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
tdngueyn@cs.rutgers.edu

Ricardo Bianchini
Microsoft Research
ricardob@microsoft.com

Kathryn S. McKinley
Google
ksmckinley@gmail.com

ABSTRACT
Interactive service providers impose strict requirements on high percentile latency (tail latency) to meet user expectations. If providers meet latency targets with less energy, they improve profits, because energy is a significant operating expense. A promising approach to optimizing tail latency and energy efficiency of interactive services is to execute long requests at faster speeds and short requests at slower speeds with less energy. This paper shows how to use Asymmetric Multi-core Processors (AMPs) where distinct core types offer distinct performance and power characteristics to realize this approach. AMPs offer potential energy savings over Dynamic Voltage and Frequency Scaling (DVFS) because AMPs increase core capacity at the same power budget. AMPs and DVFS would be the same problem if the desired AMP core type was always available, but in practice, AMPs have many few core types than total cores and thus scheduling AMP is harder because requests compete for core types, e.g., a slow core type may not be available for a short request and a fast core type may not be available for a long request.

We introduce the Adaptive Slow-to-Fast scheduling framework which prioritizes long requests to faster cores, 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 AMP and DVFS. This policy starts requests on slow cores and migrates them to faster cores. When a desired AMP core speed is not available but a faster core is, the scheduler migrates the longest request on any core type to the faster core and assigns the shorter request to an s core type. EETL for AMPs delivers the same tail latency, energy savings of 18% to 50%, and capacity (throughput) gains of 32% to 82% over per-core DVFS systems. Our framework effectively exploits dynamic DVFS and static AMP heterogeneity to reduce provisioning and operational costs.

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

KEYWORDS
Tail Latency, Heterogeneous Processors, Energy Efficiency

1 INTRODUCTION
Interactive services, such as Web search, financial trading, games, and social networking, require consistently low response times to attract and retain users [17, 50]. Service providers thus define strict targets for 95th-percentile or higher response times, commonly called tail latency [10, 11, 20, 24, 61]. Services typically distribute requests and aggregate responses from multiple servers. To meet a 95th-percentile latency target when aggregating responses from five servers requires them to deliver the target at an even more stringent $\approx 99$th-percentile.

Datacenters for interactive services incur substantial provisioning and operating costs. Sheth 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 [51]. With prices per KW hour of 4 to 21 cents [18], the U.S. datacenter energy bill is 1 to 6 billion dollars. Large providers, such as Facebook [31], Google [4], and Microsoft are highly motivated to reduce energy consumption, since even a 1% reduction saves them $1 to 5 million.

Improving tail latency and energy efficiency are conflicting goals. Interactive services however provide an opportunity to combine them since the computation time in the tail is often 10× longer than the average, and 100× longer than the median [10, 24, 26]. This characteristic makes increasing the energy efficiency of short requests and spending more energy on long requests appealing. Computationally intensive requests are a primary factor in tail latency [10, 24, 26, 30, 36], although queueing due to bursty load, hardware failures, paging, and network congestion are sometimes root causes of tail latency as well. To optimize tail latency with energy efficiency, this paper explores accelerating long or delayed requests with faster hardware using (i) Dynamic Voltage and Frequency Scaling (DVFS) per-core on a chip multiprocessor, creating...
dynamic core frequency heterogeneity, and (ii) Asymmetric Multi-core Processors (AMPs) with \( n \) core types that differ in their static peak performance, power, and energy characteristics, where \( n \) is much less than the total number of cores. We refer to these architectures collectively as heterogeneous multicores. Scheduling DVFS and AMP is the same problem if the desired AMP core speed and DVFS core speed are always available, but in practice, AMP is more challenging than DVFS because it has finite numbers of each core type and requests compete for core types. AMP however offers substantially more power reductions and capacity benefits than DVFS through static microarchitectural techniques [3, 14, 32, 39].

Accelerating long requests with AMP and DVFS is challenging because (i) individual request demand is hard to predict and prediction is never free or perfect [23, 24, 30, 37]; (ii) the desired core speed or type may not be available; and (iii) load is bursty. We use request progress to dynamically detect and accelerate long requests. DVFS control does not require a scheduler, since each core may independently set its speed. AMP requires a scheduler to resolve competition for core types. For instance, when a new request arrives, starting it on a slow core is most energy efficient, but only a fast core may be available. To satisfy this competition for slow cores, migrating an oldest request currently on a slow core to the available fast core and assigning the new request to the newly available slow core optimizes tail latency and energy efficiency. Although research shows that AMPs deliver much better energy efficiency than DVFS [32, 39], because servers already have DVFS, more prior work optimizes for it [23, 29, 36, 57].

Table 1 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 cores (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 these problems posed by AMP. Only Ren et al. [47, 48] implement the required scheduler. Ren et al. 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 either use load to configure the entire system, wasting energy on short requests [36, 45]; use request progress to configure individual cores [23, 57]; or use both load and request progress to configure individual cores [29]. Lacking a scheduler, they do not unlock the full potential of AMP. Section 2 describes all prior work in more detail. The two key practical challenges for AMP that this paper addresses are (1) when to migrate a request from slow to fast cores based on request progress and system load, and (2) what to do when a core with the desired speed is not available.

We introduce the Adaptative Slow-to-Fast scheduling framework that exploits heterogeneous multicores to optimize the energy and tail latency of interactive services. We consider heterogeneity deriving from DVFS and static core heterogeneity in AMPs [3, 7, 13, 14, 22, 32, 33, 35, 39, 40, 46, 55]. Our experimental results show our framework more effectively uses AMPs compared to prior AMP algorithms [45, 48] and more effectively uses per-core DVFS technologies [1, 44] compared to the state-of-the-art DVFS algorithms [23, 36].
These results show that, even though our scheduler focuses on op-erating cores may not be true in the future. It is not actually a scheduler — it reacts to scheduling decisions. 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 [25] 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, 30, 37]. 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 AMP 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 [6, 28, 49, 52, 56, 58, 59].

Hardware trends. We believe AMPs and DVFS are compatible with the growing adoption of hardware accelerators for key computations. Indeed, DVFS cores already co-exist with accelerators, such as FPGAs, ASICs, GPUs, and TPUs. A drawback of accelerators is once hardware manufacturers deliver an accelerator or change it, developers must specify which code should run on which accelerator and then hand specialize the code, although compilers are emerging to help with this task. We seek to unlock the potential of general purpose heterogeneity and thus benefit a larger range of workloads. DVFS is everywhere and AMPs are already present in mobile systems [13, 46] and server prototypes [7]. Prior work suggests heterogeneous systems are a promising general architecture [3, 14, 22, 32, 33, 35, 39, 40, 55]. This paper shows heterogeneous systems offer significant advantages that further motivate their design and purchase for the large fraction of datacenters that host high revenue interactive services, such as search and ads.

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 [36, 45]; (2) it uses request progress, in contrast to prediction [23, 29]; 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 [36, 45, 47, 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, and we leave their combination for future work.

Tail latency with DVFS multicores. Prior work manages tail latency and energy with DVFS [23, 29, 36, 57]. Pegasus uses a feedback-based DVFS controller based on total system load that operates at a one second granularity [36]. At low load, it reduces the voltage/frequency of all servers, increasing average latency, while main-taining 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 [29]. It assumes a desired core frequency is always available, which in power limited cores may not be true in the future. It is not actually a scheduler —
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.

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.) Both slow and fast cores easily meet the 200-ms tail latency target for the shortest 50% of requests, but only the fast core meets the target for the longest requests. The slow core substantially improves energy efficiency of short requests, by almost a factor of 3, with tail latencies of roughly 83 ms, well below the target. Thus, by using energy-efficient slow cores for short requests — and most requests are short [10, 19, 24, 26] — service providers have a significant opportunity to improve energy efficiency without compromising tail latency.

Effective Scheduling for Heterogeneity. Prior work proposed migrating requests to faster cores as long requests reveal themselves [48]. Their basic slow-to-fast (S2F) algorithm has the following desirable qualities.

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

It 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 when a core of the desired speed is not available and does not adapt to load. At higher loads, an effective policy should adapt, using fast cores more often to meet the tail latency target by migrating requests sooner. Figure 2 illustrates the limitations of 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.

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 dynamic and static performance/energy heterogeneity available from per-core DVFS and 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. 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 without a power cap, 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. The online scheduler uses the controllers to migrate requests (threads).
Table 2: Gain values for the three PID controllers based on load (measured in RPS) for Lucene.

<table>
<thead>
<tr>
<th>Load interval</th>
<th>Representative load</th>
<th>( K_p )</th>
<th>( K_i )</th>
<th>( K_d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 50</td>
<td>25</td>
<td>0.11</td>
<td>0.49</td>
<td>0.00</td>
</tr>
<tr>
<td>50–70</td>
<td>60</td>
<td>0.15</td>
<td>0.25</td>
<td>0.00</td>
</tr>
<tr>
<td>&gt; 70</td>
<td>80</td>
<td>0.19</td>
<td>0.17</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Based on the thresholds and request progress. Feedback control mitigates many transient factors, such as request interference and workload demand variability, while providing stable behavior. Each request self-schedules, so the framework incurs very low overhead and scales with the number of cores.

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. The key challenge is 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.

**EETL: Energy Efficiency with Target Tail Latency.** Our main objective is to meet a target tail latency as energy-efficiently as possible. 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 also consider the extreme settings for the migration threshold in two additional policies: **TL (Tail Latency)** and **EE (Energy Efficiency).** TL sets the migration threshold to zero, executing requests as fast as possible. EE sets the migration threshold to infinity, using fast cores only if 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 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 threshold and load, 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. 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. So, we use Gain Scheduling [21] to 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 [21] with Proportional-Integral-Derivative (PID) control, because it is effective and thus popular [53, 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 [41]. 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 slow to fast cores.

4.3 Online Scheduling

At run time, each request self-schedules, periodically executing Algorithm 1. When (i) the request execution time crosses the threshold, (ii) the request is not executing on a fast core, and (iii) a fast core is available, the request migrates to the fast core (Line 5–9). An older request migrates to a fast core (if available) before its execution time crosses the threshold if the total number of requests in the system exceed the number of cores (Line 10–14). A new request executes with lower priority if all cores or all slow cores are occupied (Lines 16–17). Otherwise, a new request starts at the regular priority on a slow core (Lines 18–20). We track request age using a synchronized
We divide the procedure into two steps. We first determine the

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
5 if (i <= |F| and a >= t) then // migrate after crossing t
6 if (c \in 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 \in F) then
12 f = free core from F
13 migrate from c to f
14 end
15 else if (c is None) then // very high load or sync delay
16 if (i > |F| + |S|) or No free core in S) then // regular request arrival
17 start running with low priority
18 else
19 s = free core from S
20 start executing at s
21 end
22 end

4.4 Controller Design for N Core Types

This section describes how to create a 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 im-
prove energy efficiency by striking better power/performance trade-
offs [7, 22, 32, 33, 35]. 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 thresh-
olds 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 prob-
lem, where the search space grows exponentially with the number
of speeds. To simplify it, we introduce a decision variable that con-
verts it into a single-dimensional 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
formulation only applies to low load because it assumes no core
contention—each request may execute on a core of type c_{i+1} when-
ever 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 [16].

Table 3: Definitions for computing migration thresholds.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>r \in R</td>
<td>Request profiles</td>
</tr>
<tr>
<td>slow_r</td>
<td>Runtime of request r on slowest core</td>
</tr>
<tr>
<td>s_i</td>
<td>Avg speedup of requests on core type c_i</td>
</tr>
<tr>
<td>p_i</td>
<td>Peak dynamic power on c_i</td>
</tr>
<tr>
<td>target_{1}</td>
<td>Target tail latency</td>
</tr>
<tr>
<td>V = [0, v_1, v_2, ..., v_{N-1}]</td>
<td>Intermediate representation of t such that v_i = (t_i - t_{i-1})</td>
</tr>
<tr>
<td>\epsilon_r(V)</td>
<td>Energy used by r with schedule V</td>
</tr>
<tr>
<td>Latency_{\beta}(V, \beta-tail)</td>
<td>\beta-th percentile latency of R with schedule V</td>
</tr>
<tr>
<td>Latency_{\gamma}(V, \beta-tail)</td>
<td>\beta-th percentile latency of R with schedule V</td>
</tr>
</tbody>
</table>

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

The second step uses the thresholds determined for low load as a
starting point for finding thresholds at higher loads. Given the solu-
tion of the problem in Figure 5 is of the form t^* = \{0, t_1^*, t_2^*, ..., t_{N-1}^*\}, we use
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 thresh-
olds. 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 execut-
ing 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 Sec-
tion 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 dependent 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 popular open-source Apache Lucene [2], an enterprise
search engine, and a Monte Carlo finance server [5, 8, 20]. Both
servers use a standard pool of worker threads. A worker thread
dequests a queue from the arrival queue and processes the request
to completion.

We configure the Java Lucene search engine with 33+ million
Wikipedia English web pages [38, 60]. 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 [19]. We use 8 K requests to design the feedback controller.
For each performance experiment, a client issues a random subset of 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 (through statically linked C module) to call sched_setaffinity for pinning requests to specific cores. We also add an 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 [5, 8, 20]. 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 parameter values, 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 validate our framework for 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 heterogeneity (AMP and per-core DVFS) on it using duty-cycling threads. Emulation is the best option for our evaluation since (1) AMPs are currently available only in mobile systems [13, 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 Power Measurement and Model

For each performance experiment, we evaluate the调度器 in existing systems and evaluate on emerging systems. We validate our approach on an 8-core Broadwell processor. Since Broadwell did not have enough cores and we also want to evaluate spectrum of core performance/energy tradeoffs, we leverage a 16-core Haswell Xeon processor.

We report real measured processor power for the Broadwell processor in our validation experiments. We use ACPI userspace power governor and modify “cpufreq/scaling_setspeed” file to change the frequency. Frequency changing of a core takes around 50–90 µs. For the Xeon processor, we report energy based on measured performance and a model of static and dynamic power. We use the five DVFS settings from the Xeon processor. Because the Xeon 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 relative AMP performance characteristics from the ARM big,LITTLE specification [13]. ARM big (Cortex-A15) cores delivers roughly 2x performance over little (Cortex-A7) cores. Our default configuration (2F-13S) has two core types: 2 fast (big) and 13 emulated slow (little) cores. The remaining core of the 16 issues the requests, running the client.

For Haswell processor, we use McPAT [34] to model power consumption of DVFS, slow cores, fast cores, and the last-level cache. The architectural details are taken from the literature [43], where 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 ∝ performance^α) similar to [42].
Using the numbers from McPAT for slow and fast cores as base cases, $\alpha$ 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.

<table>
<thead>
<tr>
<th>Core type</th>
<th>Core name</th>
<th>Performance ($c_i$)</th>
<th>Peak power ($p_i$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Slow</td>
<td>1</td>
<td>1.7 GHz, 1</td>
</tr>
<tr>
<td>M</td>
<td>Medium−</td>
<td>1.44</td>
<td>2.7</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>1.59</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td>Medium+</td>
<td>1.71</td>
<td>4.5</td>
</tr>
<tr>
<td>F</td>
<td>Fast</td>
<td>2</td>
<td>3.4 GHz, 6.8</td>
</tr>
<tr>
<td>DVFS</td>
<td>Slow</td>
<td>1</td>
<td>1.7 GHz, 1</td>
</tr>
<tr>
<td></td>
<td>Medium−</td>
<td>1.3</td>
<td>2.2 GHz, 2.7</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>1.6</td>
<td>2.7 GHz, 3.8</td>
</tr>
<tr>
<td></td>
<td>Medium+</td>
<td>1.9</td>
<td>3.3 GHz, 5.3</td>
</tr>
<tr>
<td></td>
<td>Fast</td>
<td>2</td>
<td>3.4 GHz, 6.8</td>
</tr>
</tbody>
</table>

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

With our default power model and configuration, slow cores are $\sim 3x$ more energy-efficient than fast cores, which is consistent with prior measurements ($3.3x$ [25], $2.7x$ [12], $3.5x$ [13]), and simulation ($3.01x$ [39]). 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 [36]. We focus on processor energy because it dominates: consuming 67% of the server power [4]. 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. We 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.

5.5 Scheduling Policies

Tail Latency (TL). This policy minimizes tail latency by running requests at the highest possible speed. TL for DVFS (TL-TV) uses all cores at the highest Voltage/Frequency (V/F) level. TL for AMPs (TL-H) starts executing requests on the fastest available core. When a fast core $c_1$ becomes available, the oldest request executing on $c_1 = 1$ migrates to $c_1$. 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-IV) uses controllers to set the V/F of each core independently. EETL for AMP (EETL-H) 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 [36]. Pegasus selects a single power setting for all cores, using RAPL [9] 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 $\{S_1, S_1, S_1, \ldots, S_1, S_2, \ldots, S_N, \{F_1, F_2, \ldots, F_L\}, \ldots, \{F_1, F_2, \ldots, F_M\}\}$, where $L$ fast cores provide equal or higher performance than $N$ slow cores. For 13 slow and 2 fast cores (slow core $= 0.5x$ fast core), there are only two possible orderings: $\{S_1, S_1, S_2, \ldots, F_1, F_2\}$ and $\{S_1, S_1, S_2, \ldots, F_1, F_2\}$. 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 against Octopus-Man.

Request Clairvoyant (RC). The RC policy has perfect knowledge of request demand and system load. It executes the longest $x$ requests on fast cores (or the 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.

6 RESULTS

We start by validating our approach on system with seven per-core DVFS-enabled cores using our 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 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 this value is mentioned in literature [50]; with sensitivity analysis
Figure 8: Lucene (200 ms target) on Broadwell per-core DVFS with TL-TV (as fast as possible), Pegasus-AV (all-core DVFS) and EETL-IV (per-core DVFS). \(^\dagger\) EETL delivers the same tail latency as Pegasus with substantially less energy.

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

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

\(^\dagger\)EETL-H on 2F-13S AMP delivers significant energy and/or throughput improvements over Pegasus and TL. AV: All 4 cores with same DVFS setting; IV: Independent per-core DVFS, TV: all cores at top V/F.

for different tail latency targets. We then explore workload and hardware sensitivity, and AMPs with three core types.

6.1 Validation With Real 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. We report 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-TV, which runs at top voltage-frequency. (This result is lower than the measurements in [36], 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-IV consumes less energy than Pegasus-AV: 11% at 10 RPS and 6% at 70 RPS. EETL-IV 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 per request for Lucene and the finance server, respectively. Since both workloads exhibit similar behavior, we focus on Lucene.

Pegasus-AV saves significant energy (up to 40%) compared to TL-TV in this case. This result is consistent with [36]. The discrepancy in Pegasus benefits derives from the higher fraction of static power of more recent processors (e.g., Broadwell) compared to earlier ones (e.g., Haswell). This motivates the importance and effectiveness of AMP even further (over DVFS), since AMP can reduce static power via simpler microarchitectures for some of the cores. In this hardware, EETL-IV also performs better and consumes less energy than Pegasus-AV: 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-IV consumes 18% more energy at 10 RPS and 2x more energy at 50 RPS than EETL-H. EETL-H sustains almost twice as much load as EETL-IV and (Pegasus) under the 200-ms tail latency target. AMPs have an advantage even when only optimizing for tail latency: TL-H sustains twice as much load and consumes half the power than policy TL-TV.

Optimizing latency without considering energy consumption wastes a lot of energy. Comparing EETL-IV to TL-TV or EETL-H to TL-H reveals this insight. Figure 9(c) shows that TL-H consumes 2.45x more energy at 10 RPS and 1.5x at 66 RPS than EETL-H. EETL-H 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-H, 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-H consumes the most energy at low and moderate loads: it substantially reduces average latency, but reducing average latency has a relatively small influence on tail latency. TL-H 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-H-6.5x is the default 2F-13S (EETL-H in Figure 9). It achieves the best energy and throughput results. EETL-H-5x (2F-10S) satisfies the tail latency target up to 77 RPS, consuming 14% more energy at low load than EETL-H-6.5x. EETL-H-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-H-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 the 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-H-2F-13S does not meet the target latency, but EETL-H-2F-13S is only 20% above the minimum energy. More surprising is that the oracular RC-H-2F-13S policy meets the latency constraint with only 10% less energy than EETL configurations, but it is not possible to implement.

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

6.5 Comparing Against 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.

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-1F-8S 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, 1F-1M(+)-3S. 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 [4, 51]. 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% (67%) of the peak dynamic power, TL consumes 42% (24%) more energy than EETL at low load.

Effect of Modified Workload and Tail Latency Targets. Figure 17 shows EETL performance for four variants of Lucene each with an applicable tail latency target. We create four different versions of Lucene index by varying its 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. EETL satisfies this wide spectrum of tail latency targets. The effectiveness of EETL is bounded by the overhead of migration technology. 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 intervals for gain scheduling. We obtain similar results when using six load intervals, but using only one load interval (i.e., without gain scheduling) results in significant overshoots at high loads (>70 RPS).

7 Conclusion
This paper introduced Adaptive Slow-to-Fast, a general and efficient scheduling framework for executing interactive services in multicore servers. We configure Adaptive Slow-to-Fast to judiciously leverage static and dynamic heterogeneity to manage tail latency and energy consumption. Our results show that Adaptive Slow-to-Fast improves energy efficiency and throughput quite significantly, especially on AMP hardware, while meeting tail latency targets.

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

References


