c_i for each core i as independent of the core frequency.

3.2. Memory Performance Model

The memory is subdivided into K equally sized memory banks. Each of the K memory banks serves requests that arrive within its address range. After serving one request, the retrieved data is sent back to the corresponding core through a common bus that

is shared by all memory banks. The bus is used in a first-come-first-serve manner: any request that is ready to leave a bank must queue behind all other requests from other banks that finish earlier before it can acquire the bus. Furthermore, each memory bank cannot process the next enqueued request until its current request is transferred to the appropriate core. In queuing-theoretic terminology, this memory subsystem exhibits a "transfer blocking" property [2], [3]. In Figure 1, we illustrate the transfer blocking property via an example.

We use the *mean response time*, the average amount of time a request spends in the memory (cf. Figure 2), as the performance metric for the memory. To the best of our knowledge, no closed-form expression exists for the mean response time in a queuing system with the transfer blocking property. Instead of deriving an explicit form for the mean response time, we use the following approximation.

When a request arrives at a bank, let Q be the expected number of requests enqueued at the bank (including the newly arrived request). When the request has been processed and is ready to be sent back to the requesting core, let U be the expected number of enqueued requests waiting for the bus, including the departing request itself. Denote by s_m the average memory access time at each bank. Denote by s_b the bus transfer time. We approximate the mean response time of the memory subsystem as

$$R(s_b) \approx Q(s_m + U s_b). \tag{1}$$

Since all cores contend for the memory access (and the contention is reflected in queue size U and Q), Equation (1) can be treated as the mean response time *seen by all cores.* A previous study [13] has found this calculation to be a good approximation to the response time of the memory subsystem.

The memory DVFS method dynamically adjusts the bus frequency [14]. This translates to a scaled bus transfer time. Denote by $\overline{s_b}$ the minimum bus transfer time at the maximum bus frequency—the ratio $\overline{s_b}/s_b \in [0, 1]$ is the bus frequency scaling factor. We assume the bus frequency can take M values. Determining a frequency setting for the memory is equivalent to determining the transfer time s_b.

3.3. Power Models

We model the power drawn by core i as

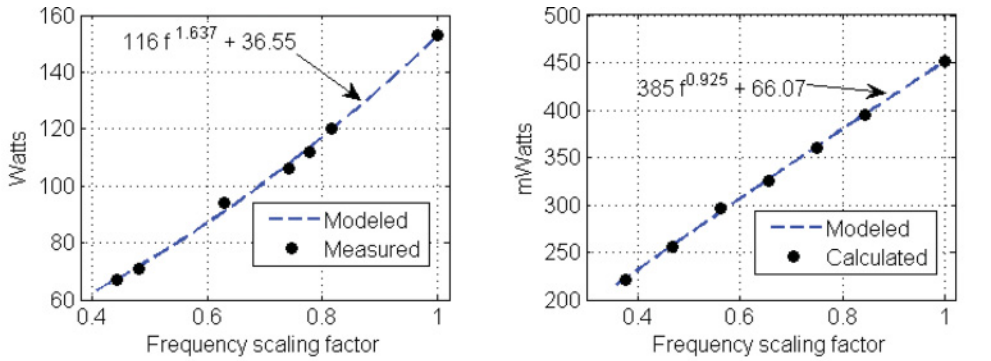
$$P_i \left(\frac{\overline{z_i}}{z_i}\right)^{\alpha_i} + P_{i,static}, \tag{2}$$

where P_i is the maximum dynamic power consumed by the core, α_i is some exponent typically between 2 and 3, and $P_{i,static}$ is the static (voltage/frequency-independent) power the core consumes at all times. At runtime, FastCap periodically recomputes P_i and α_i by using power estimates for core i running at different frequencies and solving the instances of Equation (2) for these parameters. We note that many prior papers used simple models (e.g., assuming the power is always linearly dependent on the frequency) that do not account well for different workload characteristics [29, 37].

We model the memory power as

$$P_m \left(\frac{\overline{s_b}}{s_b}\right)^{\beta} + P_{m,static}, \tag{3}$$

where P_m is the maximum memory power. The memory also consumes some static power $P_{m,static}$ that does not vary with the memory frequency. At runtime, we periodically recompute P_m and β by using power estimates for the memory running at different frequencies and solving the instances of Equation (3) for the parameters.



(a) Measured and modeled CPU power on a production SandyBridge server

(b) Computed and modeled memory power on Micron DDR3 DRAM

Fig. 3. Validation of CPU and memory power models.

We include all the sources of power consumption that do not vary with either core or memory frequencies into a single term P_s. This term includes the static power of all cores $\sum_i P_{i,static}$, the memory's static power $P_{m,static}$, the memory controller's static power, the L2 cache power, and the power consumed by other system components, such as disks and network interface.

The power models in Equation (2) and Equation (3) are *inputs* to FastCap and FastEnergy. Since α_i, β, P_i, $P_{i,static}$, P_m, and $P_{m,static}$ are all platform-specific variables, the power models can cover a wide range of systems. Our measurements of power as a function of voltage/frequency for two generations of Intel CPUs reveal that Equation (2) is an accurate model. To illustrate this claim, Figure 3(a) presents the measured and modeled CPU power under different frequencies on a production SandyBridge server. Clearly, the measurements validate our CPU power model. A recent study on power versus frequency/voltage on Intel's 32 and 22nm CPUs also revealed that Equation (2) is a good model [39]. To validate the memory power model, we calculate the memory power using a detailed datasheet for DDR3 DRAM from Micron [32] and plot both calculated and modeled power values as a function of frequency in Figure 3(b). Again, these results demonstrate that our memory power model is accurate.

To study the accuracy of our power model under dynamically changing workloads, we simulate both CPU- and memory-bound workloads on a 16-core system using the methodology described in Section 4.3.1. We collect full-system average power for every $5ms$ window. In the first $300\mu s$ of each window, we use our model to predict the average power for the rest of the window. Figure 4 compares the average power observed and the one our model estimates. Over the course of execution, the average modeling error less than 5%.

3.4. Model Discussion

The preceding presentation assumes a many-core system with in-order cores and one outstanding L2 miss per core. However, our model can easily adapt to out-of-order cores with multiple outstanding misses per core, by making z_i represent the time between two consecutive *blocking* memory accesses, assuming nonblocking accesses are off the critical path, just as cache writebacks. Since both blocking and nonblocking accesses contend for the memory, the response time model in Equation (1) does not change and the contention is reflected in the queue sizes Q and U. We discuss our out-of-order implementation in Section 4.3.2.

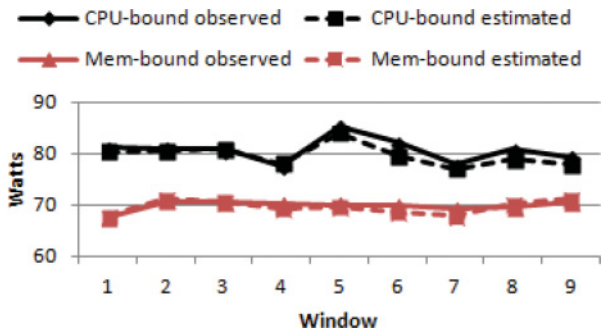


Fig. 4. Average full-system power compared to the one-model estimates for a 16-core system running CPU-bound and memory-bound workload, respectively.

The model discussed previously also assumes a single memory controller. It can be easily adapted to multiple controllers by considering different response times for different controllers. In this scenario, the probability of each core using each controller (i.e., the access pattern) has to be considered. We defer the discussion of multiple controllers to Section 4.3.2.

Finally, for simplicity, our model treats each hardware thread as a core; that is, we represent a dual-threaded core as two cores. The model is oblivious to the number of software threads running on each hardware thread.

4. FASTCAP

In this section, we introduce FastCap, an optimization framework and search algorithm for performance-aware full-system power capping developed based on the queuing model discussed in Section 3. FastCap considers maximizing performance under some system-wide power budget. The goal for FastCap is to allocate power to cores and memory fairly such that all applications degrade by the same fraction of their maximum performance as a result of the less-than-peak power. Thus, FastCap seeks to prevent "performance outliers," that is, applications that get degraded much more than others.

4.1. FastCap Optimization Algorithm

Based on the queuing model (cf. Figure 2), we use the time interval between two memory accesses (we call it *turn-around time*, i.e., $z_i + c_i + R(s_b)$) as the performance metric. Since a certain number of instructions is executed during a given think time z_i, the shorter the turn-around time is, the higher the instruction throughput and thus the better the performance. Based on this metric, we propose the following optimization for FastCap:

$$\text{Maximize} \quad D \tag{4}$$

$$\text{subject to} \quad \frac{z_i + c_i + R(s_b)}{\overline{z_i} + c_i + R(\overline{s_b})} \leq 1/D \quad \forall i \in \mathcal{N} \tag{5}$$

$$\sum_i P_i \left(\frac{\overline{z_i}}{z_i}\right)^{\alpha_i} + P_m \left(\frac{\overline{s_b}}{s_b}\right)^{\beta} + P_s \leq B\overline{P} \tag{6}$$

$$s_b \geq \overline{s_b}, \quad z_i \geq \overline{z_i}, \quad s_b, z_i \in \mathbb{R} \quad \forall i \in \mathcal{N}. \tag{7}$$

The optimization is over z_i and s_b. The objective is to maximize the performance (or to minimize the performance degradation $1/D$ as much as possible). The constraint in Equation (5) specifies that each core's average turnaround time can only be at most $1/D \geq 1$ of the minimum average turnaround time for that core. (Recall that a higher

turnaround time means lower performance.) To guarantee fairness, we apply the same upper-bound $1/D$ for all cores with respect to their best possible performance (highest core and memory frequencies). The constraint in Equation (6) specifies that the total power consumption (core power plus memory power plus system background power) should be no higher than the power budget. The budget is expressed as the peak full-system power $\overline{P}$ multiplied by a given budget fraction $0 < B \leq 1$. The constraints in Equation (7) specify the range of each variable. Since the objective function and each constraint are convex, the optimization problem is convex.

Note that the optimization problem is constrained by the overall system budget. However, it can be extended to capture per-processor power budgets by adding a constraint similar to the constraint in Equation (6) for each processor.

FastCap solves the optimization problem for z_i and s_b, and then sets each core (memory) frequency to the value that, after normalized to the maximum frequency, is the closest to $\overline{z_i}/z_i$ ($\overline{s_b}/s_b$). For the cores and memory controller, a change in frequency may entail a change in voltage as well. Thus, the power consumed by each core and memory is always dynamically adjusted based on the applications' performance needs. The coupling of the objective in Equation (4) and constraint in Equation (5) seeks to minimize the performance degradation of the application that is furthest away from its best possible performance. Since each core has its own minimum turnaround time and the same upper-bound proportion is applied to all cores, we ensure fairness among them and mitigate the performance outlier problem.

The optimization problem can be solved quickly using numerical solvers such as CPLEX. However, it can be solved substantially faster using the following observations.

THEOREM 4.1. *Suppose the solution D^*, s_b^* and $z_i^*, i \in \mathcal{N}$ are the optimal solution to the optimization problem. Then, the inequalities in Equations (5) and (6) must be equalities.*

We defer the proof to the appendix.

Theorem 4.1 suggests that the optimal solution must consume the entire power budget and each core must operate at $1/D$ times of its corresponding target. With the constraints in Equations (5) and (6) as equalities, the optimal think time z_i can be solved in linear time $O(N)$ for a given bus time s_b. This is because z_i can be written as

$$z_i = \frac{\overline{z_i} + c_i + R(\overline{s_b})}{D} - c_i - R(s_b). \tag{8}$$

We then substitute Equation (8) into the constraint in Equation (6) and solve for D using the equality condition for the constraint in Equation (6). Then, all optimal z_i can be computed in linear time using Equation (8).

We can then exhaustively search through M possible values for s_b to find the globally optimal solution. However, since the optimization problem is convex, we only need to find a local optimal. Since we can find an optimal solution for each bus transfer time s_b, we can simply perform a binary search across all M possible values for s_b to find the local optimal. This results in the $O(N \log M)$ algorithm shown in Algorithm 1.

4.2. FastCap Implementation

4.2.1. Operation. FastCap splits time into fixed-size epochs of L milliseconds each. It collects performance counters from each core $300\mu s$ into each epoch and uses them as inputs to the frequency selection algorithm. We call this $300\mu s$ the *profiling phase*, and empirically, we find its length enough to capture the latest application behaviors. During the profiling phase, the applications execute normally.

Given the inputs, the OS runs the FastCap algorithm and, depending on the outcome, may transition to new core and/or memory voltage/frequencies for the remainder of the

ALGORITHM 1: FastCap $O(N \log M)$ Algorithm

Inputs: $\{P_i\}$, $\{\alpha_i\}$, P_m, β, P_s, $\{\overline{z_i}\}$, $\overline{s_b}$, Q, U, s_m, B, $\overline{P}$, and an ordered array of M candidate values for s_b.
Outputs: $\{z_i\}$ and s_b
Let $\ell := 0$ and $r := M - 1$.
while $\ell \neq r$ **do**
  $m := (\ell + r)/2$.
  Solve the optimal D for the m^{th} s_b value.
  Solve the optimal D for the $(m \pm 1)^{th}$ s_b values. Let the optimal D be denoted as D^+ and D^-, respectively.
  if $D < D^+$ **then**
    $\ell := m$
  else if $D^- > D$ **then**
    $r := m$
  else
    break
  end if
end while
Set each core (memory) frequency to the closest frequency to $\overline{z_i}/z_i$ ($\overline{s_b}/s_b$) after normalization.

epoch. During a core's frequency transition, the core does not execute instructions. To adjust the memory frequency, all memory accesses are temporarily halted, and PLLs and DLLs are resynchronized. The core and memory transition overheads are negligible (tens of microseconds) compared to the epoch length (in milliseconds).

4.2.2. Collecting Input Parameters. Several key FastCap parameters, such as P_i, α_i, P_m, β, the minimum think time $\overline{z_i}$, and queue sizes Q and U, come directly or indirectly from performance counters. Now, we detail how we obtain the inputs to the algorithm from the counters. To compute $\overline{z_i}$, we use the following counters: (1) TPI_i, *Time Per Instruction* for core i during profiling; (2) TIC_i, *Total Instructions Executed* during profiling; and (3) TLM_i, *Total Last-Level Cache Misses* (or number of memory accesses) during profiling. The ratio between TIC_i and TLM_i is the average number of instructions executed between two memory accesses. We then set $\overline{z_i}$ as

$$TPI_i \times \frac{TIC_i}{TLM_i} \tag{9}$$

and then scale it by the ratio between the maximum frequency and the frequency used during profiling.

To obtain P_i, α_i, P_m, and β, FastCap keeps data about the last three frequencies it has seen and periodically recomputes these parameters. To obtain Q and U, we use the performance counters that log the average queue sizes at each memory bank and bus. We obtain Q by taking the average queue size across all banks. We obtain U directly from the corresponding counter. To obtain s_m, we take the average memory access time at each bank during the profiling phase. The minimum bus transfer time $\overline{s_b}$ is a constant and, since each request takes a fixed number of cycles to be transferred on the bus (the exact number depends on the bus frequency), we simply divide the number of cycles by the maximum memory frequency to obtain $\overline{s_b}$. All background power draws (independent of core/memory frequencies or workload) are measured and/or estimated statically.

4.2.3. Hardware and Software Costs. FastCap requires no architectural or software support beyond that in [13]. Specifically, core DVFS is widely available in commodity hardware, although today one may see fewer voltage domains than cores. Research has shown this is likely to change soon [23, 41]. Existing DIMMs support multiple frequencies and can switch among them by transitioning to power-down or self-refresh states [22], although this capability is typically not used by current servers. Integrated CMOS memory controllers can leverage existing DVFS technology. One needed change is for the memory controller to have separate voltage and frequency control from other processor components. In recent Intel architectures, this would require separating shared cache and memory controller voltage control.

In terms of software, the OS must periodically invoke FastCap and collect several performance counters. When FastCap adjusts the frequency of a component, the operation in that component is suspended briefly. However, FastCap operates at the granularity of milliseconds and transition latencies are in the tens of microseconds, so the overheads are negligible. Furthermore, as we show in Section 4.3.2, the algorithm execution is also in the tens of microseconds and thus negligible as well.

4.3. FastCap Evaluation

4.3.1. Methodology. We adopt the simulation infrastructure used in [13]. We assume per-core DVFS, with 10 equally spaced frequencies in the range 2.2 to 4.0GHz. We assume a voltage range matching Intel's Sandybridge, from 0.65V to 1.2V, with voltage and frequency scaling proportionally, which matches the behavior we measured on an i7 CPU. We scale memory controller frequency and voltage, but only frequency for the memory bus and DRAM chips. The on-chip four-channel memory controller has the same voltage range as the cores, and its frequency is always double that of the memory bus. We assume that the bus and DRAM chips may be frequency scaled from 800MHz down to 200MHz, in steps of 66MHz. The infrastructure simulates in detail the aspects of cores, caches, memory controllers, and memory devices that are relevant to our study, including memory device power and timing, and row buffer management. Table II lists our default simulation settings.

We model the power for the non-CPU, nonmemory components as a fixed 10W. Under our baseline assumptions, at maximum frequencies, the CPU accounts for roughly 60%, the memory subsystem 30%, and other components 10% of system power.

We construct the workloads by combining applications from the SPEC 2000 and SPEC 2006 suites. We group them into the same mixes as [14, 42]. The workload classes are memory intensive (MEM), compute intensive (ILP), compute-memory balanced (MID), and mixed (MIX, one or two applications from each other class). We run the best 100M-instruction simulation point for each application (selected using Simpoints 3.0). A workload terminates when its slowest application has run 100M instructions. Table III describes the workloads and the L2 misses per kilo-instruction (MPKI) and writebacks per kilo-instruction (WPKI) for $N = 16$. We execute $N/4$ copies of each application to occupy all N cores.

4.3.2. Results. We first run all workloads under the maximum frequencies to observe the peak power the system ever consumed. We observe the peak power $\overline{P}$ to be 60 watts for four cores, 120 watts for 16 cores, 210 watts for 32 cores, and 375 watts for 64 cores. We present results for a 16-core system in which FastCap is called every $L = 5ms$. (The $5ms$ epoch length matches a common OS time quantum.)

Power consumption. We first evaluate FastCap under a 60% power budget fraction; that is, B in Equation (6) equals 60%. Figure 5 shows the average power spent by FastCap running each workload on the 16-core system. FastCap successfully maintains overall system power to be just under 60% of the peak power.

Table II. Main System Settings

Feature		Value
CPU cores		N in-order, single thread, 4GHz Single IALU IMul FpALU FpMulDiv
L1 I/D cache (per core)		32KB, 4-way, 1 CPU cycle hit
L2 cache (shared)		16MB, N-way, 30 CPU cycle hit
Cache block size		64 bytes
Memory configuration		4 DDR3 channels for 16 and 32 cores 8 DDR3 channels for 64 cores 8 2GB ECC DIMMs
Time	tRCD, tRP, tCL	15ns, 15ns, 15ns
	tFAW	20 cycles
	tRTP	5 cycles
	tRAS	28 cycles
	tRRD	4 cycles
	Refresh period	64ms
Current	Row buffer	250mA (read), 250mA (write)
	Precharge	120mA
	Active standby	67mA
	Active pwrdown	45mA
	Precharge standby	70mA
	Precharge pwrdown	45mA
	Refresh	240mA

Table III. Workload Descriptions

Name	MPKI	WPKI	Applications ($\times N/4$ each)			
ILP1	0.37	0.06	vortex	gcc	sixtrack	mesa
ILP2	0.16	0.03	perlbmk	crafty	gzip	eon
ILP3	0.27	0.07	sixtrack	mesa	perlbmk	crafty
ILP4	0.25	0.04	vortex	gcc	gzip	eon
MID1	1.76	0.74	ammp	gap	wupwise	vpr
MID2	2.61	0.89	astar	parser	twolf	facerec
MID3	1.00	0.60	apsi	bzip2	ammp	gap
MID4	2.13	0.90	wupwise	vpr	astar	parser
MEM1	18.22	7.92	swim	applu	galgel	equake
MEM2	7.75	2.53	art	milc	mgrid	fma3d
MEM3	7.93	2.55	fma3d	mgrid	galgel	equake
MEM4	15.07	7.31	swim	applu	sphinx3	lucas
MIX1	2.93	2.56	applu	hmmer	gap	gzip
MIX2	2.55	0.80	milc	gobmk	facerec	perlbmk
MIX3	2.34	0.39	equake	ammp	sjeng	crafty
MIX4	3.62	1.20	swim	ammp	twolf	sixtrack

Figure 6 shows the FastCap behavior for three power budgets (as a fraction of the full-system peak power) for the MEM3 workload as a function of epoch number. The figure shows that FastCap corrects budget violations very quickly (within 10ms), regardless of the budget. Note that MEM3 exhibits per-epoch average powers somewhat lower than the cap for $B = 80\%$. This is because memory-bound workloads do not consume 80% of the peak power, even when running at the maximum core and memory frequencies.

Application performance. Recall that, under tight power budgets, FastCap seeks to achieve similar (fractional) performance losses compared to using maximum

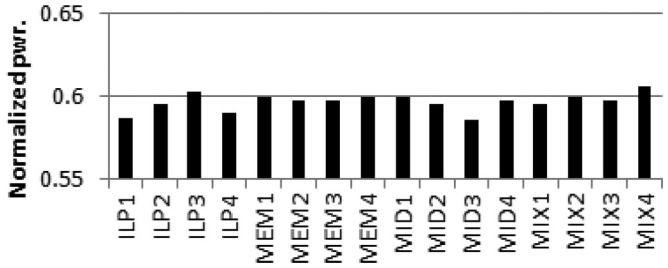


Fig. 5. Average power normalized to the peak power. Power budget is 60% of the peak power. FastCap successfully maintains power consumption at the specified 60% level for a wide range of applications.

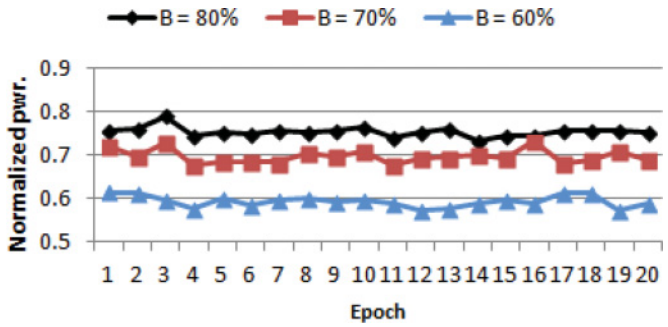


Fig. 6. Average power, normalized to the peak power when running MEM3 as a function of time and power budget.

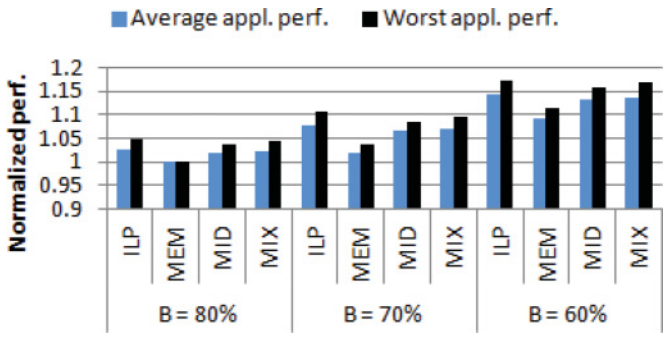


Fig. 7. Average and worst application performance for each workload class and three power budgets. Values above 1 represent the percentage application performance loss.

frequencies for all applications. So, where we discuss a *performance loss* later, we are referring to the performance degradation (compared to a run at maximum frequency) due to power capping, and *not* to the absolute performance.

Figure 7 shows the average and worst application performance (in cycles per instruction or CPI) normalized to the baseline system (maximum core and memory frequencies) for all ILP, MEM, MID, and MIX workloads. The higher the bar, the worse the performance is compared to the baseline. For each workload class, we compute the average and worst application performance across all applications in workloads of the class. For example, the ILP average performance is the average CPI of all applications in ILP1, ILP2, ILP3, and ILP4, whereas the worst performance is the highest CPI

among all applications in these workloads. In the figure, values above 1 represent the percentage application performance loss.

Figure 7 shows that the worst application performance differs only slightly from the average performance. This result shows that FastCap is fair in its (performance-aware) allocation of the power budget to applications. The figure also shows that the performance of memory-bound workloads (MEM) tends to degrade less than that of CPU-bound workloads (ILP) under the same power budget. This is because the MEM workloads usually consume less full-system power than their ILP counterparts. Thus, for the same power budget, the MEM workloads require smaller-frequency reductions, and thus exhibit smaller fractional performance losses.

Core/memory frequencies. FastCap smartly adjusts the core and memory frequencies based on the application needs. For instance, in the CPU-bound workload ILP1, the cores run at high frequency (around 3.5GHz), while the memory runs at low frequency (around 200MHz). In the memory-bound workload MEM1, the cores run at low frequency (around 3.0GHz), while the memory runs at high frequency (around 800MHz). In workload MIX4, which consists of both CPU- and memory-bound applications, memory frequencies are in the middle of the range (around 500MHz). The exact frequencies FastCap selects vary in epochs depending on the workload dynamics in each epoch.

FastCap compared with others policies. We now compare FastCap against other power capping policies. *All policies are capable of controlling the power consumption to around the budget*, so we focus mostly on their performance implications. We first compare against policies that do *not* use memory DVFS.

"CPU-only" sets the core frequencies using the FastCap algorithm for every epoch but keeps the memory frequency fixed at the maximum value. The comparison to CPU-only isolates the impact of being able also to manage memory subsystem power using DVFS. All prior power capping policies suffer from the lack of this additional capability.

"Freq-Par" is a control-theoretic policy from [29]. In Freq-Par, the core power is adjusted in every epoch based on a linear feedback control loop; each core receives a frequency allocation that is based on its power efficiency. Freq-Par uses a linear power-frequency model to correct the average core power from epoch to epoch. We again keep the memory frequency fixed at the maximum value.

Figure 8 shows the performance comparison between FastCap and these policies on a 16-core system. FastCap performs at least as well as CPU-only in both average and worst application performance, showing that the ability to manage memory power is highly beneficial. Setting memory frequency at the maximum causes the cores to run slower for CPU-bound applications, in order to respect the power budget. This leads to severe performance degradation in some cases. For the MEM workloads, FastCap and CPU-only perform almost the same, as the memory subsystem can often be at its maximum frequency in FastCap to minimize performance loss within the power budget. Still, it is often beneficial to change the power balance between cores and memory, as workloads change phases. FastCap is the only policy that has the ability to do so.

The comparison against Freq-Par is more striking. FastCap (and CPU-only) performs substantially better than Freq-Par in both average and worst application performance. In fact, Freq-Par shows significant gaps between these types of performance, showing that it does not allocate power fairly across applications (inefficient cores receive less of the overall power budget). Moreover, Freq-Par's linear power-frequency model can be inaccurate and causes the feedback control to overcorrect and undercorrect often. This leads to severe power oscillation, although the long-term average is guaranteed by the control stability. For example, the power oscillates between 53% and 65% under Freq-Par for MIX3.

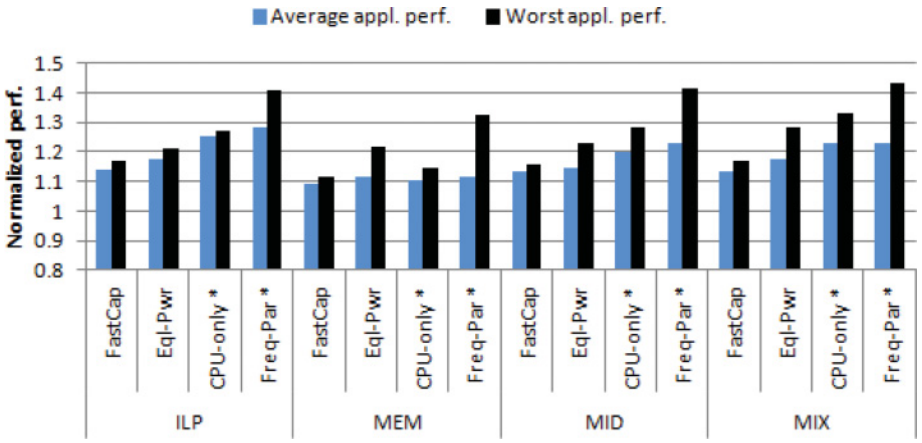


Fig. 8. FastCap compared with CPU-only,* Freq-Par,* and Eql-Pwr in normalized average/worst application performance. Values above 1 represent the percentage application performance loss. "*" indicates fixed memory frequency. Power budget is 60% of the peak power.

Next, we study policies that use DVFS for *both* cores and the memory subsystem. These policies are inspired by prior works, but we add FastCap's ability to manage memory power to them:

"Eql-Pwr" assigns an equal share of the overall power budget to all cores, as proposed in [35]. We implement it as a variant of FastCap: for each memory frequency, we compute the power share for each core by subtracting the memory power (and the background power) from the full-system power budget and dividing the result by N. Then, we set each core's frequency as high as possible without violating the per-core budget. For each epoch, we search through all M memory frequencies and use the solution that yields the best D in Equation (4).

"Eql-Freq" assigns the same frequency to all cores, as proposed in [20]. Again, we implement it as a variant of FastCap: for each epoch, we search through all M and F frequencies to determine the pair that yields the highest D in Equation (4).

"MaxBIPS" was proposed in [21]. Its goal is to maximize the total number of executed instructions in each epoch, that is, to maximize the throughput. To solve the optimization, [21] exhaustively searches through all core frequency settings. We implement this search to evaluate all possible combinations of *core and memory* frequencies within the power budget.

Eql-Pwr ignores the heterogeneity in the applications' power profiles. By splitting the core power budget equally, some applications receive too much budget and even running at the maximum frequency cannot fully consume it. Meanwhile, some power-hungry applications do not receive enough budget and thus result in performance loss. This is most obvious in workloads with a mixture of CPU-bound and memory-bound applications (e.g., MIX4). As a result, we observe in Figure 8 that Eql-Pwr's worst application performance loss is often much higher than FastCap's.

Eql-Freq also ignores application heterogeneity. In Eql-Freq, having all core frequencies locked together means that some applications may be forced to run slowly, because raising all frequencies to the next level may violate the power budget. This is a more serious problem when the workload consists of a mixture of CPU- and memory-bound applications on a large number of cores. To see this, Figure 9(a) plots the normalized average and worst application performance for FastCap and Eql-Freq when running the MIX workloads on a 64-core system. (We choose to show results on a 64-core system

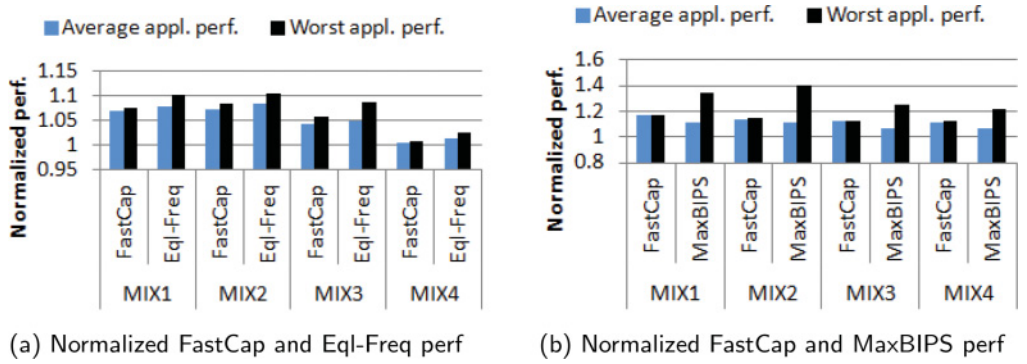


Fig. 9. FastCap compared with EqlFreq and maxBIPS. Values above 1 represent the percentage application performance loss.

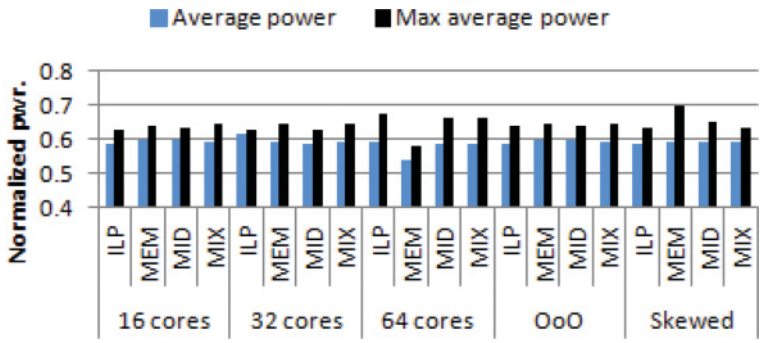


Fig. 10. FastCap average power and maximum average power, both normalized to the peak power in many configurations. Power budget is 60% of the peak power.

to magnify the comparison.) The figure shows that Eql-Freq is more conservative than FastCap and often cannot fully harvest the power budget to improve performance.

Finally, besides its use of exhaustive search, the main problem of MaxBIPS is that it completely disregards fairness across applications. Figure 9(b) compares the normalized average and worst application performance for the MIX workloads under a 60% budget. Because of the high overhead of MaxBIPS, the figure shows results for only four-core systems. The figure shows that FastCap is slightly inferior in average application performance, as MaxBIPS always seeks the highest possible instruction throughput. However, FastCap achieves significantly better worst application performance and fairness. To maximize the overall throughput, MaxBIPS may favor applications that are more power efficient, that is, have higher throughput at a low power cost. This reduces the power allocated to other applications and the outlier problem occurs. This is particularly true for workloads that consist of a mixture of CPU- and memory-bound applications.

Impact of number of cores. We now study the impact of the number of cores on FastCap's ability to limit power draw (Figure 10) and to provide fair application performance (Figure 11).

Figure 10 depicts pairs of bars for each workload class on systems with 16, 32, and 64 cores, under a 60% power budget. The bar on the right of each pair is the maximum average power of any epoch of any application of the same class normalized to the peak power, whereas the bar on the left is the normalized average power *for the workload*

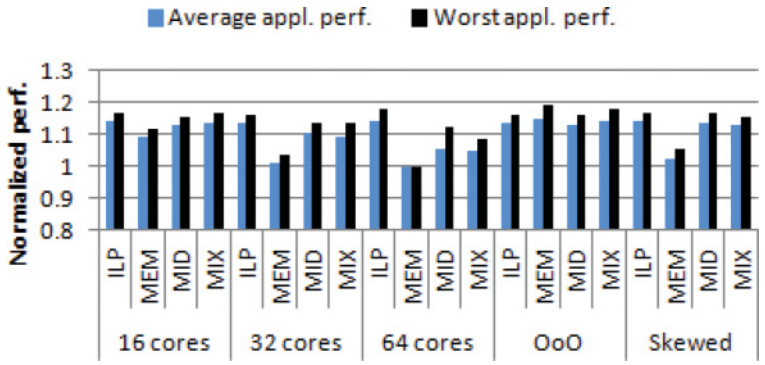


Fig. 11. Normalized FastCap average and worst application performance in many configurations. Values above 1 represent the percentage application performance loss. Power budget is 60% of the peak power.

with the maximum average power. Comparing these bars determines whether FastCap is capable of respecting the budget even when there are a few epochs with slightly higher average power. The figure clearly shows that FastCap is able to do so (all average power bars are at or slightly below 60%), even though increasing the number of cores does increase the maximum average power slightly. This effect is noticeable for workloads that have CPU-bound applications on 64 cores. In addition, note that the MEM workloads do not reach the maximum budget on 64 cores, as these workloads do not consume the power budget on this large system even when they run at maximum frequencies.

Figure 11 also shows pairs of bars for each workload class under the same assumptions. This time, the bar on the right of each pair is the normalized worst performance among all applications in a class, and the bar on the left is the normalized average performance of all applications in the class. The figure shows that FastCap is very successful at allocating power fairly across applications, regardless of the number of cores; the worst application performance is always only slightly worse than the average performance. The figure also shows that application performance losses decrease as we increase the number of cores, especially for MEM workloads. (Recall that we are comparing performance *losses* due to power capping; absolute performance cannot be compared because the baselines are different.) The reason is that MEM workloads on the larger numbers of cores are bottlenecked by the memory subsystem even when they execute at maximum frequencies.

Epoch length and algorithm overhead. By default, FastCap runs at the end of every OS time quantum (5*ms* in our experiments so far in the article). The overhead of FastCap scales linearly with the number of cores. Specifically, we run the FastCap algorithm for 100*k* times and collect the average time of each execution. The average time is 33.5μs for 16 cores, 64.9μs for 32 cores, and 133.5μs for 64 cores. For a 5*ms* epoch length, these overheads are 0.7%, 1.3%, and 2.7% of the epoch lengths, respectively. If these levels of overhead are unacceptable, FastCap can execute at a coarser granularity. Using our simulator, we studied epoch lengths of 10*ms* and 20*ms*. We find that these epoch lengths do not affect FastCap's ability to control average power and performance for the applications and workloads we consider.

Out-of-order (OoO) execution. Our results so far have assumed in-order cores. However, FastCap can be easily extended to handle the OoO executions. In OoO, the core may be able to continue executing the next instruction after issuing a memory access without stalling. In FastCap's terminology, the think time thus becomes the

interval between two core stalls (not between two main memory accesses). The workload becomes more CPU bound.

Unfortunately, our trace-based methodology does not allow detailed OoO modeling. However, we can approximate the latency hiding and additional memory pressure of OoO. Specifically, we simulate idealized OoO executions by assuming a large instruction window (128 entries) and disregarding instruction dependencies within the window. This models an upper bound on the memory-level parallelism (and has no impact on instruction-level parallelism, since we still simulate a single-issue pipeline).

Figure 10 shows four pairs of bars for the OoO executions of the workload classes on 16 cores and under a 60% power budget. The results can be compared to the bars for 16 cores on the left side of the figure. This comparison shows that FastCap is equally successful at limiting the power draw to the budget, regardless of the processor execution mode.

Similarly, Figure 11 shows four pairs of bars for OoO executions on 16 cores, under a 60% budget. These performance loss results can also be compared to those for 16 cores on the left of this figure. The comparison shows that workloads with memory-bound applications tend to exhibit higher performance losses in OoO execution mode. The reason is that the performance of these applications improves significantly at maximum frequencies, as a result of OoO; both cores and memory become more highly utilized. When FastCap imposes a lower-than-peak budget, frequencies must be reduced and performance suffers more significantly. Directly comparing frequencies across the execution modes, we find that memory-bound workloads tend to exhibit higher core frequencies and lower memory frequency under OoO than under in-order execution. This result is not surprising since the memory can become slower in OoO without affecting performance because of the large instruction window. Most importantly, FastCap is still able to provide fairness in power allocation in OoO, as the performance losses are roughly evenly distributed across all applications.

Multiple memory controllers. So far we have assumed a single memory controller. In many-core systems, there may be multiple memory controllers, each handling a subset of the memory channels. For FastCap to support multiple memory controllers (operating at the same frequency), we use the existing performance counters to keep track of the average queue sizes Q and U of each memory controller. Thus, different memory controllers can have different response times (cf. Equation (1)). We also keep track of the probability of each core's requests going through each memory controller. In this approach, the response time R in Equation (5) becomes a weighted average across all memory controllers and different cores experience different response times.

To study the impact of multiple memory controllers, we simulate four controllers in our 16-core system. In addition, we simulate two memory interleaving schemes: one in which the memory accesses are uniformly distributed across memory controllers, and one in which the distribution is highly skewed. In the uniformly distributed case, all memory controllers have roughly the same response time and all cores see the same response time. In the skewed distribution, some memory controllers are overloaded.

Figure 10 shows four pairs of bars for the skewed distribution on 16 cores, under a 60% budget. Compare these results to the 16-core data on the left side of the figure. The skewed distribution causes higher maximum power in the MEM workloads. Still, FastCap is able to keep the average performance for the workload with this maximum power slightly below the 60% budget.

Again, Figure 11 shows four pairs of the skewed distribution on 16 cores, under a 60% budget. We can compare these performance losses to the 16-core data on the left of the figure. The comparison shows that FastCap provides fair application performance even under multiple controllers with highly skewed access distributions.

Finally, FastCap uses the same frequency for all memory controllers. However, it may be more desirable to have different controllers running at different frequencies. This would introduce extra algorithmic complexity and we leave it as future work.

5. FASTENERGY

In this section, we introduce FastEnergy, an optimization framework and search algorithm for energy conservation developed based on the queuing model discussed in Section 3.

The goal of FastEnergy is to conserve as much energy as possible without degrading the performance of applications by more than a user-defined (percentage) bound. Accomplishing this online is challenging for two reasons: (1) the future amount of workload is not known in advance, and (2) there is a combinatorial space of possible power/performance states from which to choose in many-core systems. Our solution is to leverage the queuing model from Section 3 to select power/performance states efficiently, while limiting the performance degradation to the bound. As we show in Table I, FastEnergy has the lowest algorithmic time complexity than the state-of-the-art approaches to energy conservation. We show in Section 5.3 that FastEnergy also conserves more energy among those approaches without violating the performance loss bound.

5.1. FastEnergy Optimization Algorithm

Based on the queuing model (cf. Figure 2), we design FastEnergy to minimize the average energy required for all cores to complete one memory request each (or equivalently, execute all the instructions between two memory requests), which is

$$\sum_i P_i \left(\frac{\overline{z_i}}{z_i}\right)^{\alpha_i} z_i + \left[P_m \left(\frac{\overline{s_b}}{s_b}\right)^{\beta} + P_s\right] \frac{\sum_i (R(s_b) + z_i + c_i)}{N}, \tag{10}$$

subject to the per-core delay constraints:

$$R(s_b) + z_i + c_i \leq T_i \quad \forall i \in \mathcal{N}. \tag{11}$$

The parameter T_i constrains the maximum allowable performance degradation and specifies the performance requirement for each core in terms of the average time to complete one memory access (cf. Figure 2).

The optimization variables are z_i and s_b, which must satisfy

$$\overline{s_b} \leq s_b, \quad \overline{z_i} \leq z_i \quad \forall i \in \mathcal{N}. \tag{12}$$

The first term in Equation (10),

$$\sum_i P_i \left(\frac{\overline{z_i}}{z_i}\right)^{\alpha_i} z_i,$$

corresponds to the energy consumed by the cores during instruction execution (cf. Equation (2)). The energy associated with the static (voltage/frequency-independent) power is captured by P_s in the second term. The second term,

$$\left[P_m \left(\frac{\overline{s_b}}{s_b}\right)^{\beta} + P_s\right] \frac{\sum_i (R(s_b) + z_i + c_i)}{N},$$

corresponds to the energy consumed by the memory and all other power components that do not depend on CPU/memory frequencies. The second term is averaged over the number of cores N because it is the background power seen by all the cores. Because of the term $P_m z_i (\overline{s_b}/s_b)^{\beta}$, the objective function in Equation (10) is nonconvex in z_i and s_b.

For out-of-order cores, Equation (10) can be interpreted as the average energy needed for all cores to complete one *blocking* memory request each.

To derive FastEnergy's algorithm, we make the following key observation in Theorem 5.1.

THEOREM 5.1. *For each given bus transfer time s_b, the optimization problem (Equations (10) to (12)) is conditionally convex in the z_i, and the optimal think time z_i can be determined in linear time $O(N)$ and in closed form. Specifically, each optimal z_i can be one of only three values z_i', $\widehat{z_i}$, and $\overline{z_i}$ (the minimum think time), where*

$$z_i' = T_i - c_i - R(s_b),$$

$$\widehat{z_i} = \left(\frac{(\alpha_i - 1) N P_i \overline{z_i}^{\alpha_i}}{P_s + P_m (\overline{s_b}/s_b)^{\beta}} \right)^{\frac{1}{\alpha_i}}.$$

Intuitively, since the bus transfer time is experienced by all N cores, it is natural first to determine its value and then, conditioned on the bus transfer time, determine the think time for each core. We defer the proof of Theorem 5.1 to the appendix.

Motivated by Theorem 5.1, FastEnergy implements the following algorithm. For each bus transfer time s_b, it computes the think times z_i for all cores in linear time. Then, it evaluates the objective function in Equation (10) using the computed z_i and s_b. After exhausting all the M possible values for s_b, it returns the outputs with the minimum objective value. The algorithm complexity is $O(NM)$, and we outline the pseudo-code in Algorithm 2.

The optimality of Algorithm 2 is shown in Theorem 5.2.

THEOREM 5.2. *Algorithm 2 computes the global optimal solution for the nonconvex optimization problem (Equations (10) to (12)).*

We defer the proof of Theorem 5.2 and the derivation of Algorithm 2 to the appendix. Note that the optimization problem (Equations (10) to (12)) itself is nonconvex (it only becomes convex for a given bus transfer time). However, as the bus transfer time can only take M discrete values, we are able to exhaustively search for the global optimal solution.

5.2. FastEnergy Implementation

5.2.1. Operation. We operate FastEnergy the same way as we operate FastCap in Section 4.2.1. The optimization parameters, such as P_i, α_i, P_m, and β; the minimum think time $\overline{z_i}$; and queue sizes Q and U are computed the same way as in FastCap.

5.2.2. Performance Management. We now discuss how we set the performance constraint T_i. This constraint is recalculated at the beginning of each epoch. Similar to the approach proposed in [25], our policy is based on the notion of performance slack. The performance slack is the difference between the execution time of a baseline implementation and the execution time targeted by FastEnergy. The baseline implementation does not use energy management. Rather, it keeps the frequencies of all cores and of the memory at the maximum. The amount of slack controls the tradeoff between performance and energy consumption. In contrast to the prior works [25], we use the number of memory accesses to manage performance.

To understand how we do so, recall that $R(s_b) + z_i + c_i$ is the average time between two consecutive memory accesses by core i. Thus, $L/(R(s_b) + z_i + c_i)$ is the average number of accesses issued by core i during an $L ms$ epoch. Let $R(\overline{s_b})$ denote the average memory access time under the maximum memory frequency, that is,, $R(\overline{s_b}) = Q(s_m + U\overline{s_b})$ (cf. Equation (1)), and then $L/(R(\overline{s_b}) + \overline{z_i} + c_i)$ is the average number of accesses in the baseline system. FastEnergy's performance target is to complete a fraction of

ALGORITHM 2: FastEnergy$O(NM)$ Algorithm

Inputs: $\{P_i\}$, $\{\alpha_i\}$, P_m, β, P_s, $\{\overline{z_i}\}$, $\overline{s_b}$, Q, U, s_m, B, $\overline{P}$, and an ordered array of M candidate values for s_b.
Outputs: $\{z_i\}$ and s_b
Initialize $s_b = \overline{s_b}$ and $z_i = \overline{z_i}$, for all $i \in \mathcal{N}$.
Compute performance constraint T_i, for all $i \in \mathcal{N}$.
for each s_b in an increasing order in $\mathcal{M}$ **do**
  for each core $i \in \mathcal{N}$ **do**
    Compute $\widehat{z_i} = \left(\frac{(\alpha_i - 1)NP_i\overline{z_i}^{\alpha_i}}{Ps + Pm(\overline{s_b}/s_b)^{\beta}}\right)^{\frac{1}{\alpha_i}}$.
    Compute $z_i' = T_i - c_i - R(s_b)$.
    if $\overline{z_i} > z_i'$ **then**
      return s_b and z_i, for all $i \in \mathcal{N}$.
    else if $\widehat{z_i} < z_i'$ **and** $\widehat{z_i} > \overline{z_i}$ **then**
      $z_i = \widehat{z_i}$.
    else if $\overline{z_i} < z_i'$ **and** $\frac{(\alpha_i - 1)P_i\overline{z_i}^{\alpha_i}}{z_i'^{\alpha_i}} - \frac{Ps + Pm(\overline{s_b}/s_b)^{\beta}}{N} > 0$ **then**
      $z_i = z_i'$.
    else if $\overline{z_i} < z_i'$ **and** $\frac{Ps + Pm(\overline{s_b}/s_b)^{\beta}}{N} - (\alpha_i - 1)P_i > 0$ **then**
      $z_i = \overline{z_i}$.
    else
      $z_i = \overline{z_i}$.
    end if
  end for
  Evaluate Equation (10) and compare with the best objective value. Update if the best objective value is larger.
end for
return s_b and z_i for all $i \in \mathcal{N}$.

at least $1 - \gamma$ of the memory accesses in the baseline system, where $0 \leq \gamma < 1$. The parameter γ can be interpreted as the maximum allowed fractional performance slowdown/degradation. When $\gamma > 0$, each epoch adds some amount of performance slack.

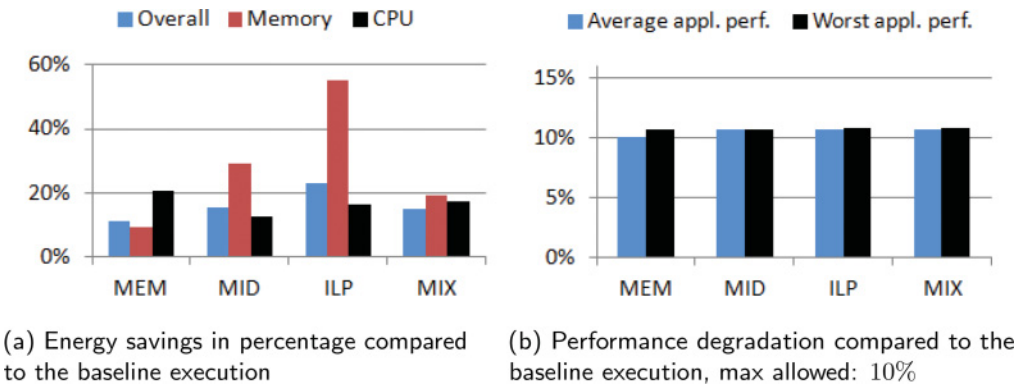
To set the performance constraints (and compute T_i), at the beginning of every epoch and for every core, FastEnergy computes

$$(1 - \gamma)\left(\frac{L}{R(\overline{s_b}) + \overline{z_i} + c_i}\right) + X_i, \tag{13}$$

where the value X_i measures the difference between the target performance and the actual FastEnergy execution in number of memory accesses *so far*. A positive X_i means FastEnergy is falling behind, and thus needs to perform more memory accesses in the next epoch. A negative X_i means FastEnergy is running faster than the target, and the execution can be slowed down if doing so would save energy. Equation (13) provides the desired bound for $L/(R(s_b) + z_i + c_i)$. Rearranging the terms, we arrive at our performance constraint T_i:

$$R(s_b) + z_i + c_i \leq \frac{L}{(1 - \gamma)\left(\frac{L}{R(\overline{s_b}) + \overline{z_i} + c_i}\right) + X_i} =: T_i \quad \forall i. \tag{14}$$

It is possible that Equation (14) cannot be satisfied, meaning that one or more cores cannot catch up in a single epoch. In such a situation, FastEnergy runs those cores at



(a) Energy savings in percentage compared to the baseline execution

(b) Performance degradation compared to the baseline execution, max allowed: 10%

Fig. 12. FastEnergy energy and performance on $N = 16$ cores.

the maximum frequency. With the slack added in the next epochs, the equation will eventually be satisfied again.

5.2.3. Hardware and Software Costs. FastEnergy requires the same features and has the same costs as FastCap.

5.3. FastEnergy Evaluation

To evaluate FastEnergy, we use the same evaluation methodology as for FastCap in Section 4.3.1.

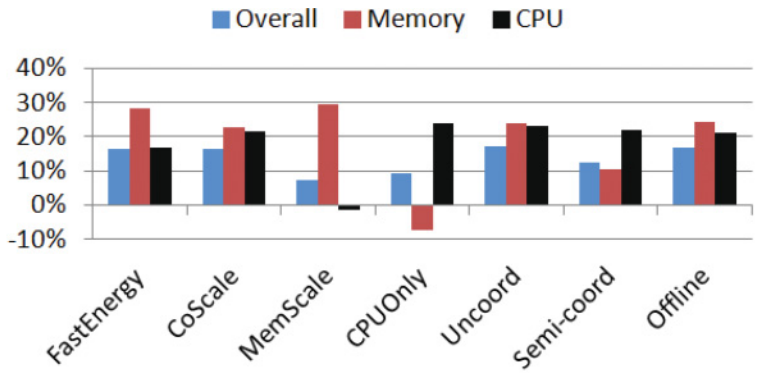
FastEnergy energy and performance. In Figure 12(a), we plot the FastEnergy energy savings (full system, CPU, and memory) as a percentage of the baseline execution (in maximum core and memory frequencies) of our workloads. Figure 12(b) shows the performance degradation compared to the baseline, where the maximum allowed degradation is $\gamma = 0.1$ (constraints in Equation (13)); that is, FastEnergy is allowed to slow applications down by at most 10% compared to the baseline.

As one would expect, Figure 12(a) shows that FastEnergy obtains more energy savings from the memory subsystem in CPU-intensive workloads (long think times z_i) and more savings from the cores in memory-intensive workloads (short think times z_i). The full-system energy savings averaged across all the workloads are 16%.

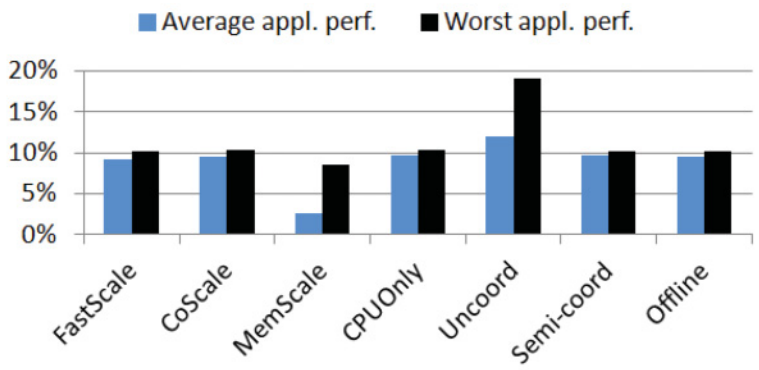
In Figure 12(b), we note that FastEnergy maintains the maximum performance degradation of any application in a workload very close to 10%, the maximum allowed slowdown. For some workloads, both the average and worst-case degradations are lower than 10%. The reason is that, for those workloads, running slower would have *increased* energy consumption (the small savings from the lower dynamic energy would have been outweighed by the static energy).

FastEnergy compared to other methods. In Figure 13, we compare FastEnergy with other energy management methods, again for $N = 16$ cores with 10% maximum allowed performance degradation ($\gamma = 0.1$). The energy savings in the figure are the average savings over all workloads.

We have already described CoScale. MemScale represents the scenario in which the system uses only memory subsystem DVFS. CPUOnly represents the scenario with CPU DVFS only. To be optimistic about this alternative, we assume that it considers all possible combinations of core frequencies and selects the best. In both MemScale and CPUOnly, the performance-aware energy management policy assumes that the behavior of the components that are not being managed will stay the same in the next epoch as in the profiling phase. Uncoordinated (or Uncoord for short) applies to both



(a) Energy savings in percentage compared to the baseline execution



(b) Performance degradation compared to the baseline execution, maximum allowed: 10%

Fig. 13. FastEnergy compared with other methods, $N = 16$.

MemScale and CPU DVFS, but in a completely independent fashion. In determining the performance slack available to it, the CPU power manager assumes that the memory subsystem will remain at the same frequency as in the previous epoch, and that it has accumulated no performance degradation; the memory power manager makes the same assumptions about the cores. Hence, each manager believes that it alone influences the slack in each epoch, which is not the case. Semicoordinated (or simply Semi-coord) increases the level of coordination slightly by allowing the CPU and memory power managers to share the same overall slack; that is, each manager is aware of the past performance degradation produced by the other. However, each manager still tries to consume the entire slack independently in each epoch (i.e., the two managers account for one another's past actions but do not coordinate their estimate of future performance). Finally, Offline relies on a perfect offline performance trace for every epoch and then selects the best frequency for each epoch by considering all possible core and memory frequency settings. As the number of possible settings is exponential, Offline is impractical and is studied simply as an upper bound on how well FastEnergy can do. However, Offline is not necessarily optimal, since it uses the

same epoch-by-epoch greedy decision making as CoScale (i.e., a hypothetical oracle might choose to accumulate slack in order to spend it in later epochs).

We observe that FastEnergy's energy savings and performance degradation are comparable to those of CoScale. The MemScale method only saves memory subsystem energy, as it does not optimize the cores' frequencies. CPU-only does the opposite, only saving energy in the CPU without adapting memory's frequency. Although Uncoordinated can save substantial energy, it is unable to keep performance within the maximum allowed degradation. In some cases, the performance degradation reaches 19%, nearly twice the maximum allowed. Semicoordinated keeps the worst performance degradation within the bound, because both the CPU and memory frequency managers share the same performance estimate. However, because of frequent oscillations and settling at suboptimal solutions, Semicoordinated consumes more energy than FastEnergy. The comparison to Offline shows that both FastEnergy and CoScale behave extremely well in terms of energy savings and performance.

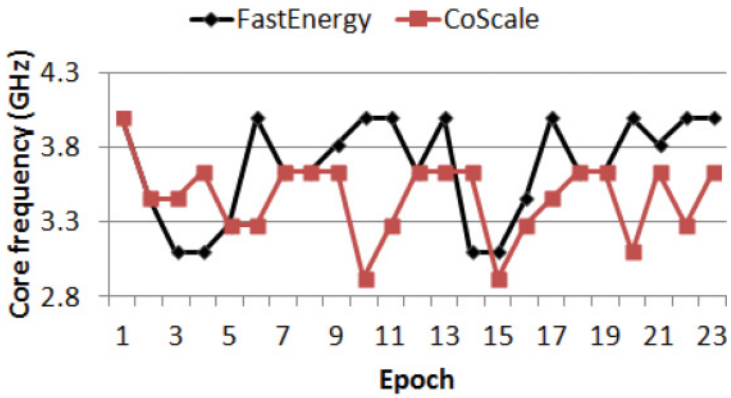
Dynamic behavior. In Figure 14, we present the optimal frequencies selected by FastEnergy alongside the ones selected by CoScale. We plot the frequencies of the core running application applu in MIX2 and the frequencies of the memory over time.

We note that FastEnergy selects different frequencies than CoScale for both the memory and the core. However, the full-system energy savings for FastEnergy running MIX2 is 14.7%, almost the same as the 14.6% for CoScale. This is because, although FastEnergy computes the *global* optimal solution for the optimization (Equations (10) and (11)), the optimization formulation itself is only an approximation of the real system. Algorithms that do not leverage FastEnergy's optimization framework may settle on a different solution with comparable energy savings and performance results.

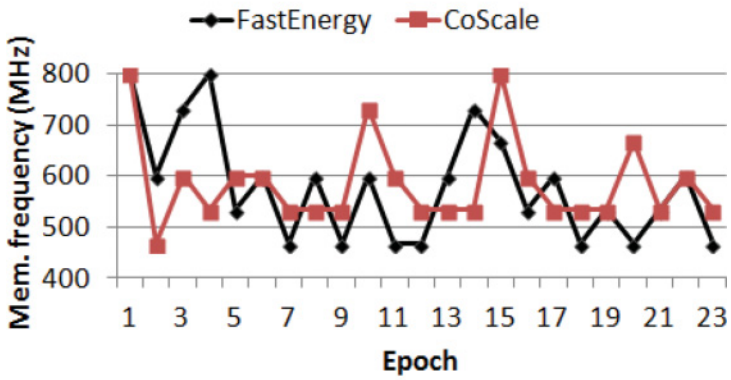
Impact of the number of cores. In Figure 15, we study the (average) energy savings and (average and worst-case) performance degradations of FastEnergy and CoScale running on $N = 16$, 32, 64, and 128 cores. Our results show that both FastEnergy and CoScale are able to maintain the worst-case performance almost exactly within the 10% bound. Their average energy savings are also very similar for all numbers of cores. Interestingly, we note that, as the number of cores increases, the energy savings from the memory subsystem decrease, while the savings from cores increase. This is because, for large N, the memory is almost always busy (i.e., large R), leaving little opportunity for the memory to slow down.

Algorithm overhead. In Figure 16, we plot the algorithm overhead (i.e., how long it takes to search for the next frequency configuration) in μs for FastEnergy and CoScale, as a function of N. Since FastEnergy's complexity is $O(MN)$, it scales linearly with the number of cores, whereas CoScale scales quadratically ($O(M + FN^2)$, where F is the number of core frequencies). In the figure, we also include FastEnergy-H, a heuristic algorithm to further reduce the complexity of FastEnergy to $O(N \log M)$. FastEnergy-H is based on the observation that the memory frequencies can be traversed via a binary search for a near-optimal solution to the optimization problem (Equations (10) to (12)). Most of the overhead in FastEnergy and FastEnergy-H is due to floating-point computations, such as computing $\widehat{z_i}$ (line 8 in Algorithm 2).

As we simulate $5ms$ epochs in these results, the overhead of CoScale is more than 10% with 256 cores, which is clearly unacceptable. To amortize this overhead, we would need substantially longer epochs, which may encompass multiple workload behaviors and render the information collected during profiling (and the predictions made based on it) obsolete. In contrast, FastEnergy achieves the same energy and performance results as CoScale at a much lower overhead. FastEnergy-H performs



(a) Core frequencies

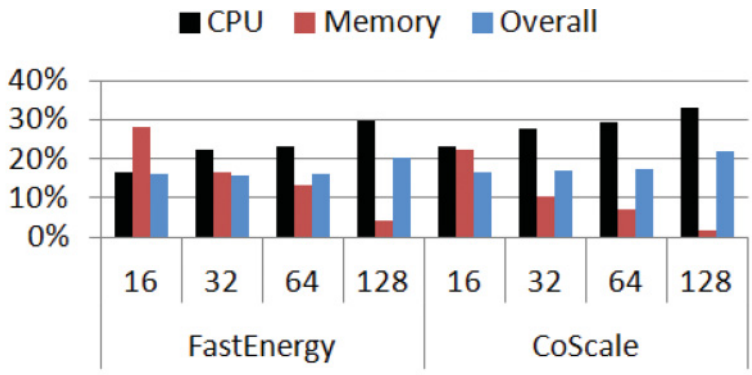


(b) Memory frequencies

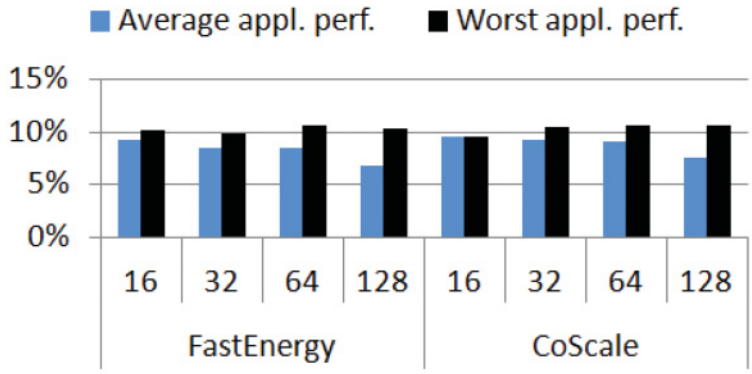
Fig. 14. Frequencies selected while running MIX2. The core frequency plots the core that runs application applu. Similar to CoScale, FastEnergy dynamically adjusts core/memory frequencies to adapt to workload changes.

even better than FastEnergy with almost the same energy and performance. Since $M = 10$ in our experiments, it is roughly 3× faster across the spectrum. Compared to CoScale, FastEnergy-H is roughly 10× faster. In terms of energy and performance, FastEnergy-H's average full-system energy savings are 16.1% (vs. 16% for FastEnergy), and average performance degradation is 8.9% (vs. 9.2% for FastEnergy) for our workloads on 16 cores. (Since FastEnergy-H and FastEnergy only differ significantly in terms of algorithm overhead, we only present FastEnergy results in our other figures.) *These results demonstrate that FastEnergy and FastEnergy-H are practical for many-core systems, whereas CoScale is not.*

Impact of the epoch length. We also study the energy savings and performance degradation under different epoch lengths ($L = 5$, 10, 15, and 20*ms*) on 64 cores. The longer the epochs, the less frequently the frequency selection algorithm executes. Running the algorithm less frequently may lead to worse performance for applications that have fast-changing dynamics.



(a) Energy savings in percentage compared to the baseline execution



(b) Performance degradation compared to the baseline execution, maximum allowed: 10%

Fig. 15. FastEnergy and CoScale with $N = 16$, 32, 64, and 128.

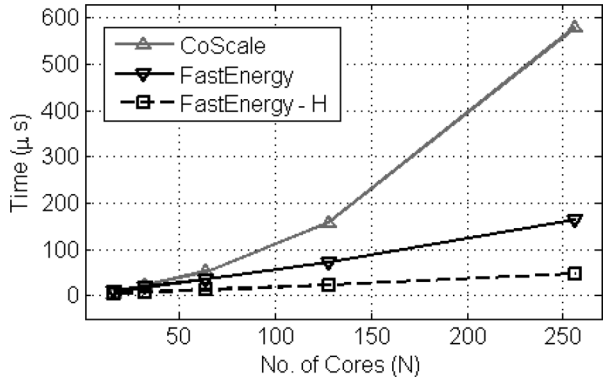
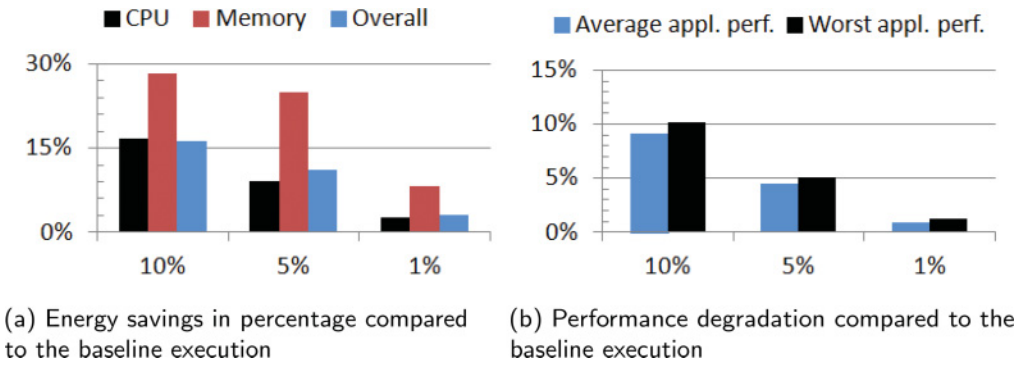


Fig. 16. Algorithm overhead, as a function of core counts N.



(a) Energy savings in percentage compared to the baseline execution

(b) Performance degradation compared to the baseline execution

Fig. 17. FastEnergy for different performance bounds, $N = 16$.

Compared to CoScale, FastEnergy is slightly better at limiting the maximum (and average) performance degradations as the epoch length increases. For example, for $L = 20ms$, the max performance degradation is 10.9% for FastEnergy versus 12.1% for CoScale. This is because the queuing and optimization framework used in FastEnergy model the average behavior of the system, making it less vulnerable to fast-changing workload dynamics. However, FastEnergy's average energy savings are slightly lower (16%) than those of CoScale (17.2%) for $L = 20ms$.

Impact of maximum allowed performance loss. Thus far, we have assumed a performance bound of 10% ($\gamma = 0.1$). Figure 17 plots the average FastEnergy results for different performance bounds (10%, 5%, and 1%) on 16 cores. FastEnergy adapts to different bounds while conserving energy. For example, with a bound of just 1%, FastEnergy conserves as much as 3.1% energy on average.

6. CONCLUSION

In this article, we proposed two algorithms for performance-aware management of active low-power modes in many-core systems. Our first algorithm, called FastCap, maximizes the performance of applications under a full-system power cap, while promoting fairness across applications. Our second algorithm, called FastEnergy, maximizes the full-system energy savings under predefined application performance loss bounds. Both algorithms embody the general queuing model and a nonlinear optimization framework. Our results show that FastCap achieves better application performance and fairness than prior power capping techniques for the same power budget, whereas FastEnergy conserves more energy than prior energy management techniques for the same performance constraint. FastCap and FastEnergy together demonstrate the applicability of the queuing model for managing the abundant active low-power modes in many-core systems.

We leave a few directions for future work. First, it would be interesting to theoretically bound the worst application performance. Second, it would be helpful to implement fine-grained tuning on the worst performing core. For example, a possible solution could be giving extra "weights" for cores that are most underperforming in each epoch. Finally, although this article focuses on DVFS in a single server, our queuing models and optimization approach could be extended for multiserver clusters [16, 17].

7. APPENDIX

7.1. Proof of Theorem 4.1

We first show that the constraint in Equation (6) must be an equality.

Suppose otherwise, and then we can always reduce the optimal bus speed s_b^* such that the performance of each core is improved (because of the decrease in $R(s_b^*)$). As a result, we can achieve a better objective, larger than D^*. This leads to a contradiction. Thus, the power budget constraint must be an equality.

Now, we show that the constraint in Equation (5) must also be an equality. Suppose otherwise, that is, that there exists a j such that the constraint in Equation (5) is strictly smaller than $1/D^*$. Then, we can increase z_j^*. The power budget saved from this core can be redistributed to other cores that have equalities in the constraint in Equation (5). As a result, we can achieve an objective that is larger than D^*. This leads to a contradiction as well.

7.2. Proof of Theorem 5.1

With s_b fixed, the optimization problem reduces to minimizing

$$\sum_i \left(\frac{P_i(\overline{z_i})^{\alpha_i}}{z_i^{\alpha_i - 1}} \right) + \frac{1}{N} \sum_i \left[P_s + P_m \left(\frac{\overline{s_b}}{s_b} \right)^{\beta} \right] z_i, \tag{15}$$

subject to

$$QUs_b + z_i \leq T_i' \quad \forall i \in \mathcal{N}, \tag{16}$$

where $T_i' = T_i - Qs_m - c_i$, and we have substituted $R(s_b)$ by Equation (1). The variable constraints are

$$\overline{s_b} \leq s_b, \quad \overline{z_i} \leq z_i, \quad \forall i \in \mathcal{N}. \tag{17}$$

Since each additive term in Equation (15) is convex in z_i, Equation (15) itself is convex in z_i. Also since the constraints in Equations (16) and (17) are linear in z_i (and thus convex), the optimization problem is convex in z_i with each fixed s_b.

To solve for the optimal z_i, we rely on the Karush-Kuhn-Tucker (KKT) conditions that the optimal z_i must satisfy. Since the optimization problem (Equation (15)) under the constraints in Equations (16) and (17) is convex, the z_i satisfying the KKT conditions are the global optimal solution. The KKT conditions can be written as

$$\frac{P_s + P_m(\overline{s_b}/s_b)^{\beta}}{N} + \lambda_i - \gamma_i = \frac{(\alpha_i - 1)P_i(\overline{z_i})^{\alpha_i}}{z_i^{\alpha_i}} \quad \forall i \in \mathcal{N} \tag{18}$$

$$\lambda_i(T_i' - z_i - QUs_b) = 0 \quad \forall i \in \mathcal{N} \tag{19}$$

$$\gamma_i(z_i - \overline{z_i}) = 0 \quad \forall i \in \mathcal{N} \tag{20}$$

$$z_i + QUs_b \leq T_i' \quad \forall i \in \mathcal{N} \tag{21}$$

$$\lambda_i, \gamma_i \geq 0 \quad z_i \geq \overline{z_i} \quad \forall i \in \mathcal{N}. \tag{22}$$

Based on these KKT conditions, we make the following observations:

1) If $\lambda_i = \gamma_i = 0$, then $z_i = \widehat{z_i}$ provided that $\widehat{z_i}$ satisfies Equations (21) and (22). This corresponds to lines 11 to 12 in Algorithm 2.

2) If $\lambda_i > 0$ and $\gamma_i = 0$, then $z_i = T_i' - QUs_b = T_i - c_i - R(s_b)$, provided that this value satisfies Equations (18) and (22). This corresponds to lines 13 to 14 in Algorithm 2.
3) If $\lambda_i = 0$ and $\gamma_i > 0$, then $z_i = \overline{z_i}$, provided that $\overline{z_i}$ satisfies Equations (18) and (21). This corresponds to lines 15 to 16 in Algorithm 2.
4) If $\lambda_i > 0$ and $\gamma_i > 0$, then $z_i = \overline{z_i} = T_i - c_i - R(s_b)$. This corresponds to line 18 in Algorithm 2.
5) Finally, if $\overline{z_i} > T_i' - QUs_b$, then there is no solution since the optimization is infeasible. This corresponds to lines 9 to 10 in Algorithm 2.

As a result, the optimal z_i can only take one of the three values, $\overline{z_i}$, z_i', or $\widehat{z_i}$, defined in Theorem 5.1. This completes the proof.

7.3. Proof of Theorem 5.2

Note that Theorem 5.1 computes the global optimal z_i for a given s_b. If we exhaust all the possible s_b choices and compare the objective value, then we find a global optimal solution to the original optimization problem (Equations (10) to (12)). This completes the proof of Theorem 5.2.

REFERENCES

[1] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. 2010. Energy proportional datacenter networks. In *ACM Proceedings of International Symposium on Computer Architecture*.

[2] I. F. Akyildiz. 1988. On the exact and approximate throughput analysis of closed queuing networks with blocking. *IEEE Transactions on Software Engineering* 14, 1, 62–70.

[3] S. Balsamo, V. D. N. Persone, and R. Onvural. 2001. *Analysis of Queuing Networks with Blocking*. Springer.

[4] N. Bansal, T. Kimbrel, and K. Pruhs. 2007. Speed scaling to manage energy and temperature. *Journal of the ACM* 54, 1, Article No. 3.

[5] R. Begum, M. Hempstead, G. P. Srinivasa, and G. Challen. 2016. Algorithms for CPU and DRAM DVFS under inefficiency constraints. In *Proceedings of the IEEE International Conference on Computer Design*.

[6] R. Bergamaschi, G. Han, A. Buyuktosunoglu, H. Patel, and I. Nair. 2008. Exploring power management in multi-core systems. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference*.

[7] P. Bose, A. Buyuktosunoglu, J. A. Darringer, M. S. Gupta, M. B. Healy, H. Jacobson, I. Nair, J. A. Rivers, J. Shin, A. Vega, and A. J. Weger. 2012. Power management of multi-core chips: Challenges and pitfalls. In *Proceedings of the IEEE Design, Automation and Test in Europe*.

[8] E. V. Carrera, E. Pinheiro, and R. Bianchini. 2003. Conserving disk energy in network servers. In *ACM Proceedings of International Conference on Supercomputing*.

[9] J. M. Cebrian, J. L. Aragon, and S. Kaxiras. 2011. Power token balancing: Adapting CMPs to power constraints for parallel multithreaded workloads. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*.

[10] M. Chen, X. Wang, and X. Li. 2011. Coordinating processor and main memory for efficient server power control. In *Proceedings of the ACM International Conference on Supercomputing*.

[11] H. David, C. Fallin, E. Gorbatov, U. Hanebutte, and O. Mutlu. 2011. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the ACM International Conference on Autonomic Computing*.

[12] M. Dayarathna, Y. Wen, and R. Fan. 2015. Data center energy consumption modeling: A survey. In *IEEE Communications Surveys & Tutorials*.

[13] Q. Deng, D. Meisner, A. Bhattacharjee, T. F. Wenisch, and R. Bianchini. 2012. CoScale: Coordinating CPU and Memory DVFS in server systems. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture*.

[14] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. 2011. MemScale: Active low-power modes for main memory. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[15] X. Fan, C. S. Ellis, and A. R. Lebeck. 2003. The synergy between power-aware memory systems and processor voltage scaling. In *Proceedings of the ACM International Conference on Power-Aware Computer Systems*.

[16] A. Gandhi and M. Harchol-Balter. 2011. How data center size impacts the effectiveness of dynamic power management. In *Proceedings of the IEEE Allerton Conference on Communication, Control, and Computing*.

[17] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. 2012. AutoScale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems* 30, 4, Article No. 14.

[18] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. 2003. DRPM: Dynamic speed control for power management in server class disks. In *Proceedings of the ACM International Symposium on Computer Architecture*.

[19] M. Harchol-Balter. 2013. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press.

[20] S. Herbert and D. Marculescu. 2007. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of the IEEE/ACM International Symposium on Low Power Electronics and Design*.

[21] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. 2006. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture*.

[22] JEDEC. 2009. DDR3 SDRAM Standard. (2009).

[23] W. Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks. 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceedings of the IEEE Symposium on High Performance Computer Architecture*.

[24] X. Li, R. Gupta, S. Adve, and Y. Zhou. 2007. Cross-component energy management: Joint adaptation of processor and memory. *ACM Transactions on Architecture and Code Optimization* 4, 3, Article No. 14.

[25] X. Li, Z. Li, F. M. David, P. Zhou, Y. Zhou, S. V. Adve, and S. Kumar. 2004. Performance-directed energy management for main memory and disks. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[26] Y. Liu, G. Cox, Q. Deng, S. C. Draper, and R. Bianchini. 2016. FastCap: An efficient and fair algorithm for power capping in many-core systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*.

[27] Y. Liu, S. C. Draper, and N. S. Kim. 2013. Queuing theoretic analysis of power-performance tradeoff in power-efficient computing. In *Proceedings of the IEEE Conference on Information Sciences and Systems*.

[28] Y. Liu, S. C. Draper, and N. S. Kim. 2014. SleepScale: Runtime joint speed scaling and sleep states management for power efficient data centers. In *Proceedings of the ACM International Symposium on Computer Architecture*.

[29] K. Ma, X. Li, M. Chen, and X. Wang. 2011. Scalable power control for many-core architectures running multi-threaded applications. In *Proceedings of the ACM International Symposium on Computer Architecture*.

[30] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. 2011. Power management of online data-intensive services. In *Proceedings of the ACM International Symposium on Computer Architecture*.

[31] K. Meng, R. Joseph, R. P. Dick, and L. Shang. 2008. Multi-optimization power management for chip multiprocessors. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation*.

[32] Micron. 2007. DDR3 SDRAM System-Power Calculator. Retrieved from http://tinyurl.com/hcddfw5.

[33] A. K. Mishra, S. Srikantaiah, M. Kandemir, and C. R. Das. 2010. CPM in CMPs: Coordinated power management in chip multiprocessors. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*.

[34] H. Sasaki, A. Buyuktosunoglu, A. Vega, and P. Bose. 2016. Mitigating power contention: A scheduling based approach. In *IEEE Computer Architecture Letters*.

[35] J. Sharkey, A. Buyuktosunoglu, and P. Bose. 2007. Evaluating design tradeoffs in on-chip power management for CMPs. In *Proceedings of IEEE/ACM International Symposium on Low Power Electronics and Design*.

[36] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. 2011. Green governors: A framework for continuously adaptive DVFS. In *Proceedings of the IEEE International Green Computing Conference*.

[37] R. Teodorescu and J. Torrellas. 2008. Variation-aware application scheduling and power management for chip multiprocessors. In *ACM Proceedings of International Symposium on Computer Architecture*.

[38] A. Wierman, L. L. H. Andrew, and A. Tang. 2012. Power-aware speed scaling in processor sharing systems. In *Performance Evaluation*, Vol. 69. 601–622.

[39] H. Wong. 2012. A Comparison of Intel's 32nm and 22nm Core i5 CPUs: Power, Voltage, Temperature, and Frequency. Retrieved from http://tinyurl.com/z7rxjy3.

[40] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song. 2016. Dynamo: Facebook's data center-wide power management system. In *Proceedings of the ACM International Symposium on Computer Architecture*.

[41] G. Yan, Y. Li, Y. Han, X. Li, M. Guo, and X. Liang. 2012. AgileRegulator: A hybrid voltage regulator scheme redeeming dark silicon for power efficiency in a multicore architecture. In *Proceedings of the IEEE Symposium on High Performance Computer Architecture*.

[42] H. Zheng, J. Lin, Z. Zhang, and Z. Zhu. 2009. Decoupled DIMM: Building high-bandwidth memory system using low-speed DRAM devices. In *Proceedings of the ACM International Symposium on Computer Architecture*.

Received September 2016; revised January 2017; accepted April 2017