

Slowing Down for Performance and Energy: An OS-Centric Study in Network Driven Workloads

Han Dong, Sanjay Arora*, Yara Awad, Tommy Unger, Orran Krieger, Jonathan Appavoo
Red Hat, Boston University*

Abstract

This paper studies three fundamental aspects of an OS that impact the performance and energy efficiency of network processing: 1) batching, 2) processor energy settings, and 3) the logic and instructions of the OS networking paths. A network device’s interrupt delay feature is used to induce batching and processor frequency is manipulated to control the speed of instruction execution. A baremetal library OS is used to explore OS path specialization. This study shows how careful use of batching and interrupt delay results in 2X energy and performance improvements across different workloads. Surprisingly, we find polling can be made energy efficient and can result in gains up to 11X over baseline Linux. We developed a methodology and a set of tools to collect system data in order to understand how energy is impacted at a fine-grained granularity. This paper identifies a number of other novel findings that have implications in OS design for networked applications and suggests a path forward to consider energy as a focal point of systems research.

1 Introduction

There has been a large body of work in systems research focused using OS path specialization techniques to accelerate network applications [3, 11, 41, 45, 58, 61, 62, 68–70, 74, 76, 78, 79]. While performance is the main motivator behind these systems, we find their impact on energy is not as clearly understood. Our work seeks to start filling this gap by offering a detailed OS-centric study of performance-energy in network applications under different OS structures.

There are three fundamental aspects of an OS that impact the performance and energy efficiency of network processing: 1) batching, 2) processor energy settings, and 3) the logic and instructions of the OS networking paths. Delaying packet processing improves overall software stack efficiency as system overheads such as interrupt processing, OS book-keeping, and cache misses are amortized or eliminated by the batched handling of packets. However, the benefits of batching are typically weighed against its impact on workload latency [21]. Similarly, processor energy settings impacts the efficiency of network processing by trading off instruction execution speed with a reduction in energy use. In addition, batching and processor energy settings interacts with the software stack, its policies, and workload performance requirements to impact the energy saved by processor sleep states during idle periods between packet arrivals. Lastly,

specializing OS paths offers the chance to handle packets with optimized OS logic and data structures, thus improving overall network processing efficiency. In this paper, our goal is to study how all three of these aspects interact together to impact network processing performance and energy use.

A network interface controller’s (NIC) interrupt delay feature [37] is used to induce batching. A processor’s Dynamic Voltage Frequency Scaling (DVFS) [20] feature is used to control its frequency and energy setting to explores trade-offs in execution speed and energy use. An open sourced library OS (EbbRT [82]), ported to run baremetal, is used as a platform to contrast against a general purpose Linux to study how different OS structures are impacted by the two mechanisms listed.

While our data-driven study reveals a wealth of results in §5 with different impacts on system design, below, we summarize three of the main example findings and suggest implications for how OSES can improve energy efficiency while supporting high performance network applications:

1. **Finding:** By manually setting both DVFS and NIC interrupt delay values in an exhaustive search, we were able to find optimal performance and energy efficiency points in both OSES, e.g. rather than setting DVFS and interrupt delay values using its default policies, Linux configured with manually searched values can improve tail latency by 2X or energy savings by 55% in a network driven TPC-C style transactional database workload (fig. 8). **Implication:** Careful coordination among different hardware features should be used towards common objectives to explore new trade-offs that achieve even further efficiencies over today’s isolated policies.
2. **Finding:** Polling not only improves latency, but can be made energy efficient (for small payloads) under specialized OS paths using slowed DVFS. For example, in §5.2.1, we show 27% improved tail latency for a key value store with 35% less energy than when interrupts are used and 11X improvement in energy efficiency for a closed loop benchmark in §5.1.3). For other workloads that involve larger payloads or more application work, polling can result in negligible performance improvement while consuming dramatically more energy. For example in fig. 8, a baremetal library OS using poll achieved best case tail latency but used over 60% more energy than Linux. **Implication:** Specialized systems that use polling to achieving

low-latency [2, 11, 13, 41, 62, 68, 70, 74, 76, 82] can be made energy-efficient with careful use of DVFS, further, these results suggests the importance for energy aware OS-level optimizations that can switch between poll and interrupt-driven IO in response to changes in demand and workload behavior.

3. **Finding:** Exploiting OS path specialization via a baremetal library OS yields improvements in not only performance but also energy over Linux (88% energy efficiency improvement for NodeJS webserver in fig. 3, 2X throughput for memcached in fig. 7, and TPC-C in fig. 8). Surprisingly, even in application-heavy workloads, OS path specialization can still result in significant energy savings (up to 85% in fig. 8). **Implication:** There is enormous value to adopting specialized OS or path specialization (even in general purpose OSES) beyond virtual environments for both performance and energy efficiency.

In order to arrive at these findings, we conducted an extensive experimental study over the two OSES with thousands of experimental combinations resulting in a dataset over 5 TB. Given this large dataset, we developed a methodology and visualization tool to help us identify the performance-energy trade-offs in a fine-grained manner and to understand the causal relations between the hardware mechanisms and its impact on different OS structures. We believe the methodology, dataset, and tools are all contributions to OS research in systems and will help other researchers to develop new insights of the impact of OS changes on energy use. We plan to open source both the dataset and the tools.

In order to better reason about data we’ve collected, we begin by constructing a simple generic timeline of how packets are processed in a typical system in §2, we then discuss performance and energy in §3 in the context of how we break this timeline down. Next, we discuss our experimental and software setup in §4 and then present our experimental data in §5. Related work is discussed in §6 and we conclude in §7.

2 Processing Break Down

From an OS perspective we break down network driven processing into stages that allows us to organize and reflect the OS and application interaction with the workload request timeline. This break down is illustrated in figure 1 and shows a generic set of stages that all requests must go through.¹

2.1 Quiescent

Given the packet and transactional nature of network driven services, a quiescent period, in which no requests are present at the server, precedes activity on the server. The nature of the workloads drive the length of quiescent period and the

¹Although this is drawn and discussed from the perspective of a single core, our analysis and evaluation assumes that multiple cores could be concurrently used to shorten servicing times.

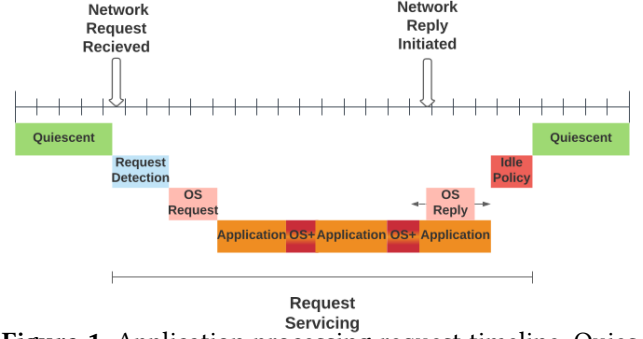


Figure 1. Application processing request timeline. Quiescent are periods between packet arrivals. Request Servicing includes all software components.

nature of the work itself required to service the request. Network services tends to fall into two broad categories, Open and Closed loop [83].

2.1.1 Open Loop: In an open loop scenario like a Memcached workload, the external request rate induces an inter-arrival gap that will drive the quiescent period – longer at lighter loads (lower queries-per-second (QPS)) and shorter at heavier loads (higher QPS). The arrival rate can largely be considered independent of the time required to service a request. Providers often set a Service-level Agreement (SLA) target, such as some percentage of requests to be completed under a stringent time budget, and there has been a wealth of research in using these SLA headrooms to lower data-center energy use mainly by decreasing processor frequencies [9, 22, 30, 31, 59, 60, 92].

2.1.2 Closed Loop: Examples of closed loop workloads are snapshotting a database to a remote server, video streaming or a middle tier service within a data center [7–9, 22, 59, 64]. The work to be done is a sequence of requests that have an inter-dependency on each other. Specifically, the arrival of the next request depends on how fast it takes to service the current request. From a server’s perspective, the quiescent period will be bounded by time to transmit both the request and the reply, as well as the time on the client to generate the next request. In the closed loop scenario, one would like the server to complete every request quickly so that the overall time to complete a task is minimized and ideally use less energy in the process.

2.2 OS Request Detection

Fundamental to any operating system is how it detects and schedules processing in response to IO device activity. At the two extremes are interrupt and poll driven detection.

2.2.1 Interrupt driven IO. Using interrupts has three important implications: 1) it can be used to wake a processor from a halted state, which the OS entered to sleep the processor previously, in response to external activity, 2) allow an OS to arbitrate processing across competitive devices in a multi-programmed/multi-device setting, 3) interrupts have

inherent performance costs associated with them – latency in starting to handle a request, either because of the costs associated with preempting work [2] or Intel c-state exit penalties[77]. This can also have a negative impact on the instruction efficiency, such as Instructions Per Cycle (IPC), due to induced micro-architecture hazards such as the inability to pre-fetch or speculatively execute across an interrupt.

2.2.2 Hybrid driven IO. A general purpose OS typically exploits some form of hybrid IO strategy alternating between interrupts and polling when servicing high speed NICs. A common strategy is to use interrupts when the load is low and switch to polling when load is high and back to interrupts when load reduces. A general purpose OS, even under sustained high load, bounds the poll phase to avoid starving other devices and software. Linux’s New API (NAPI)[86] framework implements this hybrid scheme.

2.2.3 Poll driven IO. In contrast, specializing an OS to support the execution of a single application can explore more extreme strategies like aggressive polling. Most modern NICs devices expose a cache-friendly interface that permits the processors to read a per-core memory address to determine if the device has received data that requires processing by the core. This allows software to directly poll the device and initiate software handling without an interrupt. This approach reduces latency and other performance penalties associated with interrupt driven IO but requires a busy CPU. As a result, there have been a large body of work to address latency sensitive applications through the judicious use of polling [2, 11, 13, 41, 62, 68, 70, 74, 76, 82]. In the extreme, a customized OS path supporting a single application can run a poll loop on every core to constantly check for work, conduct the work and then go back to polling for new work and thus never halting the processors.²

2.3 OS Request Processing

Once the OS detection mechanism identifies the NIC has data to process, several components of OS functionality must be run in accordance with the execution model of the OS. Normally, a network stack parses the packet header and eventually passes the payload to application for processing.

This work on a general purpose OS is typically split between two levels of scheduling; 1) interrupt level in which minimal work is done but at highest critical priority and is run-to-completion (typically called the top-half processing), 2) the so-called bottom-half uses kernel facilities to execute both device driver logic and protocol processing in a manner that can be preempted and rate limited. Regardless, all this work is done at the OS privilege level and ultimately

prepares data for application processing (pre-emptable), and is independently scheduled at lower privilege and priority.

An application specific library OS stack sheds much of the above complexity, both shortening the path and eliminating the above privilege scheduling domains [11, 70, 82]. It exploits short-cuts that allow run-to-completion execution of all the logic, including application processing in response to detecting device activity.

2.3.1 Application Processing. As illustrated in figure 1, during application processing, OS logic may be interleaved. This work roughly falls into two categories, synchronous work done in service of this application request (page-faults, system calls, etc) and asynchronous work not having to do with this request (OS background work, processing of other requests or processes). Library OS’s can often avoid interleaving asynchronous work, unrelated to the request handling, and thus minimize jitter and improve IPC.

2.3.2 OS Reply Processing. At some point during application processing, a reply is generated and submitted to the OS for transmission. This can be handled in an asynchronous fashion depending on the OS semantics; the OS can initiate protocol processing and device transmission in parallel with the remaining application logic (eg. book keeping, cleanup and preparation for the next request). This overlap reveals a potential opportunity for performance-energy trade-off. Specifically, it is possible given a particular packet arrival rate that slowing down causes the remaining application work to coincide with the time for the next request to arrive in both a closed and open loop setting, therefore keeping the processor busy. As such it may be possible that trade-offs in sleep state latency, interrupt overheads and polling leads to better performance at lower energy consumption.

2.3.3 Idle Policy. If all processing is complete, no traffic is pending and aggressive polling is not in use, the OS can use a policy that selects a hardware sleep state, such as Intel C-states [34], to halt the core. Various policies around optimizing them have been studied as well [16, 52, 65]. Each sleep state has an associated reduction in static power consumption. In the extreme, the deepest sleep states can flush micro-architectural state such as caches and power down these structures. However, each sleep state also imposes a progressively larger wake-up latency and potential impact on execution efficiency given the possible flushing of state [93].

There is clearly a relationship between the Idle Policy and Request Detection processing. For a general purpose OS the normative assumption is both are interrupt driven. Where an inter-dependency between the halt and interrupt mechanisms of the processor is exploited. In this study, we allow Linux’s scheduler and default idle policy to decide if a core should be halted and to what state. This policy exploits various statistics to estimate how long the core is likely to be idle. It takes into account an estimate of when

²It is worth noting processors also have the ability to halt in a way that an update to a cache line will awaken it, there exists the possibility of implementing the poll in combination with sleep states. We do not explore this possibility, leaving it for future work.

the next interrupt will likely occur from any source. This is a subtle implementation that interacts across many layers of the OS software, including the device driver. The idle driver framework also includes code provided by processor manufacturers to evaluate latency penalties and suggested minimum residency times. This allows us to see the impacts of making informed decisions regarding sleep states.

In contrast, the library OS explores two simple policies: 1) when there is no work to process on a core, the processor is put into the deepest c-state (C7), thus ignoring any trade-offs in use of other sleep states³ so that we can focus on the interaction of slowing down the processor and adjusting interrupt delays with the use of a fixed deep sleep, and 2) using an aggressive poll loop on all cores to check for IO events such that the processor is always kept busy and no idle policy is used.

3 Performance and Energy

In this section, we discuss the decomposition of figure 1 timeline relating to slowing down the processor, delaying requests, and OS specialization for network processing.

3.1 Interactions with Slowing Down the processor

The use of DVFS in a processor allows software to adjust the energy consumption of CMOS based logic while trading off instruction execution speed. As noted in [12, 15, 47, 52], static or leakage energy consumption (i.e. caches, TLBs) is not particularly affected by DVFS but induces a base cost for keeping a fixed core architecture active.⁴ An implication is that workloads which primarily use memory operations will suffer fewer performance penalties induced by a slowed processor while gaining energy saving benefits.

Our study explores how "slowing down" processing via DVFS interacts with the processing of network driven software stacks and the resultant energy and performance realized. We view DVFS as a speed control setting that can dilate CPU processing components of the request timeline in exchange for reduction in energy consumption.

From this perspective, the three obvious components that can be affected are OS Request, Application and OS Reply processing. For any given OS, there will be a hot-path instruction sequence that will be commonly exercised to process each request packet. The OS implementation will determine the type of instructions that will comprise of this path for a particular workload. As such, at the fastest DVFS setting there will be a characteristic mean number of cycles that will be required and thus an instructions-per-cycle (IPC) efficiency realized. It is important to note that better IPC does not necessarily imply better or worse performance or energy. What matters more is the amount of application work done

Name	Scenarios	Nature	CPU
NetPIPE	64B,8KB,64KB,512KB	CL	Low
NodeJS	na	CL	High
Memcached	200K, 400K, 600K	OL	Low
Memcached-Silo	50K, 100K, 200K	OL	High

Table 1. Workload configurations. The column *Nature* indicates open (OL) -versus-closed (CL) loop nature and *CPU* indicates application work demand.

per energy spent; as shown in §5.2.2, a more efficient implementation that uses less instructions, though with worse IPC, can still result in better performance and energy efficiency.

3.2 Interactions with Delaying Interrupts

As observed in [15, 21], the "latency-slack" between the mean time to service a request and the SLA target of an application creates an opportunity for energy-performance trade-offs. Further, these works suggest creating energy management policies that delaying request processing in order to interact with DVFS and c-states in latency sensitive workloads to yield useful trade-offs. Both these works build an application specific prediction model of the time that is required to service requests and take a specification of the required tail latency SLA target. The controller strategies exploit a combination of delaying processing and slowing down to find an optimal setting that ensures acceptable tail latencies while reducing the energy consumption. The intuition is that there are advantages to using batching to consolidate idle time, therefore lengthening the time that the processor is in deeper sleep states and additional energy savings from slowing down processing given the latency slack.

A common feature of modern high speed NICs is the ability to delay the delivery of interrupt when an event such as packet arrival or transmission completion occurs. By manipulating this setting, software can limit the minimum time between interrupts or in other words the maximum rate at which the NIC events can interrupt the processor. The NIC used in this study exposes this mechanism via an Interrupt Throttling (ITR) setting [1]. Software uses the ITR register to configure a delay in $2\mu s$ increments. If the spacing of events, such as packet reception, is less than $2\mu s \times ITR$ the NIC will delay assertion. If on the other hand events are sufficiently separated an interrupt will be asserted immediately. By default the Linux device driver attempts to automatically set this interrupt delay value to reduce interrupt overheads. We disable this feature and manually control its value to explore the impact of delaying interrupts on performance and energy. Delaying interrupts introduces an additional control that can interact with OS code and packetized payloads induced by Message Transmission Unit (MTU) constraints. If a request requires several MTU's then delaying interrupts can help reduce the interrupt processing overheads. Similarly, this can also induce prolonged quiescence periods in which processor idle policies can take advantage of.

³We have explored shallower c-states but focused on C7 as it had the most energy savings with minimal performance degradation.

⁴Opposed to big-little or re-configurable core architectures.

3.3 Interaction with Specializing OS Paths

In this study, a baremetal library OS is used to reveal the value of OS path specialization against a general purpose OS, we believe this will help further motivate the adaption of specialization even in general purpose systems. OS specialization for a single application means that in figure 1, all of the Request Servicing in the timeline is affected. As the system no longer needs to support other processes and multiplex different devices, the entire software can be dedicated towards one use-case, furthermore, more application work can thus be done per instruction. In the case of a service oriented workload that has significant application work, such that the fraction of the instructions composed by OS network processing is small, there is a potential for improved performance and energy. Customized OS paths can both reduce the time spent in the OS processing and improve the application code IPC (fig. 9) due to reduction in architectural hazards associated with interrupts, protection domain crossing, etc. This time reduction can also increase the utility of using DVFS and delaying interrupts to find optimal settings for different workloads.

Specialization also enables exploring alternate policies such as removing the Request Detection, Quiescent, and Idle Policy categories from figure 1 altogether to keep the CPU always busy with a polling loop. This alternative policy simplifies the complex control problem of managing interrupt delays and sleep states usage, in addition, given that polling is a CPU operation and will interact with processor speed settings, it is thus possible that with efficient OS paths and a slowed processor via DVFS, polling can be used to find a more energy efficient way of supporting different workloads (§5.1.3 §5.2.1).

4 Experiment Setup

4.1 Hardware Platform

Our experimental cluster consists of seven nodes, each having 16-core processors of either Intel(R) Xeon(R) CPU E5-2690 @2.90GHz or Intel(R) Xeon(R) CPU E5-2650 @2.60GHz type. All processors have Intel 82599ES 10-Gigabit SFI/SFP+ NICs, and are configured with a mix of 126 GB and 250 GB RAM. The node used to boot into the baremetal library OS, EbbRT, and Linux uses a Intel(R) Xeon(R) CPU E5-2690 @2.90GHz processor with 126 GB of RAM. While the hardware used in this study are not modern, the two mechanisms used are still commonly supported [4, 29, 66].

We ensured hardware hosting Linux and the library OS are setup in a similar way by carefully configure IA-32 Architectural MSRs and processor specific MSRs (see Tables 35-2 and 35-18 in [36]) as well as NIC features: direct-cache injection (DCA) disabled, receive-side scaling (RSS) enabled (to distribute packets for multi-core processing), and hardware checksum offloading enabled. We also match the values of the number of NIC transmit and receive descriptors and

write-back thresholds for packet transmissions. Additionally, to minimize system noise, hyperthreads and TurboBoost are disabled on all processors. While prior studies have included TurboBoost in performance-energy studies [15, 28, 47, 75], there have also been reports of energy anomalies when used with different sleep states [52].

4.2 OS Software

4.2.1 Linux. We build a set of application-specific Linux *appliances* for the four workloads listed in table 1. These appliances are specially constructed to run a RAM-based filesystem and contain only a small set of system libraries and kernel modules required to run their constituent applications. We construct these appliances from a base Debian 10.4 distribution and use a custom 5.5.17 kernel which we built using a modified configuration file created for supporting high performance; following suggestions from previous work that studied Linux core operation costs [80]. To avoid scheduling overheads and noise, we pin all applications to physical cores. In addition, we disable Linux *irqbalance* and affinitize packet receive interrupts to their respective cores.

4.2.2 Library OS. We ported EbbRT [82], an open sourced library OS, to run baremetal by developing a device driver for the Intel 82599 NIC [38]. EbbRT is used as a platform for exploring specialization of OS paths and our findings are extensible to other systems since EbbRT shares similar structural properties (such as run-to-completion, event-driven execution model, single execution domain, and compile-time optimization) with other high performance OSes and systems developed for accelerating network workloads [3, 11, 41, 45, 58, 61, 62, 68–70, 74, 76, 78, 79, 91].

EbbRT consists of specialized components written in C++⁵, with a NIC driver, a custom TCP/IP stack, virtual and physical memory allocators⁶, and a generic I/O buffer⁷. It is packaged as a library of configurable modules and *gcc-5.3.0*-based tool-chain targeting the base components of the OS.

Applications are ported to it by configuring the necessary OS components and compiling the application source along with any dependent libraries using this tool-chain. This generates a single application-specific binary that is compile and link-time optimized with the OS code. Our port enables application-specific binaries to boot directly on our hardware platform. Once booted, OS and application code is executed under a single supervisor privilege domain.

Given the design and implementation for single-application, non-preemptive processing via an optimized OS and application binary, library OS components can avoid many checks and streamline execution, ranging from interrupt dispatch

⁵All components are multi-core functional and optimized to aggressively use per-core memory and fine grain locking.

⁶Memory allocators make aggressive use of large pages and pinned memory to avoid page-faults.

⁷I/O buffers are designed to enable zero-copy application data processing.

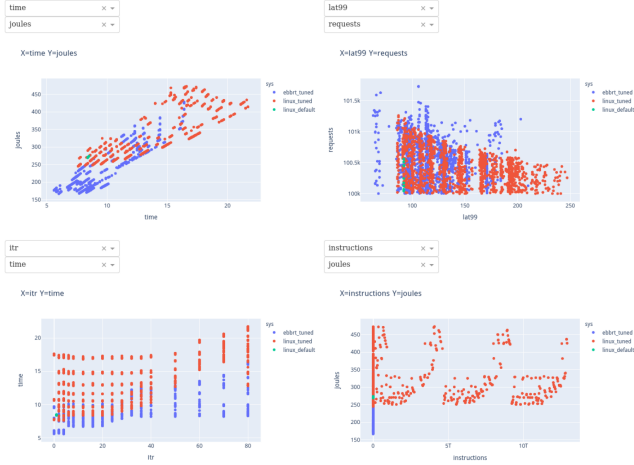


Figure 2. Small example of web visualization tool.

to application logic. The NIC device driver totals over 3000 lines of code and interfaces with EbbRT’s multi-core TCP/IP network stack⁸. EbbRT provides an interface for statically setting interrupt delay values. We use this interface in our study as we sweep across interrupt delay values. The NIC driver also exposes a configurable constant (set to 64 for all our experiments) that is used to control how many packets can be processed in a single interrupt invocation before returning to the event-loop of the core on which the interrupt was processed. This behaviour, in turn, introduces a simple bounded per-cpu device-level poll.

4.2.3 NIC polling without sleep. The simple run-to-completion, and lightweight event-driven execution model of EbbRT allows us to also explore the performance-energy trade-offs of slowing down the processor in the context of a polling loop for packet processing. We use standard techniques to auto clear hardware interrupts and enable a tight polling loop. The loop checks a in-memory data structure in which the NIC updates whenever new packet descriptors are to ready be processed. Due to this tight loop, EbbRT will never halt the processor and thus will not use any sleep states.

4.3 Per-Interrupt Log Collection and Visualization

In order to better understand the interactions of interrupt-delay and DVFS under a workload, we instrument fine-grained per-interrupt log collection in both Linux and EbbRT’s network device driver. We collect the following information in the NIC’s interrupt handler code: received and transmitted bytes, received and transmitted descriptors, sleep state statistics, and the current timestamp (via rdtsc instruction). In addition, we instrument per-core performance monitoring counters (PMCs) to collect a set of hardware statistics

⁸The device driver programs the NIC using per-cpu queues and interrupts, maintaining the affinity of TCP connections to their respective cores.

after every millisecond⁹ of elapsed time: instructions, cycles, last-level cache misses, and standard RAPL hardware registers on Intel processors to read per package energy values [35] as it has been experimentally validated for accuracy in previous works [18, 19, 46, 94]. While we have validated results against rack-level energy measures (slowing DVFS and interrupt delay resulted in rack level energy savings), we use RAPL instead because the granularity of the rack level measurements (on the order of seconds) made it difficult to attribute detailed energy use to specific system events.

Given these collected log traces, we built a web visualization tool using Dash [72] that enables a user to dynamically examine system behaviour across a wide range of configurable settings, for example, figure 2 shows how one can view the data at different dimensions (via dropdown boxes) of interrupt delay value, processor frequency, instructions, cycles, time, etc. With a fine-grained log trace, we also used the tool to zoom in on specific events that transpired in-between hardware interrupts to 1) gain better insights at a fine-grained manner, and 2) to generalize these insights into broader findings as will be discussed in §5. Having this tool gave us the ability compare and contrast different OS behaviors and was also immeasurably helpful to visually understand the structure in the data.

5 Experimental Analysis

The main methodology we used in our study is that of manually setting DVFS and interrupt delay values to all possible values as an exhaustive search to find optimal performance and energy trade-off points for both Linux and the library OS, EbbRT. We refer to these OS setups as *Linux-tuned* and *LibOS-tuned*, respectively, in the figures below. To better understand the degree of trade-offs in Linux, we also ran experiments on a base configuration which we refer to as *Linux-default*; in this mode, Linux’s interrupt delay and processor speed are both controlled dynamically by its built-in policies [20, 37]. We also explored a version of slowing down the processor by replacing network interrupts with a polling loop (whereby no sleep states are used) in EbbRT; we refer to this setup as *LibOS-poll* in the figures.

Figures 3, 5, 8 shows overviews of all the experimental runs gathered across the different applications and their respective loads as listed in table 1. Each data point represents a single experimental run and each experiment is repeated ten times for stability. For each workload, we break down the trade-offs observed by manually setting DVFS and interrupt delay in both OSes into measurements of performance (e.g. time for closed-loop workloads and 99% tail latency for open-loop workloads) and measurements of energy use. In order to reason about these trade-offs, we use two graphical mechanisms to highlight the differences:

⁹The millisecond gap is due to sampling granularity of RAPL.

1. The *size* of each point represents the degree with which interrupt delay is used; the *larger* the size, the more interrupt delay value is *increased* while the *smaller* the size the more it is *decreased* (e.g. faster IO interrupts).
2. The *color gradient* of each point represents the degree of slowing down processor speeds; the **darker** the color the more the processor has been *slowed* (less energy use) and vice-versa *faster* when the color is *lighter* (more energy use).

Finally, for each of the system configurations studied, the configuration that yields the best performance and lowest energy is indicated with **+** and **X** respectively.

5.1 Closed Loop Workloads

Figure 3 illustrates the set of closed-loop workloads that we study, all of which are run on a single core with a single connection. Netpipe [84] involves sending messages of identical size between two systems for a fixed number of iterations. We run Netpipe in a symmetric configuration, whereby the client and server sides run the same software stack and are configured with the same performance parameters. This approach allows us to analyze Netpipe performance precisely, as it eliminates any potential ambiguity in the measurements that may arise from computational differences between the communicating client and server. While Netpipe isn't a realistic workload in the datacenter, it allows us to explore different message sizes, opening up the scope of how the DVFS and interrupt delay affects time and energy. Linux runs NetPIPE-3.7.1 while the library OS uses a custom version ported to its interfaces. We fix the iteration count at 5000 and show results for a range of message sizes¹⁰. As message size increases, the workload becomes more network bound; Linux suffers an additional memory copy from kernel to userspace compared to the libOS.

NodeJS [42] consists of a JavaScript HTTP Webserver running inside a nodejs runtime. A single client running the *wrk-4.0.2* [26] benchmark¹¹ sends requests to the server for a fixed period of time. The server responds to each request with a small static payload of size 148 bytes. Linux runs nodejs-0.10.46, and the library OS runs the same version ported to support baremetal nodejs by providing OS interfaces that link with the V8 [27] JavaScript engine and libuv [57].

Given the nature of closed loop workloads, one would ideally minimize both time and energy. Therefore we use a single value, the product of time and energy, as a measure of energy efficiency for comparison between the two OSes.

5.1.1 Reducing time to save energy for small payloads.

One mechanism to reduce time across all the closed loop workloads is to always use a low interrupt delay value as shown in figure 4(c). With small payloads, reducing time

equates to reducing energy (see figure 4(b)) as well. For nodejs and netpipe 64B, setting a low interrupt delay ($2\mu s$) resulted in efficiency improvements by 2X in Linux-tuned and a further 85% in LibOS-tuned. This is due to the lightweight nature of the payloads and in this case, simply getting the work done fast leads to best energy efficiency.

5.1.2 Effects of interrupt delay induced batching on performance and energy.

Polling (dark datapoints) results in best energy efficiency for NodeJS and NetPIPE at 64 B, 8 KB message sizes and for interrupt-based configurations, a fast interrupt delay value is used. In contrast, as netpipe payload sizes increased to 8KB, 64KB, and 512KB, the interrupt delay value that yielded best energy efficiency also became slower (up to $28\mu s$ at 512 KB). A 10 GbE NIC, assuming no network jitter and switching cost, can transmit at an optimal rate of 1250 bytes/ μs . Therefore, the interrupt delay value can be used to effectively determine how much payload the software should process in a fixed quantum. With larger message sizes, one can imagine portions of its payload being transmitted over the wire and processed by software asynchronously. The interrupt delay value that yields best efficiency is indicating a "sweet spot" with which the software should pace packet processing and save energy by sleeping during the its quiescent periods.

By adjusting interrupt delay values in accordance with particular payload sizes, Linux-tuned exhibits improved energy efficiency over Linux-default up to 80%. Due to specialization of the library OS, figure 4(a) shows that it always uses fewest instructions, even in computationally heavy workloads such as nodejs. This efficiency, coupled with a custom interrupt delay, enables LibOS-tuned to improve its energy efficiency over Linux-tuned by another 2X.

5.1.3 Trade-offs in library OS polling. We compare the performance and energy trade-offs between slowing down the processor while the library OS is in a polling loop (LibOS-poll) and slowing down both processor and interrupt delay (LibOS-tuned). For nodejs, LibOS-poll only results in a 4% better energy efficiency than that of LibOS-tuned, primarily due to nodejs runtime already using an application-level poll to check for new packets.

The difference in energy efficiency for LibOS-poll is quite dramatic as message size increases in netpipe. With 64B and 8KB message sizes, polling improves this efficiency by 1.6X and 3X respectively over LibOS-tuned (11X over Linux-default). This is because for smaller payload sizes, getting the work done fastest results in the lowest energy use and mirrors the explanation in §5.1.1. At 64KB and 512 KB, the workload becomes more network bound and as a result, polling results in worse energy efficiency by up to 2X compared to LibOS-tuned. At these larger message sizes, polling only reduced time by around 10% while energy consumption increased over 2X than interrupt-driven LibOS-tuned, which is indicative that packets spent more time on the wire than in

¹⁰We found that the 10 GB link is close to saturation when a message of size greater 700 KB is exchanged.

¹¹We modified *wrk* to place a fixed request load of 100K.

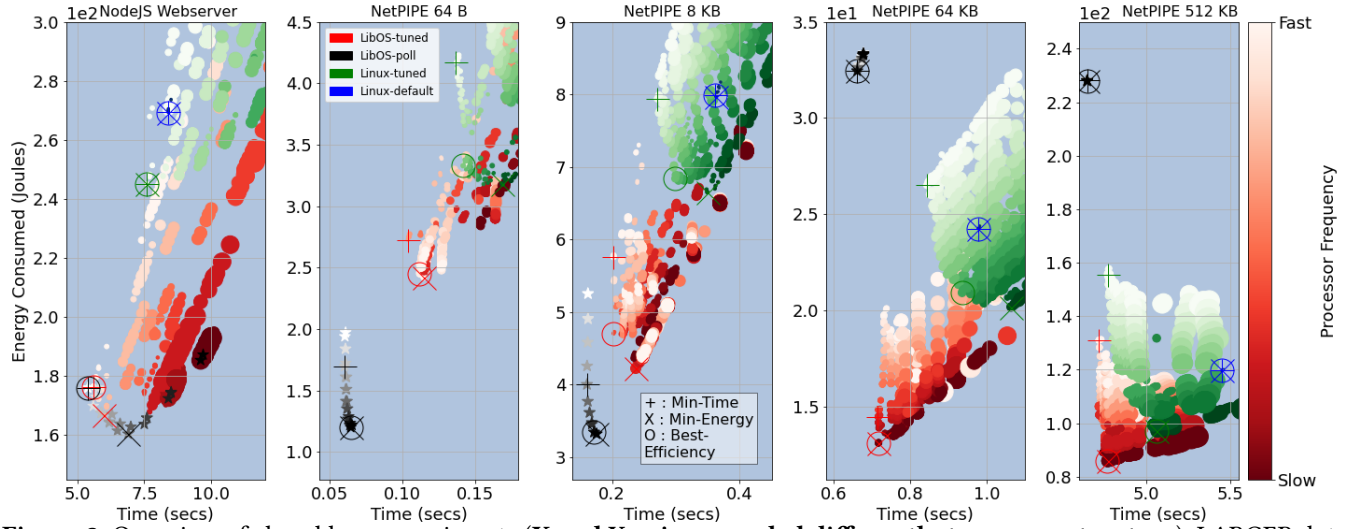


Figure 3. Overview of closed loop experiments (**X and Y axis are scaled differently to expose structure**). LARGER dots uses SLOW interrupt delay; Other visual cues: A **darker** color indicates **slow** processor frequency; **X** indicates lowest energy consumption; + indicates lowest time spent; **O** indicate best energy efficiency. Polling (grey-black datapoints) results in best energy efficiency for NodeJS and NetPIPE at 64 B, 8 KB message sizes. For interrupt-based configurations, a fast interrupt delay value is used to achieve best efficiency. At 64 KB and 512 KB for NetPIPE, best efficiency uses slow DVFS with a slow interrupt delay value that trades off speed and energy savings most efficiently. Polling largely inefficient at 64 KB and 512 KB.

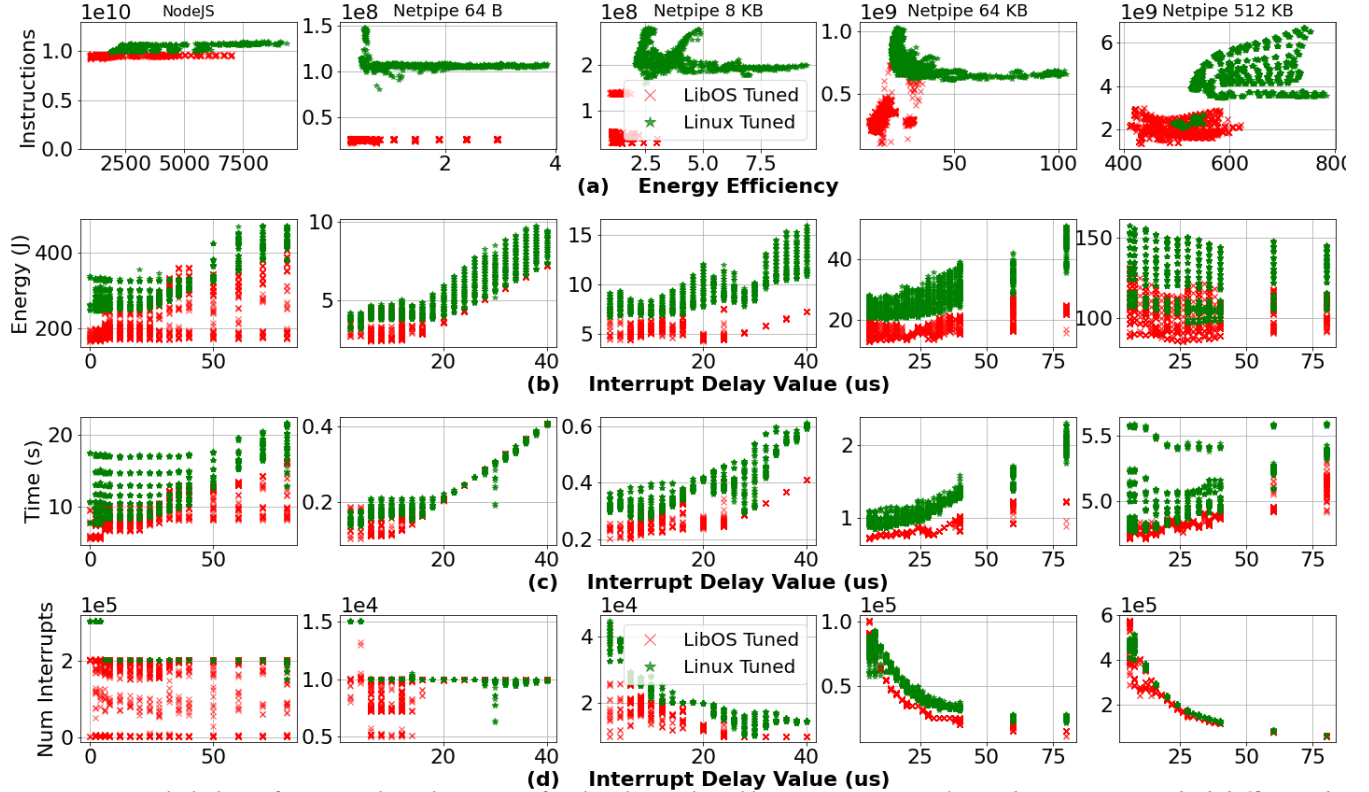


Figure 4. Detailed plots of some gathered statistics for the above closed loop experiments. (**X and Y axis are scaled differently to expose structure**) These plots only compare LibOS-tuned and Linux-tuned. All figures in a vertical slice is mapped to the same workload. Though (a) shows similar amount of instructions in both OSes, (b)(c) shows LibOS-tuned using both less energy and time to finish same workload, indicating efficiency of path specialization. In (d) for NodeJS and Netpipe 64 B, the big differences in interrupts is a result of slow-to-stay-busy effect as described in §5.1.4.

software. This phenomena suggests the importance of a hybrid strategy that switches between poll and interrupt-driven OS policies as payload size changes.

5.1.4 Overlapping work with IO. In both nodejs and netpipe with message size 64B, we find another interesting effect. In figure 4(d), for all the various interrupt delay values, the total number of interrupts can be lowered by 90%. Upon closer examination, we find slowed DVFS caused this decrease in number of interrupts.

The reason for this behavior in the library OS is described briefly in §2.3.2: The physical transmission of OS reply packets by the network driver can occur asynchronously with the unwinding of the stack back to the nodejs application and then back down to the network receive function to check for new packets. The slowing down of the processor causes this unwind path to lengthen, potentially increasing the probability that new packets have already arrived ready to be processed by the time it reaches the network receive function. Therefore, the software is able to skip one or more hardware interrupts (fired on packet receive) in order to effectively *slow-to-stay-busy* and process this new reply packet. This scenario only occurs in the library OS due to its run-to-completion nature and suggests that, for a structurally different OS, other energy saving strategies can be explored.

5.2 Open Loop Workloads

5.2.1 Memcached. This is a multi-threaded open loop workload that runs on all 16 cores of a server node [32]. It consists of an unloaded client node running mutilate [40]. This client (1) coordinates with five other mutilate agent nodes in order to generate requests to the server and (2) measures tail latency of all requests made. All five agent nodes are 16-core machines, whereby each core creates 16 connections, for a total of 1280 connections. This setup is able to saturate the single 16-core server¹².

Linux runs memcached-1.6.6 and the library OS version uses a re-implemented version of memcached, written to EbbRT's interfaces, and supports the standard memcached binary protocol. We run a representative load from Facebook [6] (ETC) which represents the highest capacity deployment. It uses 20 to 70 byte keys and 1 byte to 1 KB values and contains 75% GET requests.

Impacts of Slowing Down in Different OSes Structures:

fig. 6 shows that although LibOSTuned has worse IPC than Linux-tuned across the three QPS loads; it used on average 2.5X fewer instructions than Linux, which implies a greater fraction of its instruction were spent getting the work done. Furthermore, given that memcached is not compute heavy, most of its instructions are therefore memory bound; this also lowers the effect of a slowed processor to increase tail

latency. The vertical nature of the library OS in fig. 5 and fig. 7 illustrates this behavior. Furthermore, this suggests the logic and data structures used by a specialized library OS results in instruction mixes that can take advantage of energy saving benefits of slowed DVFS without sacrificing performance.

Figure 7 also shows that the library OS can support higher QPS loads than Linux due to its specialized paths. We see the opposite of this behavior in Linux where, at 600K QPS, it approaches 75% of its peak QPS. There is also a clear trade-off between slowing down processor speeds and an increase in tail latency (higher latency points have darker gradient color). In figure 7, memcached is scaled higher to 1500K QPS, which is 75% of the peak QPS of library OS. At this QPS rate, we can begin to see similar trade-offs in both OSes.

Impact of Slowing Down Processors on Sleep States:

Figure 6(a) shows that as a processor slows down, the energy savings from slowing down interrupts also decrease. In this figure, bold lines indicate the mean energy use at fastest interrupt delay, while dotted lines indicate mean energy use at the slowest. We find that across the QPS loads and the two OSes, the average energy savings from slowing down interrupt delay at the *slowest* processor speed is 52 J while it is 342 J at the *fastest* processor speed. As discussed in §3.1, the effect of slowing down the processor results in the lengthening of the application and OS work. Therefore, this potentially reduces the energy savings that are brought about by taking advantage of sleep states during prolonged idle periods. Furthermore, such slow-down is undesirable due to SLA requirements which result in stringent time budgets that requests must adhere to.

However, figure 5 shows that it is a combination of slow DVFS and interrupt delay that results in lowest energy use across both Linux and the library OS. Figure 6(d) shows the direct effect of interrupt delay on tail latency. The benefit of slowing down interrupts consists of 1) lowering the number of interrupts fired, which also lowers instruction use and potentially promotes better packet coalescing (see figures 6(b)(c)), and 2) ensuring a guaranteed period of quiescence such that the processor can take advantage of potentially deeper sleep states. However, these trade-offs will be different dependent on other factors such as an OS's packet processing efficiency and policies that govern the use of sleep states to maximize idle states. Moreover, the benefit of slowing down interrupts versus processor speed is subtle as the implications of slowing down the processor affects entire software stack whereas interrupt delay has a fixed impact.

Benefits of Interrupt Delays on Performance:

Figure 5 also demonstrates the ability to use a static fast interrupt delay value in order to minimize tail latency (smaller dots). Linux-tuned improved its tail latency over Linux-default by 40% at 200K QPS and 25% at 600K QPS. As discussed in section 2.2.2, a faster interrupt delay can induce a form of polling with Linux's NAPI policy by constantly waking up

¹²Mutilate is configured to pipeline up to four connections to further increase its request rate.

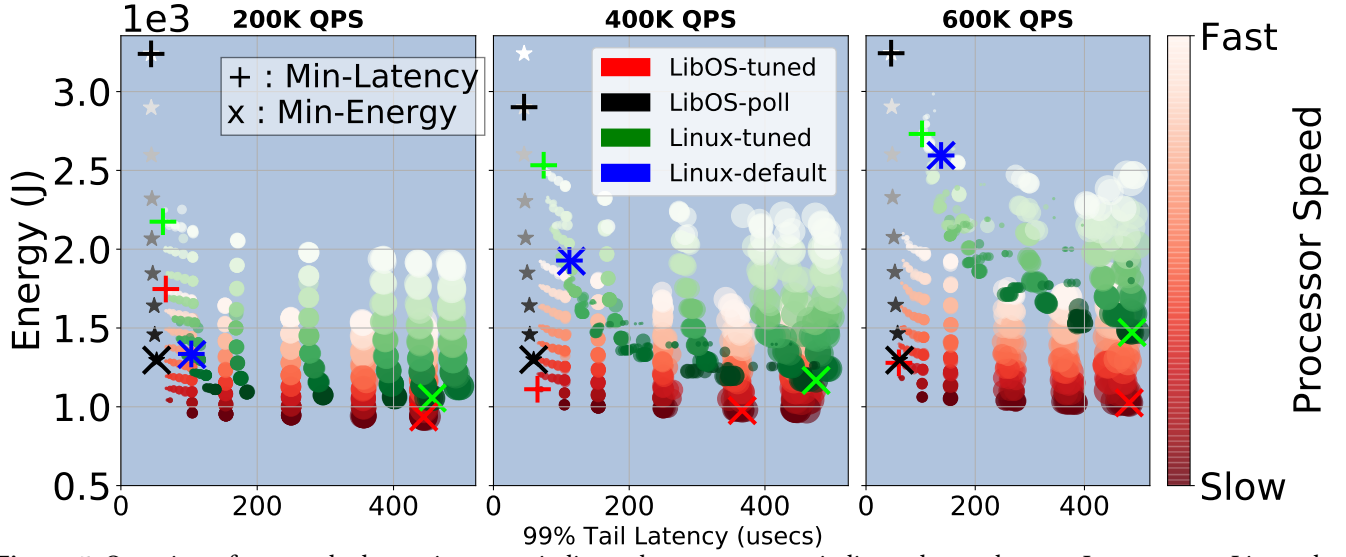


Figure 5. Overview of memcached experiments. \times indicates lowest energy. $+$ indicates lowest latency. In contrast to Linux, the vertical nature of LibOS-tuned shows its efficiency and instructions less impacted by slowed DVFS. Distance from Min-Latency to Min-Energy in contrast to Linux-default shows potential in exploring trade-offs through coordinated use of both mechanisms.

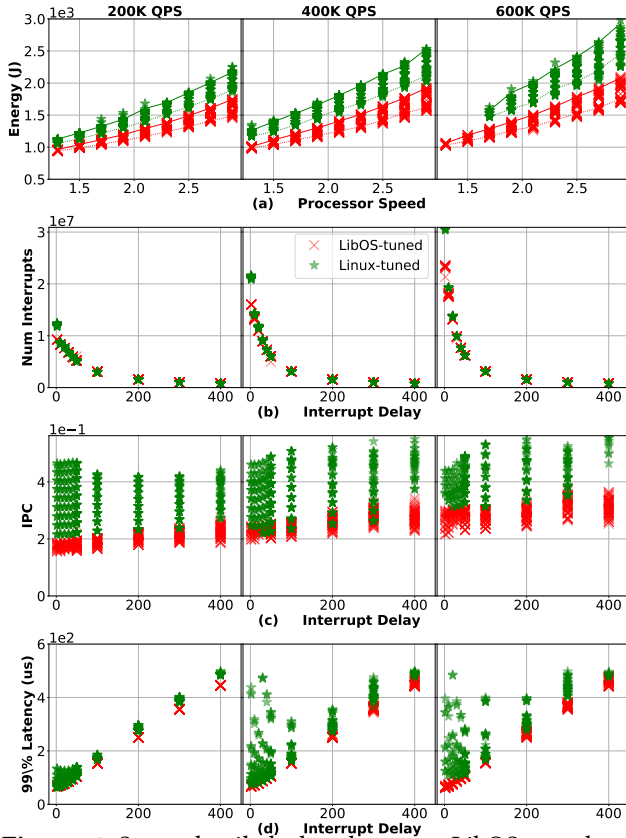


Figure 6. Some detailed plots between LibOS-tuned and Linux-tuned in memcached. Each vertical figure is mapped to the same QPS. In (a), bold lines connect points that use fastest interrupt delay and dashed lines use slowest interrupt delay; this shows additional energy savings induced by batching. In (d), one can see stability of tail latency values of LibOS-tuned compared to Linux-tuned.

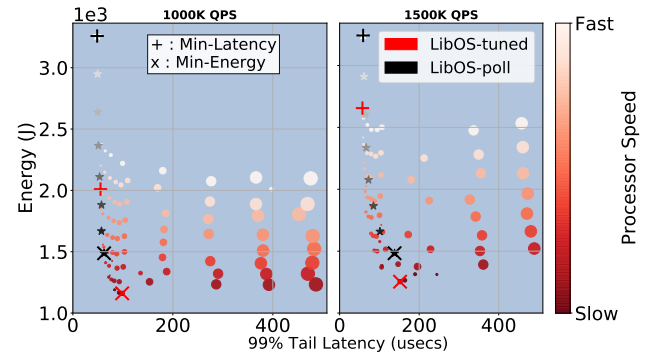


Figure 7. Overview of memcached experiments at 1000K and 1500K QPS. Vertical nature of library OS largely holds even at higher QPS loads and polling still competitive with interrupt-based in both tail latency and energy.

the processor to do the OS and application work. This induced behavior also increases energy use by 38% at 200K QPS and 5% at 600K QPS which represents another space in the energy-performance trade-off of memcached. While prior research have used static setting of a fast interrupt delay value for experimental stability [68, 70], we are the first to show its energy implications.

Polling Can be Energy Efficient: Similar to Netpipe with 64B messages (see figure 3), figure 5 shows that using an OS poll for network-bound workloads with a small payload results in the best performance (tail latency) in memcached. Although memcached is a more complex workload than Netpipe with thousands of connections and requests pipelined and multiplexed on multiple cores, LibOS-poll running memcached can still be energy efficient through slowing down the processor. Using \times point (Min-Energy) of LibOS-poll as

reference, we find that at 200K QPS, LibOS-poll improves both 99 percentile tail latency by 27% and energy by 35% over LibOS-tuned at Min-Latency point. As the load increases, by comparing the Min-Energy points, we find that while LibOS-poll consumes 11%-38% more energy across the rest of QPS loads, its tail latency was 10%-90% better. Hence a library OS poll reveals an additional trade-off space for energy and performance in memcached.

5.2.2 Memcached-silo. This is a workload built on top of the normal memcached protocol. It is more intensive both in computation and memory-use than regular memcached, as it is structured such that every memcached request triggers a corresponding set of TPC-C transaction processing logic on a in-memory database [89]. We ported the memcached-silo implementation from [73, 74] to EbbRT. The workload mix and SLA constraints of memcached-silo follow from those used in the memcached experiments in §5.2.1. However, given its heavier application nature, we only needed two 16-core client nodes at 16 connections per core to saturate a single 16-core memcached-silo server.

Slowing Down has Benefits Even in Computationally Heavy Workloads:

In contrast to memcached workload shown in figure 5, figure 8 shows that as the application work gets larger for each memcached request; the trade-offs of tail latency and energy become more discernible in both Linux and the library OS (note the color gradient darkens horizontally instead of vertically). Surprisingly even in a computationally heavy workload under a stringent SLA, it is still possible to delay both processor and interrupts to further save energy; in figure 8 at 200K QPS, Linux-tuned improved its tail latency by 21% and energy by 20% over Linux-default, further LibOS-tuned improved its tail latency by 34% and energy by 44% over Linux-tuned. Figure 9(a) shows that slowing the processor via DVFS results in the most energy savings and using interrupt delays on top leads to additional savings. However, in contrast to figure 6(a), the effects of delayed interrupts are greatly diminished as the QPS load increases given the workload nature. One can see this effect in the 100K and 200K QPS experiments where the configuration that yielded Min-Energy for the library OS is represented by a significantly small dot, implying a fast interrupt delay value. This observation harkens back to the implications of a slowed processor on the packet processing path (§3.1); furthermore, this effect is exacerbated as application logic gets heavier.

Library OS IPC Efficiency: Even though figure 9(b) demonstrates similar number of instructions between Linux-tuned and the LibOS-tuned, the IPC measurement in figure 9(c) reveals that a specialized OS can execute instructions more efficiently even in an application bound workload. This IPC benefit not only leads to energy efficiency but also creates more slack for the library OS to take advantage of slowed down processor. Figure 9(a) shows that at faster processor

speed values in the 50K and 100K QPS experiments, the library OS was still able to slow down interrupt delays to save more energy than Linux by 50% even in a computationally heavy workloads.

6 Related Work

Our work falls within a wider space of research on energy proportional computation in datacenters [9, 22, 88]. Much of this research stems from the challenges of improving the performance of network-bound datacenter workloads like MapReduce [14] and in-memory key-value stores [58, 74] while keeping energy consumption at bay. These challenges can be attributed to the complex diurnal trends that are characteristic of datacenter-level utilization, whereby idle time is common and must be optimized for [50, 63, 87] while simultaneously maintaining the ability to support high-utilization peaks and strict latency constraints [5, 15, 16, 30, 31, 44, 55, 59, 65, 74, 90, 92, 93]. Our work examines both in-memory key-value stores and its modified version with a heavier processing component as well as closed loop applications. Our goal was to gain better insight into the systemic impacts of performance and energy when slowing down network workloads using the two hardware mechanisms of DVFS and ITR delay together.

There is a wide range of work that targets energy proportionality with a focus on designing OS policies and mechanisms for power management. Most of this work presents hardware level optimizations that manipulate processor speed mechanisms such as DVFS [17, 20, 21, 23–25, 39, 48, 49, 52–54, 56, 81, 85], processor power limiting mechanisms such as RAPL [28, 31, 35, 59, 60, 71, 92], and idle power states [5, 15, 43, 47, 65, 77] (c-states) by applying feedback control mechanisms and relying on activity models. The authors of [60] and [28] go a step further, exploring and characterizing the interference of co-located latency-critical versus best-effort tasks and high versus low CPU demand tasks when subject to energy tuning via DVFS and RAPL. In doing so, they highlight limitations in using hardware features alone for power management. Similarly, the authors of [33, 87] identify a need to step away from relying entirely on hardware solutions and focusing instead on software optimizations, such as VM migration controllers for power management of an ensemble of nodes. Previous works have advocated for full-system and hardware optimizations for energy [52, 63], our work builds on their observations and assert that the OS itself plays a big role as well.

The previous research efforts present significant energy savings from well designed dynamic policies and carefully chosen static configurations, however, we are driven to explore the space beyond current findings with a focus on unveiling the role of the OS in exploiting activity and idleness and also by introducing interrupt delay as an additional knob in this exploration. We find that this exploration is timely

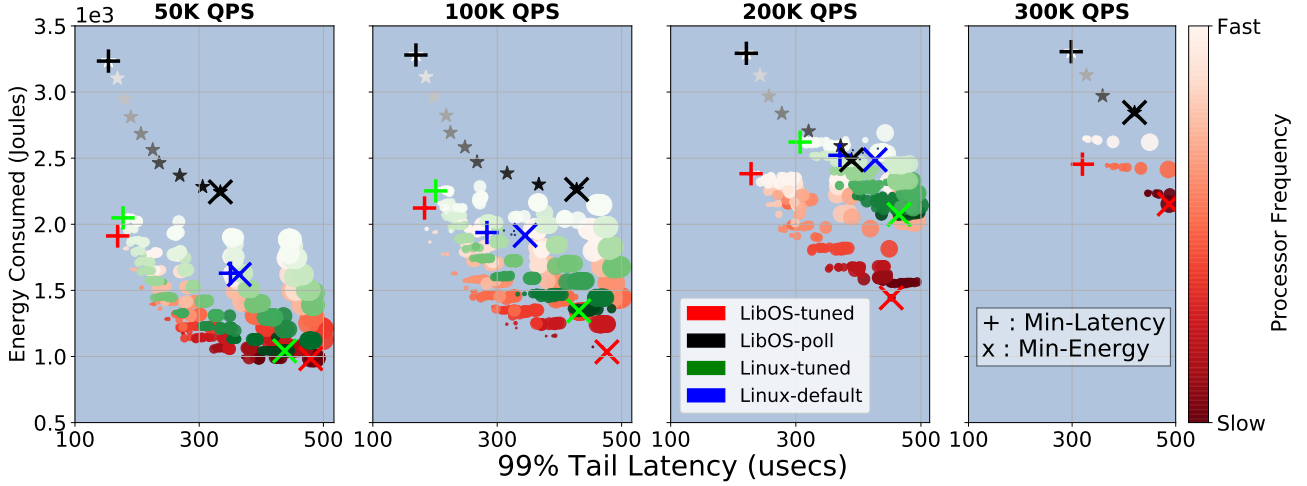


Figure 8. Overview of memcached-silo experiments across 50K, 100K, 200K and 300K QPS. As application work gets larger, horizontal nature of points indicate slowed processor affects latency more, however, can still explore performance and energy benefits with DVFS and interrupt delay induced batching in both LibOS-tuned and Linux-tuned. The base tail latency of this workload is larger than memcached due to nature of TPC-C application work. Polling is largely energy inefficient compared to interrupt-based but is competitive in tail latency.

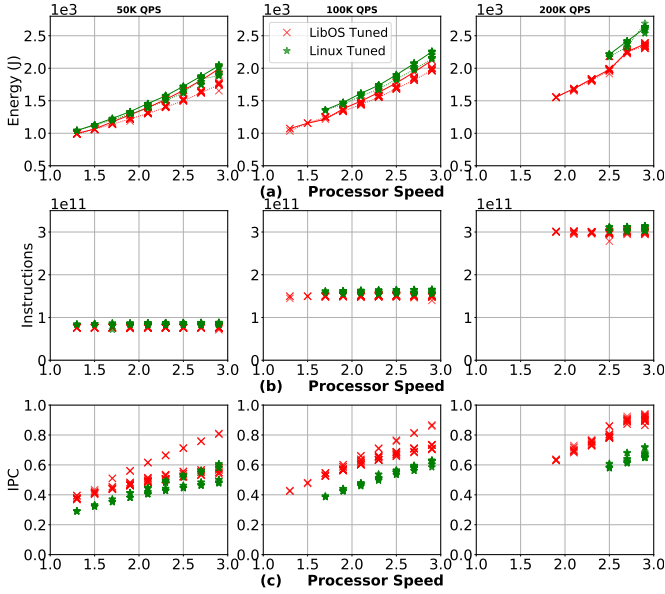


Figure 9. Detailed plots of some gathered statistics of the above QPS loads. Compares LibOS-tuned and Linux-tuned. Each vertical figure is mapped to the same QPS. In (c), with an application heavy workload, LibOS-tuned actually achieves better IPC than Linux-tuned even though (b) shows similar amount of instructions; indicating more work done per instruction for a specialized OS.

given the range of work on optimizing OS paths for performance, from NIC driver mechanisms [10, 45, 69] to the network stack [10, 41, 62] and the dataplane [11, 68, 70, 74, 82]. Our work was also influenced by previous work in energy

efficiency by slowing down both the networking and processor: μ DPM [15] is a application-level policy for memcached to delay request processing and maximize idle periods where deep sleep states can then be utilized, in [51] the authors combined bandwidth limiting in Cray clusters and scaling processor frequency to reduce energy use of HPC applications. In contrast to μ DPM, we use a hardware register on the NIC to induce batching as this can be commonly found in commercial NICs. Lastly, we are the first to conduct such an in-depth study with a baremetal specialized OS.

7 Conclusion

In this study, we conducted tens of thousands of experimental runs and accumulated over 5 TB of data. Our data also includes fine-grained timeline of various hardware/software statistics at a per-interrupt granularity (not heavily discussed in paper). This paper only scratches the surface of what can be distilled from the data and we plan to open source both the tools and data so that others can use this methodology to enable the systems research community to explore and frame performance results in context of energy as advocated by Mudge et al [67].

References

- [1] [n.d.]. Improving Measured Latency in Linux for Intel® 82575/82576 or X540/82598/82599 Ethernet Controllers. <https://www.intel.com/content/www/us/en/embedded/products/networking/82575-82576-82598-82599-ethernet-controllers-latency-appl-note.html>.
- [2] 2012. When Poll Is Better than Interrupt. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*. USENIX Association, San Jose, CA. <https://www.usenix.org/conference/fast12/when-poll-better-interrupt>

- [3] Antti Kantee, Justin Cormack. [n.d.]. Rump Kernels: No OS? No Problem! <https://www.usenix.org/publications/login/october-2014-vol-39-no-5>.
- [4] ARM. [n.d.]. <https://developer.arm.com/documentation/den0013/d/Power-Management>.
- [5] Esmail Asyabi, Azer Bestavros, Erfan Sharafzadeh, and Timothy Zhu. 2020. Peafowl: In-Application CPU Scheduling to Reduce Power Consumption of in-Memory Key-Value Stores. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) (SoCC '20). Association for Computing Machinery, New York, NY, USA, 150–164. <https://doi.org/10.1145/3419111.3421298>
- [6] Atikoglu, Berk and Xu, Yuehai and Frachtenberg, Eitan and Jiang, Song and Paleczny, Mike. 2012. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (London, England, UK) (SIGMETRICS '12). ACM, New York, NY, USA, 53–64. <https://doi.org/10.1145/2254756.2254766>
- [7] Luiz Andre Barroso, Jeffrey Dean, and Urs Hölzle. 2003. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro* 23 (2003), 22–28.
- [8] Luiz Andre Barroso and Urs Hölzle. 2009. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines* (1st ed.). Morgan and Claypool Publishers.
- [9] Barroso, Luiz André and Hölzle, Urs. 2007. The Case for Energy-Proportional Computing. *Computer* 40, 12 (Dec. 2007), 33–37. <https://doi.org/10.1109/MC.2007.443>
- [10] A. Beifuß, D. Raumer, P. Emmerich, T. M. Runge, F. Wohlfart, B. E. Wolfinger, and G. Carle. 2015. A Study of Networking Software Induced Latency. In *2015 International Conference and Workshops on Networked Systems (NetSys)*. 1–8. <https://doi.org/10.1109/NetSys.2015.7089065>
- [11] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) (OSDI'14). USENIX Association, USA, 49–65.
- [12] David M. Brooks, Pradip Bose, Stanley E. Schuster, Hans Jacobson, Prabhakar N. Kudva, Alper Buyuktosunoglu, John-David Wellman, Victor Zyuban, Manish Gupta, and Peter W. Cook. 2000. Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors. *IEEE Micro* 20, 6 (Nov. 2000), 26–44. <https://doi.org/10.1109/40.888701>
- [13] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) (EuroSys '20). Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages. <https://doi.org/10.1145/3342195.3392698>
- [14] Yanpei Chen, Sara Alspaugh, Dhruva Borthakur, and Randy Katz. 2012. Energy Efficiency for Large-Scale MapReduce Workloads with Significant Interactive Analysis. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) (EuroSys '12). Association for Computing Machinery, New York, NY, USA, 43–56. <https://doi.org/10.1145/2168836.2168842>
- [15] C. Chou, L. N. Bhuyan, and D. Wong. 2019. Î/DPM: Dynamic Power Management for the Microsecond Era. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 120–132. <https://doi.org/10.1109/HPCA.2019.00032>
- [16] Chih-Hsun Chou, Daniel Wong, and Laxmi N. Bhuyan. 2016. DynSleep: Fine-Grained Power Management for a Latency-Critical Data Center Application. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design* (San Francisco Airport, CA, USA) (ISLPED '16). Association for Computing Machinery, New York, NY, USA, 212–217. <https://doi.org/10.1145/2934583.2934616>
- [17] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. 2011. Pack & Cap: Adaptive DVFS and Thread Packing under Power Caps. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) (MICRO-44). Association for Computing Machinery, New York, NY, USA, 175–185. <https://doi.org/10.1145/2155620.2155641>
- [18] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: Memory Power Estimation and Caping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design* (Austin, Texas, USA) (ISLPED '10). Association for Computing Machinery, New York, NY, USA, 189–194. <https://doi.org/10.1145/1840845.1840883>
- [19] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. 2016. A Validation of DRAM RAPL Power Measurements. In *Proceedings of the Second International Symposium on Memory Systems* (Alexandria, VA, USA) (MEMSYS '16). Association for Computing Machinery, New York, NY, USA, 455–470. <https://doi.org/10.1145/2989081.2989088>
- [20] Dominik Brodowski, Nico Golde, Rafael J. Wysocki, Viresh Kumar. [n.d.]. CPU frequency and voltage scaling code in the Linux(TM) kernel. <https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt>.
- [21] Mootaz Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. 2003. Energy Conservation Policies for Web Servers. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4* (Seattle, WA) (USITS'03). USENIX Association, USA, 8.
- [22] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. 2007. Power Provisioning for a Warehouse-Sized Computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture* (San Diego, California, USA) (ISCA '07). Association for Computing Machinery, New York, NY, USA, 13–23. <https://doi.org/10.1145/1250662.1250665>
- [23] Krisztián Flautner, Steve Reinhardt, and Trevor Mudge. 2001. Automatic Performance Setting for Dynamic Voltage Scaling. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking* (Rome, Italy) (MobiCom '01). Association for Computing Machinery, New York, NY, USA, 260–271. <https://doi.org/10.1145/381677.381702>
- [24] Vincent W. Freeh, Tyler K. Bletsch, and Freeman L. Rawson. 2007. Scaling and Packing on a Chip Multiprocessor. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–8. <https://doi.org/10.1109/IPDPS.2007.370539>
- [25] Rong Ge, Xizhou Feng, Wu-chun Feng, and Kirk W. Cameron. 2007. CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters. In *2007 International Conference on Parallel Processing (ICPP 2007)*. 18–18. <https://doi.org/10.1109/ICPP.2007.29>
- [26] Will Glozer. 2014. wrk: Modern HTTP benchmarking tool. <https://github.com/wg/wrk>.
- [27] Google. [n.d.]. V8 JavaScript Engine. <http://code.google.com/p/v8/>.
- [28] Akhil Guliani and Michael M. Swift. 2019. Per-Application Power Delivery. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) (EuroSys '19). Association for Computing Machinery, New York, NY, USA, Article 5, 16 pages. <https://doi.org/10.1145/3302424.3303981>
- [29] Nathan Hanford, Vishal Ahuja, Matthew K. Farrens, Brian Tierney, and Dipak Ghosal. 2018. A Survey of End-System Optimizations for High-Speed Networks. *ACM Comput. Surv.* 51, 3, Article 54 (July 2018), 36 pages. <https://doi.org/10.1145/3184899>
- [30] C. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski. 2015. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 271–282.

- [31] Chang-Hong Hsu, Qingyuan Deng, Jason Mars, and Lingjia Tang. 2018. SmoothOperator: Reducing Power Fragmentation and Improving Power Utilization in Large-scale Datacenters. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Williamsburg, VA, USA) (ASPLOS '18). ACM, New York, NY, USA, 535–548. <https://doi.org/10.1145/3171362.3171390>
- [32] <https://memcached.org>. 2020. Memcached. <https://github.com/memcached/memcached>.
- [33] Inkwon Hwang and Massoud Pedram. 2016. A Comparative Study of the Effectiveness of CPU Consolidation Versus Dynamic Voltage and Frequency Scaling in a Virtualized Multicore Server. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 24, 6 (2016), 2103–2116. <https://doi.org/10.1109/TVLSI.2015.2499601>
- [34] Intel. [n.d.]. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume. <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/>.
- [35] Intel. [n.d.]. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2. <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf>.
- [36] Intel. [n.d.]. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3. <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.pdf>.
- [37] Intel. [n.d.]. Tuning Throughput Performance for Intel® Ethernet Adapters. <https://www.intel.com/content/www/us/en/support/articles/000005811/network-and-i-o/ethernet-products.html>.
- [38] Intel 82599 10 Gigabit Ethernet Controller: Datasheet. [n.d.]. <https://www.intel.com/content/www/us/en/embedded/products/networking/82599-10-gbe-controller-datasheet.html>.
- [39] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. 2006. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, USA, 347–358. <https://doi.org/10.1109/MICRO.2006.8>
- [40] J. Leverich. [n.d.]. Mutilate: high performance memcached load generator. <https://github.com/leverich/mutilate>.
- [41] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and Kyoungsoo Park. 2014. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 489–502. <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong>
- [42] Joyent. 2013. Node.js. <https://github.com/nodejs/node/tree/cc56c62ed879ad4f93b1fdab3235c43e60f48b7e>.
- [43] Svilen Kanev, Kim Hazelwood, Gu-Yeon Wei, and David Brooks. 2014. Tradeoffs between power management and tail latency in warehouse-scale applications. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 31–40. <https://doi.org/10.1109/IISWC.2014.6983037>
- [44] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. 2015. Rubik: Fast analytical power management for latency-critical systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 598–610.
- [45] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) (ASPLOS '16). Association for Computing Machinery, New York, NY, USA, 67–81. <https://doi.org/10.1145/2872362.2872367>
- [46] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. 2018. RAPL in Action: Experiences in Using RAPL for Power Measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.* 3, 2, Article 9 (March 2018), 26 pages. <https://doi.org/10.1145/3177754>
- [47] David H. K. Kim, Connor Imes, and Henry Hoffmann. 2015. Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics. In *Proceedings of the 2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA '15)*. IEEE Computer Society, USA, 78–85. <https://doi.org/10.1109/CPSNA.2015.23>
- [48] Wonyoung Kim, Meeta S. Gupta, Gu-Yeon Wei, and David Brooks. 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 123–134. <https://doi.org/10.1109/HPCA.2008.4658633>
- [49] Masaaki Kondo, Hiroshi Sasaki, and Hiroshi Nakamura. 2007. Improving Fairness, Throughput and Energy-Efficiency on a Chip Multiprocessor through DVFS. *SIGARCH Comput. Archit. News* 35, 1 (March 2007), 31–38. <https://doi.org/10.1145/1241601.1241609>
- [50] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. 2011. NapSAC: Design and Implementation of a Power-Proportional Web Cluster. *SIGCOMM Comput. Commun. Rev.* 41, 1 (Jan. 2011), 102–108. <https://doi.org/10.1145/1925861.1925878>
- [51] James H. Laros, Kevin T. Pedretti, Suzanne M. Kelly, Wei Shu, and Courtenay T. Vaughan. 2012. Energy Based Performance Tuning for Large Scale High Performance Computing Systems (HPC '12). Society for Computer Simulation International, San Diego, CA, USA, Article 6, 10 pages.
- [52] Etienne Le Sueur and Gernot Heiser. 2011. Slow down or Sleep, That is the Question. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference* (Portland, OR) (USENIXATC'11). USENIX Association, USA, 16.
- [53] Jungseob Lee and Nam Sung Kim. 2009. Optimizing Throughput of Power- and Thermal-Constrained Multicore Processors Using DVFS and per-Core Power-Gating. In *Proceedings of the 46th Annual Design Automation Conference* (San Francisco, California) (DAC '09). Association for Computing Machinery, New York, NY, USA, 47–50. <https://doi.org/10.1145/1629911.1629926>
- [54] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. 2007. Server-Level Power Control. In *Fourth International Conference on Autonomic Computing (ICAC'07)*. 4–4. <https://doi.org/10.1109/ICAC.2007.35>
- [55] Jacob Leverich and Christos Kozyrakis. 2014. Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) (EuroSys '14). Association for Computing Machinery, New York, NY, USA, Article 4, 14 pages. <https://doi.org/10.1145/2592798.2592821>
- [56] J. Li and J.F. Martinez. 2006. Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *The Twelfth International Symposium on High-Performance Computer Architecture*, 2006. 77–87. <https://doi.org/10.1109/HPCA.2006.1598114>
- [57] libuv. [n.d.]. <http://libuv.org>.
- [58] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 429–444. <https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim>
- [59] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads. In *Proceeding of the 41st Annual*

International Symposium on Computer Architecture (Minneapolis, Minnesota, USA) (ISCA '14). IEEE Press, 301–312.

- [60] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. *SIGARCH Comput. Archit. News* 43, 3S (June 2015), 450–462. <https://doi.org/10.1145/2872887.2749475>
- [61] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) (ASPLOS '13). ACM, New York, NY, USA, 461–472. <https://doi.org/10.1145/2451116.2451167>
- [62] Ilias Marinos, Robert N.M. Watson, and Mark Handley. 2014. Network Stack Specialization for Performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (Chicago, Illinois, USA) (SIGCOMM '14). ACM, New York, NY, USA, 175–186. <https://doi.org/10.1145/2619239.2626311>
- [63] David Meisner, Brian T. Gold, and Thomas F. Wenisch. 2009. PowerNap: Eliminating Server Idle Power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) (ASPLOS XIV). Association for Computing Machinery, New York, NY, USA, 205–216. <https://doi.org/10.1145/1508244.1508269>
- [64] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. 2011. Power Management of Online Data-Intensive Services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (San Jose, California, USA) (ISCA '11). Association for Computing Machinery, New York, NY, USA, 319–330. <https://doi.org/10.1145/2000064.2000103>
- [65] David Meisner and Thomas F. Wenisch. 2012. DreamWeaver: Architectural Support for Deep Sleep. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) (ASPLOS XVII). Association for Computing Machinery, New York, NY, USA, 313–324. <https://doi.org/10.1145/2150976.2151009>
- [66] Mellanox. [n.d.]. <https://community.mellanox.com/s/article/understanding-interrupt-moderation>.
- [67] T. Mudge. 2001. Power: a first-class architectural design constraint. *Computer* 34, 4 (2001), 52–58. <https://doi.org/10.1109/2.917539>
- [68] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) (NSDI'19). USENIX Association, USA, 361–377.
- [69] Aleksey Pesterev, Jacob Strauss, Nikolai Zeldovich, and Robert T. Morris. 2012. Improving Network Connection Locality on Multi-core Systems. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) (EuroSys '12). Association for Computing Machinery, New York, NY, USA, 337–350. <https://doi.org/10.1145/2168836.2168870>
- [70] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2015. Arrakis: The Operating System Is the Control Plane. *ACM Trans. Comput. Syst.* 33, 4, Article 11 (Nov. 2015), 30 pages. <https://doi.org/10.1145/2812806>
- [71] P. Petoumenos, L. Mukhanov, Z. Wang, H. Leather, and D. S. Nikolopoulos. 2015. Power Capping: What Works, What Does Not. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*. 525–534. <https://doi.org/10.1109/ICPADS.2015.72>
- [72] plotly. [n.d.]. Dash Overview. <https://plotly.com/dash/>.
- [73] George Prekas. 2017. <https://github.com/ix-project/servers/tree/master>.
- [74] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) (SOSP '17). Association for Computing Machinery, New York, NY, USA, 325–341. <https://doi.org/10.1145/3132747.3132780>
- [75] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. 2015. Energy Proportionality and Workload Consolidation for Latency-Critical Applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (Kohala Coast, Hawaii) (SoCC '15). Association for Computing Machinery, New York, NY, USA, 342–355. <https://doi.org/10.1145/2806777.2806848>
- [76] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 145–160. <https://www.usenix.org/conference/osdi18/presentation/qin>
- [77] Rafael J. Wysocki. [n.d.]. CPU Idle Time Management. <https://www.kernel.org/doc/html/v5.0/admin-guide/pm/cpuidle.html>.
- [78] Rajesh Nishtala and Hans Fugal and Steven Grimm and Marc Kwiatkowski and Herman Lee and Harry C. Li and Ryan McElroy and Mike Paleczny and Daniel Peek and Paul Saab and David Stafford and Tony Tung and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 385–398. <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala>
- [79] Ali Raza, Parul Sohal, James Cadden, Jonathan Appavoo, Ulrich Drepper, Richard Jones, Orran Krieger, Renato Mancuso, and Larry Woodman. 2019. Unikernels: The Next Stage of Linux's Dominance. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) (HotOS '19). Association for Computing Machinery, New York, NY, USA, 7–13. <https://doi.org/10.1145/3317550.3321445>
- [80] Xiang (Jenny) Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. 2019. An Analysis of Performance Evolution of Linux's Core Operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) (SOSP '19). Association for Computing Machinery, New York, NY, USA, 554–569. <https://doi.org/10.1145/3341301.3359640>
- [81] Hiroshi Sasaki, Satoshi Imamura, and Koji Inoue. 2013. Coordinated Power-Performance Optimization in Manycores. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques* (Edinburgh, Scotland, UK) (PACT '13). IEEE Press, 51–62.
- [82] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. eBBRT: A Framework for Building Per-Applcation Library Operating Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 671–688. <https://www.usenix.org/conference/osdi16/technical-sessions/presentation/schatzberg>
- [83] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open versus Closed: A Cautionary Tale. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation - Volume 3* (San Jose, CA) (NSDI'06). USENIX Association, USA, 18.
- [84] Quinn O Snell, Armin R Mikler, and John L Gustafson. 1996. Netpipe: A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*.
- [85] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. 2011. Green Governors: A Framework for Continuously Adaptive DVFS. In *Proceedings of the 2011 International Green Computing Conference and Workshops (IGCC '11)*. IEEE Computer Society, USA, 1–8. <https://doi.org/10.1109/IGCC.2011.6008552>
- [86] The Linux Foundation. [n.d.]. napi. <https://wiki.linuxfoundation.org/networking/napi>.
- [87] Niraj Tolia, Zhikui Wang, Manish Marwah, Cullen Bash, Parthasarathy Ranganathan, and Xiaoyun Zhu. 2008. Delivering Energy Proportionality with Non Energy-Proportional Systems: Optimizing the Ensemble.

In *Proceedings of the 2008 Conference on Power Aware Computing and Systems* (San Diego, California) (*HotPower'08*). USENIX Association, USA, 2.

- [88] Niraj Tolia, Zhikui Wang, Manish Marwah, Cullen Bash, Parthasarathy Ranganathan, and Xiaoyun Zhu. 2008. Delivering Energy Proportionality with Non Energy-Proportional Systems—Optimizing the Ensemble. In *Workshop on Power Aware Computing and Systems (HotPower '08)*. USENIX Association, San Diego, CA. <https://www.usenix.org/conference/hotpower-08/delivering-energy-proportionality-non-energy-proportional-systems>
- [89] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farmington, Pennsylvania) (*SOSP '13*). Association for Computing Machinery, New York, NY, USA, 18–32. <https://doi.org/10.1145/2517349.2522713>
- [90] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T. N. Vijaykumar. 2015. TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) (*MICRO-48*). Association for Computing Machinery, New York, NY, USA, 585–597. <https://doi.org/10.1145/2830772.2830779>
- [91] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct. 2001), 230–243. <https://doi.org/10.1145/502059.502057>
- [92] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. 2016. Dynamo: Facebook's Data Center-wide Power Management System. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) (*ISCA '16*). IEEE Press, Piscataway, NJ, USA, 469–480. <https://doi.org/10.1109/ISCA.2016.48>
- [93] Xin Zhan, Reza Azimi, Svilen Kanev, David Brooks, and Sherief Reda. 2017. CARB: A C-State Power Management Arbiter for Latency-Critical Workloads. *IEEE Computer Architecture Letters* 16, 1 (2017), 6–9. <https://doi.org/10.1109/LCA.2016.2537802>
- [94] Huazhe Zhang and H Hoffman. 2015. A Quantitative Evaluation of the RAPL Power Control System. *Feedback Computing* (2015).

A Mathematical Framework

A byproduct of our detailed analysis via this wealth of data collected is that we have begun to create a useful framework for analyzing and exploring energy and performance impacts accounting for OS behavior. We believe this framework will be used as a backbone towards more automated optimization of hardware and OS settings. In this section we briefly summarize our work in using our request time-line breakdown (Figure 1) to develop a mathematical framework that can be used to explain and explore software and hardware effects. In particular, we show how we model an open loop setting, with an arrival rate of λ , and explore the impacts of changes in the instruction path length and composition. A more detailed discussion of the framework and how it can be used can be found in Appendix B.

Assuming a setting where the service time is less than or equal to the time between two requests, we define:

$$\delta t = t_{\text{detect}} + t_{\text{osreq}} + t_{\text{app}} + t_{\text{idlepolicy}} + t_q = \frac{1}{\lambda}$$

δt = time between the arrivals of two consecutive requests and the remaining terms directly reflect the time-line components.

We group together the three terms that have a clear DVFS dependence and define t_{work} as $t_{\text{osreq}} + t_{\text{app}} + t_{\text{idlepolicy}}$. Excluding detection and any quiescent time, defining t_{latency} as $t_{\text{detect}} + t_{\text{work}}$ for the total service time or latency for a request.

Similarly the total energy consumed during the inter-arrival time, δt is:

$$E = P_{\text{detect}}t_{\text{detect}} + P_{\text{work}}[t_{\text{osreq}} + t_{\text{app}} + t_{\text{idlepolicy}}] + P_q t_q = P_{\text{detect}}t_{\text{detect}} + P_{\text{work}}t_{\text{work}} + P_q t_q$$

An important aspect of our model is our physically motivated abstraction of a processor's DVFS setting, Δ . While it is a single value we model its ability to have an independent impact on time (as a possible function of frequency) and power (as a possible function of voltage and frequency). Specifically, we posit t_{work} and P_{work} as follows:

$$t_{\text{work}} = A \frac{N_i}{\Delta^{1+\alpha}} \text{ and } P_{\text{work}} = B \Delta^{2+\beta}$$

where A , B are constants of proportionality, N_i = the total number of instructions and α , β are real-valued constants that describe the dependence on DVFS. This allows us, through α and β , to explore effects in which different instruction mixes of the software, are affected in different ways with respect to time and energy by DVFS settings.

As an example of the use of the framework to generate simulated plots similar to figures shown above, in figure 10, plot a) shows that when instructions are less affected by frequency one expects to see a vertical structure. Plots b) and c) show that as the paths are composed of instructions that are more frequency sensitive DVFS changes result in a more curved structure in the energy vs latency. As expected slowing the instructions starts to affect latency in these instruction mixes. Additionally in all three plots one sees that delaying interrupts via ITR, for a busy fraction, increases latency with not much improvement in energy. Remembering that for this configuration deep sleep has been configured to use zero energy. In section 5.2.1 we will examine a scenario that arises in practice that displays similar behavior.

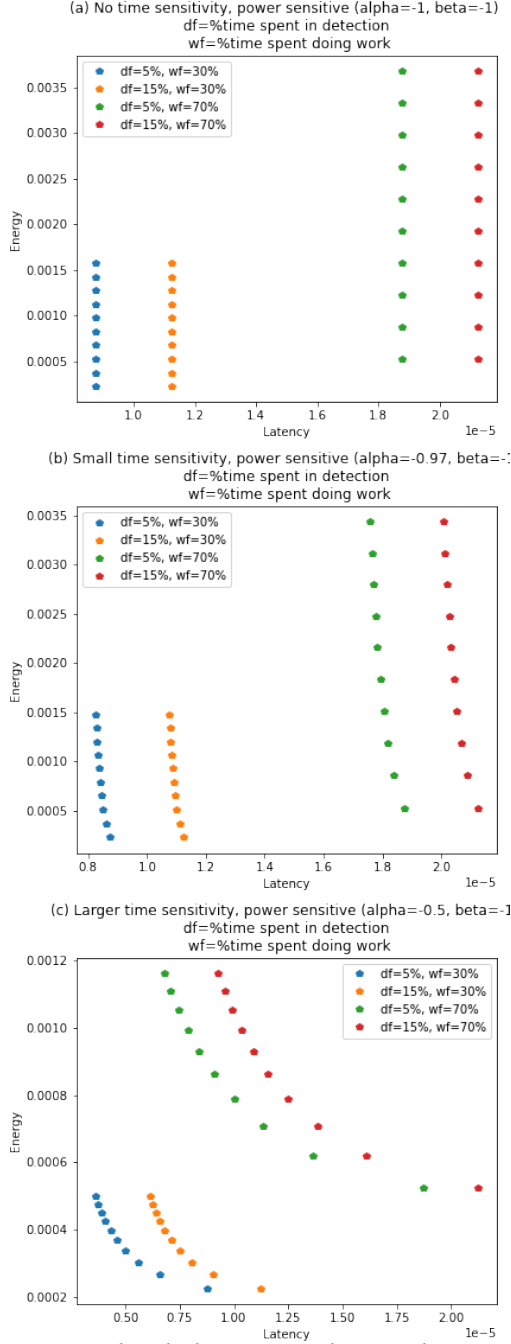


Figure 10. Simulated Plots. Note: The X and Y axis values are not realistic values but rather simulated values.

B Appendix: Mathematical Framework

Under the assumption that the various time-segments in figure 1 don't overlap, one can break down a full cycle from one request to another into disjoint time-intervals:

$$\delta t = t_{\text{detect}} + t_{\text{osreq}} + t_{\text{app}} + t_{\text{idlepolicy}} + t_q = \frac{1}{\lambda} \quad (1)$$

where, δt = time between the arrivals of two consecutive requests. The other terms represent the various intervals in figure 1.

Each term is treated as a deterministic and fixed quantity (dependent on workload, hardware, OS, parameters) as opposed to a random variable following some underlying probability distribution. This is sufficient for a qualitative treatment but a full quantitative treatment would treat these terms as part of a probabilistic graphical model.

Note that the interval δt is the interarrival time which is given by the reciprocal of the arrival rate (in queries/requests per unit time), λ . We will work in the regime where λ is low enough that each request is processed before the next request arrives. While the treatment above is for the open-loop setting, this restriction also applies to the closed loop setting where, by construction, the interarrival interval always exceeds the time spent processing a request.

To map this timeline decomposition to our experimental setup, we can group together some of the terms to define:

$$t_{\text{work}} = t_{\text{osreq}} + t_{\text{app}} + t_{\text{idlepolicy}}$$

$$t_{\text{latency}} = t_{\text{work}} + t_{\text{detect}}$$

to get:

$$\delta t = t_{\text{detect}} + t_{\text{work}} + t_q = t_{\text{latency}} + t_q$$

Intuitively, t_{work} is the time spent on processing the request outside the detection phase and outside any quiescent time, t_q , and t_{latency} is the time spent both in the detection phase and on processing for a given request.

Since the total time, δt is fixed ($= \frac{1}{\lambda}$), this implies that the quiescent time is,

$$t_q = \left[\frac{1}{\lambda} - (t_{\text{work}} + t_{\text{detect}}) \right]^+$$

where $[x]^+ = \max(x, 0)$ i.e. $[x]^+$ is the positive part of x .

In other words, if the arrival rate λ is small enough, there is an opportunity for the processor to enter a quiescent state ($t_q > 0$) but as the arrival rate increases, the time processing the request, t_{work} exceeds the inter-arrival gap leading to requests accumulating in the queue.

As stated above, these relationships also applies to the closed-loop case with the additional constraint that the arrival rate and thus the interarrival gap is no longer independent of t_{work} .

Given this time decomposition, one can compute the total energy consumed for each request as follows:

$$E = P_{\text{detect}} t_{\text{detect}} + P_{\text{work}} [t_{\text{osref}} + t_{\text{app}} + t_{\text{idlepolicy}}] + P_q t_q \quad (2)$$

The assumption is that there are three power regimes, one each for the detection phase, the work phase and the quiescent phase, respectively.

For the open-loop case, since we are studying energy, E vs latency, t_{latency} plots for various itr (t_{detect}) and DVFS values, we need to posit the dependence of these terms on DVFS. Suppose the workload needs N_i instructions. One would expect t_{work} to scale as:

$$t_{\text{work}} \propto \frac{N_i}{f}$$

where f = CPU frequency. Of course, there might be deviations from this behavior and one can posit a power law dependence,

$$t_{\text{work}} = A \frac{N_i}{f^{1+\alpha'}}$$

where A is a constant of proportionality and α' is an arbitrary parameter. $\alpha' = 0$ would fit the baseline case where time scales inversely with frequency. Since we control DVFS and not frequency directly, we can change this to

$$t_{\text{work}} = A \frac{N_i}{\Delta^{1+\alpha}}$$

where Δ = the chosen DVFS value and α is some scaling power that can be inferred from data. The other time values don't depend on DVFS in this simple model (although that assumption can be added in a straightforward way).

The total energy consumed depends on various power values which in turn can depend on DVFS. Here, we posit that P_{work} has a power law dependence on DVFS, Δ . To motivate this, the power consumed by a processor scales as:

$$P \propto V^2 f$$

where V = the operating voltage and f = CPU frequency. DVFS scales both voltage and frequency but not necessarily in a linear way. The general power law assumption is parameterized by a second parameter, β , as follows:

$$P_{\text{work}} = B \Delta^{2+\beta}$$

where B is a constant of proportionality and β can be unrestricted and is meant to be inferred from the data. Depending on the exact setup, it is possible that P_{detect} also scales with DVFS and in that case, we will set $P_{\text{detect}} = P_{\text{work}}$.

At a qualitative level, the two relationships,

$$\delta t = t_{\text{detect}} + t_{\text{work}} \left[= \frac{A N_i}{\Delta^{1+\alpha}} \right] + t_q \quad (3)$$

$$E = P_{\text{detect}} t_{\text{detect}} + P_{\text{work}} \left[= B \Delta^{2+\beta} \right] t_{\text{work}} \left[= \frac{A N_i}{\Delta^{1+\alpha}} \right] + P_q t_q \quad (4)$$

with the requirements that $\delta t = \frac{1}{\lambda}$ or equivalently, $t_q = \left[\frac{1}{\lambda} - [t_{\text{work}} + t_{\text{latency}}] \right]^+$ can be used to plot the behavior of energy consumed vs time (latency, total run-time) for various values of α and β .

We can plot some energy, E vs latency, t_{latency} curves numerically for the case:

$$P_{\text{detect}} = P_q = 0W$$

$$P_{\text{work}} = P_{\text{static}} + P_{\text{textmin}} \Delta^{2+\beta} = 10W + 20W \Delta^{2+\beta}$$

For a fixed interarrival time, δt , we assign a fraction f_{detect} to the detection phase and a maximum fraction $f_{\text{work}}^{\text{max}}$ to the work to get:

$$\begin{aligned} t_{\text{latency}} &= t_{\text{detect}} + t_{\text{work}} \\ &= f_{\text{detect}} \delta t + \frac{f_{\text{work}}^{\text{max}} \delta t}{\Delta^{1+\alpha}} \\ \Rightarrow \frac{t_{\text{latency}}}{\delta t} &= f_{\text{detect}} + \frac{f_{\text{work}}^{\text{max}}}{\Delta^{1+\alpha}} \end{aligned} \quad (5)$$

and

$$\begin{aligned} E &= P_{\text{detect}} t_{\text{detect}} + P_{\text{work}} t_{\text{work}} + P_q t_q \\ &= P_{\text{work}} t_{\text{work}} \\ &= (10W + 20W \Delta^{2+\beta}) \left(f_{\text{detect}} \delta t + \frac{f_{\text{work}}^{\text{max}} \delta t}{\Delta^{1+\alpha}} \right) \\ \Rightarrow \frac{E}{1W \delta t} &= (10 + 20 \Delta^{2+\beta}) \left(f_{\text{detect}} + \frac{f_{\text{work}}^{\text{max}}}{\Delta^{1+\alpha}} \right) \end{aligned} \quad (6)$$

For various choices of detection loads ($0 \leq f_{\text{detect}} \leq 1$), maximal work loads ($0 \leq f_{\text{work}}^{\text{max}} \leq 1$), time-scaling (α), power-scaling (β), we can plot energy (E) vs latency (t_{latency}) curves as DVFS (Δ) varies. An example is shown in figure 11. Power sensitivity to DVFS, β increases across columns (left to right) and time sensitivity to DVFS, α increases across rows (top to bottom).

While these qualitative plots indicate the ability of the model to replicate curves observed in real data, a natural next step would be to perform fits to infer the values of the model parameters including α and β and reason about them in terms of both the workload application and the underlying OS structure.

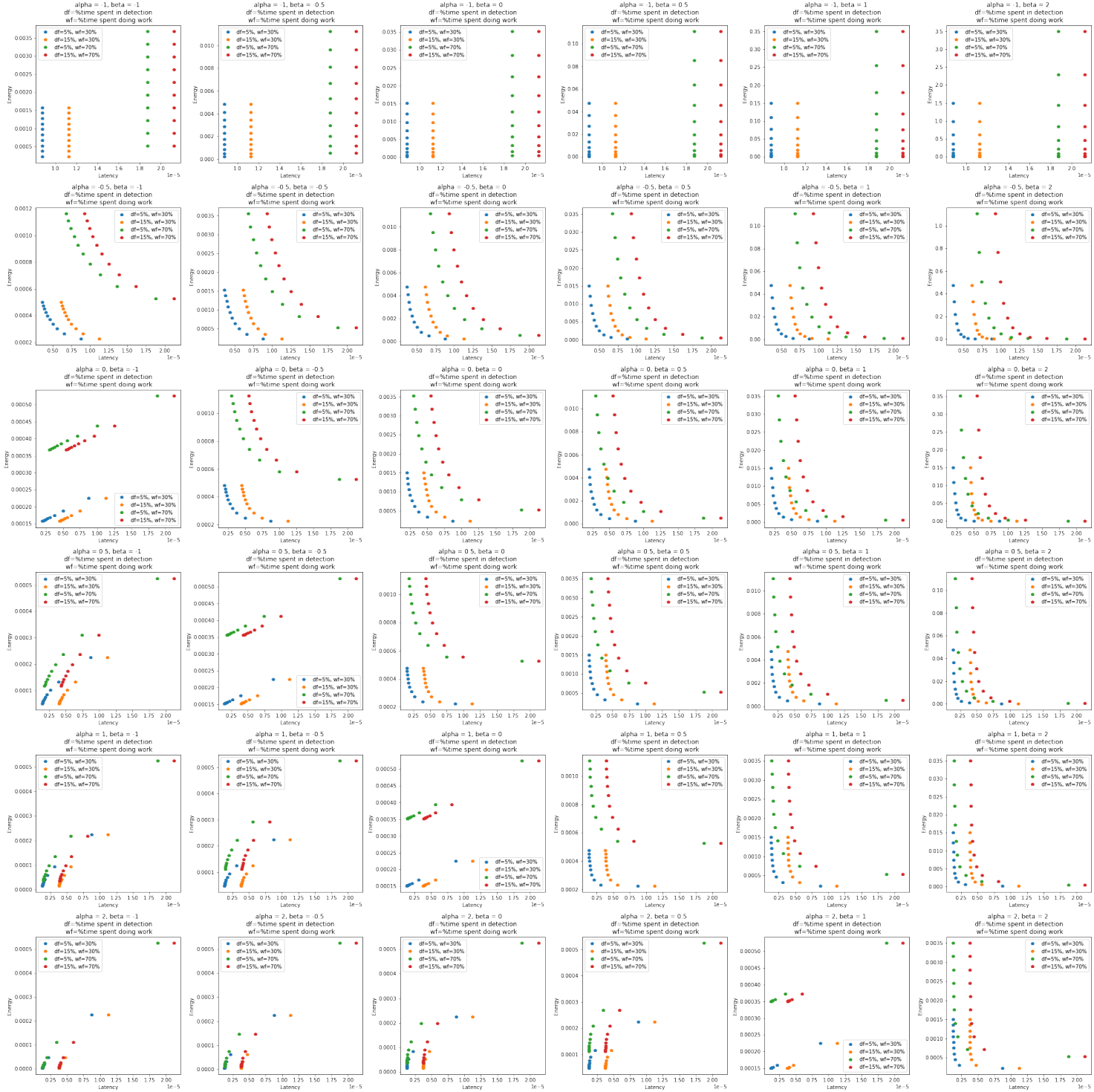


Figure 11. Simulated Energy vs Latency Curves for Open-Loop Workloads. Power sensitivity to DVFS increase across columns and time/latency sensitivity to DVFS increases across rows