# Energy Proportionality and Workload Consolidation for Latency-critical Applications

George Prekas[1], Mia Primorac[1], Adam Belay[2], Christos Kozyrakis[2], Edouard Bugnion[1]
[1]EPFL [2]Stanford

## Abstract

Energy proportionality and workload consolidation are important objectives towards increasing efficiency in large-scale datacenters. Our work focuses on achieving these goals in the presence of applications with µs-scale tail latency requirements. Such applications represent a growing subset of datacenter workloads and are typically deployed on dedicated servers, which is the simplest way to ensure low tail latency across all loads. Unfortunately, it also leads to low energy efficiency and low resource utilization during the frequent periods of medium or low load.

We present the OS mechanisms and dynamic control needed to adjust core allocation and voltage/frequency settings based on the measured delays for latency-critical workloads. This allows for energy proportionality and frees the maximum amount of resources per server for other background applications, while respecting service-level objectives. Monitoring hardware queue depths allows us to detect increases in queuing latencies. Carefully coordinated adjustments to the NIC's packet redirection table enable us to reassign flow groups between the threads of a latency-critical application in milliseconds without dropping or reordering packets. We compare the efficiency of our solution to the Pareto-optimal frontier of 224 distinct static configurations. Dynamic resource control saves 44%–54% of processor energy, which corresponds to 85%–93% of the Pareto-optimal upper bound. Dynamic resource control also allows background jobs to run at 32%–46% of their standalone throughput, which corresponds to 82%–92% of the Pareto bound.

***Categories and Subject Descriptors*** D.4 [*OS*]: Scheduling

***General Terms*** Design, Performance

***Keywords*** Latency-critical applications, Microsecond-scale computing, Energy proportionality, Workload consolidation

## 1. Introduction

With the waste in the power delivery and cooling systems largely eliminated [4], researchers are now focusing on the efficient use of the tens of thousands of servers in large-scale datacenters. The first goal is *energy proportionality*, which minimizes the energy consumed to deliver a particular workload [3, 29]. Hardware enhancements, primarily in dynamic voltage/frequency scaling (DVFS) and idle modes in modern processors [22, 45] provide a foundation for energy proportionality. The second goal is *workload consolidation*, which raises server utilization and minimizes the number of servers needed for a particular set of workloads [14, 46, 50]. Advances in cluster management [11, 32] and server consolidation using virtual machines or container systems have also played an important role in datacenter efficiency by enabling multiple workloads to be consolidated on each machine.

The two goals map to distinct economic objectives: energy proportionality reduces operational expenses (*opex*), whereas workload consolidation reduces capital expenses (*capex*). Since capital costs often dominate the datacenter's total cost of ownership (*TCO*), consolidation is highly desirable. Nevertheless it is not always possible, e.g., when one application consumes the entirety of a given resource, e.g., memory. In such cases, energy proportionality is a necessity.

Datacenter applications have evolved with the advent of web-scale applications. User-facing, large-scale applications now rely extensively on high fan-out patterns between low-latency services [9]. Such services exhibit low per-request latencies (a handful of µs for a key-value store), have strict service-level objectives (SLO, e.g. $< 500\mu s$ at the 99th percentile), and must sustain massive request rates and high client fan-in connections. Although functionally simple, these services are deployed on thousands of servers of large-scale datacenters. Because of their importance, there are several proposals to improve their throughput or latency using user-level networking stacks [20, 31], networking toolkits [16, 18, 44], or dataplane operating systems [6, 43].

*SoCC '15, August 27-29, 2015*, Kohala Coast, HI, USA.
© 2015 ACM. ISBN 978-1-4503-3651-2/15/08…$15.00.
DOI: http://dx.doi.org/10.1145/2806777.2806848

Such latency-critical services are challenging to run in a shared infrastructure environment. They are particularly sensitive to resource allocation and frequency settings, and they suffer frequent tail latency violations when common power management or consolidation approaches are used [26, 27]. As a result, operators typically deploy them on dedicated servers running in polling mode, forgoing opportunities for workload consolidation and for energy-proportional computing at below-peak utilization levels.

To gain a principled understanding of the challenges for resource management in the presence of latency-critical services, we performed an exhaustive analysis of static configurations for a latency-critical service (memcached [35]) running on a modern server. We explored up to 224 possible settings for core allocation, use of hyperthreads, DVFS frequencies, and Turbo Boost. While our experiments use a single application, the implications have broad applicability because memcached has aggressive latency requirements, short service times, and a large number of independent clients that are common among many latency-critical applications. Our experiments reveal that there is an inherent tradeoff for any given static configuration between the maximum throughput and the overall efficiency when operating below peak load. Furthermore, the experiments reveal a Pareto-optimal frontier [41] in the efficiency of static configurations at any given load level, which allows for close to linear improvements in energy-proportionality and workload consolidation factors.

Based on these insights, we introduce a set of novel dynamic resource management mechanisms and policies that improve energy proportionality and workload consolidation in the presence of latency-sensitive applications. We integrate these techniques in IX, a state-of-the-art dataplane operating system that optimizes both throughput and latency for latency-critical workloads [6].

Fig. 1 illustrates our approach: the dynamic controller (ixcp) adjusts the number of cores allocated to a latency-sensitive application running on top of IX and the DVFS settings for these cores. The remaining cores can be placed in idle modes to reduce power consumption or can be safely used to run background tasks. The controller builds upon two key mechanisms. The first mechanism detects backlog and increases in queuing delays that exceed the allowable upper bound for the specific latency-critical application. It monitors CPU utilization and signals required adjustments in resource allocation. The second mechanism quickly migrates both network and application processing between cores transparently and without dropping or reordering packets.

We evaluated our system with two control policies that optimize for energy proportionality and workload consolidation, respectively. A policy determines how resources (cores, hyperthreads, and DVFS settings) are adjusted to reduce underutilization or to restore violated SLO. The two policies are derived from the exhaustive analysis of the 224 static configurations. For the platform studied (a Xeon E5-2665), we conclude that: for *best energy proportionality*, (i) we start with the lowest clock rate and allocate additional cores to the latency-critical task as its load grows, using at first only one hyperthread per core; (ii) we enable the second hyperthread only when all cores are in use; and finally (iii) we increase the clock rate for the cores running the latency-critical task. For *best consolidation*, (i) we start at the nominal clock rate and add cores with both hyperthreads enabled as load increases; and (ii) finally enable Turbo Boost as a last resort.

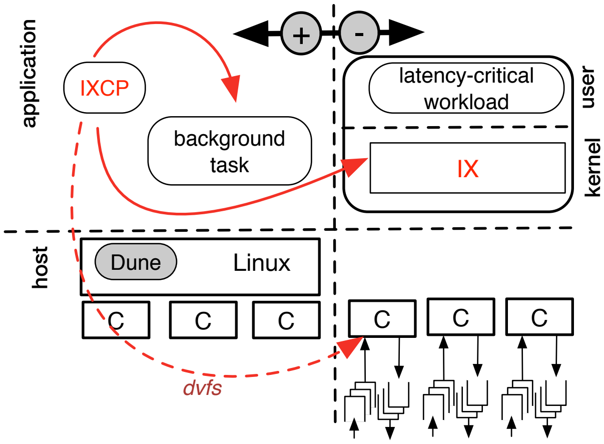


Figure 1: Dynamic resource controls with IX for a workload consolidation scenario with a latency-sensitive application (e.g., memcached) and a background batch task (e.g., analytics). The controller, ixcp, partitions cores among the applications and adjusts the processor's DVFS settings.

This paper makes the following contributions:

▷ We develop techniques for fine-grain resource management for latency-critical workloads. This includes mechanisms for detection of load changes in sub-second timescales and for rebalancing flow-groups between cores without causing packet drops or reordered deliveries. In our experiments, this mechanism completes in less than 2 ms 95% of the time, and in at most 3.5 ms.

▷ We provide a methodology that uses the Pareto frontier of a set of static configurations to derive resource allocation policies. We derive two policies for memcached that respect the SLO constraints of a latency-critical in-memory key-value store. These policies lead to 44%–54% energy savings for a variety of load patterns, and enable workload consolidation with background jobs executing at 32%–46% of their peak throughput on a standalone machine. These gains are close to the Pareto-optimal bounds for the server used, within 88% and 82%–92% respectively.

▷ We demonstrate that precise and fine-grain control of cores, hyperthreads, and DVFS has a huge impact on both energy proportionality and workload consolidation, especially when load is highly variable. DVFS is a necessary but insufficient mechanism to control latency-critical applications that rely on polling.

The rest of this paper is organized as follows. §2 describes the experimental setup. §3 characterizes resource usage using static configurations and derives insights for dynamic allocation policies. §4 describes the design and implementation of the mechanisms and policies for dynamic resource management in IX. §5 evaluates the impact of the dynamic controller for energy proportionality and workload consolidation. We discuss insights and open issues in §6, related work in §7, and conclude in §8.

## 2. Workloads and Experimental Setup

Our experimental setup consists of a cluster of 11 clients and one server connected by a low-latency 10 GbE switch. 10 clients generate load, and the 11th measures latency of requests to the latency-sensitive application. The client machines are a mix of Xeon E5-2637 @ 3.5 Ghz and Xeon E5-2650 @ 2.6 Ghz. The server is a Xeon E5-2665 @ 2.4 Ghz with 256 GB of DRAM, typical in datacenters. Each client and server socket has 8 cores and 16 hyperthreads. All machines are configured with one Intel x520 10 GbE NIC (82599EB chipset). Our baseline configuration in each machine is an Ubuntu LTS 14.0.4 distribution, updated to the 3.16.1 Linux kernel. Although the server has two sockets, the foreground and background applications run on a single processor to avoid any NUMA effects. We run the control plane on the otherwise empty second socket, but do not account for its energy draw.

Our primary foreground workload is `memcached`, a widely deployed, in-memory, key-value store built on top of the `libevent` framework [35]. It is frequently used as a high-throughput, low-latency caching tier in front of persistent database servers. `memcached` is a network-bound application, with threads spending over 75% of execution time in kernel mode for network processing when using Linux [26]. It is a difficult application to scale because the common deployments involve high connection counts for `memcached` servers and small-sized requests and replies [2, 39]. Combined with latency SLOs in the few hundreds of microseconds, these characteristics make memcached a challenging workload for our study. Any shortcomings in the mechanisms and policies for energy proportionality and workload consolidation will quickly translate to throughput and latency problems. In contrast, workloads with SLOs in the millisecond range are more forgiving. Furthermore, memcached has well-known scalability limitations [28]. To improve its scalability, we configure memcached with a larger hash table size (`-o hashpower=20`) and use a random replacement policy instead of the built-in LRU, which requires a global lock. This last change provides memcached with similar multicore scalability as state-of-the-art key-value stores such as MICA [28]. We configure `memcached` similarly for Linux and IX.

Our primary background application is a simple synthetic proxy for computationally intense batch processing applications: it encrypts a memory stream using SHA-1 in parallel threads that each run in an infinite loop and report the aggregate throughput. The workload streams through a 40 MB array which exceeds the capacity of the processor's L3-cache, and therefore interferes (by design) with the memory subsystem performance of the latency-sensitive application.

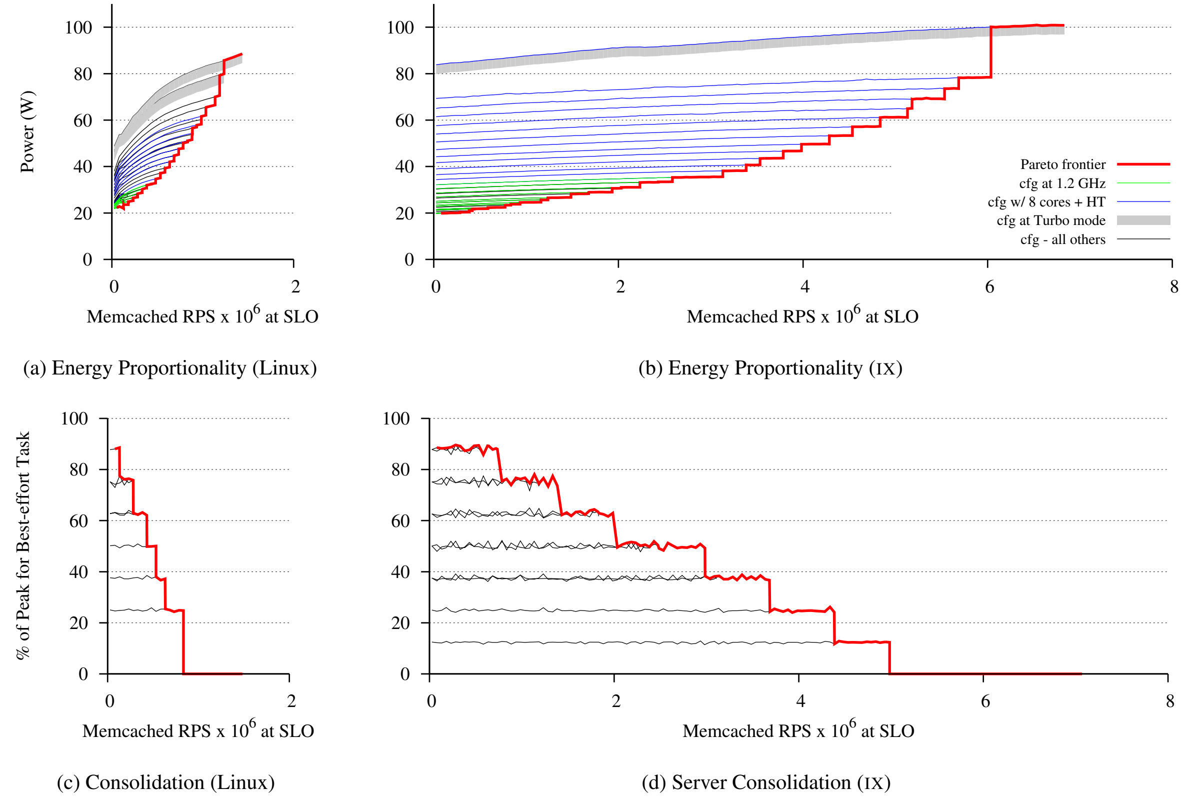
We use the `mutilate` load-generator to place a selected load on the foreground application in terms of requests per second (RPS) and measure response latency [25]. `Mutilate` coordinates a large number of client threads across 10 machines to generate the desired RPS load, and one additional, unloaded client to measure latency, for a total of 2,752 connections to the `memcached` server. We configure `mutilate` to generate load representative of the Facebook USR workload [2]: this is a large-scale deployment dominated by GET requests (99%), with short keys (<20B) and 2B values. Each request is issued separately (no `multiget` operations). However, clients are permitted to pipeline up to four requests per connection if needed to keep up with their target request rate. We enhance `mutilate` to support dynamically changing load levels used in these experiments. As the `memcached` server is multi-threaded, we also enhance the latency-measuring client to randomly choose among 32 open connections for each request, as to ensure statistically significant measurements.

We compute metrics as follows: the unloaded client measures latency by issuing one request at the time, chosen randomly among its 32 open connections. The SLO is specified so that the 99th percentile of requests issued by that unloaded client return in $\leq 500\mu s$. Latency distribution, achieved requests, and power metrics for all graphs are reported at one-second intervals. Energy consumption is measured using the Intel RAPL counters [8] for the processor in question; these provide an accurate model of the power draw of the CPU, but ignore platforms effects.

**A note on Turbo Boost:** For any given throughput level, we observe that the reported power utilization is stable for all CPU frequencies *except* for Turbo Boost. When running in Turbo Boost, the temperature of the CPU gradually rises over a few minutes from 58°C to 78°C, and with it the dissipated energy rises by 4 W for the same level of performance. The experiments in §3.1 run for a long time in Turbo Boost mode with a hot processor; we therefore report those results as an energy band of 4 W.

## 3. Understanding Dynamic Resource Usage

The purpose of dynamic resource allocation is to adapt to changes of load of a latency-critical workload by adding or removing resources as needed. The remaining resources can be placed in idle modes to save energy or used by other con-



(a) Energy Proportionality (Linux)

(b) Energy Proportionality (IX)

(c) Consolidation (Linux)

(d) Server Consolidation (IX)

Figure 2: Pareto efficiency for energy proportionality and workload consolidation, for Linux 3.16.1 and IX. The Pareto efficiency is in red while the various static configurations are color-coded according to their distinctive characteristics.

solidated applications. In general, dynamic control can be applied to any resource, including processor cores, memory, and I/O. We focus on managing CPU cores because they are the largest contributor to power consumption and draw significant amounts of energy even when lightly utilized [4, 29]. Moreover, the ability to dynamically reassign cores between latency-critical and background applications is key to meet latency SLOs for the former and to make reasonable progress using underutilized resources for the latter.

Dynamic resource control is a multi-dimensional problem on current CPUs: multi-core, hyperthreading and frequency each separately impact the workload's performance, energy consumption and consolidation opportunities. §3.1 uses a Pareto methodology to evaluate this three-dimensional space, and §3.2 discusses the derived allocation policies.

### 3.1 Pareto-Optimal Static Configurations

Static resource configurations allow for controlled experiments to quantify the tradeoff between an application's performance and the resources consumed. Our approach limits bias by considering many possible static configurations in the three-dimensional space of core, hyperthread, and frequency. For each static configuration, we characterize the maximum load that meets the SLO ($\leq 500\mu s$@99th percentile); we then measure the energy draw and throughput of the background job for all load levels up to the maximum load supported. From this large data set, we derive the set of meaningful static configurations and build the Pareto efficiency frontier [41]. The frontier specifies, for any possible load level, the optimal static configuration and the resulting minimal energy draw or maximum background throughput, depending on the scenario.

Fig. 2 presents the frontier for four experiments: the energy proportionality and workload consolidation scenarios, for both Linux and IX. The four graphs each plot the objective—which is either to minimize energy or maximize background throughput—as a function of the foreground throughput, provided that the SLO is met. Except for the red lines, each line corresponds to a distinct static configuration of the system: the green curves correspond to configuration at the minimal clock rate of 1.2 Ghz; the blue curves use all available cores and hyperthreads; other configurations are in black. In Turbo Boost mode, the energy drawn is reported as a band since it depends on operating temperature (see §2).

Finally, the red line is the Pareto frontier, which corresponds, for any load level, to the optimal result using any

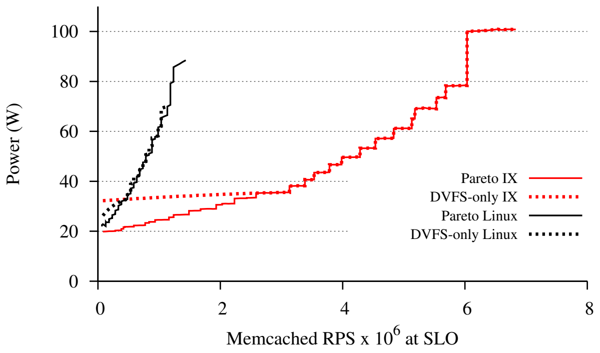


Figure 3: Energy-proportionality comparison between the Pareto-optimal frontier considering only DVFS adjustments, and the full Pareto frontier considering core allocation, hyperthread allocations, and frequency.

of the static configurations available. Each graph only shows the static configurations that participate in the frontier.

**Energy proportionality:** We evaluate 224 distinct combinations: from one to eight cores, using consistently either one or two threads per core, for 14 different DVFS levels from 1.2 Ghz to 2.4 Ghz as well as Turbo Boost. Fig. 2a and Fig. 2b show the 43 and 32 static configurations (out of 224) that build the Pareto frontier for energy proportionality. The figures confirm the intuition that: (i) various static configurations have very different dynamic ranges, beyond which they are no longer able to meet the SLO; (ii) each static configuration draws substantially different levels of energy for the same amount of work; (iii) at the low-end of the curve, many distinct configurations operate at the minimal frequency of 1.2 Ghz, obviously with a different number of cores and threads, and contribute to the frontier; these are shown in green in the Figure; (iv) at the high-end of the range, many configurations operate with the maximum of 8 cores, with different frequencies including Turbo Boost.

**Consolidation:** The methodology here is a little different. We first characterize the background job, and observe that it delivers energy-proportional throughput up to 2.4 Ghz, but that Turbo Boost came at an energy/throughput premium. Consequently, we restrict the Pareto configuration space at 2.4 Ghz; the objective function is the throughput of the background job, expressed as a fraction of the throughput of that same job without any foreground application. Background jobs run on all cores that are not used by the foreground application. Fig. 2c and Fig. 2d show the background throughput, expressed as a fraction of the standalone throughput, as a function of the foreground throughput, provided that the foreground application meets the SLO: as the foreground application requires additional cores to meet the SLO, the background throughput decreases proportionally.

**Linux vs. IX:** Fig. 2a and Fig. 2b show that Linux and IX behave differently: (i) IX improves the throughput of Linux by 4.8×; (ii) IX can clearly take advantage of hyperthreading and has its best performance with configurations with 16 threads on 8 cores. In contrast, Linux cannot take advantage of hyperthreading and has its best performance with 8 threads of control running on 8 cores; (iii) the efficiency of individual static configurations of Linux, which relies on interrupts, is much more subject to load than individual static configurations of IX, which relies on a polling model. As a consequence, an improperly configured, oversized IX dataplane will be extremely energy-inefficient at low-to-moderate loads; (iv) the comparison of the Pareto frontiers shows that IX's frontier always dominates the frontier of Linux, i.e. provides better energy efficiency for the same load level. This suggests that a properly configured dataplane design is always more energy-efficient than any Linux-based configuration. The difference is substantial, including at low-to-moderate load levels: IX's frontier is less than half of Linux's when the throughput load $\geq$ 780 KRPS ($\sim$54% of Linux's capacity).

**DVFS-only alternative:** Fig. 3 further analyzes the data and compares the Pareto frontiers with an alternate frontier that only considers changes in DVFS frequency. The comparison is limited to the energy proportionality scenario; for consolidation, available cores are necessary for the background application to run, and a DVFS-only solution would not suffice.

We observe that the impact of DVFS-only controls differs noticeably between Linux and IX: with Linux, the DVFS-only alternate frontier is very close to the Pareto frontier, meaning that a DVFS-only approach such as Pegasus [29] or Adrenaline [15] would be adequate. This is due to Linux's idling behavior, which saves resources. In the case of IX however—and likely for any polling-based dataplane—a DVFS-only scheduler would provide worse energy proportionality at low-moderate loads than a corresponding Linux-based solution. As many datacenter servers operate in the 10%-30% range [4], we conclude that a dynamic resource allocation scheme involving both DVFS and core allocation is necessary for dataplane architectures.

### 3.2 Derived Policy

We use the results from §3.1 to derive a resource configuration policy framework, whose purpose is to determine the sequence of configurations to be applied, as a function of the load on the foreground application, to both the foreground (latency-sensitive) and background (batch) applications. Specifically, given an ever-increasing (or -decreasing) load on the foreground applications, the goal is to determine the sequence of resource configurations minimizing energy consumption or maximizing background throughput, respectively.

We observe that (i) the latency-sensitive application (memcached) can scale nearly linearly, up to the 8 cores of the processor; (ii) it benefits from running a second thread on each core, with a consistent speedup of 1.3×; (iii) it is most energy-efficient to first utilize the various cores, and only then to enable the second hyperthread on each core, rather than the other way around; and (iv) it is least energy-efficient to increase the frequency.

We observe that the background application (i) also scales linearly; but (ii) does not benefit from the 2nd hyperthread; (iii) is nearly energy-proportional across the frequency spectrum, with the exception of Turbo Boost. From a total cost of ownership perspective, the most efficient operating point for the workload consolidation of the background task is therefore to run the system at the processor's nominal 2.4 Ghz frequency whenever possible. We combine these observations with the data from the Pareto analysis and derive the following policies:

**Energy Proportional Policy:** As a base state, run with only one core and hyperthread with the socket set at the minimal clock rate (1.2Ghz). To add resources, first enable additional cores, then enable hyperthreads on all cores (as a single step), and only after that gradually increase the clock rate until reaching the nominal rate (2.4Ghz); finally enable Turbo Boost. To remove resources, do the opposite. This policy leads to a sequence of 22 different configurations.

**Workload Consolidation Policy:** As a base state, run the background jobs on all available cores with the processor at the nominal clock rate. To add resources to the foreground application, first shift cores from the background thread to the foreground application one at a time. This is done by first suspending the background threads; use both hyperthreads of the newly freed core for the foreground application. Next, stop the background job entirely and allocate all cores to the foreground applications. As a final step, enable Turbo Boost. This policy leads to a sequence of 9 different configurations.

These policies closely track the corresponding Pareto frontier. For energy proportionality, (i) the 32 different static configurations of the frontier are a superset of the configurations enabled by the policy, and (ii) the difference in overall impact in terms of energy spent is marginal. For consolidation, Pareto and policy nearly identically overlap.

Obviously, these conclusions may vary with platforms and applications, but the use of underlying Pareto methodology remains valid. In fact, it should provide a robust guide to determine allocation policies for more complex workloads (e.g., with natural scalability limitations) and more flexible platforms (e.g., the newer *Haswell* processors can set the clock frequency separately for individual cores).

# 4. Dynamic Resource Controls of Dataplanes

We now present the design of the mechanisms and policies that are necessary to implement dynamic resource controls in dataplane operating systems. Although the implementation is specific to the IX dataplane operating system, the design principles generally apply to all high-throughput dataplane operating systems such as Arrakis [43] and user-level networking stacks running latency-sensitive applications [18, 20, 31]. Traditionally, such environments have not focused on dynamic resource controls, instead they focused solely on throughput with statically allocated resources, without considerations for the energy consumed. Adding resource controls in such environments poses the following challenges:

1. How to rebalance flows across cores without impacting normal performance. This is particularly challenging as dataplane environments achieve high performance because of their *coherence-free* execution model. By that, we mean a programming where all common case operations execute without requiring communication of synchronization between threads.
2. How to rebalance flows across processing cores without impacting network performance, given that dataplane operating systems implement their own TCP/IP networking stack and that TCP/IP performance is negatively impacted whenever packets are either lost or processed out of order [24]. This is particularly challenging whenever the rebalancing operation also involves the reconfiguration of the NIC, which generates inherent race conditions.
3. How to determine—both robustly and efficiently—whether resources need to be added or can be removed from the dataplane, given the complexity of modern applications, and the difficulty to estimate the impact of resource controls on application throughput.

Our design addresses these three challenges. We first provide in §4.1 the necessary background on the IX dataplane and in §4.2 the background on the few assumptions made on the hardware. §4.3 describes the implementation of the controller, which relies exclusively on application-independent queuing delay metrics to adjust resources. §4.4 describes how a dataplane operating system can have both a coherence-free execution model and at the same time allow for rebalancing of flows. Finally, §4.5 describes the mechanism that migrates flows without reordering packets.

## 4.1 Background on IX

The IX system architecture separates the control plane, responsible for the allocation of resources, from the dataplane, which operates on them, using an approach similar to Arrakis [43]. IX relies on hardware virtualization to protect the kernel from applications: it uses Dune [5] to load and execute the IX dataplane instances, and uses the Linux host as its control plane environment.

The IX dynamic controller is a Python daemon that configures and launches the IX dataplane, then monitors its execution, makes policy decisions, and communicates them to the IX dataplane and the host operating system.

Unlike conventional operating systems, which decouple network processing from application execution, the IX data-

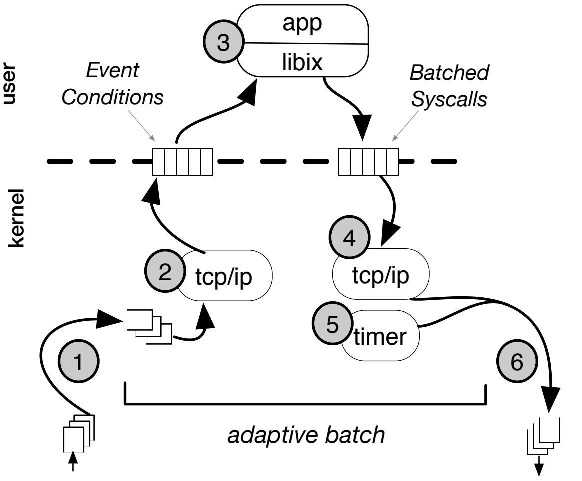


Figure 4: Adaptive batching in the IX data plane.

plane relies on a *run to completion with adaptive batching* scheme to deliver both low latency and high throughput to applications.

Fig. 4 illustrates the key run-to-completion mechanism built into the original IX dataplane [6]: each elastic thread has exclusive access to NIC queues. In step (1), the dataplane polls the receive descriptor ring and moves all received packets into an in-memory queue and potentially posts fresh buffer descriptors to the NIC for use with future incoming packets. The thread then executes steps (2) – (6) for an *adaptive batch* of packets: the TCP/IP network stack processes these packets (2) and the application then consumes the resulting events (3). Upon return from user-space, the thread processes batched systems calls and, in particular, the ones that direct outgoing TCP/IP traffic (4), runs necessary timers (5), and places outgoing Ethernet frames in the NIC's transmit descriptor ring for transmission (6).

Adaptive batching is specifically defined as follows: (i) we never wait to batch requests and batching only occurs in the presence of congestion; (ii) we set an upper bound on the number of batched packets. Using batching only during congestion allows us to minimize the impact on latency, while bounding the batch size prevents the live set from exceeding cache capacities and avoids transmit queue starvation. Batching improves packet rate because it amortizes system call transition overheads and improves instruction cache locality, prefetching effectiveness, and branch prediction accuracy. When applied adaptively, batching also decreases latency because these same efficiencies reduce head-of-line blocking.

## 4.2 Dynamic NIC Configuration

Modern NICs provide hardware mechanisms such as VMDq, Flow Director [17], and Receive Side Scaling (RSS) [36] to direct traffic to one of multiple queues, as an essential mechanism for multicore scalability. For example, the Intel x520 relies on an on-chip table (called the *RSS Redirection Table* or RETA [17]) to partition flows into multiple queues: on the data path, the NIC first computes the Toeplitz hash function of the packet 5-tuple header (as specified by RSS), and uses the lower 7 bits of the result as an index into the RETA table. A *flow group* is the set of network flows with an identical RETA index. The host updates the table using PCIe writes on a distinct control path, which can lead to packet reordering [52].

Such anomalies have severe consequences in general on the performance of the TCP/IP networking stack [24]. IX's coherence-free design exposes a further constraint: because there are no locks within the TCP/IP stack, two elastic threads *cannot* process packets from the same flow simultaneously. For correctness, packets received by the thread that does not own the flow group must be dropped. This further negatively impacts network performance.

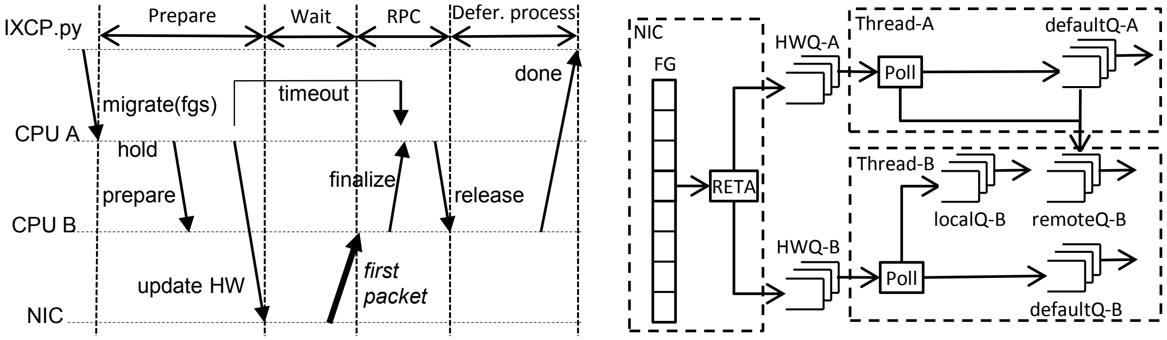
The IX dataplane implements the actual flow group migration and relies on the RETA to change the mappings. The challenge is to design a migration mechanism that updates the steering of packets to different queues (and by extension to different threads) without (i) impacting the steady state performance of the dataplane and its coherence-free design; or (ii) creating network anomalies during migration such as dropping packets or processing them out of order in the networking stack.

## 4.3 Control Loop

We first describe the control loop at the heart of IX's dynamic controller, implemented within the control plane. The controller adjusts processor resources by suspending and resuming IX elastic threads, and specifying the mapping between flow groups and threads. It migrates flow groups from multiple existing threads to a newly resumed thread, and conversely from a thread to be suspended to the remaining threads. For the server consolidation scenario, the control loop also allocates resources dedicated to the background applications. In our experimental setup, it simply issues `SIGSTOP` and `SIGCONT` signals to transparently control the background application.

The control loop does not depend on any detailed characterization of the Pareto frontier of §3.1. Instead, it merely implements the derived, high-level policy of §3.2 using queuing delay estimates. Specifically, the user sets the upper bound on the acceptable queuing delay; in our experiments, with `memcached` characterized by high packet rates and low service times, we set the acceptable bound at 300μs to minimize latency violations , given our SLO of 500μs at the 99th percentile, as measured end-to-end by the client machine.

For this, we rely on a key side effect of IX's use of adaptive batching: unprocessed packets that form the backlog are queued in a central location, namely in step (1) in the pipeline of Fig. 4. Packets are then processed in order, in bounded batches to completion through both the networking stack and the application logic. In other words, each IX core



(a) Thread-centric view

(b) Packet-centric view

Figure 5: Flow-group Migration Algorithm

operates like a simple FIFO queuing server, onto which classic queuing and control theory principles can be easily applied. In contrast, conventional operating systems distribute buffers throughout the system: in the NIC (because of coalesced interrupts), in the driver before networking processing, and in the socket layer before being consumed by the application. Furthermore, these conventional systems provide no ordering guarantees across flows, which makes it difficult to pinpoint congestion.

To estimate queuing delays, the controller monitors the iteration time $\tau$ and the queue depth $Q$. With B the maximal batch size, the tail latency is $\sim max(delay) = \lceil Q/B \rceil * \tau$. The dataplane computes each instantaneous metric every $10ms$ for the previous $10ms$ interval. As these metrics are subject to jitter, the dataplane computes the exponential weighted moving averages using multiple smoothing factors ($\alpha$) in parallel. For example, we track the queue depth as $Q(t,\alpha) = \alpha * Q_{now} + (1-\alpha) * Q(t-1,\alpha)$. The control loop executes at a frequency of 10 Hz, which is sufficient to adapt to load changes.

Deciding when to remove resources is trickier than deciding when to add them, as shallow and near-empty queues do not provide reliable metrics. Instead, we measure idle time and observe that each change in the configuration adds or removes a predictable level of throughput: (i) increasing cores from $n$ to $n+1$ increases throughput by a factor of $1/n$; (ii) enabling hyperthreads adds 30% (for our foreground workload at least); and (iii) the core's performance is largely proportional to its clock rate. We can therefore define the function $threshold[from,to]$ as the ratio of max throughput between the $from$ and $to$ configurations. When operating at configuration $from$ and the idle time exceeds $threshold[from,to]$ we can safely switch to configuration $to$ without loss of performance.

These settings are not universal, as a number of assumptions such as load changes, requests bursts, and variations in request service times, will noticeably affect the metrics. These are policies, not mechanisms. In particular, there is an inherent tradeoff between aggressive energy saving policies, which may lead to SLO violations, and conservative policies, which will consume an excessive amount of energy. Fortunately, the policy can be easily modified: the dataplane exposes naturally all relevant metrics, and the control plane daemon is ∼500 lines of Python, including ∼80 lines for the control loop.

### 4.4 Data Structures for Coherence-free Execution

We designed our memory management and data structure layout to minimize the performance impact of flow group migration and to preserve the coherence-free property of our dataplane in the steady-state. Our memory allocator draws inspiration from magazine-based SLAB allocation [7]. Each thread maintains a thread-local cache of every data type. If the size of the thread-local cache exceeds a threshold, memory items are returned, using traditional locking, to a shared memory pool. Such rebalancing events are common during flow group migration but infrequent during normal operation.

We further organize our data structures so the pointer references building linked lists, hash tables, etc., are only between objects either both permanently associated with the same thread (*per-cpu*), or both permanently associated with the same flow group (*per-fg*), or both global in scope. As a result, we can efficiently transfer ownership of an entire flow-group's data structures during migration while avoiding locking or coherence traffic during regular operation.

For example, the event condition arrays, batch syscall arrays, timer wheels, incoming and outgoing descriptor rings, posted and pending mbufs are all `per-cpu`. The protocol control blocks, 5-tuple hash table, listening sockets, `TW_WAIT` queues, out-of-order fragments, and unacknowledged sequences, are all `per-fg`. The `ARP` table is `global`. Some objects change scope during their lifetime, but only at well-defined junctures. For example, an `mbuf` is part of the `per-cpu` queue while waiting to be processed (step 1 in Fig. 4), but then becomes part of `per-fg` structures during network processing.

A global data structure, the software flow group table, manages the entry points into the per-fg structures. Intel NICs conveniently store the RSS hash result as meta-data along with each received packet; the bottom bits are used to index into that table.

We encountered one challenge in the management of TCP timers. We design our implementation to maximize normal execution performance: each thread independently manages its timers as a hierarchical timing wheel [49], i.e., as a per-cpu structure. As a consequence, a flow group migration requires a scan of the hierarchical wheels to identify and remove pending timers during the hold phase of Fig. 5a; those will be restored later in the destination thread, at the release phase of the flow group migration.

### 4.5 Flow Group Migration

When adding or removing a thread, the dynamic controller generates a set of migration requests. Each individual request is for a set of flow groups (fgs) currently handled by one elastic thread A to be handled by elastic thread B. To simplify the implementation, the controller serializes the migration requests and the dataplane assumes that at most one such request is in progress at any point in time.

Each thread has three queues that can hold incoming network packets: the *defaultQ* contains packets received during steady state operation, when no flow group migration is pending; the *remoteQ* contains packets belonging to outgoing flow groups; the *localQ* contains packets belonging to incoming flow groups. The use of these two extra queues is critical to ensure the processing of packets in the proper order by the networking stack.

Fig. 5 illustrates the migration steps in a thread-centric view (Fig. 5a) and in a packet-centric view (Fig. 5b). The controller and the dataplane threads communicate via lock-free structures in shared memory. First, the controller signals A to migrate fgs to B. A first marks each flow group of the set fgs with a special tag to hold off normal processing on all threads, moves packets which belong to the flow group set fgs from defaultQ-A to remoteQ-B and stops all timers belonging to the flow group set. A then reprograms the RETA for all entries in fgs. Packets still received by A will be appended to remoteQ-B; packets received by B will go to localQ-B.

Upon reception of the first packet whose flow group belongs to fgs by B, B signals A to initiate the final stage of migration. Then, B finalizes the migration by re-enabling fgs's timers, removing all migration tags, and pre-pending to its defaultQ-B the packets from remoteQ-B and the packets from localQ-B. Finally, B notifies the control plane that the operation is complete. There are obviously corner-cases and subtleties in the algorithm. For example, A installs a migration timer to finalize the migration even if B has not yet received any packets. We set the timeout to 1 ms.

## 5. Evaluation

We use three synthetic, time-accelerated load patterns to evaluate the effectiveness of the control loop under stressful conditions. All three vary between nearly idle and maximum throughput within a four minute period: the *slope* pattern gradually raises the target load from 0 and 6.2M RPS and then reduces its load; the *step* pattern increases load by 500 KRPS every 10 seconds; and finally the *sine+noise* pattern is a basic sinusoidal pattern modified by randomly adding sharp noise that is uniformly distributed over [-250,+250] KRPS and re-computed every 5 seconds. The slope pattern provides a baseline to study smooth changes, the step pattern models abrupt and massive changes, while the sine+noise pattern is representative of daily web patterns [48].

Fig. 6 and Fig. 7 show the results of these three dynamic load patterns for the energy proportionality and workload consolidation scenarios. In each case, the top figure measures the observed throughput. They are annotated with the control loop events that add resources (green) or remove them (red). Empty triangles correspond to core allocations and full triangles to DVFS changes. The middle figure evaluates the soundness of the algorithm and reports the 99th percentile latency, as observed by a client machine and reported every second. Finally, the bottom figures compare the overall efficiency of our solution based on dynamic resource controls with (i) the maximal static configuration, using all cores and Turbo Boost, and (ii) the ideal, synthetic efficiency computed using the Pareto frontier of Fig. 2

**Energy Proportionality:** Fig. 6 shows the behavior for the energy efficiency scenario. In each case, the workload tracks the desired throughput of the pattern and exercises the entire sequence of configurations, gradually adding cores, enabling hyperthreading, increasing the frequency and finally enabling Turbo Boost, before doing it in reverse. The step pattern is particularly challenging, as the instant change in load level requires multiple, back-to-back, configurations changes. With a few exceptions, the latencies remain well below the 500μs SLO. We further discuss the violations below. Finally, Fig. 6 (bottom) compares the power dissipated by the workload with the corresponding power levels as determined by the Pareto frontier (lower bound) or the maximum static configuration (upper bound). This graph mea-

| | Smooth | Step | Sine+noise |
|---|---|---|---|
| Energy Proportionality (W) | | | |
| Max. power | 91 | 92 | 94 |
| Measured | 42 (-54%) | 48 (-48%) | 53 (-44%) |
| Pareto bound | 39 (-57%) | 41 (-55%) | 45 (-52%) |
| Server consolidation opportunity (% of peak) | | | |
| Pareto bound | 50% | 47% | 39% |
| Measured | 46% | 39% | 32% |

Table 1: Energy Proportionality and Consolidation gains.

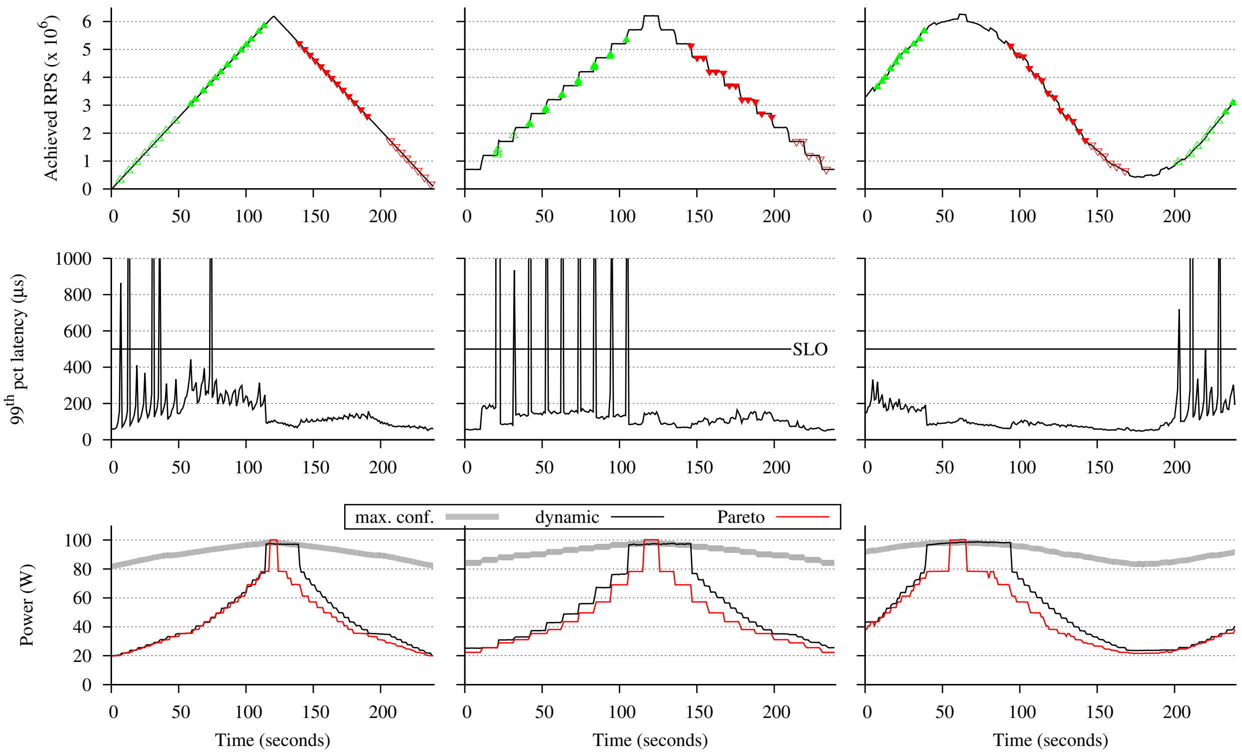


Figure 6: Energy Proportionality of `memcached` running on IX for three load patterns.

sures the effectiveness of the control loop to maximize energy proportionality. We observe that the dynamic (actually measured) power curve tracks the Pareto (synthetic) curve well, which defines a bound on energy savings. When the dynamic resource controls enters Turbo Boost mode, the measured power in all three cases starts at the lower end of the 4 W range and then gradually rises, as expected. Table 1 shows that the three patterns have Pareto savings bounds of 52%, 55% and 57%. IX's dynamic resource controls results in energy savings of 44%, 48% and 54%, which is 85%, 87% and 93% of the theoretical bound.

**Consolidation:** Fig. 7 shows the corresponding results for the workload consolidation scenarios. Fig. 7 (top) measures the observed load, which here as well tracks the desired load. Recall that the consolidation policy always operates at the processor's nominal rate (or Turbo), which limits the number of configuration changes. Fig. 7 (middle) similarly confirms that the system meets the SLO, with few exceptions. Fig. 7 (bottom) plots the throughput of the background batch application, expressed as a percentage of its throughput on a dedicated processor at 2.4 Ghz. We compare it only to the Pareto optimal upper bound as a maximum configuration would monopolize cores and deliver zero background throughput. Table 1 shows that, for these three patterns, our consolidation policy delivers 32%–46% of the standalone throughput of the background job, which corresponds to 82%–92% of the Pareto bound.

**SLO violations:** A careful study of the SLO violations of the 6 runs shows that they fall into two categories. First, there are 16 violations caused by delays in packet processing due to flow group migrations resulting from the addition of a core. Second, there are 9 violations caused by abrupt increase of throughput, mostly in the step pattern, which occur before any flow migrations. The control plane then reacts quickly (in ∼100 ms) and accommodates to the new throughput by adjusting resources. To further confirm the abrupt nature of throughput increase specific to the step pattern, we note that the system performed up three consecutive increases in resources in order to resolve a single violation. 23 of the 25 total violations last a single second, with the remaining two violations lasting two seconds. We believe that the compliance with the SLO achieved by our system is more than adequate for any practical production deployment.

**Flow group migration analysis:** Table 2 measures the latency of the 552 flow group migrations that occur during the 6 benchmarks, as described in §4.5. It also reports the total number of packets whose processing is deferred during the migration (rather than dropped or reordered). We first observe that migrations present distinct behaviors when scaling up and when scaling down the number of cores. The differ-

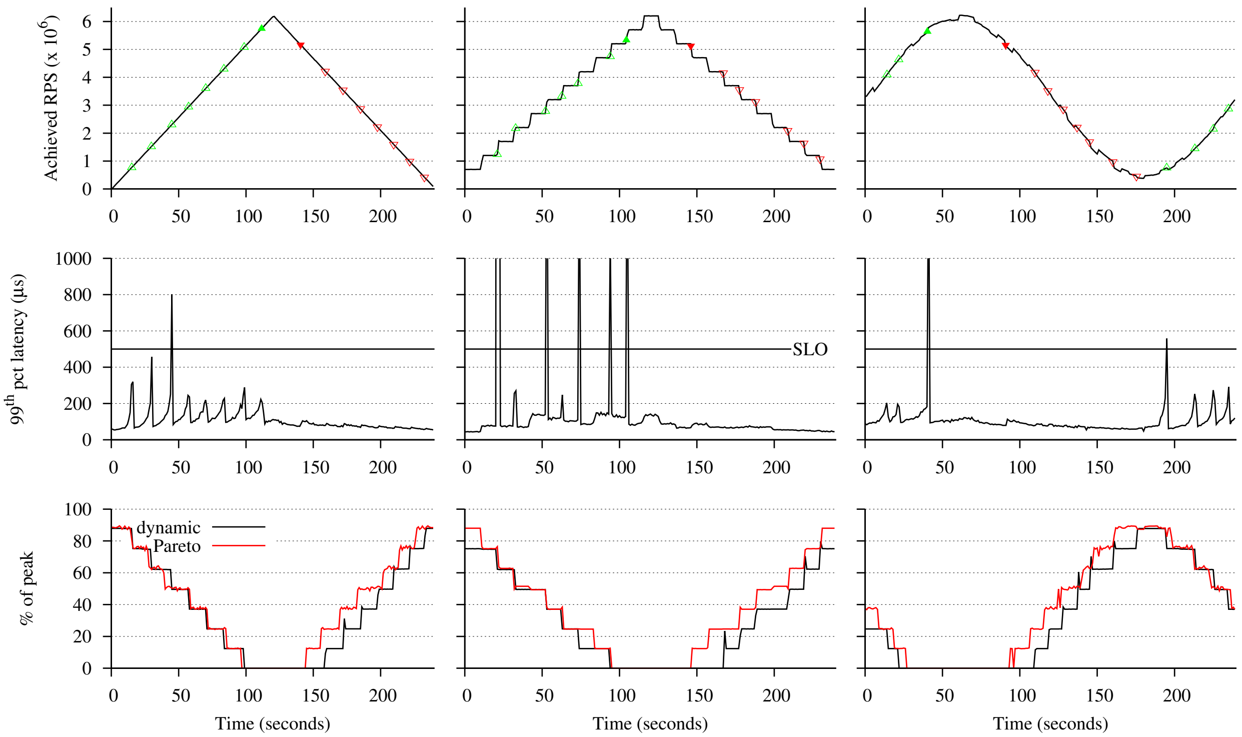


Figure 7: Consolidation of `memcached` with a background batch task.

ence can be intuitively explained since the migrations during the scale up are performed in a heavily loaded system, while the system during the scale down is partially idle. In absolute terms, migrations that occur when adding a core take 463µs on average and less than 1.5 ms 95% of the time. The outliers can be explained by rare occurrences of longer preparation times or when processing up to 614 deferred packets.

## 6. Discussion

**Using Pareto as a guide:** Even though the Pareto results are not used by the dynamic resource controller, the Pareto frontier proved to be a valuable guide, first to motivate and quantify the problem, then to derive the configuration policy sequence, and finally to evaluate the effectiveness of the dynamic resource control by setting an upper bound on the gains resulting from dynamic resource allocation. Many factors such as software scalability, hardware resource contention, network and disk I/O bottlenecks, will influence the Pareto frontier of any given application, and therefore the derived control loop policies. We will compare different workloads in future work.

**Tradeoff between average latency and efficiency:** We use a one-dimensional SLO (99th pct. $\leq$ 500µs.) and show that we can increase efficiency while maintaining that objective. However, this comes at a cost when considering the entire latency distribution, e.g. the average as well as the tail. More complex SLOs, taking into account multiple aspects of latency distribution, would define a different Pareto frontier, and likely require adjustments to the control loop.

**Adaptive, flow-centric scheduling:** The new flow-group migration algorithm leads to a *flow-centric* approach to resource scheduling, where the network stack and application logic always follow the steering decision. POSIX applications can balance flows by migrating file descriptors between threads or processes, but this tends to be inefficient because it is difficult for the kernel to match the flow's destination receive queue to changes in CPU affinity. Flow director can be used by Linux to adjust the affinity of individual network flows as a reactive measure to application-level and kernel thread migration rebalancing, but the limited size of the redirection table prevents this mechanism from scaling to large connection counts. By contrast, our approach allows flows to be migrated in entire groups, improving efficiency, and is compatible with more scalable hardware flow steering mechanisms based on RSS.

**Hardware trends:** Our experimental setup using one *Sandy Bridge* processor has a three-dimensional Pareto space. Clearly, NUMA would add a new dimension. Also, the newly released *Haswell* processors provide per-core DVFS controls, which further increases the Pareto space. Moreover, hash filters for flow group steering could benefit from recent trends in NIC hardware. For example, Intel's new

|  |  | avg | 95th pct. | max. | stddev |
|---|---|---|---|---|---|
| add core | prepare (µs) | 67 | 231 | 521 | 93 |
|  | wait (µs) | 135 | 541 | 983 | 190 |
|  | rpc (µs) | 129 | 355 | 529 | 112 |
|  | deferred (µs) | 101 | 401 | 1167 | 147 |
|  | total (µs) | 463 | 1432 | 2870 | 459 |
|  | # packets | 64 | 319 | 614 | 116 |
| remove core | prepare (µs) | 25 | 54 | 398 | 32 |
|  | wait (µs) | 34 | 101 | 170 | 32 |
|  | rpc (µs) | 12 | 25 | 49 | 7 |
|  | deferred (µs) | 19 | 43 | 108 | 14 |
|  | total (µs) | 93 | 160 | 456 | 47 |
|  | # packets | 3 | 8 | 28 | 3 |

Table 2: Breakdown of flow group migration measured during the six benchmarks.

XL710 chipset [19], has a 512 entry hash LUT (as well as independent 64 entry LUTs for each VF) in contrast to the 128 entries available in the 82599 chipset [17] used in our evaluation. This has the potential to reduce connection imbalances between cores, especially with high core counts.

## 7. Related Work

**Scheduling:** Scheduler activations [1] give applications greater control over hardware threads and provide a mechanism for custom application-level scheduling. Callisto [13] uses a similar strategy to improve the performance of co-located parallel runtime systems. Our approach differs in that an independent control plane manages the scheduling of hardware threads based on receive queuing latency indicators while the dataplane exposes a simple kernel threading abstraction. SEDA [51] also monitors queuing behavior to make scheduling decisions such as thread pool sizing. Chronos [21] makes use of software-based flow steering, but with a focus on balancing load to reduce latency. Affinity Accept [42] embraces a mixture of software and hardware-based flow steering in order to improve TCP connection affinity and increase throughput. We focus instead on energy proportionality and workload consolidation.

**Energy Proportionality:** The energy proportionality problem [3] has been well explored in previous work. Some systems have focused on solutions tailored to throughput-oriented workloads [33] or read-only workloads [23]. Meisner et. al. [34] highlight unique challenges for low latency workloads and advocate full system active low-power modes. Similar to our system, Pegasus [29] achieves CPU energy proportionality for low latency workloads. Our work expands on Pegasus by exploring the elastic allocation of hardware threads in combination with processor power management states and by basing scheduling decisions on internal latency metrics within a host endpoint instead of an external controller. Niccolini et. al. show that a software router, running on a dedicated machine, can be made energy-proportional [38]. Similar to our approach, queue length is used as a control signal to manage core allocation and DVFS settings. However, we focus on latency-sensitive applications, rather than middlebox traffic, and consider the additional case of workload consolidation.

**Co-location:** Because host endpoints contain some components that are not energy proportional and thus are most efficient when operating at 100% utilization, co-location of workloads is also an important tool for improving energy efficiency. At the cluster scheduler level, BubbleUp [32] and Paragon [10] make scheduling decisions that are interference-aware through efficient classification of the behavior of workload co-location. Leverich et. al. [26] demonstrate that co-location of batch and low latency jobs is possible on commodity operating systems. Our approach explores this issue at higher throughputs and with tighter latency SLOs. Bubble-Flux [53] additionally controls background threads; we control background and latency-sensitive threads. CPI$^2$ [54] detects performance interference by observing changes in CPI and throttles offending jobs. This work is orthogonal to ours and could be a useful additional signal for our control plane. Heracles manages multiple hardware and software isolation mechanisms, including packet scheduling and cache partitioning, to co-locate latency-sensitive applications with batch tasks while maintaining millisecond SLOs [30]. We limit our focus to DVFS and core assignment but target more aggressive SLOs.

**Low-latency frameworks:** User-level networking stacks [20, 31, 47] and RDMA applications [12, 37, 40] all rely on polling, typically by a fixed number of execution threads, to deliver µs-scale response times. These stacks and applications are evaluated based on their latency and throughput characteristics, and do not take into account energy proportionality or workload consolidation considerations.

## 8. Conclusion

We present the design and implementation of dynamic resource controls for the IX dataplane, which consists of a controller that allocates cores and sets processor frequency to adapt to changes in the load of latency-sensitive applications. The novel rebalancing mechanisms do not impact the steady-state performance of the dataplane and can migrate a set of flow groups in milliseconds without dropping or reordering packets. We develop two policies focused on optimizing energy proportionality and workload consolidation. Our results show that, for three varying load patterns, the energy proportionality can save 44%–54% of the processor's energy, or enable a background job to deliver 32%–46% of its standalone throughput. To evaluate the effectiveness of our approach, we synthesize the Pareto frontier by combining the behavior of all possible static configurations. Our policies deliver 85%–93% of the Pareto optimal bound in terms of energy proportionality, and 82%–92% in terms of consolidation.

## Acknowledgements

This work was supported by the Microsoft/EPFL Joint Research Center, the Stanford Experimental Datacenter Lab, and NSF grant CNS-1422088. George Prekas is supported by a Google Graduate Research Fellowship and Adam Belay by a Stanford Graduate Fellowship.

## References

[1] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, 1992.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 2012 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[3] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12):33–37, 2007.

[4] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[5] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th Symposium on Operating System Design and Implementation (OSDI)*, pages 335–348, 2012.

[6] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11st Symposium on Operating System Design and Implementation (OSDI)*, pages 49–65, 2014.

[7] J. Bonwick and J. Adams. Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources. In *USENIX Annual Technical Conference*, pages 15–33, 2001.

[8] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanaa, and C. Le. RAPL: memory power estimation and capping. In *Proceedings of the 2010 International Symposium on Low Power Electronics and Design*, pages 189–194, 2010.

[9] J. Dean and L. A. Barroso. The Tail at Scale. *Commun. ACM*, 56(2):74–80, 2013.

[10] C. Delimitrou and C. Kozyrakis. Quality-of-Service-Aware Scheduling in Heterogeneous Data centers with Paragon. *IEEE Micro*, 34(3):17–30, 2014.

[11] C. Delimitrou and C. Kozyrakis. Quasar: Resource-Efficient and QoS-Aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, pages 127–144, 2014.

[12] A. Dragojevic, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. In *Proceedings of the 11st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 401–414, 2014.

[13] T. Harris, M. Maas, and V. J. Marathe. Callisto: co-scheduling parallel runtime systems. In *Proceedings of the 2014 EuroSys Conference*, page 24, 2014.

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[15] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. F. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *Proceedings of the 21st IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 271–282, 2015.

[16] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proceedings of the 11st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 445–458, 2014.

[17] Intel Corp. Intel 82599 10 GbE Controller Datasheet. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/82599-10-gbe-controller-datasheet.pdf`, 2014.

[18] Intel Corp. Intel DPDK: Data Plane Development Kit. `http://dpdk.org/`, 2014.

[19] Intel Corp. Intel Ethernet Controller XL710 Datasheet. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf`, 2014.

[20] E. Jeong, S. Woo, M. A. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 489–502, 2014.

[21] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the 2012 ACM Symposium on Cloud Computing (SOCC)*, page 9, 2012.

[22] W. Kim, M. S. Gupta, G.-Y. Wei, and D. M. Brooks. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *Proceedings of the 14th IEEE Symposium on High-Performance Computer Architecture (HPCA)*, pages 123–134, 2008.

[23] A. Krioukov, P. Mohan, S. Alspaugh, L. Keys, D. E. Culler, and R. H. Katz. NapSAC: design and implementation of a power-proportional web cluster. *Computer Communication Review*, 41(1):102–108, 2011.

[24] K.-C. Leung, V. O. K. Li, and D. Yang. An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges. *IEEE Trans. Parallel Distrib. Syst.*, 18(4):522–535, 2007.

[25] J. Leverich. Mutilate: High-Performance Memcached Load Generator. `https://github.com/leverich/mutilate`, 2014.

[26] J. Leverich and C. Kozyrakis. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *Proceedings of the 2014 EuroSys Conference*, page 4, 2014.

[27] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the 2014 ACM Symposium on Cloud Computing (SOCC)*, pages 1–14, 2014.

[28] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11st Symposium on Networked Systems Design and Implementation (NSDI)*, pages 429–444, 2014.

[29] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. In *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*, pages 301–312, 2014.

[30] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd International Symposium on Computer Architecture (ISCA)*, pages 450–462, 2015.

[31] I. Marinos, R. N. M. Watson, and M. Handley. Network Stack Specialization for Performance. In *Proceedings of the ACM SIGCOMM 2014 Conference*, pages 175–186, 2014.

[32] J. Mars, L. Tang, K. Skadron, M. L. Soffa, and R. Hundt. Increasing Utilization in Modern Warehouse-Scale Computers Using Bubble-Up. *IEEE Micro*, 32(3):88–99, 2012.

[33] D. Meisner, B. T. Gold, and T. F. Wenisch. The PowerNap Server Architecture. *ACM Trans. Comput. Syst.*, 29(1):3, 2011.

[34] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, pages 319–330, 2011.

[35] memcached. memcached – a distributed memory object caching system. `http://memcached.org`, 2014.

[36] Microsoft Corp. Receive Side Scaling. `http://msdn.microsoft.com/library/windows/hardware/ff556942.aspx`, 2014.

[37] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, pages 103–114, 2013.

[38] L. Niccolini, G. Iannaccone, S. Ratnasamy, J. Chandrashekar, and L. Rizzo. Building a Power-Proportional Software Router. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 89–100, 2012.

[39] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 385–398, 2013.

[40] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–41, 2011.

[41] Pareto. Pareto efficiency. `http:http://en.wikipedia.org/wiki/Pareto_efficiency`.

[42] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proceedings of the 2012 EuroSys Conference*, pages 337–350, 2012.

[43] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. E. Anderson, and T. Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11st Symposium on Operating System Design and Implementation (OSDI)*, pages 1–16, 2014.

[44] L. Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, pages 101–112, 2012.

[45] E. Rotem, A. Naveh, A. Ananthakrishnan, E. Weissmann, and D. Rajwan. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro*, 32(2):20–27, 2012.

[46] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 2013 EuroSys Conference*, pages 351–364, 2013.

[47] Solarflare Communications. Introduction to OpenOnload: Building Application Transparency and Protocol Conformance into Application Acceleration Middleware. `http://www.solarflare.com/content/userfiles/documents/solarflare_openonload_intropaper.pdf`, 2011.

[48] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[49] G. Varghese and A. Lauck. Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility. In *Proceedings of the 11st ACM Symposium on Operating Systems Principles (SOSP)*, pages 25–38, 1987.

[50] W. Vogels. Beyond Server Consolidation. *ACM Queue*, 6(1): 20–26, 2008.

[51] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 230–243, 2001.

[52] W. Wu, P. DeMar, and M. Crawford. Why Can Some Advanced Ethernet NICs Cause Packet Reordering? *IEEE Communications Letters*, 15(2):253–255, 2011.

[53] H. Yang, A. D. Breslow, J. Mars, and L. Tang. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers. In *Proceedings of the 40th International Symposium on Computer Architecture (ISCA)*, pages 607–618, 2013.

[54] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes. CPI$^2$: CPU performance isolation for shared compute clusters. In *Proceedings of the 2013 EuroSys Conference*, pages 379–391, 2013.