Exploiting Heterogeneity for Tail Latency and Energy Efficiency

Md E. Haque
Rutgers University*
mdhaque@cs.rutgers.edu

Yuxiong He
Microsoft Research
yuxhe@microsoft.com

Sameh Elnikety
Microsoft Research
samehe@microsoft.com

Thu D. Nguyen
Rutgers University
tdnguyen@cs.rutgers.edu

Ricardo Bianchini
Microsoft Research
ricardob@microsoft.com

Kathryn S. McKinley
Google
ksmckinley@google.com

ABSTRACT

Interactive service providers have strict requirements on high-percentile (*tail*) latency to meet user expectations. If providers meet tail latency targets with less energy, they increase profits, because energy is a significant operating expense. Unfortunately, optimizing tail latency and energy are typically conflicting goals. Our work resolves this conflict by exploiting servers with per-core Dynamic Voltage and Frequency Scaling (DVFS) and Asymmetric Multicore Processors (AMPs). We introduce the *Adaptive Slow-to-Fast* scheduling framework, which matches the heterogeneity of the workload — a mix of short and long requests — to the heterogeneity of the hardware — cores running at different speeds. The scheduler prioritizes long requests to faster cores by exploiting the insight that long requests reveal themselves. We use control theory to design threshold-based scheduling policies that use individual request progress, load, competition, and latency targets to optimize performance and energy. We configure our framework to optimize Energy Efficiency for a given Tail Latency (EETL) for both DVFS and AMP. In this framework, each request self-schedules, starting on a slow core and then migrating itself to faster cores. At high load, when a desired AMP core speed s is not available for a request but a faster core is, the longest request on a s core type migrates early to make room for the other request. Compared to per-core DVFS systems, EETL for AMPs delivers the same tail latency, reduces energy by 18% to 50%, and improves capacity (throughput) by 32% to 82%. We demonstrate that our framework effectively exploits dynamic DVFS and static AMP heterogeneity to reduce provisioning and operational costs for interactive services.

CCS CONCEPTS

• **Computer systems organization → Heterogeneous (hybrid) systems**; • **Software and its engineering →** *Scheduling*;

KEYWORDS

Tail Latency, Heterogeneous Processors, Energy Efficiency

*Haque is currently a Software Engineer at Google.

MICRO-50, October 14–18, 2017, Cambridge, MA, USA

ACM ISBN 978-1-4503-4952-9/17/10…$15.00
https://doi.org/10.1145/3123939.3123956

ACM Reference format:
Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. 2017. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proceedings of MICRO-50, Cambridge, MA, USA, October 14–18, 2017,* 14 pages.
https://doi.org/10.1145/3123939.3123956

1 INTRODUCTION

Interactive services, such as Web search, financial trading, games, and social networking, require consistently low response times to attract and retain users [16, 50]. Service providers thus define strict targets for 95th or higher percentile response times, commonly called *tail latency* [9, 10, 19, 24, 61]. Services typically distribute requests and aggregate responses from multiple servers. This places even more stringent tail latency targets on the servers, as each server in a set of five servers must meet a target of $\sqrt[5]{0.95} \approx 99$th-percentile to achieve an overall 95th-percentile latency target.

Datacenters for interactive services incur substantial provisioning and operating costs. Shehabi et al. estimate the total energy consumption of U.S. datacenters in 2014 at 73 billion KW hours, with servers consuming 40%–80% of this energy [52]. With prices per KW hour of 4 to 21 cents [17], the U.S. datacenter energy bill is roughly 1 to 6 billion dollars. Many servers run interactive services, making even a 1% reduction in total energy a significant savings. Large interactive service providers, such as Amazon, Facebook, Google, and Microsoft, are all highly motivated to reduce energy consumption [3, 25, 32, 51].

A key challenge is that improving tail latency and energy efficiency are usually conflicting goals. However, interactive services provide an opportunity to combine them since most requests are short, but the computation time of long requests is often 10× longer than the average, and 100× longer than the median [9, 24, 28]. These computationally intensive (long) requests are a primary factor in tail latency [9, 24, 28, 31, 37], although queuing delay due to bursty load, paging, network congestion, and failures sometimes contribute. Our work thus focuses on these long requests. This variety in request demand makes increasing the energy efficiency of short requests and spending more energy on reducing computation time of long requests appealing.

This paper shows how to optimize tail latency and energy efficiency by matching the workload heterogeneity in the request mix to heterogeneous multicore hardware. For our heterogeneous hardware, we consider both per-core Dynamic Voltage and Frequency Scaling (DVFS) and Asymmetric Multicore Processors (AMPs). DVFS

creates dynamic core frequency heterogeneity, whereas AMPs provide static core heterogeneity by using different microarchitectures [2, 6, 12, 13, 21, 33, 34, 36, 40, 41, 46, 55].

We introduce the *Adaptive Slow-to-Fast* scheduling framework for interactive services running on servers with heterogeneous multicores. The framework addresses three key challenges: (i) the computational demand of each individual request is hard to predict and prediction is never free or perfect [22, 24, 31, 38]; (ii) load is variable and bursty; and (iii) the desired core speed or type may not be available. If the desired AMP core speed and DVFS core speed are always available, we show AMP and DVFS scheduling pose a similar problem. For AMP, we schedule requests, whereas for DVFS, we select core speeds. In practice, AMP is more challenging than DVFS, because it has a limited number of cores of each type and, at load, requests compete for cores of the same type. With DVFS, any core can take on any of the hardware-available speeds. Though AMP is harder to schedule, it offers substantially greater power reductions and capacity benefits than DVFS because it uses static microarchitectural techniques [2, 13, 33, 40].

We configure our Adaptive Slow-to-Fast framework to minimize energy consumption for short requests, while ensuring that long requests do not exceed tail latency targets. We call this configuration Energy Efficiency with Target Tail Latency (EETL). We exploit two observations. (1) Short requests complete quickly, but long requests reveal themselves during execution. Request length is the sum of queuing and computation time. (2) We only need to consider slow-to-fast policies because given an energy optimal schedule, there exists an equivalent slow-to-fast schedule, simplifying the online scheduling problem and minimizing migration [48].

We design threshold-based policies with control theory that takes as input request progress, core speeds, relative core energy consumption, workload demand distributions, and load. Building the controller takes only a few seconds and can be performed online when workloads change. Configuration parameters to our framework specify a range of energy and latency tradeoffs, including policies that minimize energy, minimize tail latency, and the EETL policy that minimizes energy for a given target tail latency. Because requests periodically self-schedule based on thresholds and our controller determines the thresholds in constant time, the system is efficient and scalable.

The EETL policy starts all requests at low frequency or on the slowest cores, where short requests consume the least amount of energy while still meeting tail latency targets. As long requests execute, each request self-schedules, increasing its core's frequency with DVFS or migrating to a faster core, when available, to meet the tail latency target. The maximum number of migrations on AMP is the number of core types n, which is much less than the total number of cores. When a request r desires a core type s and none is available, but a faster core $f > s$ is available, the oldest requests on each type $t = s + 1 \ldots f$ detects this load and migrates itself early, before its threshold expires, to the next faster type making a slower core available to execute request r. This policy for resolving core competition prioritizes youngest requests to the slowest cores and the oldest requests to the fastest cores. EETL thus addresses the two major challenges of AMP scheduling: (1) when to migrate a request from a slow to a fast core: use request progress and system load, and (2) what to do when a core with the desired speed is not available but other cores speeds are: prioritize long requests to faster cores, achieving global adjustment through self-scheduling.

Optimizes	hardware	core scheduling	core competition	request progress	system load
Pegasus [37]	DVFS				✔
OctopusMan [45]	AMP	✓			✔
Rubik [30]	DVFS			✔	✔
Adrenaline [22]	DVFS			✔	✓
TimeTrader [57]	DVFS			✔	
Slow-to-Fast [47, 48]	AMP	✔		✔	✓
EETL	DVFS/AMP	✔	✔	✔	✔

Table 1: Comparing prior work to Energy-Efficiency with Target Tail Latency (EETL). ✔ denotes a complete solution and ✓ denotes a partial solution.

Table 1 compares our approach with previous work. It classifies approaches for optimizing the tail latency and energy efficiency of interactive workloads based on hardware target (DVFS, AMP), if they schedule requests to cores as AMP requires (scheduling), if they handle competition for core types (competition), if they monitor and optimize target latency of individual requests (request progress), and if they optimize for system load (load). No prior approach handles all the problems posed by AMP. Ren et al. [47, 48] show that any optimal AMP schedule is equivalent to a slow-to-fast schedule, simplifying the solution space, but do not manage core competition or load. The other approaches configure cores: either they use load to configure the entire system, wasting energy on short requests [37, 45]; they use request progress to configure individual cores [22, 57]; or they use both load and request progress to configure individual cores [30]. Since they lack a scheduler [22, 30, 37, 45, 57], they do not unlock the full potential of AMP. Section 2 describes all prior work in more detail.

Our evaluation uses the popular Apache Lucene enterprise search engine on Wikipedia pages [1, 18] and an interactive finance server [7, 14, 27, 47]. We evaluate our approach in multiple ways. We report measurements on an 8-core Broadwell processor with per-core DVFS for multiple systems and show consistency with prior work [37]. We introduce a methodology for emulating AMP on homogeneous multiprocessors because (1) AMPs are currently available only for mobile systems [12, 46], and (2) multicore simulators are neither complete nor accurate enough to report tail latency. We emulate AMP and, for comparison, per-core DVFS by configuring a 16-core Xeon (Haswell) machine and reducing individual core performance with duty-cycling threads. We combine performance results with power models derived from Xeon measurements, McPAT [35], and the literature [43, 44]. Because emulation is orders-of-magnitude faster than simulation, we explore many hardware configurations and algorithmic sensitivities.

On Broadwell, EETL meets tail latency targets while substantially reducing energy compared to Pegasus, the state-of-the-art for DVFS [37]. Whereas Pegasus adjusts the speed of *all* cores simultaneously in response to load, our fine-grain scheduler accelerates requests based on individual request progress and load using *per-core* DVFS. This finer-grain control decreases processor energy consumption by 22% while delivering the same tail latency. Because processor energy represents about 67% of server energy [3], these savings mean about 15% total system energy savings over Pegasus.

For the same tail latency and power budget, EETL on AMP reduces processor energy an additional 18 to 50% over per-core DVFS

in emulation. Moreover, EETL meets tail latency targets at higher loads on AMPs than DVFS, increasing throughput by 32 to 82%. These results show that our scheduler effectively uses AMP to reduce both provisioning and operational costs by optimizing energy efficiency and tail latency.

In summary, our contributions are: (1) a configurable scheduling framework for optimizing tail latency and energy that exploits characteristics of interactive services on heterogeneous multicores; (2) fine grain, per-request policies that migrate requests from slower to faster cores as available or adjust core speeds; (3) a practical evaluation methodology; and (4) showing how to match workload heterogeneity to hardware heterogeneity to reduce provisioning and operational energy costs in datacenters.

2 RELATED WORK

The EETL algorithm improves over prior work because (1) it exploits heterogeneity at a finer granularity, using request progress and load to select per-request core speeds, in contrast to prior work that configures all cores based on load [37, 45]; (2) it uses request progress, in contrast to prediction [22, 30]; and (3) it manages competition for core speeds in AMPs by prioritizing the longest requests to faster cores, in contrast to prior work that assumes the desired speeds are always available [37, 45, 48]. Table 1 overviews these differences and shows that most approaches do not schedule requests on cores, which limits their ability to exploit AMP. Our scheduling framework applies to both per-core DVFS and AMP systems. (We leave their combination for future work.)

Tail latency with DVFS multicores. Prior work manages tail latency and energy with DVFS [22, 23, 30, 37, 57]. Pegasus uses a feedback-based DVFS controller based on total system load that operates at a five second granularity [37]. At low load, it reduces the voltage/frequency of all cores, increasing average latency, while maintaining the same tail latency. Processor-wide DVFS wastes energy since the speed and power consumption of all cores must be sufficient to service the tail. Section 6 shows that by targeting individual requests, our fine-grain policies reduce energy consumption while delivering the same tail latency, since only some cores must be fast enough to satisfy long requests.

Rubik considers both queue length and request progress to control the frequency of each core independently [30]. It assumes a desired core frequency is always available, which in power limited cores may not be true in the future. It decides frequency of a single core and it does not adjust the policy based on what's happening in other cores. Rubik adapts frequently to transient changes in queue length. While not desirable, DVFS may switch frequencies often, but frequent migration degrades AMP performance.

Researchers have used per-core DVFS to accelerate individual requests by predicting their latency [22] or if request have spent a long time in the network [57]. Accurate prediction of long requests is challenging and must be retrained as software evolves [24, 27, 31, 38]. Since prediction is never perfect, it is not a complete solution. These DVFS approaches do not require schedulers because they assume the desired core speed is always available.

Tail latency on AMPs. The most closely related work optimizes tail latency on heterogeneous multicores [45, 47, 48]. Ren et al. [47] show slow-to-fast (S2F) scheduling can exploit heterogeneity to improve throughput in simulation, but does not optimize energy. Ren et al. [48] prove that heterogeneous cores offer significant energy efficiency advantages over homogeneous cores when optimizing multiple objectives (e.g., tail and average latency). Their basic S2F algorithm computes migration thresholds assuming cores of the desired speed are always available. Section 3 illustrates experimentally that basic S2F does not satisfy tail latency targets when load varies or a core of the desired speed is not available; constraints our algorithm optimizes.

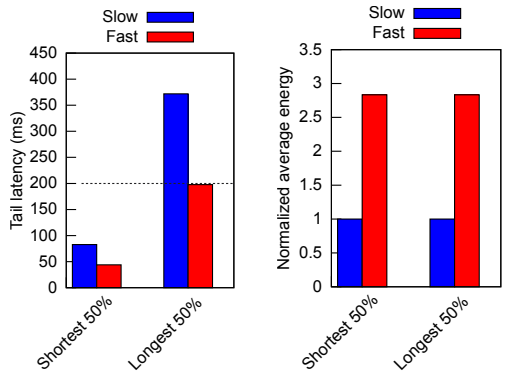
Petrucci et al.'s OctopusMan system is coarse grain — they allocate either all slow or all fast cores to an entire service, but never mix core types [45]. Their controller selects increasingly larger homogeneous configurations of slow or fast cores until it meets the target tail latency. They do not describe a scheduler. Section 6.5 compares OctopusMan with EETL. EETL uses individual request progress and system load to deliver tighter control over power/performance tradeoffs and greater energy savings.

Other workloads on AMPs. Prior optimization work for parallel and multiprogrammed workloads on AMPs does not consider tail latency [5, 29, 49, 53, 56, 58].

Hardware trends. Per-core DVFS is present in most modern systems, including those with accelerators. AMPs are already present in mobile systems and server prototypes [6, 12, 46]. Researchers have shown that heterogeneous multicores are a promising general architecture [2, 13, 21, 33, 34, 36, 40, 41, 55]. Datacenter operators have already deployed heterogeneous systems that use accelerators, such as FPGAs, ASICs, and GPUs, for specific tasks that are carefully mapped to them. For these tasks, accelerators offer substantial performance and energy efficiency that is difficult to match. However, accelerators require extensive engineering of the software and hardware, imposing a potentially prohibitive cost, especially for smaller services. In contrast, we seek to maximize the potential of general-purpose heterogeneity and thereby benefit a wide range of interactive workloads and datacenter operators.

3 SCHEDULING HETEROGENEOUS MULTICORES

This section describes challenges and opportunities for optimizing interactive services on heterogeneous multicores, and illustrates the limitations of prior work for AMP. To explore energy/performance tradeoffs as a function of workload, we divide Lucene search requests into two halves based on their length and assume a 200-ms tail latency target for the 99th percentile. Figure 1(a) reports the tail latency of the two workloads and 1(b) reports their energy consumption when executed on a representative slow and fast core. (Section 5 describes our methodology.) The slow core substantially improves energy efficiency of all requests by almost a factor of 3. Both slow and fast cores easily meet the 200-ms tail latency target for the shortest 50% of requests. Slow cores deliver roughly 83 ms for short requests, well below the target tail latency. However, only the fast core meets the target for the longest requests. Thus, by using fast cores for long requests and energy-efficient slow cores for short requests — and most requests are short [9, 18, 24, 28] —



(a) (b)

Figure 1: (a) 99th percentile (tail) latency of shortest and longest 50% of requests; (b) average energy consumption executing exclusively on a slow or fast core.

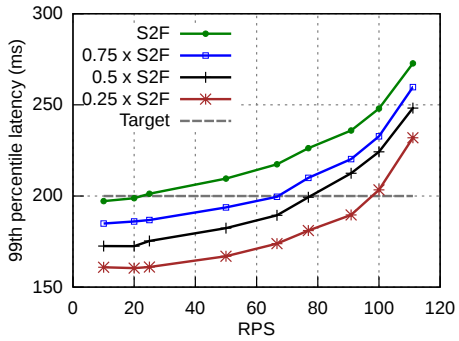


Figure 2: S2F on 2 fast and 13 slow cores (2F-13S) uses a fixed migration threshold for a tail latency target [48]. It only works at low load, when requests do not compete for cores. Decreasing the threshold (0.5×S2F migrates twice as fast), exceeds the target and wastes energy at low load, motivating an adaptive approach.

service providers have a significant opportunity to improve energy efficiency without compromising tail latency.

Basic Slow-to-Fast Scheduling for Heterogeneity. Prior work proposed migrating requests to faster cores as long requests reveal themselves [48]. This basic slow-to-fast (S2F) algorithm computes a single threshold for each core type. When a request passes each threshold, S2F migrates it from a slower to a fast core type. S2F has the following desirable qualities.

Basic S2F bounds migration. A slow-to-fast scheduler limits the migrations based on the number of different core speeds.

Basic S2F reduces energy consumption. Using slow cores first completes short requests on slow cores and does not use fast cores at all, reducing energy consumption.

Challenges in Slow-to-Fast Scheduling. Despite this potential, *basic S2F is not effective because it does not adapt to load and does manage competition for core types.* Figure 2 illustrates the limitations of the fixed thresholds in S2F on an AMP with 2 Fast and 13 Slow (2F-13S) cores. With two core speeds, S2F computes a single migration threshold based only on the request demand distribution. Since it does not consider load or core competition, it only satisfies the tail latency target at very low load (20 RPS in this example). By simply reducing the threshold, S2F meets the target at higher load, but wastes energy at low load. This observation motivates an algorithm that adapts to load and core availability. Our work exploits the following insights.

Workload demand distributions are known. Service providers already profile demand distributions (i.e., the percentage of requests of each length) to provision servers and these distributions tend to change slowly over time [18, 38, 47].

Use fast cores just enough. Our scheduler uses fast cores just enough to satisfy the tail latency target, sacrificing some average latency. Since slow cores are more energy-efficient, this strategy satisfies the target with less energy.

Adapt to load. To handle load spikes, the policy needs to adapt, using fast cores more often as load increases.

4 ADAPTIVE SLOW-TO-FAST

This section presents our Adaptive Slow-to-Fast scheduler design. We assume that the service executes each request independently and sequentially, and that multiple simultaneous requests execute concurrently on multiple cores.

Our framework handles the performance and energy heterogeneity available from dynamic per-core DVFS and static microarchitecture design in AMP. We treat these forms of heterogeneity similarly: a core type in AMP is equivalent to a core speed in per-core DVFS. With respect to the general scheduling problem, AMP migration and DVFS speed increases are equivalent. Although DVFS has more potential settings than AMP, this difference does not change the scheduling problem and in practice is constrained by transition times. However, on AMP a request must migrate across cores to take advantage of multiple types and other requests could be executing on these cores, whereas with per-core DVFS, the request can increase its speed since it continues to execute on the same core. Since scheduling AMP subsumes scheduling per-core DVFS, we present the framework for AMP without loss of generality. We assume AMP hardware that consists of N same-ISA core types with different power and performance characteristics. Slower cores are more energy-efficient than faster ones. If slower cores are not energy efficient, providers will buy only fast cores.

Our framework includes an offline and an online component. Offline, we design feedback controllers that compute migration thresholds based on measured tail latency, target tail latency, and system load. While we design the controller as an offline phase in our evaluation, it can be done online since it takes few seconds. Each request schedules itself online starting on a slow core. It makes a decision to migrate based on load and thresholds that the controllers compute using request progress and system load. Feedback control mitigates many transient factors, such as request interference and workload demand variability, while providing stable behavior. Because each request self-schedules, the framework incurs very low overhead and scales with the number of cores.

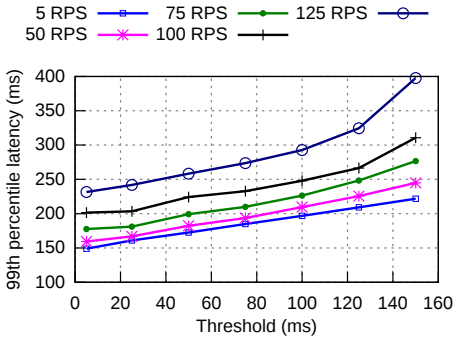


Figure 3: Tail latency, migration threshold, and load (RPS) relationship on 2F-13S.

4.1 Scheduling Objectives

We mainly consider two popular service provider objectives: (1) reducing tail latency to improve responsiveness, and (2) reducing energy for a tail latency target to reduce operating costs. We consider an energy-only objective as a lower bound on energy and tail latency only as an upper bound on latency. The key challenge is to decide *when* to migrate a request because this threshold controls tail latency and energy. A low threshold migrates requests sooner and uses fast cores more often, reducing tail latency at the expense of more energy. Higher thresholds use fast cores less often, consume less energy, and incur higher tail latencies. We show how to adjust migration thresholds to meet all these objectives.

We introduce the ***Energy Efficiency with Target Tail Latency (EETL)*** policy to meet a target tail latency as energy-efficiently as possible, a popular service provider objective. On one hand, the migration threshold must be short enough to meet the latency target. On the other hand, it must be long enough to avoid wasting energy. The threshold depends on the target tail latency, workload, processor configuration, and dynamic system load.

For comparison, we consider the two policies with the extreme settings for their migration thresholds: ***Tail Latency (TL)*** and ***Energy Efficiency (EE)***. TL sets the migration threshold to zero, executing requests as fast as possible. EE sets the migration threshold to infinity, using fast cores only it the slow cores are all busy.

4.2 Controller Design for Two Core Types

This section describes the design of a Single Input Single Output (SISO) feedback controller to determine the threshold that achieves the EETL objective for two core types. Section 4.4 generalizes the controller for N core types. We consider tail latency targets in the range of measured and thus achievable tail latencies. The controller input is the difference between the current tail latency of the system and the target. The output is the migration threshold. To determine the effect of input changes on the output, we measure tail latency as a function of *system load* and migration thresholds with a representative request trace (i.e., a good representation of the workload demand distribution).

Figure 3 illustrates the tradeoffs with Lucene. Unsurprisingly, the lower the load and threshold, the lower the potential tail latency. The highest load this system supports within a target tail latency of 200 ms is a little less than 100 RPS, using the lowest threshold.

Load interval	Representative load	K_p	K_i	K_d
< 50	25	0.11	0.49	0.00
50–70	60	0.15	0.25	0.00
> 70	80	0.19	0.17	0.03

Table 2: Gain values for the three PID controllers based on load (measured in RPS) for Lucene.

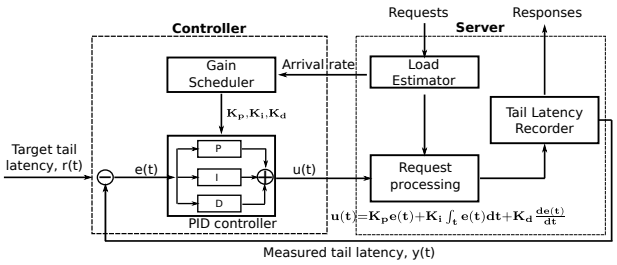


Figure 4: Diagram of the controller and interactive server.

At low load, the latency and migration threshold relation is linear and flat. As load increases, slope increases. The relationship is non-linear for high loads and thresholds. To handle these differences, we use Gain Scheduling [20] and design separate controllers for different load ranges. We divide the load range into three intervals, design a controller for each interval, and validate this choice with sensitivity analysis.

We use Gain Scheduling [20] with Proportional-Integral-Derivative (PID) control, because it is effective and thus popular [54]. For each interval and target tail latency, we determine the open-loop step response with the load at the mid-point of the interval. We compute and tune the PID control gains using MATLAB PID Tuner [42]. Table 2 shows the generated PID gains and Figure 4 shows the resulting controller block diagram.

The load estimator uses request arrivals to compute the average arrival rate. The gain scheduler uses it to select one of the three PID gain parameters. The load estimator does not need to be precise since the controller will still work without any load estimation or gain scheduling (albeit with higher overshoot and settling time). The PID controller's input is $e(t)$, the difference between the measured tail latency and the target tail latency. The output of the controller is $u(t)$, the migration threshold from a slow to a fast core. The online scheduler uses these computed thresholds to migrate requests from slow to fast cores.

4.3 Online Scheduling

At run time, each request self-schedules, periodically executing Algorithm 1. With no core competition and two core types, a new request starts running on a slow core (Lines 18–20). When (i) the request execution time crosses the threshold, (ii) the request is not already executing on a fast core, and (iii) a fast core is available, the request migrates to the fast core (Lines 5–9). We track request age and position using a synchronized priority queue.

When a new request arrives and all slow cores are occupied due to high load, the new request executes with lower priority (Lines 16–17). We rely on the older requests to move themselves to faster cores, out of the way of the new request, when faster cores are free. In the case of two core types, the oldest request on a slow core

Algorithm 1: Adaptive Slow-to-Fast for two core types. Each request self-schedules independently.

```
   input  : F & S, the set of fast & slow cores
   input  : t the migration threshold
   output : core allocation for current request
1  n = total number of active requests
2  a = age of current request
3  i = request position among all requests by decreasing age
4  c = current core                                  // c is None for a new request
5  if (i <= |F| and a >= t) then                     // migrate after crossing t
6      if (c ∉ F) then
7          f = free core from F
8          migrate from c to f
9      end
10 else if (n > |S| and i <= |F| and i <= n − |S|) then
       // migrate before crossing t because there are not enough slow cores
11     if (c ∉ F) then
12         f = free core from F
13         migrate from c to f
14     end
15 else if (c is None) then
16     if (i > |F| + |S| or No free core in S) then // very high load or sync delay
17         start running with low priority
18     else                                          // regular request arrival
19         s = free core from S
20         start executing at s
21     end
22 end
```

migrates itself early to an available fast core (Lines 10–14). Because each request frequently self-schedules, the system reacts quickly. This logic thus achieves three goals: responsiveness, scalable self-scheduling, and a global scheduling effect in which slower cores are always executing shorter requests and faster cores are executing longer requests. For the more general N core types case, we use similar algorithm and track request length and competition on each core type to make early migration decisions.

4.4 Controller Design for N Core Types

This section describes our controller for $N > 2$ core types/speeds. Because N core types/speeds can more finely match core capabilities to workloads, they have more potential to improve energy efficiency by striking better power/performance trade-offs [6, 21, 33, 34, 36]. We use our controller for five-speed per-core DVFS experiments as well as AMPs with 3 core types. Without loss of generality however, we next describe the controller in AMP terms.

Our controller for N core types specifies $N-1$ migration thresholds from core type c_i to c_{i+1} where cores of type i exhibit s_i performance (average speedup compared to the slowest core) with p_i power consumption. Without loss of generality, $0 < s_1 < s_2 < \cdots < s_N$ and $0 < p_1 < p_2 < \cdots < p_N$. Table 3 defines the symbols we use for controller design for N core types.

Determining these thresholds is a multi-dimensional search problem, where the search space grows exponentially with the number of speeds. To simplify it, we introduce a decision variable that converts it into a single-dimension problem and use an SISO controller. We divide the procedure into two steps. We first determine the thresholds for low loads based on the workload distribution and relative core speeds. We pose this optimization problem as shown in Figure 5. We seek to minimize the average energy used by all the requests while satisfying the tail latency target. This problem

Symbol	Definition
$r \in R$	Request profiles
$slow_r$	Runtime of request r on slowest core
s_i	Avg speedup of requests on core type c_i
p_i	Peak dynamic power on c_i
$target_l$	Target tail latency
$t = \{0, t_1, t_2, \ldots, t_{N-1}\}$	t_i is the migration threshold to core type c_{i+1}
$V = \{0, v_1, v_2, \ldots, v_{N-1}\}$	Intermediate representation of t such that $v_i = (t_i - t_{i-1})$
$e_r(V)$	Energy used by r with schedule V
$latency_R(V, \beta\text{-tail})$	βth-percentile latency of R with schedule V
$L_{(R,V)}[]$	List of runtimes for all $r \in R$ with schedule V in non-decreasing order

Table 3: Definitions for computing migration thresholds.

$$\text{Min. } Objective = \frac{\sum_r e_r(V)}{|R|} \quad (1)$$

$$e_r(V) = \begin{cases} slow_r \times p_1 & \text{if } slow_r \le v_1 \\ (v_1 \times p_1 + \frac{(slow_r - v_1) \times p_2}{s_2}) & \text{if } v_1 < slow_r \le (v_1 + v_2) \\ \ldots \end{cases} \quad (2)$$

$$latency_R(V, \beta\text{-tail}) \le target_l \quad (3)$$

$$latency_R(V, \beta\text{-tail}) = L_{(R,V)}[\lceil \beta \cdot |R| \rceil] \quad (4)$$

Figure 5: Threshold determination for low loads. We select t to minimize the objective function.

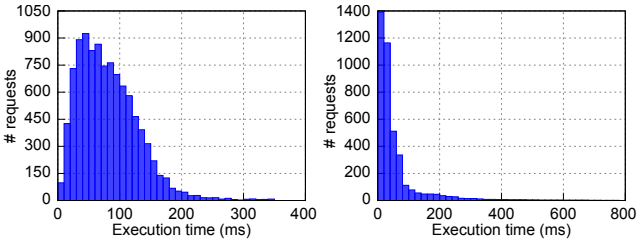
formulation only applies to low load because it assumes no core contention—each request may execute on a core of type $c_{(i+1)}$ whenever it is older than t_i. We transform the optimization problem to an equivalent convex problem. It takes 1-2 seconds with workload distributions for 4 to 8 K requests on our server using Gurobi [15].

The second step uses the thresholds determined for low load as a starting point for finding thresholds at higher loads. Given the solution of the problem in Figure 5 is of the form $t^* = \{0, t_1^*, t_2^*, ..., t_{N-1}^*\}$, we assume the thresholds for all loads take the form $t = \{0, z \times t_1^*, z \times t_2^*, ..., z \times t_{N-1}^*\}$. For low load, $z = 1$ gives the best thresholds. For high load, $z = 0$ gives the best thresholds. Intuitively, the values of thresholds are t^* at low load and, as the load increases, the thresholds decrease until they all become zero when all cores are occupied. The older a request is, the more likely it is executing on the fastest cores, because oldest requests are promoted first (see Section 4.3). Thus, $z = 1$ and $z = 0$ work well for low and high load, respectively. To select z for intermediate loads at run time, we design SISO controllers using the methodology from Section 4.2, running the request trace on the heterogeneous processor with different load and input (z) values. This results in a set of PID controllers from the Gain scheduler for various loads. Since the controller runtime overhead is not depenedent on core type or core count, this method is extremely efficient and scalable. The resulting schedule may consume unnecessary energy compared to the optimal at moderate to high loads. In practice, deviations are small across many demand distributions and core configurations.

5 EVALUATION METHODOLOGY

5.1 Workloads

We use the commercial open-source Apache Lucene [1, 18, 60], an enterprise search engine, and a Monte Carlo finance server [4, 7, 19]. Both servers use a standard pool of worker threads. A worker thread



(a) Execution time histogram of 10 K Lucene queries (b) Execution time histogram of 4 K finance requests

Figure 6: Workload demand distribution in 10 ms bins.

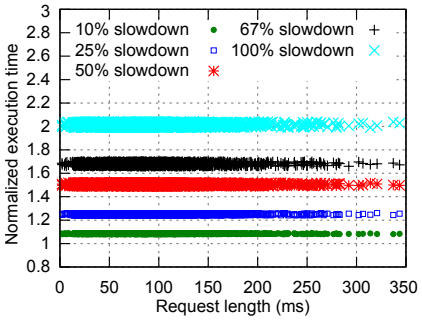


Figure 7: Normalized execution time of Lucene requests for different planned slowdown.

dequeues a request from the arrival queue and processes the request to completion.

We configure the Java Lucene search engine with 33+ million Wikipedia English Web pages [39, 59]. The search index consumes 10 GB. We use 10 K term search requests from Lucene's nightly regression tests to build the workload demand distribution in Figure 6(a). Prior work shows Lucene shares characteristics with Bing search [18]. We use 8 K requests to design the feedback controller. For each performance experiment, a client issues a random subset of the other 2 K requests, following a Poisson process in an open loop. We vary system load by changing the request per second (RPS) arrival rate. We add periodic calls (at a configurable 1 ms frequency) during request processing from Lucene to our scheduling logic. Our scheduler uses JNI, statically linking a C module that calls sched_setaffinity for pinning requests to specific cores. We add online profiling that tracks tail latency and load (about 500 lines of Java). We design our controller using MATLAB PID tuner.

We also use a Monte Carlo finance server from prior work that computes financial derivatives for path-dependent Asian options and is computationally intensive [4, 7, 19]. Banks and fund managers use such derivatives to make interactive trading decisions. Each request estimates an option price under various economic scenarios with different interest rates, strike prices, dividend yields, and volatility. Processing time varies widely based on request parameters, for example, sampling more for trades with higher volatility or total monetary value. Figure 6(b) depicts the service demand distribution of 4 K synthetic requests. The finance server is about 150 lines of C++ and we add about 400 lines of C++ code to implement our scheduler.

5.2 Hardware

We use two hardware platforms and heterogeneity emulation. To evaluate our framework on multicores that support per-core DVFS, we use an 8-core 2GHz Broadwell processor and 64GB RAM. To experiment with larger systems, we use a 16-core 2.3GHz Xeon (Haswell) and 16GB of RAM. In both cases, we turn off hyper-threading to avoid interference among threads sharing a physical core. Unfortunately, the larger server does not support full per-core DVFS (only per-socket), so we emulate AMP and per-core DVFS heterogeneity on it using duty-cycling threads. Emulation is the best option for our evaluation since (1) AMPs are currently available only in mobile systems [12, 46], which cannot handle large memory-intensive workloads like Lucene, and (2) cycle-accurate multicore simulators have not achieved the speed or accuracy necessary to report tail latency.

Our emulator uses a single slowdown parameter to capture the performance difference from DVFS, microarchitectural, and cache hierarchy behavior. We slowdown cores by alternating between executing request threads and a background thread. The background thread executes a CPU-bound computation and then sleeps. We control the slow down factor by a duty-cycle setting. For example, a background thread executing 50% of the time and sleeping 50% of the time increases the request processing time by a factor of 2. We perform duty cycling in 100–500 microsecond periods. Since our shortest requests are a few milliseconds, this period is sufficient to affect all requests equally, and long enough to avoid excessive context switching. Figure 7 shows the measured and planned slowdown in processing time as a function of request length for several duty-cycling settings for Lucene requests. Although the variance increases with higher slowdown settings, it is low compared to other sources of non-determinism in the system. The emulator is quite accurate in increasing the execution time of requests, corresponding to the desired core speed.

5.3 Core Performance and Power

We use relative heterogeneous core performance characteristics from the ARM big.LITTLE specification [12]. ARM big (Cortex-A15) cores delivers roughly 2x performance over little (Cortex-A7) cores. For DVFS experiments, We use five DVFS settings with the lowest frequency being roughly half of the highest frequency. Our default configuration for AMP is *2F-13S* and has two core types: 2 fast (big) and 13 emulated slow (little) cores. We also study medium cores with varying performances between slow and fast cores. In all of our experiments, we spare one full core to issue the requests, running the client.

We report real measured processor power on the Broadwell with per core DVFS. We use the ACPI userspace power governor and modify the "cpufreq/scaling_ setspeed" file to change the frequency. For Haswell, we report energy based on measured performance and a model of static and dynamic power. Because it implements per-socket DVFS, not per-core DVFS, we measure power using RAPL and model per-core DVFS. Measured processor power at the lowest speed is roughly one-fourth that of the highest.

We use McPAT [35] to model AMP power consumption of slow cores, fast cores, and the last-level cache using architectural details from the literature [44]. The slow cores are single-issue in-order, and the fast cores are out-of-order with a 48-entry instruction window, a 168-entry re-order buffer, and 2 load/store ports. Each core has 32KB L1 instruction and data caches, private 256KB L2 cache and shared 2MB LLC. We assume 22nm technology to compute dynamic power and static power (with power gating enabled). To study medium cores, we develop an analytic model ($Power \propto performance^{\alpha}$) similar to [43]. Using the numbers from McPAT for slow and fast cores as base cases, α is 2.73 for dynamic power and 1 for static power. Table 4 shows the performance and peak power consumption of the cores normalized to slow cores.

		Performance (s_i)		Peak
Core type	**Core name**	**Relative**	**Frequency**	**power (p_i)**
S	**Slow**	1	1.7 GHz	1
M	**Medium−**	1.44		2.7
	Medium	1.59		3.5
	Medium+	1.71		4.5
F	**Fast**	2	3.4 GHz	6.8
DVFS	**Slow**	1	1.7 GHz	1.7
	Medium−	1.3	2.2 GHz	2.7
	Medium	1.6	2.7 GHz	3.8
	Medium+	1.9	3.3 GHz	5.3
	Fast	2	3.4 GHz	6.8

Table 4: Performance and peak power for emulated DVFS and AMP. Our default AMP (2F-13S) configuration is peak power comparable to 4 Fast DVFS cores.

With our default power model and configuration, slow cores are ~3x more energy-efficient than fast cores, which is consistent with prior measurements (3.3x [26], 2.7x [11], 3.5x [12]), and simulation (3.01x [40]). Using our power model for DVFS, Pegasus improves energy consumption by up to 40% compared to the baseline. This result is consistent with their reported measurements [37]. We focus on processor energy because it dominates: consuming 67% of the server power [3]. Section 6.8 explores the sensitivity of our results to our model. Even with much more conservative models, our improvements are substantial.

5.4 Overheads and Performance Measurements

Each worker thread executes an algorithm like Algorithm 1 that is of constant complexity. Online overhead of PID controller is also constant, regardless of the total number of cores. DVFS frequency changing of a core takes around 50–90 μs and does not impose any extra overhead. We also measure and perform request migration from emulated slow to fast AMP cores. To account for migration energy, we pessimistically assume requests consume full power as soon as the request determines it needs to migrate. All results include migration time (<500 μs) and energy. Each request migrates at most $N-1$ times for N core types. Migration time and energy are negligible for our 100–200 ms tail latency targets.

We report tail and average latency, which include queuing delay and execution time. We use Java *NanoTime* in Lucene and *clock_gettime* in the finance server. Since tail latency is an order statistic, we measure it over a moving window of the last 2000 requests and report the 99th percentile. Since the controllers take the tail latency as input, we set their control epoch to 500 requests. At high load, epochs are consequently shorter than at low load.

5.5 Scheduling Policies

Tail Latency (TL). This policy minimizes tail latency by running requests at the highest possible speed. TL for DVFS (TL-DVFS) uses all cores at the highest Voltage/Frequency (V/F) level. TL for AMPs (TL-AMP) starts executing requests on the fastest available core. When a fast core c_i becomes available, the oldest request executing on $c_i - 1$ migrates to c_i. This configuration reduces tail and average latency.

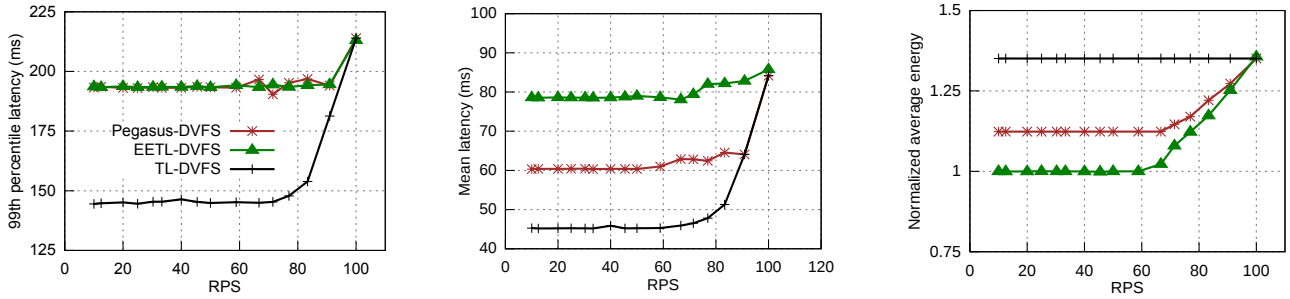
Energy Efficiency with Target Tail Latency (EETL). This policy uses thresholds from the feedback controllers for a specified target latency. EETL for per-core DVFS (EETL-DVFS) uses controllers to set the V/F of each core independently. EETL for AMP (EETL-AMP) migrates a request to a faster core once request execution time exceeds the appropriate threshold at the current load.

Energy Efficient (EE). This policy configures Adaptive Slow-to-Fast with the maximum (infinite) threshold. Each request executes on the slowest available core and only migrates to a faster core when it is the oldest request and the slower cores are insufficient to serve the load. This policy consumes the least energy but usually misses the tail latency target. It illustrates a lower bound on dynamic energy.

Pegasus. We implement the state-of-the-art DVFS policy [37]. Pegasus selects a single power setting for all cores at five second intervals using RAPL [8] and a feedback controller. It adjusts power to meet a tail latency target as a function of system load, using more power at higher loads. Since Lo et al. do not present their controller design, we implement an ideal version. For each load, we determine the best power setting by sweeping through all possible settings and constructing a look-up table. Our experiments index this table by load to determine the best power level. A feedback controller can only do worse.

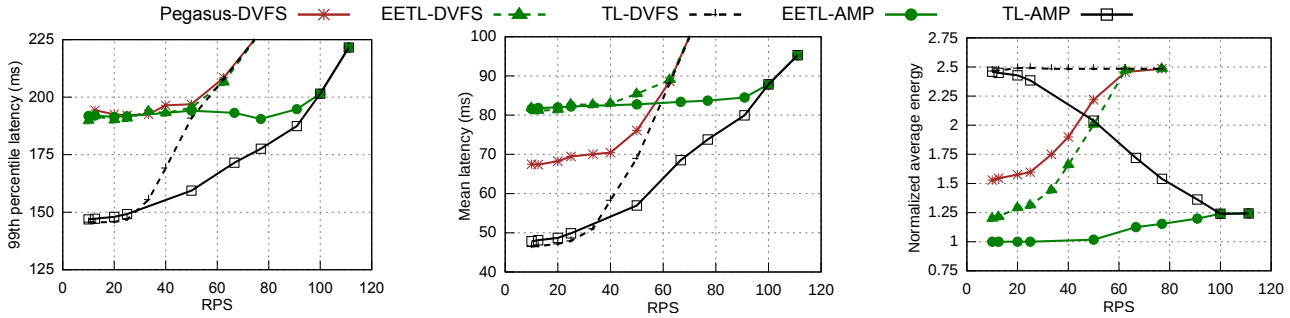
Octopus-Man. We implement Octopus-Man, which selects core configurations (core type and count) based on system load [45]. However, it only selects all small or all big cores; it never chooses a mix of core types. It assumes a strict ordering of core configurations by compute performance. For N slow cores and M fast cores, the usable configurations are $\{\{S_1\}, \{S_1, S_2\}, \ldots, \{S_1, S_2, \cdots, S_N\}, \{F_1, F_2, \cdots, F_L\}, \ldots, \{F_1, F_2, \cdots, F_M\}\}$, where L fast cores provide equal or higher performance than N slow cores. For 13 slow and 2 fast cores (slow core = 0.5× fast core), there are only two possible orderings: $\{\{S_1\}, \{S_1, S_2\}, \{F_1\}, \{F_1, F_2\}\}$ and $\{\{S_1\}, \{S_1, S_2\}, \{S_1, S_2, S_3\}, \{F_1, F_2\}\}$. We use the first ordering since it matches exactly with the original paper. We constrain our approach to 2 slow and 2 fast cores to compare to Octopus-Man.

Request Clairvoyant (RC). The RC policy has perfect knowledge of request demand and load. It executes the longest x requests on fast cores (or highest V/F) that satisfies the target tail latency. When a long request arrives and no fast core is available, it executes on a slow core until a fast core becomes available. All other requests run exclusively on slow cores. We experimentally select the best value of x for minimum energy at the target tail latency for every load.



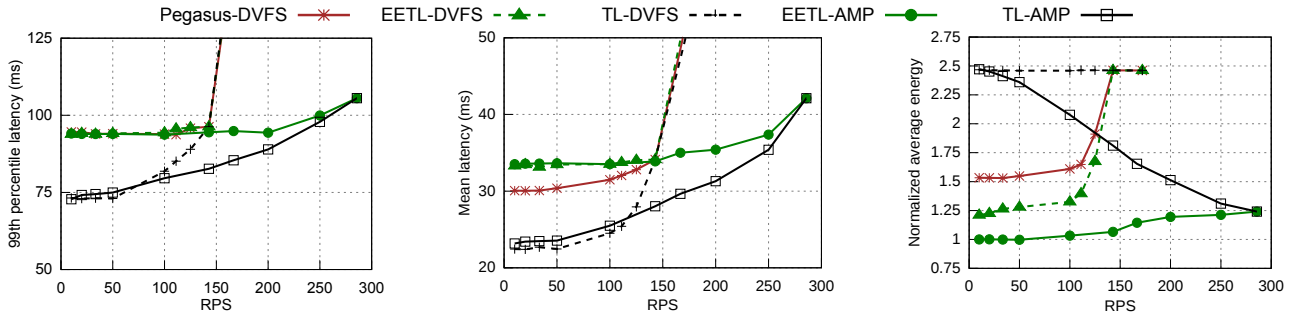
(a) 99th percentile latency (b) Average latency (c) Normalized energy consumption

Figure 8: Lucene (200 ms target) on seven per-core DVFS Broadwell cores with TL-DVFS (as fast as possible), Pegasus-DVFS (all cores at the same DVFS settings) and EETL-DVFS (per-core DVFS). EETL delivers the same tail latency as Pegasus with substantially less energy.



(a) 99th percentile latency (b) Average latency (c) Normalized energy consumption

Figure 9: Lucene (200 ms target) on emulated AMP. EETL-AMP meets the tail latency target better than all other hardware and scheduling options at lower energy.



(a) 99th percentile latency (b) Average latency (c) Normalized energy consumption

Figure 10: Finance (100 ms target) on emulated AMP. EETL-AMP meets the tail latency target at lower energy better than all other hardware and scheduling options.

6 RESULTS

We start by evaluating our system on seven per-core DVFS-enabled cores on the Broadwell server. We then focus on both DVFS and AMP via emulation; comparing four DVFS cores and AMP with two fast and thirteen slow (2F-13S) cores, which have similar peak power. A core at the highest (lowest) DVFS setting has the same computation capacity as a fast (slow) core at the AMP setting, but the AMP small cores are more energy efficient. We use a target tail latency at the 99th-percentile of 200 ms for Lucene and 100 ms for Finance unless otherwise noted. We use 200 ms target because Lucene supports it on our hardware and the literature establishes it as a good value for human interactivity [50]. We also perform a sensitivity analysis for different tail latency targets. We then explore workload and hardware sensitivity, and AMPs with three core types.

6.1 Measured Results on Broadwell with Per-Core DVFS

Figure 8 plots 99th-percentile latency (left), average latency (middle), and normalized average energy per request (right), as a function of load in Requests Per Second (RPS) for Lucene running on

the Broadwell server reported by the energy performance counters. We plot energy consumption per request normalized to the setting with the minimum energy to meet the target (EETL at 10 RPS).

Comparing the policies, we observe that exploiting DVFS can save energy. Pegasus-AV saves up to 17% energy compared to TL-DVFS, which runs at top voltage-frequency. This result is lower than the measurements in [37], but the next section shows the numbers are consistent with a more comparable hardware platform.

More interestingly, *per-core DVFS reduces energy consumption at the same tail latency.* Until the load requires all policies to set all cores to their highest V/F, EETL-DVFS consumes less energy than Pegasus-DVFS: 11% at 10 RPS and 6% at 70 RPS. EETL-DVFS configures each core as needed, and trades average latency for energy savings while achieving the same tail latency target. Since Pegasus uses the same speed for all requests, it wastes energy on short requests without improving tail latency.

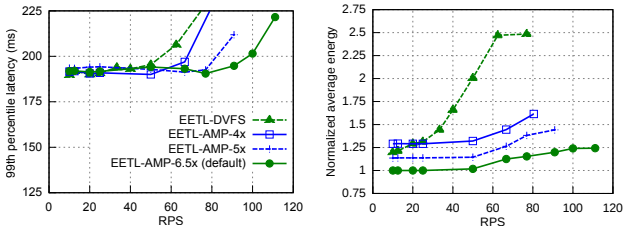
6.2 Heterogeneity for Tail Latency And Energy

Figures 9 and 10 plot 99th percentile latency, average latency, and normalized average energy on emulated AMP and DVFS per request for Lucene and the finance server, respectively. Since both workloads exhibit similar behavior, we focus on Lucene.

Pegasus-DVFS saves significant energy, up to 40%, compared to TL-DVFS. This result is consistent with [37]. The benefits of Pegasus differ slightly because more recent processors (e.g., Broadwell) have a higher fraction of static power compared to the earlier hardware (e.g., Haswell) in their study. This result further motivates the importance and effectiveness of AMP over DVFS, since AMP can reduce static power via simpler microarchitectures for slow cores. In this hardware, EETL-DVFS also performs better and consumes less energy than Pegasus-DVFS: 22% at 10 RPS and 10% at 50 RPS.

AMPs deliver significant energy savings and improved throughput over DVFS at the same tail latency. Since slow AMP cores are significantly more energy-efficient than homogeneous cores using DVFS, chip designers may add more cores to the chip for the same thermal design power. Both EETL and TL exploit this additional capacity. EETL-DVFS consumes 18% more energy at 10 RPS and 2x more energy at 50 RPS than EETL-AMP. EETL-AMP sustains almost twice as much load as EETL-DVFS (and Pegasus) under the 200-ms tail latency target. AMPs have an advantage even when only optimizing for tail latency: TL-AMP sustains twice as much load and consumes half the power than policy TL-DVFS.

Optimizing latency without considering energy consumption wastes a lot of energy. Comparing EETL-DVFS to TL-DVFS or EETL-AMP to TL-AMP reveals this insight. Figure 9(c) shows that TL-AMP consumes 2.45x more energy at 10 RPS and 1.5x at 66 RPS than EETL-AMP. EETL-AMP uses more energy as the load increases. EETL makes good use of this energy—it meets tail latency as the load increases by shortening the migration threshold until the threshold reaches 0. At this point, it behaves exactly the same as TL-AMP, with the two policies attaining the same tail latency and consuming the same amount of energy at 100 RPS and beyond. Figure 9(b) shows why TL-AMP consumes the most energy at low and moderate loads: it substantially reduces average latency, but reducing



(a) 99th percentile latency (b) Normalized energy

Figure 11: Lucene (200 ms target): Effect of microarchitectural efficiency on EETL.

average latency has a relatively small influence on tail latency. TL-AMP wastes energy by processing short requests on a fast core without benefiting tail latency.

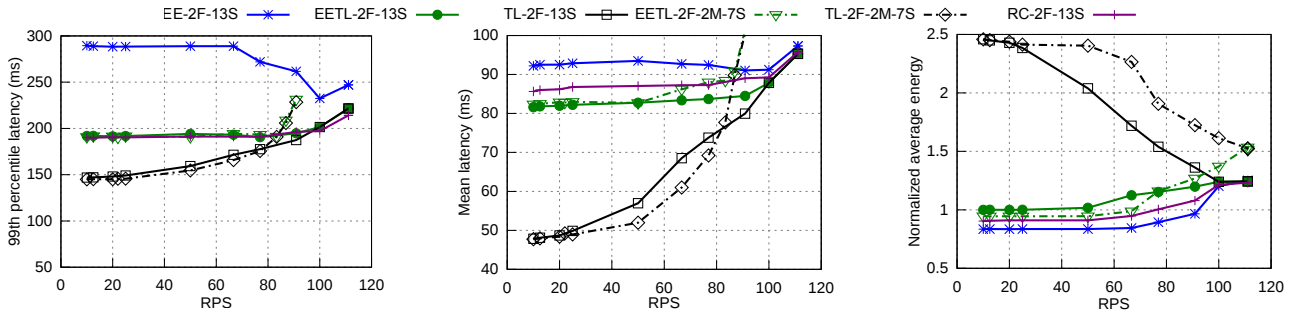
EETL effectively trades average latency for improvements in energy efficiency while meeting the target tail latency. Because EETL uses thresholds, it waits to migrate long requests to the fast core such that the request just meets the specific target. For short requests, this policy increases average latency, but reduces energy consumption, since short requests never execute on a fast core. Long requests are slightly more energy efficient as well because they execute on multiple core types. EETL is thus much more energy efficient than TL at low and moderate loads.

6.3 Sensitivity to DVFS and AMP Power Differentials

We now explore the relative energy efficiency of slow cores in AMP. The slowest DVFS setting consumes 4x less power than the fastest one. Since AMP provides better efficiency through microarchitectural differences, we use AMP configurations where slow cores consume 4x, 5x, and 6.5x less power than fast cores. With less efficient slow cores, we must reduce their number to hold peak power constant. We hold the two AMP fast cores constant. Figure 11 compares EETL on 3 AMP configurations and per-core DVFS. EETL-AMP-6.5x is the default 2F-13S (EETL-AMP in Figure 9). It achieves the best energy and throughput results. EETL-AMP-5x (2F-10S) satisfies the tail latency target up to 77 RPS, consuming 14% more energy at low load than EETL-AMP-6.5x. EETL-AMP-4x (2F-8S) sustains higher load (66 RPS) than EETL-IV (50 RPS) using per-core DVFS, even though each core in EETL-IV has five potential settings and EETL-AMP-4x has only two and an extremely conservative power model. These results strengthen the conclusion that AMPs offer significantly better energy efficiency and throughput than DVFS for same tail latency target.

6.4 Sensitivity to More Core Types in AMPs

Figure 12 explores AMP configurations with three core types (2F-2M-7S) and the two-type default (2F-13S) with EETL, TL, Energy Efficiency (EE) only, and Request Clairvoyant (RC). Optimizing only energy with EE-2F-13S does not meet the target latency, whereas EETL-2F-13S is only 20% above the minimum energy. More surprising is that the oracular RC-2F-13S policy meets the latency constraint with only 10% less energy than EETL configurations, but it is not possible to implement.



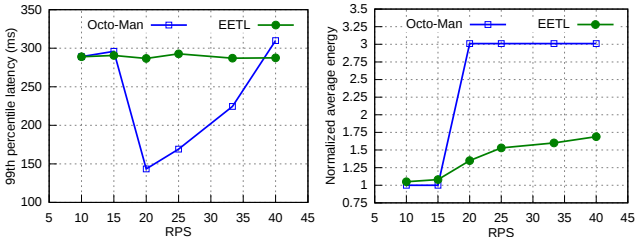
(a) 99th percentile latency (b) Average latency (c) Normalized energy consumption

Figure 12: Lucene with two core types (2F-13S) and three core types (2F-2M-7S) for 200 ms target with EETL, TL, EE, and RC. More core types deliver more energy savings.

More core types offer the potential of more efficiency with finer grain control under low and moderate load. Comparing EETL-2F-13S to EETL-2F-2M-7S reveals that more core types are more energy efficient at low to moderate load. At the highest loads, AMPs with more core types degrade tail latency due to insufficient numbers of fast cores. In addition, they consume more energy, as all cores must be active to process requests.

6.5 Comparing to Octopus-Man on AMPs

We now compare EETL to Octopus-Man [45]. Octopus-Man considers one core type at a time and orders configurations according to their computing capacity. We study Octopus-Man with 2 slow and 2 fast cores, allocating more powerful configurations as load increases, that is, transitioning from 1 slow to 2 slow to 1 fast to 2 fast as a function of load. Octopus-man does not use slow and fast cores at the same time. Figure 13 shows that Octopus-Man starts using both fast cores at 20 RPS, at which point the tail latency drops dramatically and energy consumption increases to 3x. However, there are numerous short requests that can run on slow cores without affecting tail latency. Octopus-Man misses this opportunity. On the other hand, EETL uses a mix of slow and fast cores to satisfy the tail latency target at very low energy consumption. Note that a lower tail latency target (e.g., 200 ms) would not help Octopus-Man; it would use fast cores for all requests, while EETL would reduce energy by using slow cores for short requests.



(a) 99th percentile latency (b) Normalized energy

Figure 13: Lucene (300 ms target): EETL-2F-2S judiciously uses slow and fast cores at the same time to optimize energy. Octopus-Man-2F-2S uses 1 slow, 2 slow, 1 fast, *or* 2 fast cores.

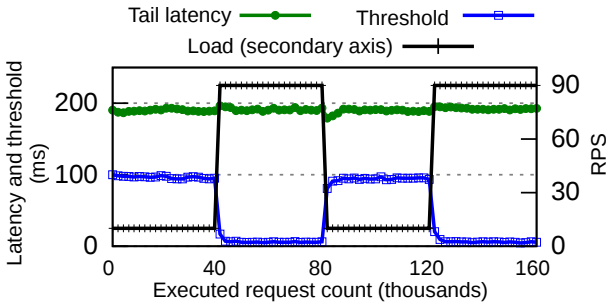


Figure 14: Lucene (200 ms target): EETL-2F-13S adapts to widely varying load while delivering the target tail latency.

6.6 Adapting to Load Spikes

Figure 14 shows that even when load varies significantly, EETL's feedback controllers dynamically adapt to meet the target tail latency. We subject EETL to extreme variations between low (10 RPS) and high (90 RPS) loads. Each point is the 99th-percentile latency of the last 2000 requests. EETL dynamically and quickly (in 500 request epochs) adjusts thresholds to meet the tail latency target consistently. The epoch size depends on the workload characteristics and latency accounting period. Smaller epochs respond faster to load changes and longer ones are more stable. The service provider should empirically select a suitable epoch size.

6.7 EETL AMP Energy Proportionality

We use our models to explore core configurations giving minimum energy as a function of tail latency target and load, demonstrating that our approach produces energy proportionality and helps choose good AMP configurations. Figure 15 plots six core configurations with the same peak power: 2F, 1F-8S, 1F-1M(-)-5S, 1F-1M-4S, 1F-1M(+)-3S, 15S. The figure shows that EETL on AMP produces energy proportional results when it can select cores based on load and tail latency target. With less stringent tail latency targets and lower load, the system consumes proportionally less energy. Furthermore, service providers can achieve energy scaling and better adapt the computing capacity, if they employ efficient small cores first, and then the faster cores.

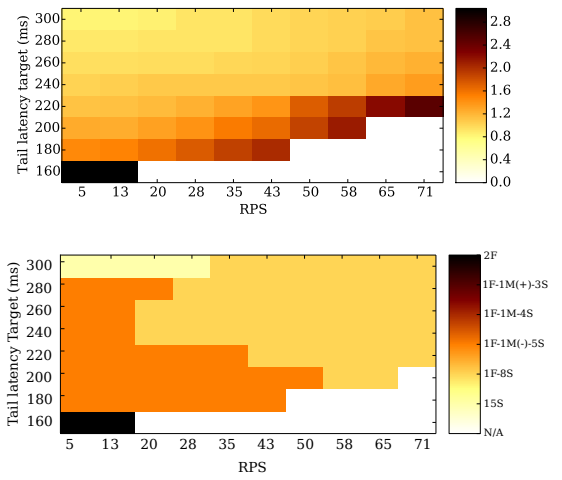


Figure 15: Lucene: EETL-AMP normalized minimum average energy (top) and core configurations (bottom).

6.8 Algorithmic Sensitivities

This section summarizes AMP (2F-13S) results that explore the effects of our static power model, tail latency constraint, workload characteristics, and policy configurations.

Effect Of Non-CPU Power. We model both static and dynamic processor power. Prior work shows that memory and other components (we call them non-CPU for short) consume at most 33% of total server power today [3, 52]. Not surprisingly, Figure 16 shows that, when non-CPU power becomes a larger fraction, the energy savings of EETL decreases. Nevertheless, even when non-CPU power is 50% and 67% of the peak dynamic power, TL consumes 42% and 24% more energy, respectively, than EETL at low load.

Effect of Modified Workload and Tail Latency Targets. We create four different workloads with Lucene by varying the search index size and consequently the achievable average and tail latency. We select the EETL tail latency target to 67% of the tail latency measured on a slow core. Figure 17 shows EETL performance for these four variants of Lucene, each with an applicable tail latency target. EETL satisfies this wide spectrum of tail latency targets generated by the four workload variants. The effectiveness of EETL is bounded by the overhead of migration. The faster the requests can migrate, the lower tail latency target EETL can support.

Effect of Workload Distribution. Previous work shows that differentiating between short and long requests is more effective as the gap increases between the mean and the tail service demand. The more uniform the request distribution, the less room for DVFS and AMP with EETL to improve energy or throughput. Our results for a modified Lucene workload with a mean of 30 ms and 99th percentile service demand of 180 ms (a larger gap) are consistent with previous findings. EETL conserves more energy compared to TL by executing even fewer short requests on the fast core.

Effect of Gain Scheduling. The EETL controller uses three load intervals for gain scheduling. We obtain similar results when using six load intervals, but using only one load interval (i.e., without gain scheduling) overshoots at high loads (>70 RPS).

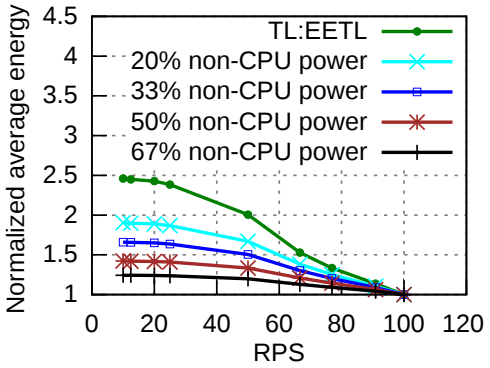


Figure 16: Lucene: TL-2F-13S energy normalized to EETL-2F-13S for different fractions of memory and other non-CPU power.

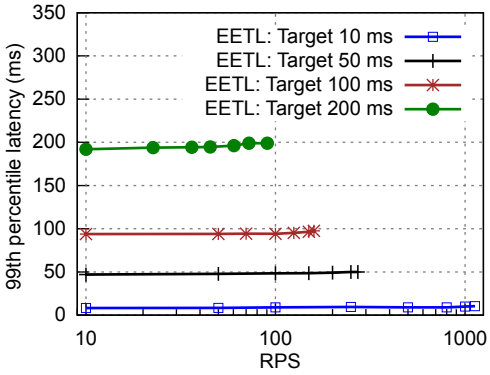


Figure 17: Lucene: EETL-2F-13S with different index sizes and tail latency targets on 2 fast and 13 slow cores.

7 CONCLUSION

This paper introduced Adaptive Slow-to-Fast, a general and efficient scheduling framework for executing interactive services on heterogeneous multicore servers. We configure it to judiciously leverage static and dynamic heterogeneity to manage tail latency and energy consumption. The resulting EETL algorithm meets tail latency targets and improves energy efficiency and throughput quite significantly compared to prior work on real and emulated hardware, especially for AMP.

ACKNOWLEDGEMENTS

We thank Santosh Nagarakatte, David Lo, Jason Mars, Liqun Cheng, and anonymous reviewers for their helpful suggestions.

REFERENCES

[1] Apache Lucene. 2014. http://lucene.apache.org/. (2014). Retreived July 2014.
[2] Omid Azizi, Aqeel Mahesri, Benjamin C. Lee, Sanjay J. Patel, and Mark Horowitz. 2010. Energy-performance Tradeoffs in Processor Architecture and Circuit Design: A Marginal Cost Analysis. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*. ACM.
[3] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Morgan & Claypool.
[4] Mark Broadie and Paul Glasserman. 1993. Estimating security price derivatives using simulation. *Management Science* 42, 2 (1993), 269–285.

[5] Ting Cao, Stephen M. Blackburn, Tiejun Gao, and Kathryn S. McKinley. 2012. The Yin and Yang of Power and Performance for Asymmetric Hardware and Managed Software. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[6] Nagabhushan Chitlur, Ganapati Srinivasa, Scott Hahn, P K. Gupta, Dheeraj Reddy, David Koufaty, Paul Brett, Abirami Prabhakaran, Li Zhao, Nelson Ijih, Suchit Subhaschandra, Sabina Grover, Xiaowei Jiang, and Ravi Iyer. 2012. QuickIA: Exploring Heterogeneous Architectures on Real Prototypes. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA)*.

[7] Gonzalo Cortazar, Miguel Gravet, and Jorge Urzua. 2006. The valuation of multidimensional American real options using the LSM simulation. *Computers and Operations Research* (2006), 113–129.

[8] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. 2010. RAPL: Memory power estimation and capping. In *International Symposium on Low-Power Electronics and Design (ISLPED)*.

[9] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (2013), 74–80.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles (SOSP)*. 205–220.

[11] Hadi Esmaeilzadeh, Ting Cao, Yang Xi, Stephen M. Blackburn, and Kathryn S. McKinley. 2011. Looking back on the language and hardware revolutions: Measured power, performance, and scaling. In *ACM International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*.

[12] Peter Greenhalgh. 2011. Big. little processing with ARM Cortex-A15 & Cortex-A7. *ARM White paper* (2011).

[13] Ed Grochowski, Ronny Ronen, John Shen, and Hong Wang. 2004. Best of Both Latency and Throughput. In *Proceedings of the IEEE International Conference on Computer Design*.

[14] Ron Guida. 2010. Parallelizing a Computationally Intensive Financial R Application with Zircon Technology. In *The R User Conference*.

[15] Gurobi Optimization Inc. 2016. Gurobi Optimization. (2016). http://www.gurobi.com.

[16] James Hamilton. 2009. The Cost of Latency. (2009). http://perspectives.mvdirona.com/10/31/TheCostOfLatency.aspx.

[17] Ronald Hankey. 2014. Electric Power Monthly with Data for December 2014. (2014). http://www.eia.gov/electricity/monthly/current_year/december2014.pdf.

[18] Md E. Haque, Yong H. Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *ACM International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*.

[19] Y. He, S. Elnikety, J. Larus, and C. Yan. 2012. Zeta: Scheduling Interactive Services with Partial Execution. In *ACM Symposium on Cloud Computing (SOCC)*.

[20] J. Hellerstein, Y. Diao, S. Parekh, and D Tilbury. 2004. *Feedback Control of Computing Systems*. Wiley-IEEE Press.

[21] M.D. Hill and M.R. Marty. 2008. Amdahl's Law in the Multicore Era. *Computer* 41, 7 (2008), 33–38.

[22] C. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, L. Tang, J. Mars, and R. Dreslinski. 2015. Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[23] Hsu, Chang-Hong and Zhang, Yunqi and Laurenzano, Michael A. and Meisner, David and Wenisch, Thomas and Dreslinski, Ronald G. and Mars, Jason and Tang, Lingjia. 2017. Reining in Long Tails in Warehouse-Scale Computers with Quick Voltage Boosting Using Adrenaline. *ACM Trans. Comput. Syst.* 35, 1, Article 2 (March 2017), 33 pages. https://doi.org/10.1145/3054742

[24] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. 2013. Speeding Up Distributed Request-response Workflows. In *SIGCOMM '13*.

[25] Sean James. 2016. Energy Efficiency and Designing the Datacenters of the Future. (2016). https://blogs.microsoft.com/green/2016/10/27/energy-efficiency-and-designing-the-datacenters-of-the-future/.

[26] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. 2010. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[27] Myeongjae Jeon, Yuxiong He, Hwanju Kim, Sameh Elnikety, Scott Rixner, and Alan L. Cox. 2016. TPC: Target-Driven Parallelism Combining Prediction and Correction to Reduce Tail Latency in Interactive Services. In *ACM International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*.

[28] Myeongjae Jeon, Saehoon Kim, Seung-Won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. 2014. Predictive parallelization: Taming tail latencies in web search. In *ACM Conference on Research and Development in Information Retrieval (SIGIR)*.

[29] Ivan Jibaja, Ting Cao, Stephen M. Blackburn, Tiejun Gao, and Kathryn S. McKinley. 2016. Portable Performance on Asymmetric Multcore Processors. In *ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*.

[30] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. 2015. Rubik: Fast Analytical Power Management for Latency-Critical Systems. In *Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[31] Saehoon Kim, Yuxiong He, Seung-Won Hwang, Sameh Elnikety, and Seungjin Choi. 2015. Delayed-Dynamic-Selective (DDS) Prediction for Reducing Extreme Tail Latency in Web Search. In *ACM International Conference on Web Search and Data Mining (WSDM)*.

[32] Data Center Knowledge. 2016. The Facebook Data Center FAQ. (2016). http://www.datacenterknowledge.com/the-facebook-data-center-faq-page-three/

[33] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. 2003. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[34] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. 2004. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[35] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, and Dean M. Tullsen. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *ACM/IEEE International Symposium on Microarchitecture (MICRO)*.

[36] Tong Li, Dan P. Baumberger, David A. Koufaty, and Scott Hahn. 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *ACM/IEEE Conference on Supercomputing*.

[37] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards Energy Proportionality for Large-scale Latency-critical Workloads. In *In Proceeding of the 41st Annual International Symposium on Computer Architecuture*.

[38] Jacob R. Lorch and Alan Jay Smith. 2001. Improving Dynamic Voltage Scaling Algorithms with PACE. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. 50–61.

[39] Lucene Nightly Benchmarks. 2014. http://people.apache.org/~mikemccand-/lucenebench. (2014). Retreived June 2014.

[40] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Ronald Dreslinski, Jr., Thomas F. Wenisch, and Scott Mahlke. 2014. Heterogeneous Microarchitectures Trump Voltage Scaling for Low-power Cores. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. 237–250.

[41] M. Demler. 2013. MediaTek Steps Up to Tablets: MT8135 Brings Heterogeneous Multiprocessing to Big.Little. (Aug 2013). Microprocessor Report, The Linley Group.

[42] Mathworks. 2015. http://www.mathworks.com/discovery/pid-tuning.html. (2015). Retreived July 2015.

[43] David Meisner, Brian T. Gold, and Thomas F. Wenisch. 2009. PowerNap: Eliminating Server Idle Power. In *ACM International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*.

[44] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the Potential of Heterogeneous Von Neumann/Dataflow Execution Models. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[45] Vinicius Petrucci, Michael A. Laurenzano, John Doherty, Yunqi Zhang, Daniel Mosse, Jason Mars, and Lingjia Tang. 2015. Octopus-Man: QoS-Driven Task Management for Heterogeneous Multicore in Warehouse Scale Computers. In *Proceedings of the 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*.

[46] Qualcomm. 2014. Snapdragon 810 Processors. (2014). https://www.qualcomm.com/products/snapdragon/processors/810

[47] Shaolei Ren, Yuxiong He, Sameh Elnikety, and Kathryn McKinley. 2013. Exploiting Processor Heterogeneity in Interactive Services. In *USENIX International Conference on Autonomic Computing (ICAC)*. 45–58.

[48] Shaolei Ren, Yuxiong He, and Kathryn McKinley. 2014. A Theoretical Foundation for Scheduling and Designing Heterogeneous Processors for Interactive Applications. *The International Symposuium on Distributed Computing (DISC)* LNCS 8784 (2014), 152–166.

[49] Juan Carlos Saez, Manuel Prieto, Alexandra Fedorova, and Sergey Blagodurov. 2010. A Comprehensive Scheduler for Asymmetric Multicore Processors. In *In Proceedings of the 5th ACM European Conference on Computer Systems (EuroSys.*

[50] Eric Schurman and Jake Brutlag. 2009. The user and business impact of server delays, additional bytes, and HTTP chunking in web search. In *Velocity Conference*.

[51] Amazon Web Services. 2017. AWS and Sustainability. (2017). https://aws.amazon.com/about-aws/sustainability/

[52] A. Shehabi, Sarah Josephine Smith, Dale A. Sartor, Richard E. Brown, Magnus Herrlin, Jonathan G. Koomey, Eric R. Masanet, Nathaniel Horner, InÃ‌s Lima Azevedo, and William Lintner. 2016. United States Data Center Energy Usage Report. (2016). http://eta.lbl.gov/sites/all/files/lbnl-1005775_v2.pdf

[53] Taejoon Song, Daniel Lo, and G. Edward Suh. 2016. Prediction-Guided Performance-Energy Trade-off with Continuous Run-Time Adaptation. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design (ISLPED)*.

[54] Jinho Suh and Michel Dubois. 2009. Dynamic MIPS Rate Stabilization in Out-of-order Processors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*.

[55] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 253–264. https://doi.org/10.1145/1508244.1508274

[56] Christopher Torng, Moyang Wang, and Christopher Batten. 2016. Asymmetry-Aware Work-Stealing Runtimes. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[57] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T. N. Vijaykumar. 2015. TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *Proceedings of the 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[58] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel S Emer. 2012. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[59] Wikipedia: Database download. 2014. http://en.wikipedia.org/-wiki/Wikipedia:Database_download#English-language_Wikipedia/. (2014). Retreived May 2014.

[60] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2016. Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading. In *USENIX Annual Technical Conference (USENIX ATC)*. 309–322.

[61] Jeonghee Yi, Farzin Maghoul, and Jan Pedersen. 2008. Deciphering mobile search patterns: A study of Yahoo! mobile search queries. In *ACM International Conference on World Wide Web (WWW)*. 257–266.