Rubik: Fast Analytical Power Management for Latency-Critical Systems

Harshad Kasture Davide B. Bartolini Nathan Beckmann Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{hkasture, db2, beckmann, sanchez}@csail.mit.edu

ABSTRACT

Latency-critical workloads (e.g., web search), common in datacenters, require stable tail (e.g., 95^{th} percentile) latencies of a few milliseconds. Servers running these workloads are kept lightly loaded to meet these stringent latency targets. This low utilization wastes billions of dollars in energy and equipment annually.

Applying dynamic power management to latency-critical workloads is challenging. The fundamental issue is coping with their inherent *short-term variability*: requests arrive at unpredictable times and have variable lengths. Without knowledge of the future, prior techniques either adapt slowly and conservatively or rely on application-specific heuristics to maintain tail latency.

We propose Rubik, a fine-grain DVFS scheme for latency-critical workloads. Rubik copes with variability through a novel, general, and efficient statistical performance model. This model allows Rubik to adjust frequencies at sub-millisecond granularity to save power while meeting the target tail latency. Rubik saves up to 66% of core power, widely outperforms prior techniques, and requires no application-specific tuning.

Beyond saving core power, Rubik robustly adapts to sudden changes in load and system performance. We use this capability to design RubikColoc, a colocation scheme that uses Rubik to allow batch and latency-critical work to share hardware resources more aggressively than prior techniques. RubikColoc reduces datacenter power by up to 31% while using 41% fewer servers than a datacenter that segregates latency-critical and batch work, and achieves 100% core utilization.

Categories and Subject Descriptors

C.5.5 [**Computer system implementation**]: Servers

Keywords

DVFS, power management, colocation, latency-critical, tail latency, interference, isolation, quality of service

MICRO-48, December 05–09, 2015, Waikiki, HI, USA

ACM 978-1-4503-4034-2/15/12 ...$15.00
DOI: http://dx.doi.org/10.1145/2830772.2830797

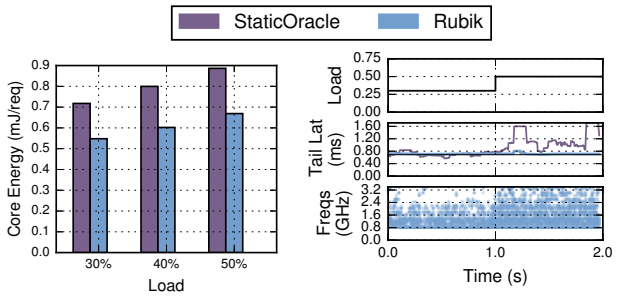
1. INTRODUCTION

Latency-critical applications are common in current datacenters and pose new challenges for system designers. *Tail latency*, not average latency, determines the performance of these applications. For example, web search leaf nodes must provide 99^{th} percentile latencies of a few milliseconds [7,56]. Servers running latency-critical work are kept lightly loaded to meet these latency targets, with typical utilizations between 5% and 30% [1,8,38]. This low utilization wastes billions of dollars in infrastructure and, since servers are not energy-proportional, terawatt-hours of energy annually [1,33,41].

Dynamic voltage/frequency scaling (DVFS) can reduce power consumption at low utilization, but it is hard to apply without degrading tail latency. The key challenge is coping with the inherent *short-term variability* of latency-critical applications: requests arrive at unpredictable times and are often bursty, causing short-term spikes and queuing delays that dominate tail latency [22,25]; and the amount of work per request often varies by an order of magnitude or more [16,25].

Without knowledge of the future, DVFS schemes must somehow cope with the uncertainty of latency-critical workloads when they adjust frequencies to avoid tail latency degradation. Traditional DVFS schemes focus on long-term throughput and ignore short-term variability, violating tail latency [33,39]. Recent work has proposed techniques that account for uncertainty, but leave significant performance on the table (Sec. 2). Pegasus [33] uses a feedback controller to adjust voltage and frequency every few seconds in response to changes in measured latency. Pegasus adapts to diurnal variations, but not to short-term, sub-millisecond variability. Adrenaline [16] sets voltage and frequency at a per-query granularity, using application-level information to identify and selectively speed up long queries. Adrenaline adapts much faster than Pegasus. However, it does not consider queuing delays, which often dominate tail latency, and relies on application-specific heuristics and tuning.

In this work we propose *Rubik*, a scheme that uses fine-grain DVFS to quickly adapt to both long- and short-term variability, minimizing power consumption while meeting tail-latency bounds (Sec. 4). Rubik uses a statistical model to account for the uncertainty of latency-critical applications, including queuing delays and per-request compute requirements. Rubik uses lightweight online profiling to periodically update this model.



(a) Core Energy (b) Response to load change

Figure 1: Rubik and StaticOracle on `masstree`: (a) Energy per request at 30–50% loads. (b) Response to load changes: input load over time (top panel), tail latency over a 200 ms window (middle panel), and Rubik's frequencies over time (bottom panel).

Rubik queries the model *on each request arrival and completion* to determine the lowest safe frequency to use given the current system state.

Fig. 1 shows Rubik in action on `masstree`, a high-performance key-value store [36] (see Sec. 5.1 for methodology details). Fig. 1a compares the energy per request of Rubik and StaticOracle, an oracular scheme that chooses the lowest static frequency for a given request trace, at three different loads: 30%, 40% and 50%. Rubik outperforms StaticOracle because it adjusts frequencies at sub-millisecond granularity, reducing core energy by up to 23% over StaticOracle. Fig. 1b shows how these schemes respond to a sudden change in load, from 30% to 50%, at $t = 1s$. While StaticOracle misses the latency target when load increases (middle panel), Rubik adapts quickly, immediately choosing higher frequencies (bottom panel) to maintain a flat tail latency.

Because Rubik adapts quickly, it can counter not only short-term variability in the workload, but performance variability due to shared hardware resources. We use this capability to design RubikColoc, a scheme that colocates and manages batch and latency-critical applications in the same server, allowing them to share resources more aggressively than prior techniques (Sec. 6). Specifically, RubikColoc uses memory-system partitioning to guard against interference in shared caches and main memory, and shares cores among batch and latency-critical applications, using Rubik to guard against short-term performance degradation. RubikColoc achieves 100% core utilization without degrading tail latency.

In summary, we make the following contributions:

- We develop a novel statistical model that accurately predicts the instantaneous performance needs of latency-critical applications, accounting for per-request variability and queuing delays. Our model is general and does not use application-specific heuristics.
- We present *Rubik*, a lightweight fine-grain DVFS scheme that leverages this model to minimize power while meeting strict latency bounds for latency-critical applications. We evaluate Rubik in simulation and on a real system. Rubik reduces active core power by up to 66%, and outperforms state-of-the-art techniques.
- We present *RubikColoc*, a colocation scheme that uses Rubik to share resources among latency-critical and batch work. RubikColoc reduces datacenter power by up to 31% and uses 41% fewer servers than a datacenter that segregates latency-critical and batch work.

2. BACKGROUND AND RELATED WORK

Anatomy of latency-critical applications: Large-scale online services (e.g., web search) operate on massive datasets spread among many nodes. Thousands of nodes collaborate in serving each request [2, 7, 33], and the few slowest nodes determine overall service latency. Single-node latencies must therefore be small, tightly distributed, and uniform across nodes. Interactivity requirements dictate that these services achieve end-to-end latencies of about 100 ms [7, 52], which requires individual leaf nodes to have *tail latencies* (e.g., 95^{th} or 99^{th} percentile latencies) of a few milliseconds [7, 56] or lower. These applications thus need *guaranteed short-term performance*. Since tail latency increases quickly with load, nodes run at low utilization to avoid large queuing delays and handle traffic spikes gracefully [25, 33].

These strict requirements preclude conventional dynamic power management and colocation techniques, which hurt tail latency. We now discuss prior work that focuses on these problems.

2.1 Dynamic Power Management

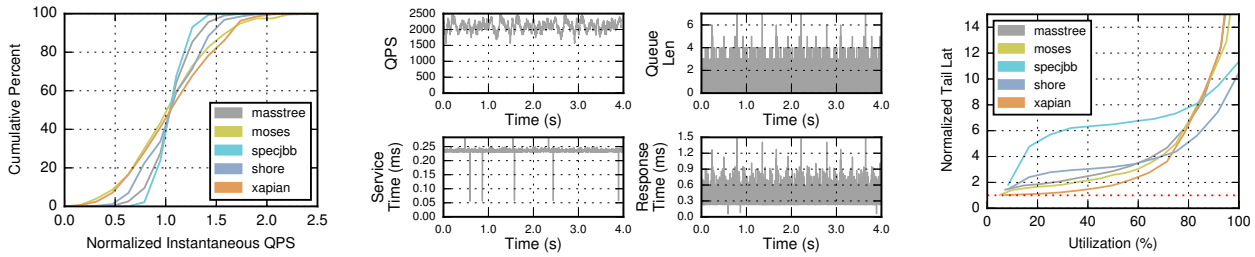
Prior work has proposed DVFS schemes to improve efficiency in multithreaded [6, 17, 21] and multiprogrammed [32, 51] batch applications. However, recent work has shown that these schemes are unsuitable for latency-critical applications since they focus on long-term performance and severely hurt tail latency [24, 33, 34, 39].

The strict latency requirements of these applications also make it hard to use sleep states to reduce idle power. Full-system idle power modes have transition times on the order of seconds and are therefore unsuitable. Deep CPU sleep states also hurt tail latency, since they flush significant microarchitectural state (e.g., the last-level cache), causing long wakeup latencies [25], while shallow sleep states save limited power [24]. In the context of datacenters, recent work has proposed using coordinated deep sleep modes with reduced transition times [38, 41, 57]. However, latency-critical applications have extremely short idle periods (a few ms or less) even at low utilization, and would need to batch tasks to create sufficiently long idle periods to benefit from deep sleep techniques, causing latency violations [33, 39].

While off-chip regulators can take tens to hundreds of microseconds to adjust voltage [27, 39], recent techniques based on on-chip voltage regulators [5, 12, 27, 44] have sub-µs delays (e.g., 500 ns on Haswell [5]). Rubik leverages these fast voltage transition times, updating voltage/frequency at sub-millisecond granularity to counter short-term load variations (Sec. 4).

2.2 DVFS for Latency-Critical Applications

In designing Rubik, a key challenge was predicting how frequency changes affect tail latency. This is hard



(a) Input QPS over time

(b) Execution trace for `masstree`: QPS, service times, queue lengths and response latencies over time

(c) Tail latency vs load

Figure 2: Analysis of main factors that add short-term variability and contribute to tail latency.

Application	Service Time	Inst. QPS	Queue Length
`masstree`	0.03	0.09	**0.94**
`moses`	0.08	0.40	**0.93**
`specjbb`	0.40	0.08	**0.66**
`shore`	0.56	0.17	**0.63**
`xapian`	0.50	0.32	**0.75**

Table 1: Pearson correlation coefficients of end-to-end response latency with service time, instantaneous QPS, and queue length.

because tail latency depends on the complex interplay of arrival, queuing, and service time distributions. Prior work has avoided this problem in three ways.

First, schemes that optimize for responsiveness in embedded systems, such as PACE [35,58] and Grace [60], try to satisfy each request by a given deadline, and do not consider queuing time. Ignoring queuing works well in interactive systems that run one task at a time, but is not applicable to datacenter servers, where queuing is significant and unavoidable [22, 25].

Second, Pegasus [33] takes a feedback-based approach: it measures tail latency periodically and adjusts frequency every few seconds to keep tail latency within a given bound. Pegasus uses a workload-specific feedback controller to drive its decisions. Relying exclusively on feedback is simple, but it is unresponsive to short-term variability, since reliably measuring tail latency takes a large number of requests. Thus, pure feedback-based approaches adapt to long-term variations (e.g., diurnal patterns), but cannot exploit short-term variability. Instead, Rubik combines short-term, analytical adaptation and long-term, feedback-based adaptation. These complementary techniques let Rubik improve efficiency further without degrading tail latency.

Third, Adrenaline [16] relies on the intuition that long requests are more likely to contribute to the tail, and uses application-level hints to identify these requests. These long requests are *boosted*, i.e., run at a higher frequency than other requests. The optimal frequency settings for boosted and unboosted queries at each load level are determined via a dynamic search process. Changes in load are sensed by a load monitor at coarse granularity (tens of minutes), which triggers the dynamic search process. Adrenaline is thus unable to respond to sudden load changes. Additionally, Adrenaline relies on application-level hints to identify long requests before they start being served. As we show in Sec. 3, not all applications are amenable to hints. Finally, tail latency is often largely determined by queuing delay, even at moderately low loads (Sec. 3). Adrenaline does not account for queuing delays explicitly, so it needs to choose conservative frequency settings to avoid tail latency violations.

3. SHORT-TERM VARIABILITY IN LATENCY-CRITICAL WORKLOADS

Rubik uses fast, per-core DVFS to quickly adapt to the short-term performance variability inherent to latency-critical applications. To do this, Rubik must determine the appropriate frequency that maintains tail latency in the presence of multiple sources of uncertainty: variability in compute requirements across requests, changes in input load, and queuing. Statistical modeling is a natural way to achieve this, but for the model to be practical, it needs to be simple enough to be used online, and accurate enough to capture the key factors that affect response time. We first analyze these factors.

Fig. 2a shows the cumulative distribution function (CDF) of instantaneous load, measured in queries per second (QPS) over a rolling 5 ms window, for five latency-critical applications (see Sec. 5.1 for methodology details). For each application, the load is normalized to the average load over the entire run. Instantaneous load varies from nearly zero to more than twice the average load. Intuitively, applications should run at low frequency in low-load periods to save power, and at high frequency in high-load periods to maintain tail latency.

However, determining the right frequency can be hard in practice. Fig. 2b shows a four-second execution trace of `masstree`, a high-performance key-value store [36]. We report input QPS (top left), request *service times*, i.e., request latencies disregarding queuing (bottom left), queue lengths (top right) and end-to-end *response times* (bottom right). Average server utilization over the course of the trace is 50%. Input QPS fluctuates appreciably over time, but is only weakly correlated with response times. Request service times are fairly stable for `masstree`, with only a few requests being appreciably longer or shorter than the rest. Service times, therefore,

offer no useful hints about end-to-end response times and the right frequency setting. Note, however, that queue lengths are very well correlated with response latency. This trend is not limited to `masstree`. Table 1 shows the Pearson correlation coefficients for response latency with service times, instantaneous QPS, and queue lengths for five latency-critical applications. In each case, response latency is strongly correlated with queue lengths. Its correlation with service times and input QPS, however, is much weaker, and is essentially zero for some applications (e.g., `masstree`).

The reason response latency is so well correlated with queue lengths is that *queuing delay often dominates tail latency*. Fig. 2c plots the tail latency (defined as the 95^{th} percentile of the response time distribution) at various loads. For each application, the tail latency is normalized to the 95^{th}-percentile service latency for that application. In the absence of queuing, the response time distribution would be identical to the service time distribution, and the normalized tail latency would be 1.0. However, Fig. 2c shows that normalized tail latency is significantly higher even at low loads (e.g., `specjbb` at 20% load), indicating that queuing plays a substantial role in determining tail latency. The contribution of queuing delay to tail latency increases rapidly with load; queuing times account for more than half the tail latency for four of our five applications beyond 40% load (Fig. 2c). For applications with fairly uniform service times (e.g. `masstree`, `moses`), response latency is determined almost completely by queuing (Table 1). For applications that exhibit greater variability in service times (e.g., `shore`, `xapian`), both service times and queuing delay determine the response latency (Table 1).

4. RUBIK: FAST DVFS FOR LATENCY-CRITICAL APPLICATIONS

We leverage the insights above to design Rubik. Rubik's statistical model uses request service time distributions, collected online, to account for the uncertainty in compute requirements of individual requests. Rubik then uses these distributions to predict the lowest frequency that does not violate tail latency, given the number of currently-queued requests and their arrival times. Each time a new request arrives or is serviced, Rubik makes a new prediction and changes the core's frequency, as shown in Fig. 3. To make these predictions cheaply, Rubik periodically precomputes two small lookup tables, called the *target tail tables*. These tables encapsulate the beliefs of the statistical model, and are consulted to make each prediction.

We first present Rubik's statistical model, then describe how the target tail tables are computed and used to support lightweight, fine-grain frequency adaptation.

4.1 Fast Analytical Frequency Control

Fig. 4 shows a concrete example of Rubik's operation. The application has received three requests, R0, R1, and R2, which arrived t_0, t_1, and t_2 time ago (Fig. 4a). The application is currently processing R0, and has spent ω cycles doing so; R1 and R2 are queued. For this

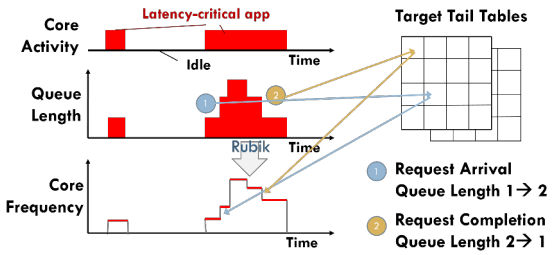


Figure 3: Rubik adjusts core frequency on each request arrival and completion to enforce the tail latency bound.

example, assume that requests are not memory-bound, so performance scales linearly with frequency (we address memory later). Fig. 4a shows that Rubik spans two parallel timelines, measured in time (i.e., seconds) and core cycles. The tail latency bound is given in time, but requests complete after some number of core cycles dictated by their compute requirements. Core frequency (f) connects the two. Our goal is to set f to meet the tail with minimal power.

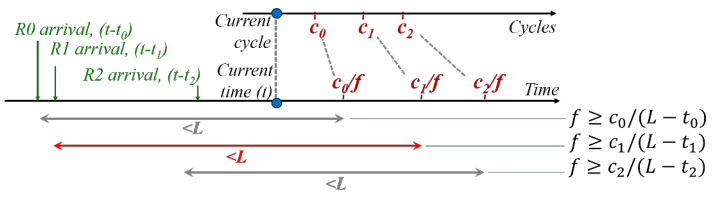
Rubik achieves this by finding the lowest f that satisfies the tail latency bound L for the current requests. This bound is specified as a percentile. In this example, 95% of requests must be served by $L = 2\,\text{ms}$. Rubik exploits the probabilistic nature of the problem as follows. First, Rubik treats the completion *cycle* for each request Ri as a random variable, S_i, with probability distribution $\text{P}[S_i = c]$. Fig. 4b shows these distributions for R0, R1, and R2 (we discuss how to compute them later). Second, Rubik finds the tail completion cycle of each request, c_i (the 95^{th} percentile of each $\text{P}[S_i = c]$), shown in red in Fig. 4b. The timelines in Fig. 4a show how frequency scaling maps each c_i in cycles to c_i/f in time. Request Ri has already spent t_i time in the system, and the 95^{th} percentile will be served by time c_i/f. Satisfying the tail bound for request Ri requires $t_i + c_i/f \leq L$, so to satisfy *all* current requests:

$$f \geq \max_{i=0\ldots N} \frac{c_i}{L - t_i} \qquad (1)$$

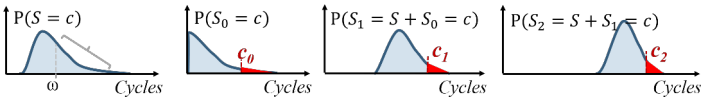
In this example, request R1 has the most stringent constraint—the longest time between t_1 and c_1/f—and sets the frequency. Rubik computes f from Eq. 1 each time a request arrives or completes, quickly adapting to changing conditions. For example, if a new request, R4, were to arrive at the current time, Eq. 1 would be invoked for each of R0-R4.

Notice that the last request in the queue often does not set the frequency, for two main reasons. First, later requests arrived more recently (e.g., $t_N \approx 0$), so they have more headroom than earlier requests (e.g., $t_0 \approx L$). Second, the completion time of queued requests often becomes more tightly distributed the longer the queue length, which shortens the tail of their distributions (e.g., compare $\text{P}[S_1 = c]$ and $\text{P}[S_2 = c]$ in Fig. 4b).

Computing the distributions: Rubik computes the completion cycle distributions by assuming the work for each request is drawn independently from a single distribution, $\text{P}[S = c]$. S gives how many cycles it takes



(a) Cycles/seconds timelines, frequency constraints to meet tail (L).



(b) Probability distributions of service cycles (S), cycles to serve the running request (S_0) and the queued requests (S_1 and S_2), used to compute constraints.

Figure 4: Rubik example with three requests and no memory-bound cycles.

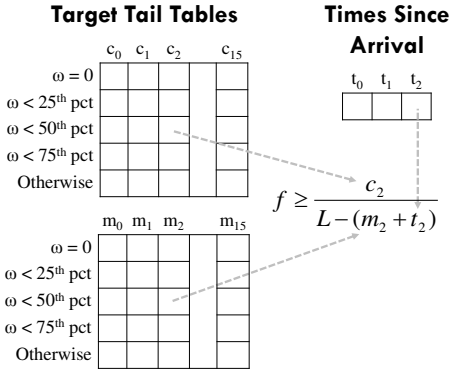


Figure 5: Rubik implementation: Precomputed target tail tables (for core cycles and memory times) make it cheap to compute the frequency constraints.

to process one request, not including queuing time. Independence is a reasonable assumption for datacenter nodes, because each node serves requests from many users, and caches serve repeated requests before they reach leaf nodes, reducing temporal correlations [3].

The completion time distribution of the current request, $\mathrm{P}[S_0 = c]$, is the distribution $\mathrm{P}[S = c]$ conditioned on ω cycles having already elapsed for R0:

$$\mathrm{P}[S_0 = c] = \mathrm{P}[S = c + \omega | S > \omega] = \frac{\mathrm{P}[S = c + \omega]}{\mathrm{P}[S > \omega]}$$

This scales and shifts $\mathrm{P}[S = c]$ at ω, as shown in Fig. 4b.

From S_0 and S, we can derive the completion time distributions of all queued requests. In order to service the i^{th} request, we must first service the $i-1$ requests preceding it. Hence $S_1 = S_0 + S$, and $S_2 = S_1 + S = S_0 + S + S$, and in general $S_i = S_0 + \sum_{j=1}^{i} S$. Using independence, each S_i is distributed according to the *convolution* $(*)$ of S and S_{i-1} [13]. Thus if $\mathrm{P}_X(x) = \mathrm{P}[X = x]$:

$$\mathrm{P}_{S_i} = \mathrm{P}_{S_{i-1}} * \mathrm{P}_S = \mathrm{P}_{S_0} * \overbrace{\mathrm{P}_S * \ldots * \mathrm{P}_S}^{i \text{ times}}$$

Core DVFS and memory: Core frequency does not affect stalls on LLC and main memory accesses, limiting the impact of core DVFS. Rubik therefore uses two probability distributions: per-request *compute cycles*, $\mathrm{P}[C = c]$, and per-request *memory-bound times*, $\mathrm{P}[M = t]$. Work per request (in cycles) is the sum of these two random variables at the current frequency: $S = C + Mf$. Computing the lowest acceptable frequency exactly would require considering the joint distribution of C and M. Instead, Rubik makes the conservative approximation that the tail of S is no better than the combination of the tails of C and M (triangle inequality). For each request, Rubik computes the tails of each distribution C_i and M_i, as discussed above for S_i. This yields tail compute cycles c_i and tail memory time m_i until completion of request Ri. The m_i values are a fixed cost that DVFS cannot affect, so Eq. 1 becomes:

$$f \geq \max_{i=0\ldots N} \frac{c_i}{L - (t_i + m_i)} \qquad (2)$$

4.2 Rubik Implementation

Rubik is implemented as a software runtime and requires minimal hardware support: fast, per-core DVFS, and performance counters for computing CPI stacks.

Estimating probability distributions: Prior work has shown that, using performance counters, one can produce *CPI stacks* that separate compute and memory-bound cycles, even for complex cores with significant memory-level parallelism [11]. CPI stacks have been used to perform memory-aware DVFS with batch apps [10, 26, 42, 47]. Similarly, Rubik uses performance counters to estimate per-request compute and memory-bound cycles, and uses them to produce $\mathrm{P}[C = c]$ and $\mathrm{P}[M = t]$.

Target tail tables: Computing the c_i and m_i percentiles from scratch on each frequency adjustment would be very expensive, but fortunately they can be precomputed. Periodically, the runtime updates the service cycle and time distributions, performs the convolutions, and fills in the c_i and m_i values in the *target tail tables*, as shown in Fig. 5. Each row has the c_i and m_i values for selected quantiles of the service time distribution (quartiles in Fig. 5, octiles in our implementation). On each request arrival and completion, Rubik picks the appropriate row and computes the minimum f as shown in Fig. 5. Computing each constraint requires few instructions (Eq. 2), so updates take negligible time.

Large queues: In theory, the number of queued requests is unbounded. In practice, we rarely observe more than 10 queued requests in our benchmarks; still, large queues could build up with lax tail bounds. Fortunately, by Lyapunov's Central Limit Theorem [4], at large i, $\mathrm{P}[S_i = c]$ converges to a Gaussian distribution with mean $\mathrm{E}[S_0] + i \cdot \mathrm{E}[S]$ and variance of $\mathrm{var}[S]$. By precomputing the tail value for the zero-centered Gaussian with variance of $\mathrm{var}[S]$, each c_i and m_i for large i can be computed by adding the mean. We use this formulation for $i \geq 16$, avoiding long tail tables.

Cost: Every 100 ms in our implementation, Rubik updates the service cycle and time distributions and uses them to compute the target tail tables. We use 128-

Cores	6 x86-64 cores, detailed Westmere-like OOO [50]
L1 caches	32 KB, 4-way set-associative, split D/I, 1-cycle latency
L2 caches	256 KB private per-core, 16-way set-associative, inclusive, 7-cycle latency
L3 cache	6 banks, 12 MBs total, 4-way 52-candidate zcache [48], 20 cycles, inclusive, LRU (Rubik) / Vantage [49] partitioning (RubikColoc)
Coherence protocol	MESI protocol, 64-byte lines, in-cache directory, no silent drops, TSO
Memory	48 GB, 6 DDR3-1066-CL7 channels, unpartitioned (Rubik) / partitioned [43] (RubikColoc), 8.6 GB/s per core
Power	2.4 GHz nominal frequency; Haswell-like FIVR [5] per-core DVFS: 0.8–3.4 GHz frequency range, in 200 MHz steps, 4 μs V/F transition latency; core sleep with L1s & L2 flushed to LLC (Haswell C3 [20]); 65 W TDP

Table 2: Configuration of the simulated CMP.

bucket distributions, and use FFTs to accelerate convolutions. Each update of the target tail tables takes 0.2 ms, resulting in a 0.2% overhead for periodic 100 ms updates. Moreover, updates happen when the application is idle, so they do not impact its performance.

Feedback-based fine-tuning: Rubik as described so far will satisfy the desired tail latency, provided the available frequencies allow it to. However, since its estimates are conservative, it may waste power by using somewhat higher frequencies than needed. To improve efficiency, we use a simple PI controller [55] that observes the difference between the measured and predicted tail latencies over a rolling 1-second window and adjusts Rubik's internal latency target. Adjustments are minor, as the analytical model typically needs little correction.

5. RUBIK EVALUATION

We now evaluate Rubik in simulation and on a real system. We first present exhaustive simulation results evaluating Rubik's power savings at various loads, as well as its responsiveness to load changes. We then present real system results for two representative applications, and show that Rubik saves significant power despite the limitations of current hardware.

5.1 Experimental Methodology

Simulated system: We extend zsim [50] to perform microarchitectural simulation of a 6-core system with parameters shown in Table 2. This configuration is representative of modern high-performance servers [31, 53]. The system supports per-core DVFS and sleep states modeled after Haswell [5, 14]. We conservatively model voltage/frequency transition latencies of 4 μs, longer than the 0.5 μs achievable with FIVR [5].

Power model: We evaluate Rubik's power savings using an accurate power model that we construct with a methodology similar to prior work [9, 18, 30, 54]. We fit the model to a Supermicro 5018D-MTF server with a 4-core Haswell (Xeon E3-1240v3 [20]) and 32GB of unbuffered DDR3-1600 ECC DRAM. We run the full SPEC CPU2006 suite using different numbers of cores and frequencies, and measure key performance counters (instructions, cycles, and memory and cache accesses), per-component power (cores, uncore, DRAM) using RAPL counters [46], and wall-plug power using a WattsUp meter. We sample RAPL counters every 25 ms and the WattsUp meter every 1 s. We perform least-squares regression to train a full-system power model that depends on frequency, voltage, and performance counters. The model reports power for cores, uncore, main memory, and other components (power supply, HDD, etc.). The DVFS range and step of our modeled system (Table 2) match the tested Haswell chip; uncore and DRAM power were scaled to account for the larger LLC and main memory.

We use k-fold cross-validation [29] on SPEC CPU2006 mixes to test model accuracy. On 20,000 25 ms samples of SPEC CPU2006 mixes at different frequencies, the model has 5.1% mean, and 11% worst-case absolute power error; core, uncore, and DRAM have lower errors (1.5% mean, 4% worst-case).

App	Workload configuration	Requests
`xapian`	English Wikipedia, zipfian query popularity	6000
`masstree`	mycsb-a (50% GETs/PUTs), 1.1GB table	9000
`moses`	opensubtitles.org corpora, phrase mode	900
`shore`	TPC-C, 10 warehouses	7500
`specjbb`	1 warehouse	37500

Table 3: Configuration and number of requests for latency-critical applications.

Benchmarks: We use five diverse latency-critical applications, similar to prior work [25]: `xapian`, a web search engine configured as a leaf node [7, 45]; `masstree`, a high-performance key-value store [36]; `moses`, a statistical machine translation system configured to perform real-time translation (e.g., as in Google Translate) [28]; `shore`, an online transaction processing database running TPC-C [23]; and `specjbb`, a Java real-time middleware benchmark. Table 3 shows their input sets and number of simulated requests.

To measure tail latency in simulation, we integrate server and client under the same process. The client produces a request stream with exponentially distributed interarrival times at a given rate (i.e., a Markov input process, common in datacenter workloads [39, 40]). Client overheads are negligible (~150 ns/request).

Tail latency: We define tail latency as the 95^{th}-percentile latency, which is typical [33]. To ensure statistically significant results, we perform enough runs per experiment to achieve 95% confidence intervals below 1%.

5.2 Power Savings

We first evaluate Rubik's power savings for each of our five latency-critical applications. Our baseline scheme, Fixed-frequency, always runs at nominal frequency (2.4 GHz for our simulated system, Table 2). For each application, the latency target is set at the tail latency of the Fixed-frequency scheme at 50% load. We compare Rubik with two oracular schemes: *StaticOracle* and *AdrenalineOracle*. At each load, StaticOracle chooses the lowest

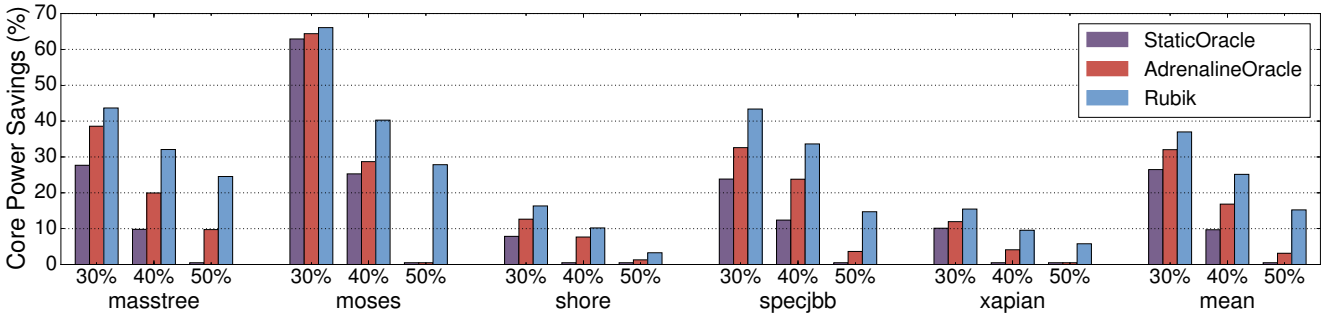
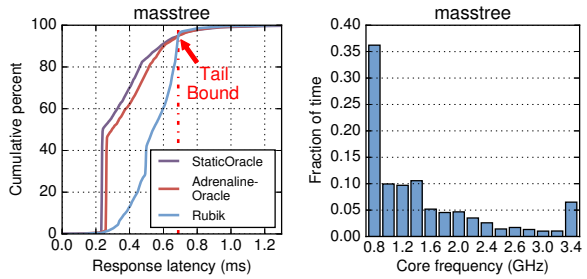
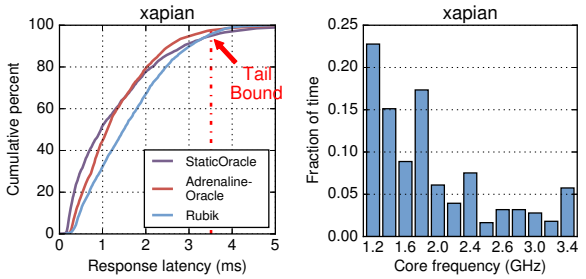


Figure 6: Core power savings for various schemes and applications at 30%, 40% and 50% loads.



(a) Response latency CDF (b) Frequency histogram

Figure 7: Rubik on `masstree`: (a) Rubik serves requests later without degrading tail latency, and (b) Rubik uses low frequencies often to save power.



(a) Response latency CDF (b) Frequency histogram

Figure 8: Rubik on `xapian`: (a) Rubik serves requests later without degrading tail latency, and (b) Rubik uses low frequencies often to save power.

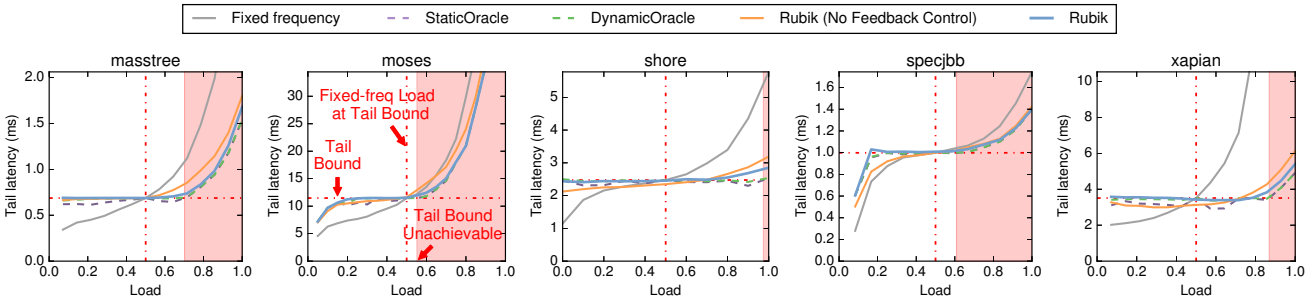
static frequency that satisfies the latency target. StaticOracle is an upper bound on the efficiency of feedback-based controllers such as Pegasus [33]: real controllers need to be more conservative to maintain tail latency (e.g., using guardbands [33]). In fact, StaticOracle is identical to the oracular iso-latency scheme that upper-bounds the power savings from Pegasus [33, Sec. 4.1]. AdrenalineOracle implements an idealized, oracular version of Adrenaline [16]. AdrenalineOracle can perfectly distinguish long requests from short ones (Adrenaline as presented in [16] relies on application-level hints to approximate this). In tuning AdrenalineOracle, we sweep different values of the threshold between long and short requests, and pick the best one. The frequency settings for boosted and non-boosted requests are determined in an offline training phase, and are different for each application and load.

Fig. 6 reports power savings for each application at three loads: 30%, 40%, and 50%. To simplify the discussion, we report active core power only (pipeline, L1s, and L2). At each load, the reported power savings are relative to the power consumed by the Fixed-frequency scheme at that load. All three schemes save significant power at low (30%) load. Rubik performs the best for each application, reducing power by up to 66% (37% average across all applications). While energy savings for StaticOracle and AdrenalineOracle are similar to Rubik's for some applications (e.g., `moses`), Rubik significantly outstrips them for others (e.g., `specjbb`).

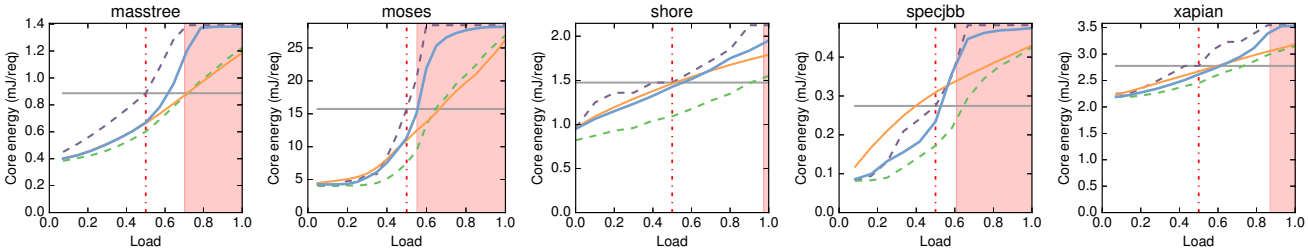
As load increases, Rubik's savings over the other schemes become more pronounced. At 50% load, StaticOracle saves no additional power over Fixed-frequency, while AdrenalineOracle achieves significant power savings for just one application (`masstree`), saving 2% power on average. By contrast, Rubik saves 15% power on average, and up to 28%.

Rubik outperforms StaticOracle and AdrenalineOracle because it is able to respond quickly and accurately to short-term load variations. Rubik runs at low frequency whenever possible, only increasing frequency during periods of significant queuing, where its accurate estimate of queued work allows it to determine the lowest viable frequency. Moreover, since these estimates are highly accurate, Rubik can respond to load changes as early as possible, on each request arrival/completion, further improving energy efficiency. By contrast, StaticOracle must set frequency conservatively to account for transient queuing. AdrenalineOracle mitigates this problem somewhat, implicitly relying on the heuristic that longer requests are more likely to cause queuing. However, since it does not explicitly adapt to queuing, its frequency settings must still be conservative.

We note that Rubik and Adrenaline, of which AdrenalineOracle is an oracular version, are complementary techniques: Rubik focuses on short-term load variations, while Adrenaline leverages knowledge of request classes with varying compute requirements. These approaches could be combined to further improve efficiency.



(a) Load-latency diagrams, showing 95^{th} percentile tail latencies.



(b) Load-energy diagrams, showing core energy per request (includes dynamic and static for pipeline, L1s, and L2).

Figure 9: Tail latencies and core energy per request for each latency-critical application under a fixed frequency, StaticOracle, DynamicOracle, and Rubik. The tail latency at fixed-frequency under 50% load is the tail latency bound for all other schemes. In the shaded areas, load is high enough that no scheme can meet the tail bound.

Rubik on `masstree` and `xapian`: We now characterize Rubik in more detail for two representative apps, `masstree` and `xapian`. Fig. 7a compares the response latency CDFs for Rubik, StaticOracle and AdrenalineOracle on `masstree` at 50% load. All three schemes meet the tail bound. However, Rubik's fine-grain adaptation allows it to push the lower end of the CDF to the right, delaying short requests by running them at low frequencies to save significant power. Indeed, Rubik's frequency histogram (Fig. 7b) shows that most time is spent at low frequencies. By distinguishing between short and long requests, AdrenalineOracle is able to shift the CDF to the right somewhat. However, the shift is much less pronounced than Rubik's, and thus its power savings are lower (Fig. 6). Fig. 8 presents a similar analysis for `xapian`. `xapian` has more variable service times, causing Rubik to choose more conservative frequency settings due to the increased uncertainty. Thus, while the low part of the response latency CDF does shift to the right (Fig. 8a), the shift is less significant. As before, AdrenalineOracle shifts the low part of the CDF to the right by a smaller amount. However, the increased variability for `xapian` necessitates more conservative frequency settings, so that the upper part of the CDF shifts to the *left*, consuming more power than necessary.

5.3 Trace-Driven Characterization

We use trace-driven experiments to gain further insight into Rubik's operation. We capture per-request arrival times, core cycles, memory-bound times, and performance counters in zsim, and replay the trace under different schemes. This setup allows us to compare Rubik against two oracular schemes, *StaticOracle* and *DynamicOracle*. StaticOracle operates as described above. *DynamicOracle* finds the frequency schedule that minimizes power while staying within latency bounds. It first computes, for each request, the lowest frequency that meets the latency bound. Then, it progressively reduces frequencies until 5% of the requests are above the tail bound (if achievable), prioritizing the reductions that save most power.

Fig. 9 shows the tail latency and average core energy per request as a function of load. A load of 100% corresponds to the maximum request rate at nominal frequency (2.4 GHz). Each plot characterizes an app, and each line shows a single scheme. The fixed-frequency results run at nominal frequency, while the oracles and Rubik can use all available frequencies (0.8–3.4 GHz). We show Rubik with and without feedback control. As before, we use the tail latency of the fixed-frequency scheme at 50% load as the target for all other schemes.

Focusing on Fig. 9a, the fixed-frequency results show that tail latency is highly sensitive to load. By contrast, both oracles lower frequencies to match the latency bound, producing a flat tail latency curve until at least 50% load in all apps. The oracles match the latency bound as far as possible beyond 50% load; the region where even the oracles cannot meet tail latency bounds is shaded red in the graphs. Rubik without feedback closely tracks the desired behavior: with loads below 50%, it achieves a near-flat tail, but its conservative approximations produce a slightly lower tail than necessary, especially for `specjbb` which has highly variable service times; at high loads (shaded region), tail latency is slightly above the minimum achievable tail, as set by the oracles. Rubik's feedback controller fixes these small

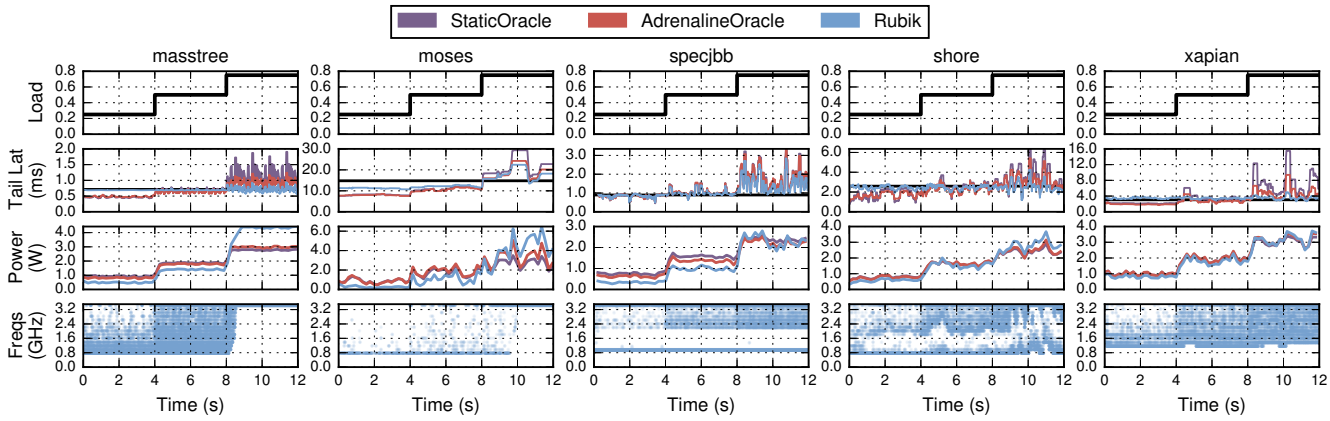


Figure 10: Tail latency and active power consumption for StaticOracle, AdrenalineOracle and Rubik, and frequency distribution for Rubik. For each latency-critical app, input load goes from 25% to 50% at $t = 4s$, and to 75% at $t = 8s$.

deviations and matches the latency curves of the oracles.

Fig. 9b shows the core energy per request of each scheme. At a fixed frequency, active energy per request does not change with load. Below 50% load, the three adaptive schemes reduce frequencies often and lower energy. Comparing StaticOracle and DynamicOracle reveals the benefit of short-term adaptation: across the range, DynamicOracle saves significantly more energy than StaticOracle. At 50% load, DynamicOracle often saves 20–45% of energy consumed by StaticOracle. Rubik outperforms StaticOracle and reaps most of the benefits of DynamicOracle, especially in apps with tightly clustered service times (`masstree`, `moses`). With more variable service times (`shore`, `specjbb`), Rubik saves less power than DynamicOracle because it lacks knowledge of the future and must guard against long requests. Variability increases with load, so Rubik and DynamicOracle achieve larger power savings over StaticOracle as well. Above 50% load, all three schemes often use higher frequencies and more energy to keep the tail latency as close to the target as possible. Rubik continues to use less power than StaticOracle even at these high loads.

5.4 Responsiveness to Load Changes

In addition to increased power savings in steady state (Fig. 6), Rubik's design allows it to respond near-instantaneously to sudden changes in input load. This happens because higher loads immediately translate to longer queues, and *Rubik reacts to changes in queue length immediately*. Note that the target tail tables depend on the service-time distributions, not on queue lengths. Thus, a sudden change in load makes Rubik immediately shift to higher frequencies without any tuning. By contrast, Pegasus reacts to load changes when it measures a higher tail latency [33], which takes time and hampers its responsiveness; and Adrenaline must both accurately detect a change in input load and go through a dynamic search phase to select new frequencies [16].

Fig. 10 shows how Rubik, StaticOracle, and AdrenalineOracle respond to sudden load changes. The top row shows the input load, which grows in steps over a span of 12 seconds, going from 25% to 50% and eventually to 75%. As in Sec. 5.2, the latency bound is set at the tail latency achieved when running at the nominal frequency at 50% load. The second row shows the 95^{th} percentile latency and the third row shows the active power consumption for the three schemes, both over a rolling 200 ms window. The bottom row shows Rubik's frequency settings over time.

Both StaticOracle and AdrenalineOracle are slow to respond. At 25% load, they use overly aggressive frequency settings, yielding unnecessarily low tail latencies and wasting power. On the other hand, as load exceeds 50%, both schemes choose frequency settings that are *too low*, suffering large latency violations.

By contrast, Rubik achieves stable tail latencies when the load is 50% or lower. At low load (25%), Rubik chooses lower frequencies to achieve tail latency close to the latency bound, saving additional power. Rubik adapts quickly when load changes, choosing progressively higher frequencies as load increases. Beyond 50% load, Rubik chooses higher frequencies to meet tail latency requirements as best as possible. Even at 75% load, Rubik suffers minimal latency degradation for `masstree` and `xapian`. Even for apps for which latency violation at 75% load is inevitable (e.g., `shore`), Rubik achieves the lowest latency degradation of the three schemes.

5.5 Real-System Evaluation

We implement and evaluate Rubik on a real system and explore the limitations of current hardware. We use the same 4-core Haswell server used to train our power model (Sec. 5.1). This chip has integrated voltage regulators (FIVR [5]), but lacks per-core DVFS, so we use a single core and turn the rest off. Applications are less memory-bound than on the simulated system, since they have the full 8 MB LLC to themselves.

Despite the ∼500 ns voltage transition latencies advertised by FIVR [5], we observe transition latencies of up to 130 μs, even when setting frequencies by directly writing MSRs. We conjecture that this delay is due to the Power Control Unit [5], the internal microcon-

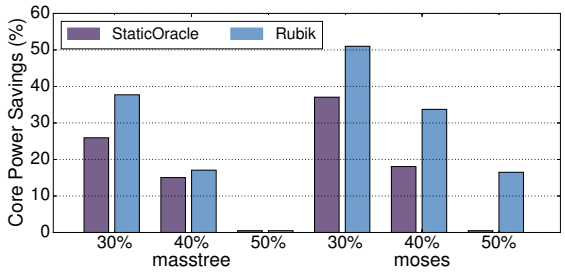


Figure 11: Core power savings for Rubik and StaticOracle on a real system.

troller that manages FIVR, for three reasons. First, transition times are highly variable, and uncorrelated with either the magnitude or direction of the frequency change. Second, we also tested a Broadwell system (an 8-core Xeon D-1540) with second-generation FIVR, and found transition times were about 4× *higher* and similarly variable. Third, MSR write latencies are small (1.2 μs). High DVFS transition latencies limit Rubik's gains somewhat. We hope that this and other recent work that requires fast DVFS [12, 16] will motivate the adoption of low-latency DVFS interfaces.

We implement Rubik as described in Sec. 4.2. Our implementation is entirely in userspace. Request arrivals are handled by a separate *interrupt thread*, scheduled on a hyperthread that shares the same physical core as the application. This thread dequeues network requests and changes core frequency in response if needed. The interrupt thread has negligible CPU utilization even at high loads. We chose this design to avoid kernel modifications; the interrupt thread could be avoided with a few new kernel interrupts. We use the RAPL interface [19] to measure core power, and set the nominal frequency to 2.4 GHz as in our simulations.

Since long DVFS transition latencies have a larger impact with short requests, we focus on the applications with the shortest (`masstree`) and longest (`moses`) tail latency targets (730 μs and 5.54 ms, respectively). Fig. 11 reports the core power savings over the fixed-frequency scheme achieved by Rubik and StaticOracle at various loads on both applications. Rubik meets the tail latency requirement in all cases (not shown). We were somewhat surprised by Rubik's resilience to DVFS lag, especially for `masstree`, which has very short requests (median service time 240 μs). The lag does diminish Rubik's power efficiency for `masstree` somewhat: since frequency changes take longer, Rubik is forced to take more conservative decisions to avoid degrading latency, particularly at high loads where frequent request arrivals force rapid frequency changes. Thus, while Rubik saves significant power over StaticOracle for `masstree` at low (30%) load, these additional savings diminish as load increases, and Rubik and StaticOracle perform identically at high (50%) load. For `moses`, which has longer requests (median service time 3.95 ms), Rubik saves significant power over StaticOracle even at high load. Note that both StaticOracle and Rubik achieve somewhat lower power savings than the simulated system on `moses`. This is because the server's larger LLC makes `moses` more compute-bound and causes more variable service times, so both StaticOracle and Rubik set higher frequencies. Nevertheless, Rubik saves substantial power: 51% at low load, and 17% at high load.

6. RUBIKCOLOC: COLOCATING BATCH AND LATENCY-CRITICAL WORK

While Rubik substantially reduces dynamic core power, it does not reduce idle power. Even when servers are fully idle, resources such as disks, network cards and DRAM consume a significant amount of power [1, 38, 41]. This limits the efficiency gains from any DVFS scheme. Fig. 12 shows the total system power savings achieved by Rubik at 30% load. These savings, while substantial, are modest relative to the savings in core power (Fig. 6).

An attractive way to reduce idle power is to colocate latency-critical and batch applications in the same server. Current datacenters execute a significant amount of batch work (e.g., analytics) that only requires high long-term throughput. In principle, batch applications could run when latency-critical applications are idle, amortizing idle power over a much larger amount of work and reducing the number of datacenter servers. However, this is not possible in conventional systems because colocated applications contend on shared resources, causing large tail latency degradations [8, 25, 37, 59]. As a result, current datacenters often segregate latency-critical and batch applications to avoid interference, and suffer from high idle power.

We observe that the key challenge in colocating batch and latency-critical work is the same as for DVFS: uncertainty. While in DVFS this uncertainty arises from the unknown request lengths and arrival times of the latency-critical workload, in colocation this uncertainty is due to the unknown performance degradation caused by interference. Solutions to both problems should be complementary. Indeed, we now show that Rubik can be used to perform more aggressive colocation than prior work.

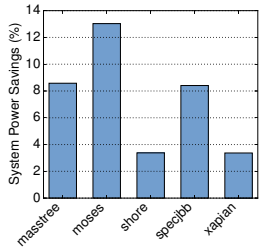
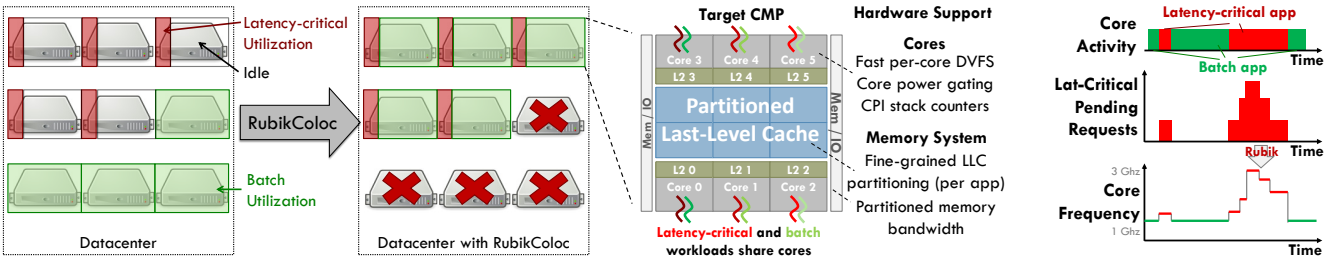


Figure 12: Rubik's full-system power savings at 30% load.

Prior colocation schemes allow latency-critical and batch applications to share *memory system resources* (e.g., last level caches, DRAM bandwidth), using either memory system partitioning [25] or conservative coscheduling [37, 59] to mitigate performance degradation in latency-critical applications. However, while these schemes share memory system resources, *they do not share cores*. This limits overall utilization. For example, if half of the cores are dedicated to latency-critical work that runs at 10% load, overall core utilization cannot exceed 55%. It would be more efficient to time-multiplex latency-critical and batch applications on cores, having batch applications run when latency-critical applications are idle.



(a) RubikColoc in the datacenter (b) System architecture (c) Sharing-aware DVFS

Figure 13: RubikColoc uses fine-grain DVFS to colocate batch and latency-critical apps without degrading tail latency. (a) By increasing server utilization and reducing power, RubikColoc improves datacenter efficiency and reduces provisioned servers. (b) RubikColoc requires modest hardware extensions over commodity systems. (c) RubikColoc adjusts core frequency on each request arrival and completion to enforce the tail latency bound.

Unfortunately, sharing cores introduces uncertainty that is hard to model: core microarchitectural state (branch predictor, TLBs, L1s, L2, etc.) becomes a shared resource and causes interference. Fortunately, core state is small enough that DVFS can compensate for its inertia. For example, private caches can be refilled from a "warm" LLC in microseconds, compared to tens of milliseconds needed to refill the LLC from main memory [25]. Other state has similarly low inertia. Thus, *given a warm LLC partition*, judicious DVFS can maintain tail latency.

We use this insight to propose *RubikColoc*, a scheme that builds on Rubik to enable sharing of cores between latency-critical and batch applications (Fig. 13b). RubikColoc seeks to maximize the throughput per watt (TPW) of batch applications without degrading tail latency for latency-critical applications. RubikColoc partitions shared memory system resources (LLC capacity and memory bandwidth) among latency-critical and batch apps (as in prior work [25,43]) to avoid interference in these resources, and uses Rubik to mitigate contention in core microarchitectural state. RubikColoc prioritizes latency-critical apps over batch apps: latency-critical apps run whenever they have pending requests, and yield the core to batch apps when idle (Fig. 13c). Rubik sets the core frequency for latency-critical apps, while batch apps run at the frequency that maximizes their TPW. This allows batch apps to safely utilize spare core cycles on latency-critical servers. RubikColoc significantly reduces datacenter power consumption while using fewer machines than traditional segregated datacenters.

7. RUBIKCOLOC EVALUATION

Experimental setup: To evaluate RubikColoc, we first consider a baseline datacenter that segregates batch and latency-critical apps. As shown in Fig. 14, this datacenter has 1000 servers that run the 5 latency-critical apps, with 200 servers dedicated to each app. Each latency-critical server runs 6 copies of the app at nominal frequency. This datacenter also has 1000 servers running batch work. We produce 20 mixes of six randomly chosen SPEC CPU2006 apps, and dedicate 50 servers to each mix. Each batch app is fast-forwarded 5 billion

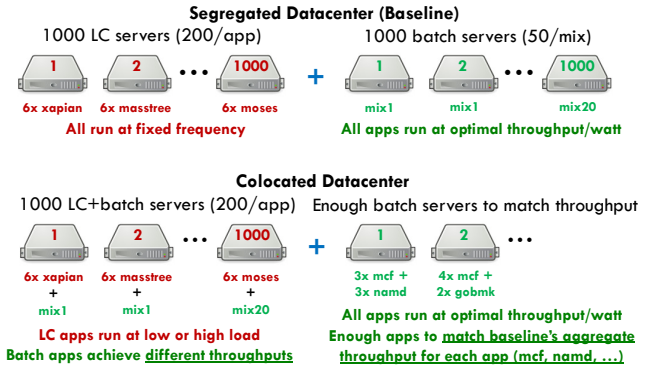


Figure 14: Experimental setup used to compare colocation schemes.

instructions, is simulated for 400 million instructions, and is then restarted; the mix finishes when all apps have restarted at least five times. Each batch app runs at its optimal throughput per watt; because all servers use a partitioned memory system, the optimal frequency for each batch app does not depend on the apps it is colocated with (batch apps do not run above the nominal frequency to stay within the TDP).

We then evaluate a datacenter that uses RubikColoc for colocation, as shown in Fig. 14. First, each latency-critical server now also runs the mix from the corresponding batch server of the segregated datacenter, becoming a colocated server. Mixes are interleaved so that each latency-critical app is co-scheduled with all batch mixes equally. Second, because each app in the batch mixes gets less throughput when colocated, we provision a variable number of batch-only servers and run additional copies of each batch app to match the throughput of the segregated datacenter for each batch app.

This experiment is carefully designed to have three desirable properties: it is *fixed-work* [15] (both RubikColoc and the baseline segregated datacenter run matching batch and latency-critical work), it allows comparing *end-to-end metrics* (tail latencies, datacenter power, and servers used), and, by interleaving mixes, it exposes each latency-critical app to all batch apps. We do not claim this is the best approach to manage datacenters with

latency-critical/batch mixes; it is just a controlled and fair way to evaluate the benefits of RubikColoc.

Colocation schemes: In addition to RubikColoc, we evaluate three other colocation schemes: *StaticColoc*, *HW-T*, and *HW-TPW*. StaticColoc runs latency-critical apps at the frequency determined by StaticOracle and batch apps at their optimal TPW. HW-T and HW-TPW are hardware-controlled schemes that perform coordinated per-core DVFS: HW-T sets frequencies to maximize aggregate system throughput (IPC) while staying below TDP; HW-TPW maximizes aggregate throughput per watt. These schemes adapt every 100 µs, and represent hardware-controlled DVFS schemes typical of modern chips (e.g., Turbo Boost [34, 46]). All schemes use a partitioned memory system.

7.1 Impact of Colocation on Tail Latencies

Fig. 15 shows the distribution of tail latencies achieved by different colocation schemes when latency-critical apps run at a load of 60%. Each line represents a single scheme, and the x-axis represents the $5 \times 20 = 100$ latency-critical/batch mixes in the 1000 colocated servers. For each scheme, mixes are sorted from highest to lowest tail latency, relative to the tail bound (lower is better). HW-T and HW-TPW grossly violate tail latencies, suffering degradations of up to 8.2× and 3.2×, respectively. These hardware-managed DVFS schemes are oblivious to the requirements of each application, so they introduce interference among apps and are not suitable for colocation. Even StaticColoc degrades tail latency for 40% of the mixes (by up to 42%) due to interference in core microarchitectural state. By contrast, RubikColoc maintains tail latency across all mixes, making up for core sharing by automatically using higher frequencies when needed. Thus, while naive colocation can violate latency bounds, Rubik's fine-grain DVFS allows RubikColoc to share cores safely.

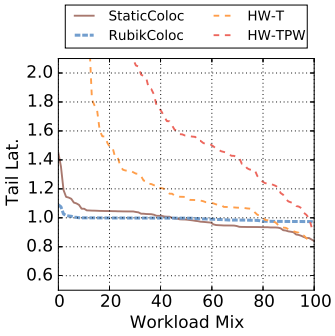


Figure 15: Distributions of tail latency, relative to the baseline, at 60% load (lower is better).

7.2 Efficiency Gains from Colocation

We now evaluate the efficiency gains from colocation by comparing two datacenters: a colocated datacenter managed by RubikColoc, and a segregated datacenter, representative of non-colocating schemes like Pegasus [33], where the batch apps execute at their optimal throughput-per-watt and StaticOracle sets the frequencies of latency-critical apps (Sec. 5.2). We sweep the latency-critical load from 10% to 60% to model diurnal variations [1, 33]. Fig. 16 reports total datacenter power consumption and number of servers used for both schemes, normalized to the values of the segregated datacenter at 60% load. For each metric, the contribution from batch servers is hatched.

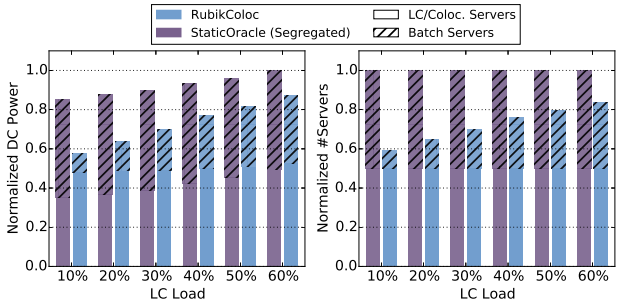


Figure 16: Datacenter power and number of servers for segregated and colocated datacenters as the latency-critical load changes.

For the segregated datacenter, StaticOracle uses lower frequencies at low loads, reducing dynamic power consumption on latency-critical servers by over 90% as load decreases from 60% to 10%. This corresponds to only a 33% reduction in total power for these servers, since idle power remains significant. Moreover, since the power consumed by batch servers remains unchanged, the reduction in total datacenter power is only 17%.

By contrast, RubikColoc migrates more batch work to colocated servers as latency-critical load decreases, using fewer batch servers and reducing static power. Even at high (60%) load, colocation confers significant efficiency gains (17% reduction in datacenter power, 19% fewer servers than the segregated datacenter). These savings are even more pronounced at lower loads; at 10% load, RubikColoc uses 43% less power and 41% fewer servers than the baseline datacenter (segregated, 60% load). Relative to the segregated datacenter at the same load (10%), RubikColoc reduces datacenter power consumption by 31% while using 41% fewer servers.

8. CONCLUSIONS

We have presented Rubik, an analytical, fine-grain power management technique that reduces active core power consumption by up to 66%. Rubik uses a novel statistical model to account for uncertainty in the behavior of latency-critical applications, and does not require application-specific heuristics. We use Rubik to design RubikColoc, a scheme that allows aggressive colocation of latency-critical and batch apps without degrading tail latency. RubikColoc reduces datacenter power consumption by up to 31% while using 41% fewer machines than conventional, segregated datacenters.

9. ACKNOWLEDGMENTS

We thank Christina Delimitrou, Mark Jeffrey, Webb Horn, Anurag Mukkara, Suvinay Subramanian, Guowei Zhang, and the anonymous reviewers for their helpful feedback. This work was supported in part by NSF grant CCF-1318384 and a Google research award. Davide B. Bartolini was supported in part by a Roberto Rocca Doctoral Fellowship.

10. REFERENCES

[1] L. Barroso and U. Hölzle, "The case for energy-proportional computing," *IEEE Computer*, 40(12), 2007.

[2] L. Barroso, J. Clidaras, and U. Hölzle, *The datacenter as a computer*, 2nd ed. Morgan & Claypool, 2013.

[3] L. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The Google cluster architecture," *IEEE Micro*, 23(2), 2003.

[4] P. Billingsley, *Probability and measure*. Wiley, 2008.

[5] E. Burton, G. Schrom, F. Paillet *et al.*, "FIVR: Fully integrated voltage regulators on 4th generation Intel Core SoCs," in *APEC-29*, 2014.

[6] R. Cochran, C. Hankendi, A. Coskun *et al.*, "Pack & Cap: Adaptive DVFS and thread packing under power caps," in *MICRO-44*, 2011.

[7] J. Dean and L. Barroso, "The tail at scale," *Comm. ACM*, 56(2), 2013.

[8] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and QoS-aware cluster management," in *ASPLOS-XIX*, 2014.

[9] Q. Deng, D. Meisner, A. Bhattacharjee *et al.*, "CoScale: Coordinating CPU and memory system DVFS in server systems," in *MICRO-45*, 2012.

[10] S. Eyerman and L. Eeckhout, "A counter architecture for online DVFS profitability estimation," *IEEE Trans. on Computers*, 59(11), 2010.

[11] S. Eyerman, L. Eeckhout, T. Karkhanis *et al.*, "A performance counter architecture for computing accurate CPI components," in *ASPLOS-XII*, 2006.

[12] W. Godycki, C. Torng, I. Bukreyev *et al.*, "Enabling realistic fine-grain voltage scaling with reconfigurable power distribution networks," in *MICRO-47*, 2014.

[13] C. Grinstead and J. Snell, *Introduction to probability*. American Mathematical Soc., 1998.

[14] P. Hammarlund, A. Martinez, A. Bajwa *et al.*, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, 34(2), 2014.

[15] A. Hilton, N. Eswaran, and A. Roth, "FIESTA: A sample-balanced multi-program workload methodology," in *MoBS*, 2009.

[16] C.-H. Hsu, Y. Zhang, M. Laurenzano *et al.*, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *HPCA-21*, 2015.

[17] M. Huang, J. Renau, S.-M. Yoo *et al.*, "A framework for dynamic energy efficiency and temperature management," in *MICRO-33*, 2000.

[18] W. Huang, C. Lefurgy, W. Kuk *et al.*, "Accurate fine-grained processor power proxies," in *MICRO-45*, 2012.

[19] *Intel 64 and IA-32 architectures software developer's manual*, Intel, 2014.

[20] *Intel Xeon processor E3-1200 v3 product family datasheet*, Intel, 2014.

[21] C. Isci, A. Buyuktosunoglu, C.-Y. Cher *et al.*, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *MICRO-39*, 2006.

[22] M. Jeon, Y. He, S. Elnikety *et al.*, "Adaptive parallelism for web search," in *EuroSys*, 2013.

[23] R. Johnson, I. Pandis, N. Hardavellas *et al.*, "Shore-MT: A scalable storage manager for the multicore era," in *EDBT*, 2009.

[24] S. Kanev, K. Hazelwood, G.-Y. Wei *et al.*, "Tradeoffs between power management and tail latency in warehouse-scale applications," in *IISWC*, 2014.

[25] H. Kasture and D. Sanchez, "Ubik: Efficient cache sharing with strict QoS for latency-critical workloads," in *ASPLOS-XIX*, 2014.

[26] G. Keramidas, V. Spiliopoulos, and S. Kaxiras, "Interval-based models for run-time DVFS orchestration in superscalar processors," in *CF*, 2010.

[27] W. Kim, M. S. Gupta, G.-Y. Wei *et al.*, "System level analysis of fast, per-core DVFS using on-chip switching regulators," in *HPCA-14*, 2008.

[28] P. Koehn, H. Hoang, A. Birch *et al.*, "Moses: Open source toolkit for statistical machine translation," in *ACL-45*, 2007.

[29] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *IJCAI*, 1995.

[30] K. Koukos, D. Black-Schaffer, V. Spiliopoulos *et al.*, "Towards more efficient execution: A decoupled access-execute approach," in *ICS'13*, 2013.

[31] N. Kurd, S. Bhamidipati, C. Mozak *et al.*, "Westmere: A family of 32nm IA processors," in *ISSCC*, 2010.

[32] J. Li and J. Martínez, "Dynamic power-performance adaptation of parallel computation on chip multiprocessors," in *HPCA-12*, 2006.

[33] D. Lo, L. Cheng, R. Govindaraju *et al.*, "Towards energy proportionality for large-scale latency-critical workloads," in *ISCA-41*, 2014.

[34] D. Lo and C. Kozyrakis, "Dynamic management of TurboMode in modern multi-core chips," in *HPCA-20*, 2014.

[35] J. Lorch and A. Smith, "Improving dynamic voltage scaling algorithms with PACE," *SIGMETRICS PER*, 29(1), 2001.

[36] Y. Mao, E. Kohler, and R. Morris, "Cache craftiness for fast multicore key-value storage," in *EuroSys*, 2012.

[37] J. Mars, L. Tang, R. Hundt *et al.*, "Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *MICRO-44*, 2011.

[38] D. Meisner, B. Gold, and T. Wenisch, "The PowerNap server architecture," in *ASPLOS-XIV*, 2009.

[39] D. Meisner, C. Sadler, L. Barroso *et al.*, "Power management of online data-intensive services," in *ISCA-38*, 2011.

[40] D. Meisner and T. Wenisch, "Stochastic queuing simulation for data center workloads," *EXERT*, 2010.

[41] D. Meisner and T. Wenisch, "DreamWeaver: Architectural support for deep sleep," in *ASPLOS-XVII*, 2012.

[42] R. Miftakhutdinov, E. Ebrahimi, and Y. Patt, "Predicting performance impact of DVFS for realistic memory systems," in *MICRO-45*, 2012.

[43] S. Muralidhara, L. Subramanian, O. Mutlu *et al.*, "Reducing memory interference in multicore systems via application-aware memory channel partitioning," in *MICRO-44*, 2011.

[44] N. Pinckney, M. Fojtik, B. Giridhar *et al.*, "Shortstop: An on-chip fast supply boosting technique," in *VLSI*, 2013.

[45] V. Reddi, B. Lee, T. Chilimbi *et al.*, "Web search using mobile cores: quantifying and mitigating the price of efficiency," in *ISCA-37*, 2010.

[46] E. Rotem, A. Naveh, A. Ananthakrishnan *et al.*, "Power-management architecture of the Intel microarchitecture code-named Sandy Bridge," *IEEE Micro*, 32(2), 2012.

[47] B. Rountree, D. Lowenthal, M. Schulz *et al.*, "Practical performance prediction under dynamic voltage frequency scaling," in *IGCC*, 2011.

[48] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling ways and associativity," in *MICRO-43*, 2010.

[49] D. Sanchez and C. Kozyrakis, "Vantage: Scalable and efficient fine-grain cache partitioning," in *ISCA-38*, 2011.

[50] D. Sanchez and C. Kozyrakis, "ZSim: Fast and accurate microarchitectural simulation of thousand-core systems," in *ISCA-40*, 2013.

[51] H. Sasaki, S. Imamura, and K. Inoue, "Coordinated power-performance optimization in manycores," in *PACT-22*, 2013.

[52] E. Schurman and J. Brutlag, "The user and business impact of server delays, additional bytes, and HTTP chunking in web search," in *Velocity*, 2009.

[53] B. Sinharoy, R. Kalla, W. Starke *et al.*, "IBM POWER7 multicore server processor," *IBM J. Res. Dev.*, 55(3), 2011.

[54] V. Spiliopoulos, A. Sembrant, and S. Kaxiras, "Power-sleuth: A tool for investigating your program's power behavior," in *MASCOTS-20*, 2012.

[55] J. Teo, J. P. How, and E. Lavretsky, "Proportional-integral controllers for minimum-phase nonaffine-in-control systems," *IEEE Trans. Autom. Control*, 55(6), 2010.

[56] A. Vulimiri, P. B. Godfrey, R. Mittal *et al.*, "Low latency via redundancy," in *CoNEXT*, 2013.

[57] D. Wong and M. Annavaram, "KnightShift: Scaling the energy proportionality wall through server-level heterogeneity," in *MICRO-45*, 2012.

[58] R. Xu, C. Xi, R. Melhem *et al.*, "Practical PACE for embedded systems," in *EMSOFT*, 2004.

[59] H. Yang, A. Breslow, J. Mars *et al.*, "Bubble-Flux: Precise online QoS management for increased utilization in warehouse scale computers," in *ISCA-40*, 2013.

[60] W. Yuan and K. Nahrstedt, "Energy-efficient soft real-time CPU scheduling for mobile multimedia systems," in *SOSP-19*, 2003.