BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

# A DATA-DRIVEN STUDY OF OPERATING SYSTEM ENERGY-PERFORMANCE TRADE-OFFS TOWARDS SYSTEM SELF OPTIMIZATION

by

**HAN DONG**

BS in Computer Science, University of Maryland Baltimore County, 2010
MS in Computer Science, University of Maryland Baltimore County, 2013

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2023

Approved by

First Reader

Jonathan Appavoo, PhD
Associate Professor of Computer Science

Second Reader

Orran Krieger, PhD
Professor of Electrical and Computer Engineering

Third Reader

Mark Crovella, PhD
Professor of Computer Science

Fourth Reader

Sanjay Arora, PhD
Senior Principal Software Engineer
Red Hat, Inc.

Fifth Reader

Mark Charney, PhD
Distinguished Engineer
Ampere Computing

# A DATA-DRIVEN STUDY OF OPERATING SYSTEM ENERGY-PERFORMANCE TRADE-OFFS TOWARDS SYSTEM SELF OPTIMIZATION

**HAN DONG**

Boston University, Graduate School of Arts and Sciences, 2023

Major Professor: Jonathan Appavoo, PhD

ABSTRACT

This dissertation is motivated by an intersection of changes occurring in modern software and hardware; driven by increasing application performance and energy requirements while Moore's Law and Dennard Scaling are facing challenges of diminishing returns. To address these challenging requirements, new features are increasingly being packed into hardware to support new offloading capabilities, as well as more complex software policies to manage these features. This is leading to an exponential explosion in the number of possible configurations of both software and hardware to meet these requirements.

For network-based applications, this thesis demonstrates how these complexities can be tamed by identifying and exploiting the characteristics of the underlying system through a rigorous and novel experimental study. This thesis demonstrates how one can simplify this control strategy problem in practical settings by cutting across the complexity through the use of mechanisms that exploit two fundamental properties of network processing.

Using the common request-response network processing model, this thesis finds that controlling 1) the speed of network interrupts and 2) the speed at which the re-

quest is then executed, enables the characterization of the software and hardware in a stable and well-structured manner. Specifically, a network device's interrupt delay feature is used to control the rate of incoming and outgoing network requests and a processor's frequency setting was used to control the speed of instruction execution. This experimental study, conducted using 340 unique combinations of the two mechanisms, across 2 OSes and 4 applications, finds that optimizing these settings in an application-specific way can result in characteristic performance improvements over 2X while improving energy efficiency by over 2X.

This thesis also discovers that both the model and characterization results are generic enough that an off-the-shelf sample efficient machine learning technique, Bayesian optimization, can be used to tackle the seemingly intractable problem of configuration space explosion. This thesis demonstrates how a Linux server can lower overall energy use while supporting a real-world in-memory key-value store workload over the course of 24 hours by using Bayesian optimization to automatically adapt to different request rates and performance and energy goals. This technique was able to search through a space of over 2 million possible configurations to yield operating points in Linux that resulted in energy savings of over 50%.

## CONTENTS

**Abstract** **iv**

**List of Tables** **xiii**

**List of Figures** **xv**

**List of Symbols and Abbreviations** **xx**

**1 Introduction** **1**

1.1 Problem Statement . . . . . 1

1.2 Challenges . . . . . 4

1.2.1 Energy . . . . . 4

1.2.2 Hardware . . . . . 5

1.2.3 Software . . . . . 9

1.3 Motivation . . . . . 11

1.4 Contributions . . . . . 11

1.4.1 Engineering Contributions . . . . . 13

1.4.1.1 In-situ data collection infrastructure . . . . . 13

1.4.2 Experimental Study Contributions . . . . . 14

1.4.2.1 Combining ITR and DVFS for performance and energy gains. . . . . 14

1.4.2.2 Energy study of baremetal Library OS . . . . . 15

1.4.2.3 Data-driven OS Specialization . . . . . 16

1.4.3 Experimental Analysis Contributions . . . . . 17

1.4.3.1 Developing a model to capture complex systems interactions . . . 17
1.4.4 Applying with Machine Learning . . . 18
1.4.4.1 Machine learning technique to automatically tune ITR and DVFS. . . . 18
1.5 Outline . . . 19

**2 *itrLog*: in-situ data collection infrastructure 20**
2.1 Attributes of data collection infrastructure . . . 21
2.1.1 Agnostic Infrastructure . . . 21
2.1.2 Epoch-based . . . 22
2.1.3 Data Logging Statistics . . . 24
2.1.4 Efficient Data Retrieval . . . 25
2.2 *itrLog* Details . . . 25
2.2.1 Linux Implementation . . . 26
2.2.1.1 Agnostic Infrastructure: . . . 26
2.2.1.2 Epoch-based . . . 28
2.2.1.3 Data Logging Statistics . . . 30
2.2.1.4 Efficient Data Retrieval . . . 34
2.2.2 EbbRT Implementation . . . 35
2.2.2.1 Agnostic Infrastructure . . . 35
2.2.2.2 Epoch-based . . . 36
2.2.2.3 Data Logging Statistics . . . 36
2.2.2.4 Efficient Data Retrieval . . . 37
2.3 Enabling Static Hardware Settings . . . 37
2.3.1 ITR . . . 37

2.3.2 DVFS . . . 39
2.4 Booting Baremetal OSes . . . 41
2.5 Data Collection Overview . . . 42
2.6 Visualization Tool . . . 43

**3 Experimental Setup** **45**
3.1 MOC Hardware Configuration . . . 45
3.2 Systems Software Stacks . . . 46
3.2.1 Linux Appliance . . . 46
3.2.2 EbbRT Library OS . . . 47
3.3 Application Workloads . . . 48
3.3.1 Closed Loop . . . 50
3.3.1.1 NetPIPE . . . 50
3.3.1.2 NodeJS HTTP Web Server . . . 52
3.3.2 Open Loop . . . 53
3.3.2.1 Memcached . . . 53
3.3.2.2 Memcached-Silo . . . 55

**4 Break Down of Network Processing in lieu of Analysis and Models** **57**
4.1 Quiescent Periods . . . 58
4.1.1 Idle Policies . . . 59
4.1.1.1 Linux Idle Policy . . . 59
4.1.1.2 EbbRT Idle Policy . . . 60
4.2 OS Request Detection . . . 60
4.2.1 Interrupt driven IO . . . 61
4.2.2 Poll driven IO . . . 61

4.2.3 Linux NAPI Policy . . . 62
4.2.4 EbbRT . . . 62
4.2.4.1 Interrupt-driven . . . 62
4.2.4.2 Poll-based . . . 63
4.3 Request Servicing . . . 63
4.3.1 Linux Network Processing . . . 64
4.3.2 EbbRT Network Processing . . . 64
4.4 Potential Performance-Energy Trade-offs In Different Applications . 65
4.4.1 Open-Loop . . . 65
4.4.2 Closed-Loop . . . 66
4.4.3 OS-centric . . . 67
4.4.4 Application-centric . . . 67

**5 Experimental Findings** **68**
5.1 Closed-Loop . . . 72
5.1.1 OS Changes . . . 72
5.1.2 ITR Changes . . . 72
5.1.2.1 **Detailed Finding 1: ITR can be used to induce packet processing stability in order to greatly improve performance.** . . . 74
5.1.3 Offered Load, ITR, and DVFS Changes . . . 78
5.1.3.1 **Detailed Finding 2: Combining batching with DVFS to enable energy efficient pacing of packet processing.** . . . 78
5.2 Open-Loop . . . 81
5.2.1 OS, Offered Load Changes . . . 81

5.2.2 ITR Changes . . . 82
5.2.2.1 **Detailed Finding 3: Using ITR to stabilize tail latency in open loop applications.** . . . 83
5.2.3 DVFS Changes . . . 86
5.2.3.1 **Detailed Finding 4: Combining DVFS and ITR to lower total energy use.** . . . 87
5.3 OS-centric . . . 89
5.3.1 OS, ITR, DVFS Changes . . . 89
5.3.1.1 **Detailed Finding 5: A specialized system has more headroom with DVFS to further reduce energy without sacrificing performance.** . . . 90
5.3.2 DVFS Changes . . . 93
5.3.2.1 **Detailed Finding 6: Polling can be energy efficient.** 93
5.4 Application-centric . . . 94
5.4.1 OS Changes . . . 94
5.4.1.1 **Detailed Finding 7: Energy-aware-slow-poll strategy in a run-to-completion OS.** . . . 95
5.4.2 Offered load, ITR, DVFS Changes . . . 96
5.4.2.1 **Detailed Finding 8: IPC Benefits Even in Computationally Heavy Applications.** . . . 97
5.5 Summary of Experimental Findings Towards Building a Model . . . 99

**6 Modeling the Experimental Data** **101**
6.1 Open Loop Model . . . 101
6.2 Closed Loop Model . . . 104
6.3 Model Fitting Results . . . 105

6.3.1 Open Loop Discussion . . . 110
6.3.1.1 Memcached . . . 110
6.3.1.2 Memcached-silo . . . 112
6.3.2 Closed Loop Discussion . . . 113
6.4 Model Limitations . . . 114

**7 Tuning with Machine Learning Techniques 115**
7.1 Summary of Bayesian Optimization . . . 116
7.2 Bayesian Optimization Applied to Experimental Study . . . 117
7.3 Bayesian Optimization Applied to Real-world Trace Data . . . 121
7.3.1 Experimental Setup . . . 122
7.3.2 Reward Function . . . 123
7.3.3 System Configurations . . . 124
7.3.4 Memcached Results . . . 126
7.3.5 Memcached-silo Results . . . 127
7.3.6 Bayesian Optimization Implications . . . 129
7.3.6.1 Deployment in Datacenters . . . 129
7.3.6.2 Reward Function . . . 130
7.3.6.3 SLA Objectives . . . 130
7.3.6.4 Learning Implicit Hardware Encoding . . . 130

**8 Related Work 132**

**9 Future Work 142**
9.1 Performance and energy study . . . 142
9.2 Energy reporting in systems research . . . 142
9.3 Specialized OSes and Network Path Optimizations . . . 143

9.4 NIC polling without sleep in specialized OS paths . . . . . 144
9.5 Configuring NICs for Performance . . . . . 145
9.6 Expanding on ITR-Delay mechanism . . . . . 146
9.7 Model Extension . . . . . 147

**10 Conclusion** **148**

**Bibliography** **151**

**Curriculum Vitae** **173**

## LIST OF TABLES

1.1 Some widely citepd historical research operating systems. . . . . 1

3.1 Workload configurations. The column *Nature* indicates open (OL) - versus-closed (CL) loop nature and *CPU* indicates application work demand. . . . . 48

5.1 Best static setting of ITR and DVFS across all applications and offered loads. For each cell the **(ITR, DVFS)** numbers represents the physical values used to find Perf (Highest performance) and Energy (Lowest energy) and the number below each ITR, DVFS setting is its corresponding performance (seconds for closed-loop applications and microseconds 99% tail latency measurement in open-loop applications) or energy in Joules. ITR units are in $\mu$s and DVFS units are in GHz. . . . . 71

6.1 Values for free parameters in memcached at different QPSes from doing fit with Adam optimizer. . . . . 109

6.2 Values for free parameters in memcached-silo at different QPSes from doing fit with Adam optimizer. . . . . 110

6.3 Values for free parameters in NetPIPE fromdoing fit with Adam optimizer. 114

8.1 List of related systems and the hardware parameters explored. **WOL** – Wake-On-Lan capability on certain NICs and an experimental feature. **CAT** – Cache Allocation Technology hardware feature on certain Intel CPUs. **TB** – Turbo-Boost. **CS** – C-states. . . . . 133

8.2 List of related works of applying ML to automatically configure various configurations. However, all these works were run in a simulator only. . . . . 139

8.3 List of related works of applying ML to automatically configure various configurations for datacenter scale applications. However, most of these works only on software settings only and do not have publicly available datasets. Lastly, they have applied to Linux only. . 141

9.1 Measured performance of each NIC feature configuration when running **EbbRT memcached**. TSO is separated into *single* and *multiple* categories, and for the rest of the hardware features (DCA, RSC), this table lists the peak QPS achieved for every combination while maintaining SLA of 99% tail latency < 500 $\mu s$. . . . . 146

## LIST OF FIGURES

2.1 Example figures that showcase the fidelity of fine-grained data collection of energy consumed (Joules) at a per interrupt level. . . . . 22

5.2 Pareto-optimal curves of closed loop applications. . . . . 73

5.3 Throughput measurements for NetPIPE across different message sizes in the three systems studied. The inset zooms in on message sizes between 64 B to 8 KB. . . . . 74

5.4 Three figures showing the bytes received per interrupt for **Linux** with dynamic ITR algorithm enabled when running NetPIPE @ 65536 Bytes. Each figure shows a distinct run of NetPIPE and demonstrates the dynamic behavior of Linux's ITR algorithm on the way packets are being processed (Y-axis) and overall time it takes to run a single experiment (X-axis). Out of the total of 10 different runs, we illustrate these three figures as after examining all 10 collected log datasets, we find performance of Linux with dynamic ITR mainly fluctuates between these three distinct behaviors. . . . . 75

5.5 ITR values set by Linux's dynamic ITR algorithm for a single experimental run of NetPIPE at 64 KB message size. . . . . 76

5.6 Three figures showing the bytes received per interrupt for **Linux-static** with ITR value at 10 $\mu s$ for a message size of 65536 Bytes in NetPIPE. The three figures each illustrate a distinct run of NetPIPE. . 77

5.7 Three figures showing the changes for energy (J) with different static ITR values used in both EbbRT and Linux across three different DVFS values from slowest (1.2 Ghz) to fastest (2.9 Ghz). Furthermore, the **X** markers indicate configurations that yielded best performance in order to illustrate the performance-energy trade-offs that exist in this application. **Note:** Y-Axis is not scaled to show structure of the two OSes. . . . . 80

5.9 The X-axis plots the static ITR values explored in both OSes and the Y-axis shows the measured 99%, 90%, and 50% latency in Memcached for the different QPS rates. The range of points along the vertical is indicative of different DVFS explored for each static ITR. This figure illustrates how ITR can be used to induce stability in tail latency measurements even at 99% for a dynamic Memcached workload and the stability is more pronounced in a specialized OS such as EbbRT in comparison to Linux where at fast ITR values, the different DVFS values used causes a larger difference in tail latency measures. . . . . 84

5.10 Using Linux-static as an example, the three figures show C-state counts in Memcached for different QPSes while running at a fixed DVFS of 2.5 Ghz. The X-axis shows for each C-state from C1 to C7 where C1 is the lightest and C7 is heaviest sleep state where architectural state such as caches are completely flushed. We show the count of how many times Linux's idle policy went into each sleep state given two different ITR values at a fast rate of 2 us and a slow rate of 300 us. The Y-axis are normalized against the counts of ITR at 2 us. These figures show that across the QPSes, a fast ITR rate of 2 us typically uses only C1 sleep state as it is the lightest and will be constantly woken up, as ITR increases to 300 us, the heavier sleep states begin to be used more to take advantage of the prolonged idle periods induced by the ITR mechanism. . . . . 85
5.11 Static ITR setting impact on total number of interrupts in Memcached. 86
5.13 The X-axis shows for each static DVFS setting and the Y-axis shows the measured total energy use across the two OSes. The vertical span of each DVFS setting is indicative of how different static ITR values impact energy use. The bold lines show the fastest ITR explored and the dotted line show the slowest ITR explored. These lines indicate how within a DVFS value that changing ITR also impacts energy consumption. . . . . 88
5.15 ITR impact on instruction count in memcached. Not drawn to scale in order to shown structure in data. . . . . 91

5.16 Timeline plot of non-idle ratio at per-interrupt basis for Linux-static and EbbRT-static that resulted in min energy for memcached @ 600K QPS. . . . . 92
5.17 DVFS impact on number of interrupts in NodeJS and Netpipe 64B. . . . . 95
5.18 All memcached-silo results. Note we don't have Linux results for 300K as the Linux could not support that offered load without violating the SLA objective. . . . . 96
5.19 The figures show collected hardware statistics for Memcached-silo across three QPS values. For consistency, we plot the data from the perspective of different fixed DVFS values. In (a), the Y-axis shows the count of total number of last-level cache misses between the two OSes. In (b), we illustrate that the total number of instructions executed is roughly the same even though the application runs on two different OSes. In (c), our results show that the EbbRT is executing instructions more efficiently than Linux even in a computationally heavy application. . . . . 98

6.1 Prediction of energy and performance using model for Netpipe at different message sizes. The Y-Axis consist of measured values (either performance or energy) and the X-AXis consists of predicted values using the constructed models. We draw diagonal lines and show if the dots (which are measured values) lie on the diagonal line, then it is an accurate fit of the model onto the data. . . . . 106
6.2 Prediction of energy and performance using model for Memcached. 107
6.3 Prediction of energy and performance using model for Memcached-silo. . . . . 108

7.1 Bayesian optimization for Memcached for 99% tail latency and energy. The X and Y axis represent unique ITR, DVFS pairs in a single experimental run and is also illustrated by every **O**. We show the samples that the Bayesian process undertook via the **X**. The + indicates the best case (performance/energy) ITR, DVFS configuration found by Bayesian optimization and the * is the best case found so far by the exhaustive experimental study search. . . . . 118
7.2 Bayesian optimization for 99% tail latency and energy in Memcached-silo. . . . . 119
7.3 Bayesian optimization for NetPIPE for performance and energy. . . . 120
7.4 Bayesian optimization for NodeJS for performance and energy. . . . 120
7.5 Raw requests-per-second log from Twitter cache-trace. . . . . 122
7.6 Bayesian optimization applied to Twitter cache-trace request rates over a 24 hour period. Each row represents a single SLA objective we are targeting and we display the change in energy (Joules) as QPS changes Next to each energy (Joules) figure, we also show the change in measured tail latency as QPS changes. . . . . 125
7.7 Bayesian optimization applied to Twitter cache-trace request rates where it used to optimize **only** for minimizing 99% tail latency. We show show the energy (Joules) consumption for this mode of operation. . . . . 125
7.8 Bayesian optimization applied to memcached-silo over a 24 hour period. Each row represents a single SLA objective we are targeting and we display the change in energy (Joules) as QPS changes Next to each energy (Joules) figure, we also show the change in measured tail latency as QPS changes. . . . . 128

## LIST OF SYMBOLS AND ABBREVIATIONS

| | |
|---|---|
| OS | Operating System |
| IO | Input-Output |
| CPU | Central Processing Unit |
| CC | Cache Coherent |
| NUMA | Non-Uniform Memory Access |
| MSR | Model Specific Registers |
| RAPL | Running Average Power Limit |
| DVFS | Dynamic Voltage/Frequency Scaling |
| NIC | Network Interface Card |
| RSC | Receive Side Coalescing |
| ITR-Delay | Interrupt delay |
| J | Joules |
| EDP | Energy Delay Product |
| RX | Receive |
| TX | Transmit |
| TCP | Transmission Control Protocol |
| IP | Internet Protocol |
| UDP | User Datagram Protocol |
| EbbRT | Elastic building-block Runtime |
| PMU | Performance Monitor Unit |
| GB | GigaBytes |
| TB | TeraBytes |

MTU Maximum Transmission Unit

# CHAPTER 1

# Introduction

## 1.1 PROBLEM STATEMENT

The design and use of operating systems (OSes) go hand-in-hand with the hardware on which it is deployed. One can trace this co-design through the first research OSes built in the 1960s up till the dawn of modern multiprocessors in the early 2000s (see Table 1.1). The evolution of computer hardware, in particular the decreasing cost of processors, memory and networking, compelled researchers to investigate the best way to structure a distributed OS around a set of networked workstations/processors. As part of this evolution, the microkernel model was developed by breaking from traditional monolithic designs and popularizing the restructuring of basic system resources such as memory, disks, networking, security, etc to be managed by a set of resource servers that typically use a message passing protocol to communicate these resources between different applications. These servers enabled a form of system specialization where these resources can be built with custom protocols and/or policies towards application specific require-

| OS | Target Hardware | Year |
|---|---|---|
| Multics | GE-635 | 1964 |
| UNIX | PDP-11 | 1964 |
| HYDRA | C.mmp | 1974 |
| Mach | heterogeneous workstations | 1986 |
| V | VMP Multiprocessor | 1988 |
| Sprite | SPUR Multiprocessor Workstation | 1988 |
| Amoeba | heterogeneous workstations (SPARC, x86, Sun3) | 1990 |
| Cache Kernel | ParaDiGM | 1994 |
| Tornado | NUMAchine Multiprocessor | 1999 |

**Table 1.1:** Some widely citepd historical research operating systems.

ments. The exokernel model further eliminated most OS abstractions in order to create a set of base components to directly manage the physical resources of a computer, this further improved application performance by enabling applications to also specialize hardware resources toward a single use-case.

However since the late 1990s, a variety of industry, application, and hardware trends (popularity of x86 ISAs that provide virtualization support) have moved away from this systems specialization goal and resulted in OS research (Pike, 2000) more focused on efficiently multiplexing multiple applications on similar hardware profiles. The advent of large shared memory CC-NUMA multiprocessors (Kuskin et al., 1994; Grindley et al., 2000; Laudon & Lenoski, 1997; Brewer & Astfalk, 1997; Lovett & Clapp, 1996) led to industry efforts on trying to scale traditional Unix-based OSes on these new architectures (Verghese et al., 1996; Perez, M., 1995). Part of this was due to the ubiquitous and successful development environments provided by Unix (i.e. X, Emacs, Tex, C/C++, gcc), which meant a common goal of OS research was to provide a compatible Unix emulation layer (Young et al., 1987; Ousterhout et al., 1988; Cheriton, 1988; Cheriton & Duda, 1994; Gamsa et al., 1999). In addition, the increasing role of computers in all manners of modern life meant an ever abundant set of standards, ranging from TCP/IP to various computer peripherals, that an OS must support. For example, in Plan9 (Pike et al., 1990) around 90% of the development work was invested to be compatible with various standards (Pike, 2000). In this period, monolithic kernels largely went unchanged primarily due to the performance and energy "free lunch" provided by improvements in computer silicon technology. As stated by Moore's law and Dennard scaling: as transistor counts doubled, processor power density stayed roughly the same, which means an effective doubling of performance per watt

every two years. During this period, the combination of kernel virtualization software maturity and processor hardware advances meant that it was cheaper and easier to meet computing demands by simply scaling up the commodity hardware used in data centers, i.e. economies-of-scale.

However, due to the effective end of Dennard scaling and Moore's Law (Esmaeilzadeh et al., 2011), traditional approaches of scaling up commodity software and hardware in data centers is no longer a feasible method to meet modern application performance demands (Mark Silberstein, 2017). Specifically, as we can no longer reliably rely on faster clock speeds nor the ability to power large caches, there is greater pressure to exploit alternatives such as hardware function-offloading. For example, there has been a proliferation of new smart hardware such as NICs (businesswire, 2016; Mellanox Innova SmartNIC, 2016), SSDs (Gu et al., 2016), memory (Agrawal, Sandeep R and Idicula, Sam and Raghavan, Arun and Vlachos, Evangelos and Govindaraju, Venkatraman and Varadarajan, Venkatanathan and Balkesen, Cagri and Giannikis, Georgios and Roth, Charlie and Agarwal, Nipun and Sedlar, Eric, 2017) in order to bring computation closer to the data (Barbalace, Antonio and Iliopoulos, Anthony and Rauchfuss, Holm and Brasche, Goetz, 2017; R. Balasubramonian and J. Chang and T. Manning and J. H. Moreno and R. Murphy and R. Nair and S. Swanson, 2014), and programmable accelerators (Putnam et al., 2014; **?**) in modern data centers that expose mechanisms for applications to better cater the hardware towards their specific use cases.

Regardless, as computing demand continues to grow, so does its energy consumption (Strubell et al., 2019; Fan et al., 2007; Nicola Jones, 2020); thereby providing another set of challenges for computing systems to be competitive in both performance and energy. To meet these challenges, new systems design insights

are needed. Below, we first begin by providing more context for these changes and challenges; next, we provide a high level overview of the contributions in this dissertation that try to meet these challenges, lastly, we summarize a few key experimental results and their implications on future systems research.

## 1.2 CHALLENGES

### 1.2.1 Energy

Global datacenter energy is continuing to rise (Gupta et al., 2020; Strubell et al., 2019; Fan et al., 2007; Nicola Jones, 2020) as data-intensive and interactive applications continue to grow alongside new workloads driven by Internet-of-Things (IoT) and various other edge computing devices. Further, a recent study shows that the environmental footprint of hardware manufacturing is also increasing concurrently (Gupta et al., 2020), which indicates the importance of being able to extract more value out of existing software and hardware.

While there have been a tremendous amount of previous research in understanding and reducing application energy use (Wu et al., 2016; Hsu et al., 2018; Lo et al., 2014; Hsu et al., 2015; Kasture et al., 2015; Leverich & Kozyrakis, 2014; Prekas et al., 2017; Asyabi et al., 2020; Zhan et al., 2017; Vamanan et al., 2015; Meisner & Wenisch, 2012; Chou et al., 2016, 2019; Sasaki et al., 2013; Flautner et al., 2001; Dominik Brodowski, Nico Golde, Rafael J. Wysocki, Viresh Kumar, 2022; Lefurgy et al., 2007; Cochran et al., 2011; Isci et al., 2006; Li & Martinez, 2006; Lee & Kim, 2009; Kim et al., 2008; Ge et al., 2007; Spiliopoulos et al., 2011; Kondo et al., 2007; Le Sueur & Heiser, 2011; Freeh et al., 2007; Elnozahy et al., 2003; Guliani & Swift, 2019; Tolia et al., 2008a; Hwang & Pedram, 2016), the role that the OS plays in application energy use and whether opportunities exist to achieve even better energy

efficiency is unfortunately not as well understood. We believe there are two main factors to this:

1. The popularity of using general purpose OSes such as Linux to deploy applications. The structure of a general purpose OS is typically complex, and is often treated as a black-box as it was designed to support diverse applications and is packed with many kinds of policies that dynamically adjust system policies (Intel, 2021; Mellanox, 2022; ARM, 2023; Dominik Brodowski, Nico Golde, Rafael J. Wysocki, Viresh Kumar, 2022; Rafael J. Wysocki, 2018), therefore making it difficult to reason about overall system energy efficiency given the interplay between systems policies and how it interacts with the hardware.

2. Modern systems software also often contain complex stacks of both synchronous and asynchronous software layers which include device drivers, OS-level policies, and application level work. This makes it a challenge to gather fine-grained experimental measurements of both software and hardware components during application runtime to understand the role that the OS plays.

### 1.2.2 Hardware

Traditional hardware techniques for shrinking transistors on a processor die have already hit physical limits as the era of Moore's law and Dennard scaling is effectively over (Esmaeilzadeh et al., 2011; Semiconductor Industry Association, 2015; Krste Asanovi). In data centers, this means that scaling up commodity hardware and running applications in virtualized instances of OSes is no longer a competitive method to meet compute demands with respect to monetary cost, perfor-

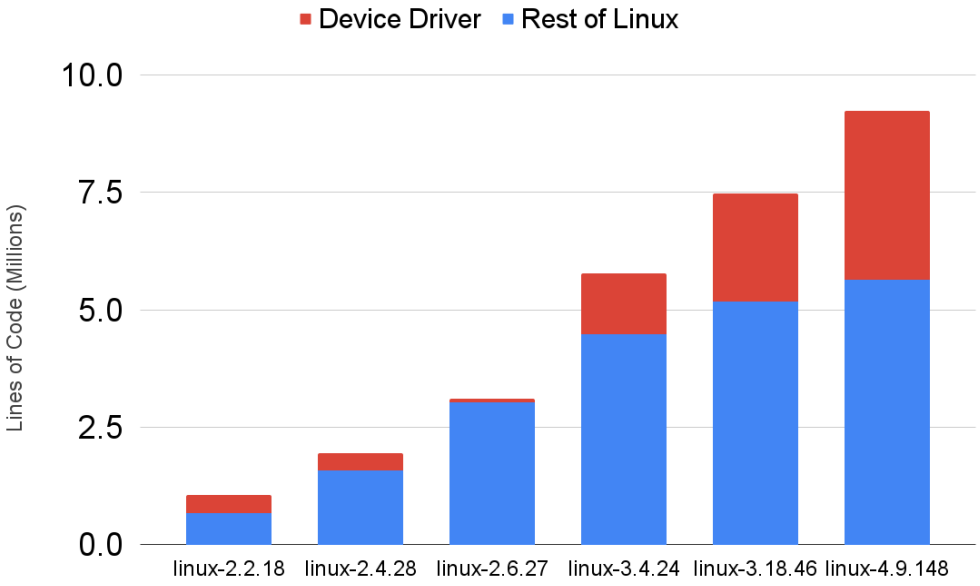


**Figure 1.1:** Growth in complexity of Linux kernel and device drivers.

mance, and energy (Mark Silberstein, 2017). To meet these demands, hardware manufacturers have mainly pursued two approaches: 1) function-offloading into various hardware devices, and 2) packing more functionality into existing devices.

- **Hardware Function-Offloading:** Modern data centers are beginning to diversify its hardware makeup by introducing new programmable hardware such as NICs (Mellanox Innova SmartNIC, 2016; businesswire, 2016), SSDs (Gu

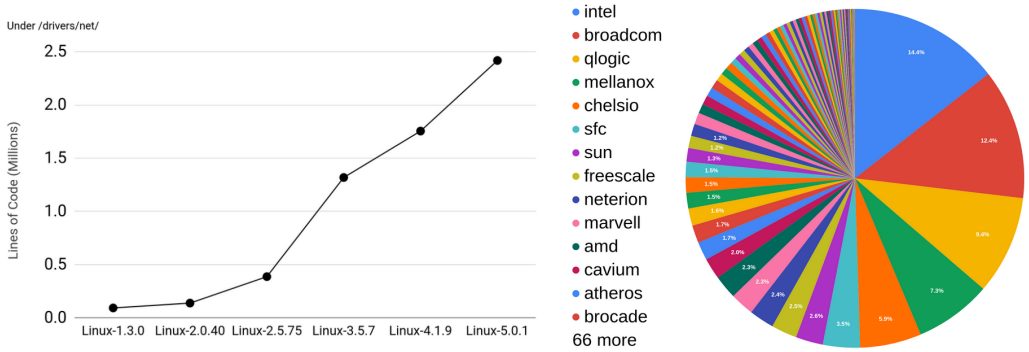


**Figure 1.2:** Growth in complexity of NIC device drivers.

et al., 2016), and memory (Agrawal, Sandeep R and Idicula, Sam and Raghavan, Arun and Vlachos, Evangelos and Govindaraju, Venkatraman and Varadarajan, Venkatanathan and Balkesen, Cagri and Giannikis, Georgios and Roth, Charlie and Agarwal, Nipun and Sedlar, Eric, 2017). This move away from the traditional CPU-centric view of programming is motivated by factors such as faster network bandwidth approaching DRAM access time (Barbalace, Antonio and Iliopoulos, Anthony and Rauchfuss, Holm and Brasche, Goetz, 2017) and processing bottlenecks induced by current processor speeds (Esmaeilzadeh et al., 2011), this problem is also worsened by increasingly constrained energy budgets (Hsu et al., 2018; Wu et al., 2016). Further, bringing computation closer to the data (R. Balasubramonian and J. Chang and T. Manning and J. H. Moreno and R. Murphy and R. Nair and S. Swanson, 2014) also has the potential to improve overall application efficiency such as reducing the need to transfer data back and forth from the CPU. Subsequently, these devices are expected to join an existing plethora of programmable accelerators (Graphics Processing Units, 2022; Field Programmable Gate Arrays (FPGAs), 2022; **?**) already within modern datacenters (Mark Silberstein, 2017). These mechanisms also represent new opportunities to address the role that the OS can play in application performance and energy such as identifying which portions of functionality can be offloaded and how to best program and manage diverse sets of hardware concurrently.

- **Diverse Hardware Features:** There has been a large body of work documenting and understanding the complexity of modern device drivers (Kadav & Swift, 2012) and various aspects of their reliability (Ball et al., 2006; LeVasseur et al., 2004; Ryzhyk et al., 2009), configuration (Renzelmann & Swift, 2009;

Ryzhyk et al., 2014; Schüpbach et al., 2011), and performance (Ganapathy et al., 2008; Ryzhyk et al., 2010). This complexity can be seen in figure 1.1 which shows the growing code size of device drivers relative to the rest of the kernel between 2012 and 2018. In figure 1.2, one can also see this growth even in a common hardware device such as the NIC. Between kernel versions 1.3.0 and 5.0.1, there has been a increase in the complexity of NIC device driver from a few hundred thousand lines of code up to 2.5 million. Part of this is due to the variety of hardware vendors releasing NICs, figure 1.2 shows over 80 different companies releasing device drivers for their respective NICs in the latest 5.0.1 kernel version. Another contributing factor to this complexity is the ever increasing amount of features that are packed into modern hardware. Figure 1.3 shows a breakdown of the various features in device driver code for one a popular network device such as the Intel 82599 family of NICs. We found there were a total of 5630 32-Bit hardware registers that can be configured in this device and at the moment, Linux device driver setup code only use around 1360 of them to bring the NIC to a working state.

Both of the approaches described above also illustrate challenges that future systems must address. In contrast to hardware designs from Table 1.1, which were mainly based on the von Neumann architecture; modern hardware is more complex with different features that may or not may be exposed to applications. Moreover, relying on device drivers to act as an abstraction over existing hardware functionality is not sufficient to meet the performance and energy goals of datacenter scale workloads as 1) they typically come prepackaged with "one-size-fits-all" algorithms to manage the hardware features and this 2) can prevent software from fully understanding and utilizing these feature towards application specific goals.

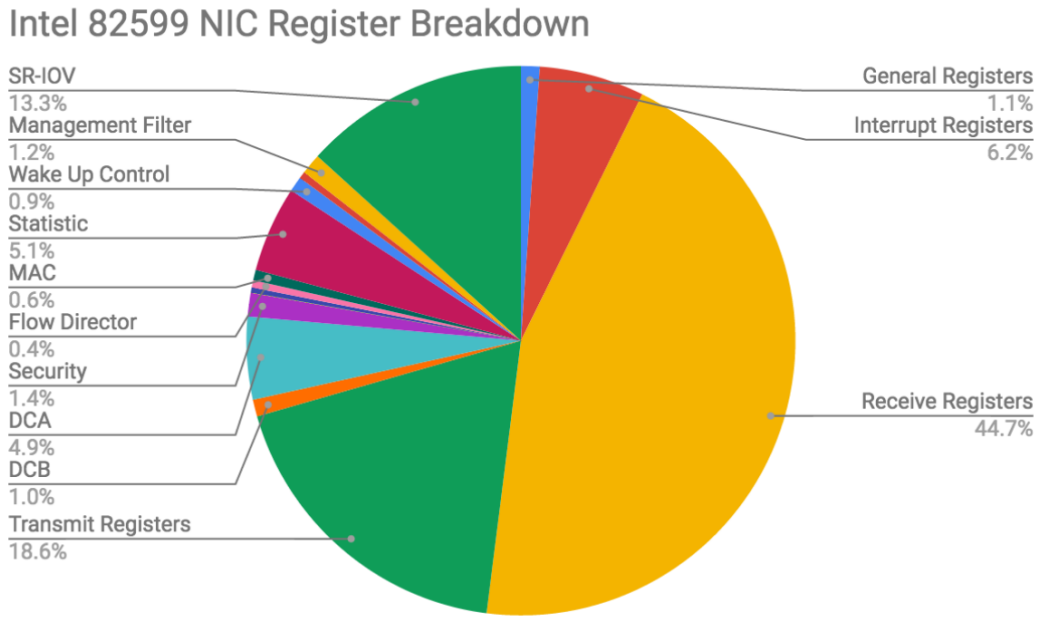


**Figure 1.3:** Complexity of Intel 82599 NIC device driver.

Lastly, the vast amount of configuration combinations exposed by these devices poses the challenges of whether an application or user can successfully explore this space without the help of more guided optimization techniques. Recent advances in machine learning may offer a path to enabling such guidance in a feasible way.

### 1.2.3 Software

As latency-critical tasks become ubiquitous across data centers, deploying them on dedicated nodes is becoming a well studied and favored decision (Prekas et al., 2015; Lo et al., 2015; Guliani & Swift, 2019; Tang et al., 2020) as this dedication prevents latency violations that might be triggered by the co-location of best-effort batch tasks. From an OS research perspective, specializing applications for a single system led to a revival in exploring previous micro-kernel and exokernel designs in the context of modern hardware and language techniques such as kernel

bypass (Belay et al., 2014; Dragojević et al., 2014; Intel Corporation, 2022) and recently proposed library OSes (Schatzberg et al., 2016; Madhavapeddy et al., 2013; Antti Kantee, Justin Cormack, 2014; Peter et al., 2015). While these techniques have demonstrated tremendous performance gains, they are typically deployed in virtualized instances and time-share the hardware with other system components. Further, we lack knowledge regarding the performance and energy benefits of running these library OSes baremetal.

Service oriented applications are typically I/O driven, and frequently exercise OS components such as the device driver and protocol processing stacks. Various OS policies and mechanisms are needed to detect requests, dispatch the work to service a request and to determine when and how long to idle the processor for potential energy savings. As such, the OS can have important impacts on both performance and energy of a workload even if there is a user level component at the end of the request. Further, the challenges posed in §1.2.1 and §1.2.2 means that there is a need to understand the role of the OS in affecting overall system energy efficiency both in its software policies and the possibilities to further improve both performance and energy when the hardware itself can also be optimized for a single application.

Recent studies of widely deployed services, such as in-memory key-value stores, from companies such as Facebook (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), Twitter (Yang et al., 2020), Netflix (Shashi Madappa, 2012), and Reddit (Daniel Ellis, 2017) reveal that these services often maintain a mean demand curve that change slowly over the course

of hours and up to days, which suggests that specialization of a single application at a specific offered load could be a realistic form of optimization by exploiting the stable regions of these demand curves.

## 1.3 MOTIVATION

In context of the challenges above, this dissertation is motivated by changes occurring in both software and hardware, and the performance and energy requirements of datacenter applications. In §1.2.3, the growing popularity of unikernels (Antti Kantee, Justin Cormack, 2014; Madhavapeddy et al., 2013; Raza et al., 2019) and the popularity of dedicating nodes for specific applications (Tang et al., 2020) in data centers implies that an eventual deployment model to achieve highest performance would be to run application specific unikernel binaries on dedicated baremetal hardware. However, the diverse nature and inherent complexity of modern hardware as shown in §1.2.2 poses a set of challenges as to how an OS should manage its features and how it can exploit these features to customize towards a specific use case. Section 1.2.1 shows that energy is also an important consideration for many applications and the energy efficiency benefits of specialized OS' and what further improvements can be achieved through hardware specialization is a largely unexplored area of research. Given the importance of reducing energy consumption and the attendant challenges (§1.2.1), this thesis is motivated to consider the combined impacts of both OS and hardware specialization.

## 1.4 CONTRIBUTIONS

This dissertation establishes that optimizing performance-energy simultaneously, despite complex OS structure, can be made a well-defined task using existing hard-

ware mechanisms. A summary of the novel contributions of this thesis is as follows:

1. We design and construct a reproducible experimental methodology that enables the impact of different OS designs and implementations on application performance and energy to be studied in a controlled fashion.

2. Using our methodology, we find that application performance and energy can be significantly optimized by externally controlling (independent of the OS) request batching and processor frequency settings using standard hardware mechanisms.

3. We find that specialized OS structure enables new and important interplay with energy consumption: **i)** our work is the first to show that specialization has a dramatic impact not only on performance but also energy, **ii)** and yields even more advantages from using the two hardware mechanisms.

4. We find that the results of our experimental study are sufficiently structured, which led us to develop a novel mathematical model that can accurately characterize the complex layers of systems software and their interactions. Moreover, we show how our model can accurately predict application performance and energy independent of the OS.

5. The results of our modeling work motivated us to explore learning systems in order to exploit these structures. We demonstrate how an off-the-shelf platform, Ax Ax:Adaptive Experimentation Platform (2022); Bakshy et al. (2018) can be used to use the dataset from our experimental study to automatically adapt the two hardware mechanisms towards performance and energy targets that are agnostic to different types of OSes and applications.

6. Lastly, we demonstrate the practicality of our learning system by applying it to on a real world in-memory key-value store trace from Twitter Juncheng Yang (2020) and demonstrate its ability to dynamically adapt to changing request rates while satisfying performance and energy goals.

Below, we first segment our contributions and findings into the following broad categories: 1) Engineering, 2) Experimental Study, 3) Experimental Analysis, 4) Applying Machine Learning and then expand on our contributions individually.

### 1.4.1 Engineering Contributions

#### *1.4.1.1 In-situ data collection infrastructure*

Given the complexity of the modern systems software layers Kadav & Swift (2012); Renzelmann & Swift (2009); Ryzhyk et al. (2014); Schüpbach et al. (2011); Ball et al. (2006); LeVasseur et al. (2004); Ryzhyk et al. (2009); Lim et al. (2014); Jeong et al. (2014); Marinos et al. (2014); BeifuSS et al. (2015); Hanford et al. (2018); Cai et al. (2021) and an application's inter-dependency on external request rates Schroeder et al. (2006), it can be a difficult task to measure how changing an OS, its policies, and the way hardware is used, can affect an application's performance and energy profile.

In this thesis, we tackle this problem by constructing a novel logging mechanism, *itrLog*, that captures the behavior of an entire software stack by using epochs based timestamp counters to conduct lossless in-situ fine-grained time series data which can reflect all packet and software interactions. These timestamps enable us to achieve precise delineation of system behavior down into the detailed packet receives and transmits at the NIC's interrupt handling logic. Furthermore in each epoch, we log additional software statistics and hardware statistics from the per-

formance monitor units (PMUs) in order to gain deeper insights into these fine-grained interactions on an application's performance and energy.

As a mechanism to induce hardware and policy changes, *itrLog* uses a NIC's hardware feature to set a periodic timer for handling interrupts, this interrupt delay (ITR) setting Intel (2021); Mellanox (2022) is used to induce batched handling of requests. Further, a processor's dynamic voltage frequency scaling (DVFS) setting ARM (2023); Dominik Brodowski, Nico Golde, Rafael J. Wysocki, Viresh Kumar (2022) is used to to modify CPU energy settings to explore application performance and energy trade-offs. Using *itrLog*, this thesis presents results from an in-depth and first-of-a-kind performance and energy study of four network applications on two different OSes through the controlled use of ITR and DVFS mechanisms.

**Implications:** *itrLog* is both application and OS agnostic as it captures system behavior at the device driver level, our approach can be easily extended to other application domains and document their performance and energy profiles.

### 1.4.2 Experimental Study Contributions

#### *1.4.2.1 Combining ITR and DVFS for performance and energy gains.*

Although DVFS has been extensively studied in energy proportional computing in Linux Sasaki et al. (2013); Flautner et al. (2001); Dominik Brodowski, Nico Golde, Rafael J. Wysocki, Viresh Kumar (2022); Lefurgy et al. (2007); Cochran et al. (2011); Isci et al. (2006); Li & Martinez (2006); Lee & Kim (2009); Kim et al. (2008); Ge et al. (2007); Spiliopoulos et al. (2011); Kondo et al. (2007); Le Sueur & Heiser (2011); Freeh et al. (2007); Elnozahy et al. (2003), this thesis presents novel results from its use with a baremetal library OS, EbbRT Schatzberg et al. (2016). Further,

while prior work have only used a static setting of a single ITR value for experimental stability Yasukata et al. (2016); Peter et al. (2015); Ousterhout et al. (2019) and demonstrated value of batching in software Chou et al. (2019); Elnozahy et al. (2003), this thesis presents finds how one can exploit this network device feature on a per-application basis to induce stability and predictability in incoming request processing in order to magnify the benefits of processor energy settings to improve performance and reduce energy use. For example in Linux, we find the right static setting of ITR and DVFS can result in 1.33X performance improvement while reducing energy consumption by 1.76X.

**Implications:** ITR and DVFS can be used to further improve application performance and energy by exploiting the stable regions of the demand curves of applications such as widely deployed in-memory key-value stores 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); Yang et al. (2020); Shashi Madappa (2012); Daniel Ellis (2017).

#### *1.4.2.2 Energy study of baremetal Library OS*

In academia, there is a huge interest in building per-application systems Belay et al. (2014); Lim et al. (2014); Prekas et al. (2017); Ousterhout et al. (2019); Ghigoff et al. (2021); Chou et al. (2016); Jin et al. (2017); Han et al. (2012); Yang et al. (2021); Yasukata et al. (2016); Cadden et al. (2020); Peter et al. (2015); Antti Kantee, Justin Cormack (2014); Madhavapeddy et al. (2013); Raza et al. (2019), such as library OSes, unikernels, kernel-bypass techniques, etc., however, they have typically only focused on performance and the energy efficiency of such systems are not as clearly

understood. We demonstrate the value of such systems in both performance and energy by porting EbbRT Schatzberg et al. (2016), a library OS written in a high-level C++ language, to run on baremetal hardware by developing a network device driver from scratch for the Intel 82599 family of NICs. EbbRT has common optimization attributes such as such as run-to-completion, event-driven execution model, single execution domain, and compile-time optimization. EbbRT running in baremetal was able to achieve up to 10X throughput improvements over Linux in a NetPIPE workload and over 2.5X improvements energy efficiency in a Memcached server.

**Implications:** Many of these previous research systems should exhibit the same energy efficiency findings as demonstrated through our EbbRT results. Further, our findings demonstrate the value of OS specialization techniques to reduce energy consumption.

#### *1.4.2.3 Data-driven OS Specialization*

This dissertation also demonstrates the use of our novel *itrLog* test bed to perform data driven driven OS exploration to discover new insights for alternate OS design and policy changes. As an example, based off our findings of using fast ITR values in Linux which enabled the best performance and energy savings; we were then motivated to explore the use of "slow" polling in EbbRT and as a result demonstrated up to 1.8X better performance while using 2X lower energy - this is in contrast to the normative assumption of OS poll whe (2012); Golestani et al. (2019) where it trades performance for higher energy use.

**Implications:** Library OSs are strong candidates to serve as tools for aggressive optimization of energy where the design and implementation space of opti-

mizations can be guided through a data driven process given the right points of introspection. Furthermore, this opens the space of new policies designs, such as switching from interrupt to poll based processing given the workload type. Our findings of OS path specialization techniques and their benefits can also be adapted in general purpose OSes.

### 1.4.3 Experimental Analysis Contributions

#### *1.4.3.1 Developing a model to capture complex systems interactions*

Using insights from our analysis of the dataset, this thesis develops a simplified request processing timeline that can be expressed mathematically and constructs a model to reflect the important interactions that the OS can have on the realized application performance and energy. Specifically, our model lets us evaluate how ITR and DVFS interacts with OS and application request processing and how this changes when the OS or offered load changes.

As an example of our model's accuracy, we can use it to predict the per-request tail latency of memcached at 600K QPS: our experimental data shows a mean tail latency of 270.91 $\mu$s $\pm$ 132.80 $\mu$s and our model's predicted $\sqrt{MSE}$ latency of 0.18 $\mu$s is much smaller than any latency measurements, indicating an accurate fit. To the best of our knowledge, this is the first type of study where hardware level metrics are collected, analyzed and subsequent model built to correctly capture behavior of real world applications Atikoglu, Berk and Xu, Yuehai and Frachtenberg, Eitan and Jiang, Song and Paleczny, Mike (2012); Prekas (2017); Joyent (2013).

**Implications:** Our results demonstrate that not only is it possible to do in-situ fine-grained data gathering and analysis of a complete full software stacks, but also correlate the fine-grained packet by packet behavior to its impacts on an

application's performance and energy profile. Furthermore, the modeling work suggests reinforcement learning approaches can be used to build control of ITR, DVFS settings.

### 1.4.4 Applying with Machine Learning

#### *1.4.4.1 Machine learning technique to automatically tune ITR and DVFS.*

While the analytical models demonstrate that one can pre-compute ideal batching and processor energy settings for some software stacks; it is not a practical approach in all but highly constrained static environments. However, the existence and accuracy of the analytic model's equations suggests the viability of using a black-box learner to exploit these structures. We present an example of this in our use of Bayesian optimization Frazier (2018); Garnett (2022) to efficiently find batching and energy settings that target performance and energy goals across the applications and OSes. Such a technique can compensate for inaccuracies in our analytical model and the need for exhaustively searched experimental data.

We then built an experiment to run a full 24 hour in-memory key-value store trace from Twitter Juncheng Yang (2020) and demonstrate Bayesian optimization's ability to adapt to changing requests rates. We find this technique was able to automatically configure Linux such that it further reduced its energy use by 50% while meeting performance goals.

**Implications:** Despite different OS and application stacks, the results in this dissertation present novel techniques for data gathering, analysis and subsequent model and automated learner that demonstrate the ability to tame these complexities in order to build smarter policies that can better adapt to changing performance and energy targets.

## 1.5 OUTLINE

The rest of the dissertation is organized as follows,

**Chapter 2:** details the design and architecture of *itrLog* and provides additional information on our experimental infrastructure.

**Chapter 3:** details the Linux and EbbRT software stacks along with the four applications used in our experimental study.

**Chapter 4:** presents an abstract network processing timeline in order to prepare for discussions of experimental results.

**Chapter 5:** lists our major experimental findings from the study.

**Chapter 6:** presents intuition for the mathematical model and results from applying it our experimental data.

**Chapter 7:** demonstrates our Bayesian optimization approach to automatically find optimal parameters for ITR and DVFS to minimize energy use.

**Chapter 8:** presents the related work section and **Chapter 9:** lists possible future work from this thesis and **Chapter 10:** concludes this thesis.

# CHAPTER 2

## *itrLog*: in-situ data collection infrastructure

This dissertation carefully examines the interactions of the OS software and hardware when considering both energy and performance in the context of network driven applications. To this end, we conducted a comprehensive study by creating a novel data collection infrastructure to collect a body of data that documents these relationships. Through this study, one goal is to reveal the fine-grained relationships between the different applications as they run on different OSes and the impact on their runtime behavior as request batching and processor speeds are externally controlled through two common hardware mechanisms. Towards this, this chapter presents the details of our data collection infrastructure, *itrLog*. This infrastructure not only contain OS independent ways to collect experimental data, but also the mechanisms required to set batching and processor frequency rates as well as the visualization tools needed for subsequent data analysis.

To begin, we summarize the following attributes that we believe to be important for an experimental data collection infrastructure:

1. **Agnostic infrastructure** that can gather data regardless of the OS and applications running on top.

2. **Epoch-based** data collection to collect fine-grained system measurements partitioned by time at the per-core granularity.

3. **Data logging statistics** that capture both important network processing data as well as system usage data (i.e. energy, instructions, cycles, etc.) while not drastically perturbing overall application performance.

4. **Efficient data retrieval** method for subsequent storage and analysis.

In the sections below, we first discuss the details of implementing these four attributes and expand on possible challenges. We then present *itrLog*, which is our data collection infrastructure whose construction is informed by these attributes. We also discuss the similarities and differences of our implementations of *itrLog* in both a general purpose Linux and a specialized OS, EbbRT.

## 2.1 ATTRIBUTES OF DATA COLLECTION INFRASTRUCTURE

The broader implications of our design choices for the data collection infrastructure is discussed below; along with prospective implementation details that one should manage.

### 2.1.1 Agnostic Infrastructure

As our infrastructure should collect data transparently regardless of the OS and application running on top, it then becomes key to place the data collection logic as close to the underlying hardware as possible such that it can be applied *generically*. As this thesis targets network oriented applications, it was intuitive to place our logic at the device driver for managing the NIC as this code is often the first piece of logic that runs after packets are received. As the NIC's device driver logic communicates with higher level network stacks to pass on the received data, this also enables the infrastructure to gather baseline network statistics such as number of physical bytes processed; therefore providing greater context around application runtime behavior.

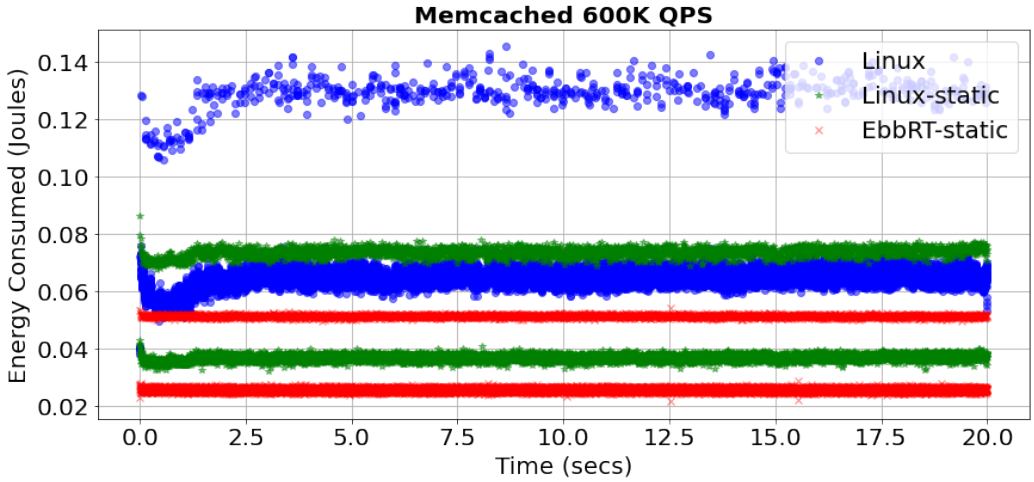


**Figure 2.1:** Example figures that showcase the fidelity of fine-grained data collection of energy consumed (Joules) at a per interrupt level.

### 2.1.2 Epoch-based

We were motivated to pursue an epoch-based approach for data collection as this approach creates a time series that partitions and records critical system events; primarily because OS interrupt and polling behavior partitions time into busy and idle periods. The OS directly controls energy consumption behavior during idle time and its design and implementation can have a significant impact on the instructions that compose busy time and their efficiency. As such, we found that local behavior of the OS, during interrupts, combines to significantly influence global system behavior and performance. Interrupt-centric energy-performance logs enable us to locate and analyze relevant power events in the span of execution of a workload. The behavior of the events can be attributed to either the involvement of OS policies, such as using just enough processor energy to satisfy a particular request or keeping the processor busy due to changes in offered loads. Typical logging approaches such as periodic PC sampling can often overlook these critical OS episodes, that while short in duration, can fundamentally dictate the net energy

```
i rx_desc rx_bytes tx_desc tx_bytes instructions cycles llc_miss c0 c1 c1e c3 c6 c7 joules rdtsc
0 0 0 0 0 39897946925 94877628846 99999926 0 0 0 0 0 0 1794002775 3050504315855
1 3 227 4 140 39899697156 94904765857 100031381 0 0 0 0 0 0 1794934193 3052903309265
2 0 0 0 0 39901734069 94933626831 100062328 0 0 0 0 0 0 1795720235 3054925781967
3 3 227 4 140 39901825203 94934928931 100063724 0 0 0 0 0 0 1795729309 3054949131714
4 2 1524 2 66 39901850973 94935475436 100064417 0 0 0 0 0 0 1795730435 3054952895062
5 1 66 0 0 39901879028 94935910059 100064812 0 0 0 0 0 0 1795737279 3054967548869
6 1 114 2 66 39901934102 94936622676 100065413 0 0 0 0 0 0 1795744082 3054986651459
7 1 66 0 0 39901978062 94937225354 100065986 0 0 0 0 0 0 1795750890 3055004402417
8 1 82 0 0 39901994602 94937502971 100066216 0 0 0 0 0 0 1795796797 3055121346880
9 3 318 6 198 39902055228 94938227855 100066926 0 0 0 0 0 0 1795802941 3055138669653
```

**Figure 2.2:** Snippet of raw log data

efficiency achieved.

Further, as real world applications such as in-memory key-value stores are typically deployed on entire nodes (Atikoglu, Berk and Xu, Yuehai and Frachtenberg, Eitan and Jiang, Song and Paleczny, Mike, 2012) and span multiple physical cores, it is also important to conduct this epoch-based data collection across these cores. In most systems, there is typically a fixed mapping between a network queue and a physical core. By taking advantage of this fixed mapping, one can create pre-allocated per-core data structures that are sized appropriately for the upcoming application and its offered load. This initial pre-allocation also eliminates additional overhead from dynamically allocating and freeing memory in the OS.

An example of the benefits of this epoch-based data collection approach is shown in fig. 2.1 (a sample of the raw log data is also shown in fig. 2.2); the distinct banding in energy consumed characterizes the behaviors of three different systems studied. In this figure, each point represents a single measurement of energy at a distinct point in time during the entire experimental run of Memcached which serves an offered load at 600K QPS for a total of 20 seconds.

### 2.1.3 Data Logging Statistics

The types of data logged can include hardware statistics such as performance monitoring units (PMUs), which are often provided on modern processors, and software statistics that are system wide and other packet processing metrics. The periodicity of the data logged can also be specified with the epoch-based approach. For example in the NIC's device driver, it is possible to instrument this periodicity at the per-interrupt level such that hardware and software statistics relevant to the performance and energy profile of a particular application and system can be logged. Further, one should also ensure that the collected metrics are OS independent enough such that they are not embedded with system specific semantics.

As this level of data collection can result in very detailed sets of data, care must be taken to ensure that it does not drastically impact the performance of the application by evicting performance sensitive cache lines. Here are a few methods to address the performance sensitivity of massive data collection: 1) simply decrease the variety and total amount of the data collected, 2) decrease the frequency of data collection, and 3) use special instructions (if available on your system) that allow for write-once-read-later mechanisms in order to avoid evicting crucial cache lines.

Depending on the application work and the performance degradation observed; all three methods can be adjusted for the application specific use cases. In the case of 1), it might not always be necessary to gather all possible system and hardware metrics and some modulation may needed to only capture the metrics that are important for the specific application. For 2), one can set specific epochs such as doing data collection after **X** number of interrupts or after **X** amount of time has passed – though care should be taken with the time counter such that its frequency is also not impacted by processor frequency changes. For 3), it largely depends on

the ISA of the system to provide such mechanisms; for example on x86_64, such capabilities are provided via non-temporal store instructions.

### 2.1.4 Efficient Data Retrieval

Suppose a large amount of in-memory data has been gathered; here are a few methods to efficiently retrieve the data for storage and analysis: 1) the data structures can be mapped into the filesystem such that basic filesystem commands can then be used to read and save the results locally and remotely, 2) the data can also be mapped to userspace memory where it is then the application's responsibility to handle storage, and 3) using TCP protocols such that a specific incoming packet request results in the data being sent over the network to a storage server; this process can be made even more efficient if the data can be immediately transmitted from the NIC with custom network protocols without needing to go up through the entire network stack again. For 1) and 2), care should be taken to pace this memory copying as to not throttle the rest of the system while 3) should ensure the protocol is simple enough and only used after the application has finished running in order to not perturb the performance of a running application.

## 2.2 *ITRLOG* DETAILS

Motivated by the attributes described above, we created *itrLog* in order to conduct an experimental study on a x86_64 platform that consists of an Intel(R) Xeon(R) CPU E5-2690 @ 2.90GHz processor with an Intel 82599ES 10-Gigabit SFI/SFP+ NIC. Figure 2.3 illustrates the key features of *itrLog* and the sections below will provide further implementation details in both Linux and EbbRT.

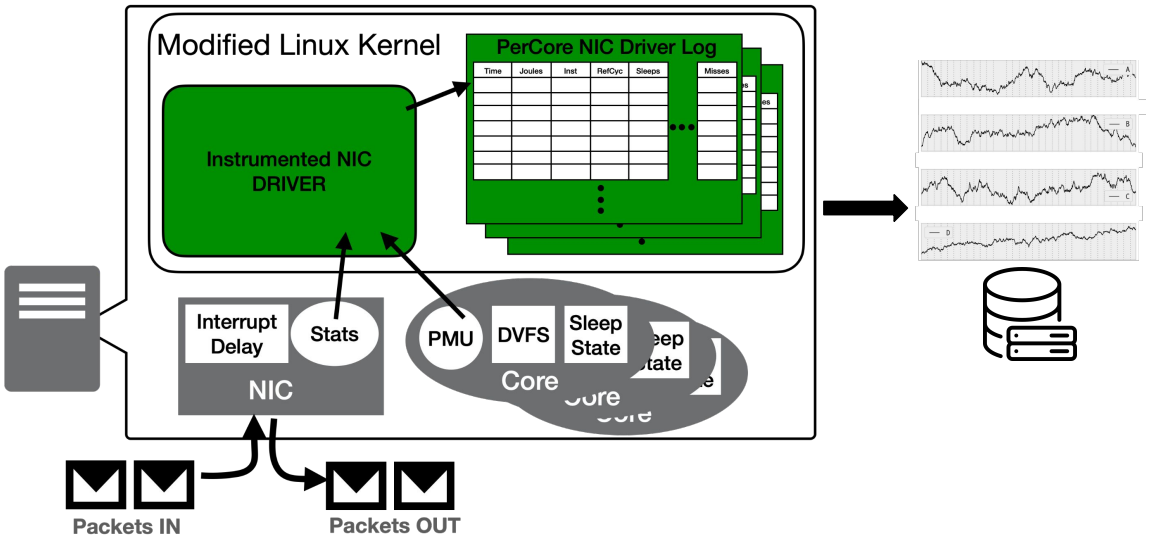


**Figure 2.3:** Data collection infrastructure of *itrLog* which uses a modified NIC device driver to collect and write fine-grained per-core log data.

### 2.2.1 Linux Implementation

Our implementation of *itrLog* is integrated with the Linux 5.4 kernel and further details of this Linux kernel and its configuration is described in chapter 3.

#### *2.2.1.1 Agnostic Infrastructure:*

In order to collect fine-grained data points, we modified the NIC's device driver code (https://github.com/handong32/linux.git). In the Linux kernel, this is located at the following directory: `drivers/net/ethernet/intel/ixgbe/`. For the modifications, we first added a new data structure in the file `ixgbe.h` called `IxgbeLogEntry`, this data structure (shown in listing 2.1) contains a set of unsigned 64-bit and 32-bit variables that is used to store statistics captured at the granularity of a single hardware interrupt or an epoch.

```
1 union IxgbeLogEntry {
```

```
2   long long data[13];
3   struct {
4     long long tsc;                  // time stamp counter
5     long long ninstructions;        // nstructions counter
6     long long ncycles;              // cycles counter
7     long long nllc_miss;            // last-level cache miss counter
8     long long joules;               // energy counter
9     long long c0;                   // c0 sleep state counter
10    long long c1;                   // c1 sleep state counter
11    long long c1e;                  // c1e sleep state counter
12    long long c3;                   // c3 sleep state counter
13    long long c6;                   // c6 sleep state counter
14    long long c7;                   // c7 sleep state counter
15    unsigned int rx_desc;           // number of receive descriptors
16    unsigned int rx_bytes;          // number of received bytes
17    unsigned int tx_desc;           // number of transmit descriptors
18    unsigned int tx_bytes;          // number of transmit bytes
19  } __attribute((packed)) Fields;
20 } __attribute((packed));
```

**Listing 2.1:** `struct IxgbeLogEntry` data structure

Both the device initialization and main functions for operating the NIC is found in the file: `ixgbe_main.c`. Inside this function, `ixgbe_open()` is the main function that first brings the NIC to a working state; at the top of this function is where we instrumented code to pre-allocate arrays of `struct IxgbeLogEntry` using a Linux specific allocator such as `vmalloc()`. These arrays represent a log of the entire performance and energy data for an application under some offered load. Correspondingly, inside the function `ixgbe_close()`, `vfree()` is used to free this memory – this is to ensure proper memory management behavior every time the modified NIC kernel module is loaded and unloaded. We confine our modi-

fications to device driver specific changes as these loadable modules enable ease of (re)deployment and giving us the capability to quickly swap out the modified device drivers with different types of data collection methodology.

#### *2.2.1.2 Epoch-based*

To enable epoch-based data collection, *itrLog* gathers fine-grained logs of hardware statistics relevant to the performance and energy profile of a particular workload and of the system overall. We instrument code to collect data using MSI-X interrupts; which notify the device driver that new events such as packets received or transmission completion have occured. This MSI-X initialization code is found in `ixgbe_main.c` with function `ixgbe_request_msix_irqs()` and inside this function, the `request_irq()` call is used to map an available hardware interrupt vector number to the device driver's generic interrupt handler function: `ixgbe_msix_clean_rings()`. Typically, this occurs on a per-core basis where every core has its own interrupt handler function called depending if there are packets to be processed for that core. Inside `ixgbe_msix_clean_rings()`, the logging code is added right before the call to `ixgbe_poll()` (which is Linux's NAPI function for processing received packets) in order store the accumulated data statistics. As stated, we log this data into a single `struct IxgbeLogEntry` entry by using a per-core counter to index into one of the pre-allocated arrays. A sample of this logging process is provided below:

```
1 static irqreturn_t ixgbe_msix_clean_rings(int irq, void *data) {
2     struct ixgbe_q_vector *q_vector = data;
3
4     // index is unique to each core
5     int index = q_vector->v_idx;
```

```
6
7       // there is an ixgbe_logs[] array for each core
8       il = &ixgbe_logs[index];
9
10      // icnt is used to index into each IxgbeLogEntry
11      icnt = il->itr_cnt;
12
13      // ensure log entries stay within pre-allocated memory bounds
14      if (icnt < IXGBE_LOG_SIZE) {
15          // access each IxgbeLogEntry
16          ile = &il->log[icnt];
17
18          // update receive and transmit bytes and descriptors
19          ile->Fields.rx_desc = q_vector->rx.per_itr_desc;
20          ile->Fields.rx_bytes = q_vector->rx.per_itr_bytes;
21          ile->Fields.tx_desc = q_vector->rx.per_itr_desc;
22          ile->Fields.tx_bytes = q_vector->rx.per_itr_bytes;
23          ......
24      }
25      /* EIAM disabled interrupts (on this vector) for us */
26      if (q_vector->rx.ring || q_vector->tx.ring)
27          napi_schedule_irqoff(&q_vector->napi);
28      return IRQ_HANDLED;
29  }
```

**Listing 2.2:** `ixgbe_msix_clean_rings` psuedocode

Each `struct IxgbeLogEntry` contains both transmit and receive *descriptors* information, where a *descriptor* is a NIC specific data structure that describes a transmit or receive network packet. In the case of receive descriptors, it contains a memory address pointing to where the received packet lives along with various other information such as checksums, bytes received, etc. Similar fields exist for

transmit descriptors but the key difference is that the address instead points to a packet that has already been sent and the NIC devices driver should then decide how to reclaim the memory. The `ixgbe_poll()` function processes both receive and transmit packets, it achieves this by calling both `ixgbe_clean_tx_irq()` and `ixgbe_clean_rx_irq()` functions in succession before returning back to the NAPI scheduler.

In order to log this descriptor data on a per-core basis, we take advantage of the multi-core nature of `ixgbe_msix_clean_rings()` as shown in listing 2.1. MSI-X maps a single interrupt vector number to its interrupt handler for each individual core. Therefore, accessing Linux data structures inside its handler is already akin to accessing per-core and per-queue local variables. In listing 2.1, we use `q_vector->v_idx` to index into our global multi-core data structure and update the corresponding log entry for the different queues.

Due to the performance implications of dynamically allocating and freeing large arrays of `struct IxgbeLogEntry` data structure inside the Linux kernel, *itrLog* pre-allocates a large chunk of memory in the device driver initialization code instead. We size the arrays appropriately to hold all data for the applications listed in table 3.1 by running initial sniff-tests on a variety of applications at different offered loads.

#### *2.2.1.3 Data Logging Statistics*

We collect the following bits of information at every interrupt: received and transmitted bytes, the current timestamp (via the `rdtsc` instruction), and various sleep state statistics. We also collect a set of hardware statistics from per-core performance monitoring units (PMUs) at a larger granularity of every millisecond: in-

structions, cycles, last-level-cache misses, and the energy consumed. All of the hardware data is collected by reading specific MSR registers by calling `rdmsrl` function in Linux.

The energy register is read from MSR_PKG_ENERGY_STATUS from Intel's Software Developer Manual (Intel, 2022c,b). This register has been experimentally validated for accuracy in previous works (David et al., 2010; Zhang & Hoffman, 2015; Khan et al., 2018; Desrochers et al., 2016). While we have validated results against rack-level energy measures where we see per-application energy savings reflected in global rack energy measurements, we chose to use MSR_PKG_ENERGY_STATUS instead as the granularity of the rack level measurements (on the order of seconds) made it difficult to attribute detailed energy use to specific system events.

For hardware PMUs such as cycles, instructions, and last-level cache misses, we use processor specific Intel Performance Monitor Events as described in Chapters 18 and 19 of the Intel manual (Intel, 2022c). In particular, we use `Unhalted References Cycles` as it counts at a fixed frequency regardless of the processor's operating frequency – which is adversely impacted by mechanisms such as DVFS which slows down a processor's frequency for decreased energy consumption. As these hardware counters need to be initialized on each core before they can be used, a single per-core variable was added to start the setup code for each counter and then on the next interrupt after a millisecond has passed will the first data entry be logged. The millisecond gap for collecting logs is due to the sampling granularity of MSR_PKG_ENERGY_STATUS register. Details of this log collection in `ixgbe_msix_clean_rings()` is shown listing 2.3.

```
1 static irqreturn_t ixgbe_msix_clean_rings(int irq, void *data) {
2     struct ixgbe_q_vector *q_vector = data;
```

```
3
4     // index is unique to each core
5     int index = q_vector->v_idx;
6
7     // there is an ixgbe_logs[] array for each core
8     il = &ixgbe_logs[index];
9
10    // icnt is used to index into each IxgbeLogEntry
11    icnt = il->itr_cnt;
12
13    // ensure log entries stay within pre-allocated memory bounds
14    if (icnt < IXGBE_LOG_SIZE) {
15        // access each IxgbeLogEntry
16        ile = &il->log[icnt];
17
18        // save timestamp
19        now = ixgbe_rdtsc();
20        write_nti64(&ile->Fields.tsc, now);
21
22        // get last saved timestamp
23        last = il->itr_joules_last_tsc;
24
25        // capture after ~1 ms has passed
26        if ((now - last) > ixgbe_tsc_per_milli) {
27            // update new saved timestamp
28            il->itr_joules_last_tsc = now;
29
30            // save joule counter
31            rdmsrl(0x611, res);
32            write_nti64(&ile->Fields.joules, res);
33
```

```
34              // save instructions counter
35              rdmsrl(0x309, tmp);
36              write_nti64(&ile->Fields.ninstructions, tmp);
37              ......
38          }
39      }
40
41      /* EIAM disabled interrupts (on this vector) for us */
42      if (q_vector->rx.ring || q_vector->tx.ring)
43          napi_schedule_irqoff(&q_vector->napi);
44
45      return IRQ_HANDLED;
46  }
```

**Listing 2.3:** `ixgbe_msix_clean_rings` data collection psuedocode

In order to correctly isolate these millisecond timing differences, a `ixgbe_rdtsc()` function was added which calls the `rdtsc` instruction to return a 64 bit value that records how many CPU ticks took place since the processor was reset. This `rdtsc` instruction is key in ensuring stability of measurements under impact of DVFS on application performance as it counts at a fixed rate regardless of different processor frequencies. This tick count can then be easily converted to microseconds by a simple division with that fixed processor frequency rate. Furthermore, the `rdtsc` instruction exists on each individual core and simplifies the multi-core data gathering approach by avoiding the use of global clock sources and synchronization.

To lessen the impact of the data collection on application performance, we use non-temporal store instructions to write collected data. As these data are not expected to be read again soon (usually only read after the experiments are over),

the non-temporal instructions ensure they do not need to follow cache-coherency rules and therefore can have less of an impact on performance. We instrumented two new functions: `write_nti64` and `write_nti32` which wrap around the `movnti` non-temporal instruction to store 64-bit and 32-bit data respectively, their use is also shown in listing 2.3.

#### *2.2.1.4 Efficient Data Retrieval*

Having integrated the logging facilities into the device driver, one must come up with a strategy for exposing the data for transport off the system. Our logging implementation in Linux exploits the `seq_file` interface (Corbet, 2003) so that file commands, such as `cat`, can be used to both save the output in a formatted way and also automatically reset values in `struct IxgbeLogEntry` for reuse. To enable this, we create a set of files for each core in order to easily retrieve the contents of the logged data. Upon initial loading of our custom NIC device driver module, we create the directories `/proc/ixgbe_stats/core/N`, where `N` is the specific core number. Next, we defined a set of file operations: `ct_open`, `ct_start`, `ct_next`, `ct_show`, `ct_close` for `seq_file` interface to work. While (Corbet, 2003) provides general examples of how to create and use custom file operations, the specific implementation in the 82599 NIC driver is summarized below:

1. `ct_open` is a function that enables opening of the specific file and we also use its unique inode identifier as a private member data to represent a per-core variable.

2. `ct_start` starts the inital reading of data and passes in a generic iterator, however we overload it and simply start a numeral counter that begins at 0 and return that counter. We also cannibalize this function to reset the log

data variable after we find it has printed all lines of data by simply checking the position counter against the number of logged data entries.

3. `ct_show` is the main function that prints the log data in a formatted manner and we use the function `seq_printf` for this use case. The position counter from `ct_start` is passed to this function and it is used to index a specific log entry that is then printed.

4. `ct_next` is then used to simply increment the position counter for the next line to be printed.

5. `ct_close` does some cleanup to free the allocated position counter pointer variable.

As we are moving data from kernel memory into a file in userspace, Linux does impose memory bandwidth limits for this data copying. Luckily, the `seq_file` implementation already contain internal mechanisms to pace this memory copying without additional modifications on our part.

### 2.2.2 EbbRT Implementation

As *itrLog* is designed to be agnostic to both the OS and applications, EbbRT's implementation reuses the exact same data structures and data collection logic as described above in Linux. In the paragraphs below, we will summarize a few key points of how EbbRT's implementation differs from Linux.

#### *2.2.2.1 Agnostic Infrastructure*

The modifications to EbbRT's 82599 device driver can be found at https://github.com/handong32/EbbRT/tree/baremetalNIC. Given that EbbRT has a much sim-

pler code base, we only needed to add data collection code in the main 82599 device driver source at `src/native/IxgbeDriver.cc` and `src/native/IxgbeDriver.h`. Similar to Linux, we initialize the exact same data structures as shown in listing 2.1 in EbbRT's device driver *init* function. As EbbRT is used to create a single application and system binary, it does not have the same loadable module support as Linux.

#### 2.2.2.2 *Epoch-based*

EbbRT has two main functions, `Send()` and `ReceivePoll()`, for network communication. `Send` contains the main functionality for packet transmission as well as clean up logic to free transmitted resources back for reuse, we also instrumented counters in this function to keep track of bytes and descriptors transmitted. `ReceivePoll()` contains the main logic for handling packet reception; as it is the main interrupt handler for the NIC; our log collection functionality is instrumented at the beginning of this function. We use the same data collection code as listing 2.2 in `ReceivePoll()` function instead. EbbRT also takes advantage of MSI-X interrupt handlers to access per-core data for logging.

#### 2.2.2.3 *Data Logging Statistics*

As EbbRT runs on the exact same hardware as Linux, we use the same Intel PMU counters, RAPL energy counters, and `rdtsc` counters in order to log the same data. EbbRT maintains its own set of software statistics such as bytes received and transmitted and we also store that into the `struct IxgbeLogEntry`. We also copied the same `write_nti64` and `write_nti32` temporal store functions introduced in Linux in order to lessen application performance impacts.

#### 2.2.2.4 *Efficient Data Retrieval*

Given that EbbRT does not have a file system with which to store the collected log data; we built a custom server that responds to queries from a Linux client by packaging the raw log data as a byte stream and transmits over TCP protocol. On the Linux client machines, we use the `socat` application to create the initial TCP connection in order to receive the log data, next, we built a custom application to parse the byte stream into the original stored format for local storage. The details of these scripts can be found at https://github.com/handong32/energy_trace_experiment_scripts/.

## 2.3 ENABLING STATIC HARDWARE SETTINGS

Having created the requisite data collection infrastructure above in Linux and EbbRT, this section details the two hardware mechanisms studied in this dissertation to enact batching and processor frequency changes, namely through the use of interrupt delay (ITR) and dynamic voltage frequency scaling (DVFS) mechanisms. Below, we discuss these mechanisms in detail as well as how they can be toggled in both OSes.

### 2.3.1 ITR

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 (Intel, 2021; Mellanox, 2022). 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. In this work, we use this mechanism on a Intel 82599 10GbE NIC (Intel 82599 10 Gigabit Ethernet Controller:

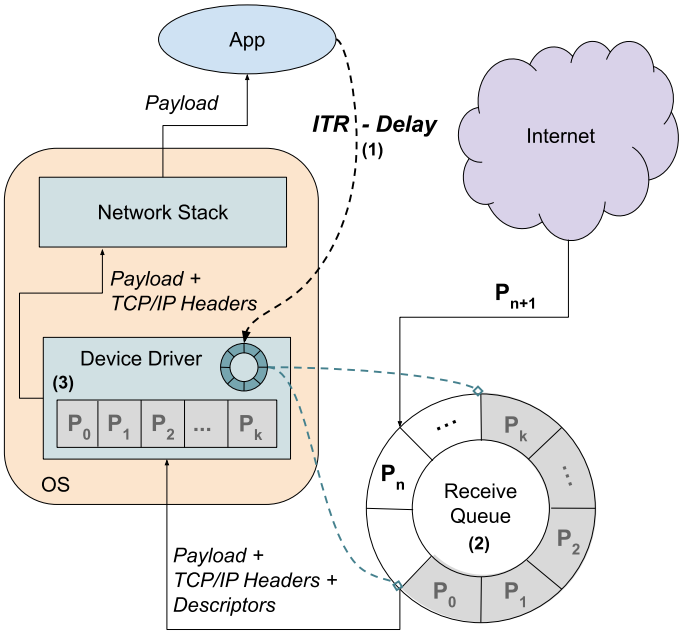


**Figure 2.4:** As ITR (1) is set, receive queue (2) buffers incoming packets as the NIC counts down to ITR. Once the interrupt is fired, device driver work on new packets (3) until it reaches its budget or no more work to do.

Datasheet, 2022) via its Interrupt Throttling (ITR) register (int, 2022). Software uses the ITR register to configure a delay in 2$\mu$s increments from a range between `0` and `1024` $\mu$s. If the spacing of events, such that packet reception is less than the ITR value, the NIC will delay interrupt assertion. If on the other hand events are sufficiently separated an interrupt will be asserted immediately. Figure 2.4 illustrates the interaction of ITR values with the rest of systems software.

ITR serves as an external control to probe and study how the OSes behave in response to externally induced batching delays. By artificially inflating the ITR values, this can enable energy savings by amortizing costs such as interrupt processing and OS book-keeping by inducing the batched handling of packets as well as prolonging idle periods that can then be exploited by processor sleep states (Le Sueur & Heiser, 2011). However, its use must be also weighed against its

impact on application latency (Elnozahy et al., 2003; Chou et al., 2019). Our goal is to expose if this type of batching, induced by ITR, has a measureable impact on application performance and energy at a offered load.

Linux's network device driver uses a dynamic algorithm that seeks to tune the ITR value such that it better reflects the current workload (Kan Liang, Andi Kleen, and Jesse Brandenburg, 2010). It is possible to disable this dynamic algorithm through the flip of a bit inside the device driver. After flipping this bit, we use `ethtool` (Tom Herbert, Willem de Bruijn, 2022) to then statically set ITR values such that interrupts can now be fired at some fixed rate and we also explore its performance and energy impacts.

One difference between EbbRT and Linux is that EbbRT inherently does not have a dynamic policy for adjusting ITR values since its implementation in Linux relies on other systemic assumptions such as jiffies and NAPI polling budgets. We enable the static setting of the ITR register in EbbRT via a simple function that is callable by any application or system code.

### 2.3.2 DVFS

DVFS is well-known hardware control knob for setting a CPU core's frequency and has been extensively studied (Sasaki et al., 2013; Flautner et al., 2001; Dominik Brodowski, Nico Golde, Rafael J. Wysocki, Viresh Kumar, 2022; Lefurgy et al., 2007; Cochran et al., 2011; Isci et al., 2006; Li & Martinez, 2006; Lee & Kim, 2009; Kim et al., 2008; Ge et al., 2007; Spiliopoulos et al., 2011; Kondo et al., 2007; Le Sueur & Heiser, 2011; Freeh et al., 2007; Elnozahy et al., 2003). DVFS states are composed of a combination of a core's clock frequency and voltage; it is often reflected by the

following equation (Le Sueur & Heiser, 2011):

$$P = CfV^2 + P_{static} \tag{2.1}$$

where P is processor's power usage, C is its switching capacitance, f is processor frequency, V is operational voltage and $P_{static}$ is the circuit leakage costs.

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 (Le Sueur & Heiser, 2011; Brooks et al., 2000; Kim et al., 2015b; Chou et al., 2019), 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 – as opposed to big-little or re-configurable core architectures. This implies that workloads which primarily use memory operations will suffer fewer performance penalties induced by a slowed processor frequency while continuing to attain energy saving benefits. We view DVFS as a speed control setting that can dilate CPU processing components of the network request timeline in exchange for reduction in energy.

This mechanism in Linux is typically set dynamically according to current processor load by a policy governor (Dominik Brodowski, Nico Golde, Rafael J. Wysocki, Viresh Kumar, 2022). This dynamic setting can be disabled through Linux boot options and static values can be written instead using the `IA32_PERF_CTL` register (Intel, 2022c) on our Intel processors via MSR tools (msr, 2022). In this dissertation, we explore static DVFS settings **from 1.3 GHz to 2.9 GHz** as they are existing limits supported by Linux's policy governor.

Similarly as before, since EbbRT does not have a dynamic policy to update DVFS due to dependence on Linux system semantics, therefore we expose a func-

tion that is callable by any application or system code in order to write to the same `IA32_PERF_CTL` register with the same static DVFS settings as Linux. We use EbbRT's own MSR tools library to write to this register.

## 2.4 BOOTING BAREMETAL OSES

In order to conduct a study of this scale, it is critical to control and automate large numbers of hardware experiments. Each experiment typically involves power cycling a hardware node, configuring and booting it with a carefully controlled software stack, conducting a series of network benchmarks, and finally retrieving and archiving the collected data. To enable this level of automation, we describe below the setup of our experimental test-bed.

We use the Mass Open Cloud (MOC) platform (Mass Open Cloud, 2022) to both run Linux (§3.2.1) and EbbRT (§3.2.2) experiments. On the MOC, we allocate a single server node to be the testing node for all the experiments by configuring it to boot into both our Linux appliance a EbbRT instance. We use a single bootstrapping node to orchestra the benchmarks and have also set up a preboot execution environment (PXE) infrastructure for booting Linux and EbbRT kernel images. On the bootstrap node, we use the package manager to install the following packages: `tftp-server`, `dhcpd`, and `xinetd`.

The Extended Internet Services Daemon (`xinetd`) is configured to listen for incoming tftp protocol traffic and subsequently start the `tftp-server` services. We use `grub2-mknetdir` and `grub2-mkimage` programs to seed the initial tftp-server boot directory with a default grub files for booting Linux and a custom boot directory that contains files for booting EbbRT. The only difference between the two is that EbbRT's boot directory requires a custom `grub.cfg` file which uses

`multiboot` command that points to an EbbRT `ELF32` binary instead of a Linux `vmlinux` file. The `tftp-server` protocol requires a `pxelinux.cfg` folder to organize grub boot menu's based on the server's MAC address; we place both EbbRT and Linux grub boot menu's in this folder.

Next, we set up `dhcpd` server to statically set the server's IP address in the file `/etc/dhcp/dhcpd.conf`. This step is important because there is a special `filename` parameter for dhcp configurations in order to point the tftpboot protocol to retrieve the correct grub boot files created earlier; this changes depending on whether Linux or EbbRT is being booted. Details of this setup can be found at https://handong32.github.io/guides/.

## 2.5 DATA COLLECTION OVERVIEW

Overall, we've conducted an extensive experimental study over the two OSes by statically sweeping up to 340 unique combinations of ITR-DVFS pairs for the four applications in table 3.1 and at different offered loads. Our study resulted in a dataset over 5 TB, and is currently open sourced at https://github.com/sesa/intlog. Given this large data set, we also 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. The layout of this dataset is in four folders: `mcd`, `mcdsilo`, `node`, `netpipe`, which reflect the four workloads. Inside each folder, there are sets of Linux and EbbRT log files, which contain the raw log data and each row contains the following set of data: `log_identifier`, `receive_bytes`, `receive_descriptors`, `transmit_bytes`, `transmit_descriptors`, `instructions`, `reference_cycles`, `last-level`

`cache miss`,`C1_counter`,`C3_counter`,`C6_counter`,`C7_counter`,`joules`, and `rdtsc_timestamp`. Each filename is also specially formatted in the following way: `*.i_Core_ITR_DVFS_RAPL`, where `i` is the experimental run number, `Core` is the processor core that the data is collected from, ITR is the actual interrupt delay value used, `DVFS` is the actual DVFS value that the processor core is set to, and `RAPL` is the RAPL power limiting value this package is set to. Other than the files that contain raw log values, there is also an output file that is generated by the client workload generators; this typically shows the overall performance of the server. There are also the `rdtsc` files that get generated, the `rdtsc` file contains two timestamps, the beginning of the actual experiment and the end, this is in order to clean up the log files such that they discard bogus logs that get created during the initialization and clean up the server such that the log data accurately reflects only the actual work itself.

## 2.6 VISUALIZATION TOOL

Given these collected log traces, we built a web visualization tool using Dash (ploty, 2022) that enables a user to dynamically examine system behaviour across a wide range of configurable settings, for example, figure 2.5 shows how one can view the data at different dimensions (via dropdown boxes) of ITR, DVFS, 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 chapter 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.

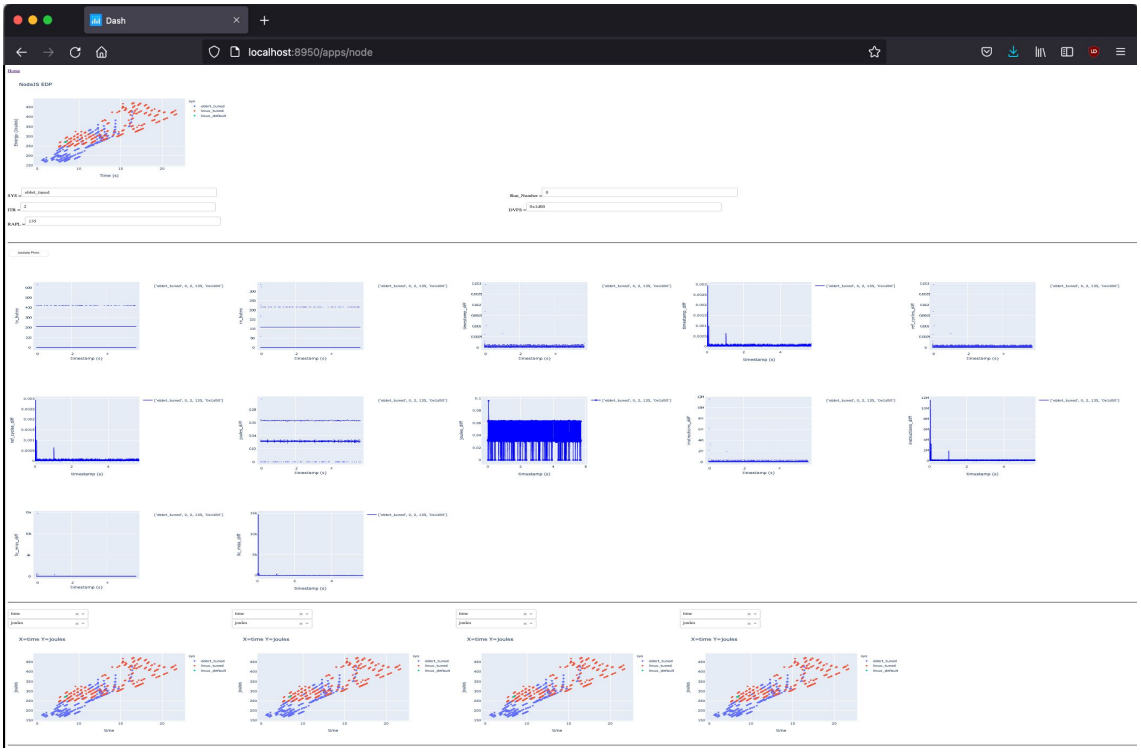


**Figure 2.5:** Small example of web visualization tool.

# CHAPTER 3

# Experimental Setup

In this chapter, we first present the hardware configuration setup of the MOC cluster used in the study. Next, the detailed system specific settings of Linux and EbbRT are discussed to provide greater context around their systems software stack. Lastly, we present how all four applications in table 3.1 are benchmarked for our experimental study.

## 3.1 MOC HARDWARE CONFIGURATION

Our MOC 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 as modern, the two mechanisms used, DVFS and ITR delay, are still commonly supported across a range of hardware manufacturers (Mellanox, 2022; ARM, 2023; Hanford et al., 2018).

We ensured the server node hosting both Linux and EbbRT kernel images are setup in a similar way by carefully configuring IA-32 Architectural MSRs and processor specific MSRs (see Tables 35-2 and 35-18 in (Intel, 2022c)) 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 work have included TurboBoost in performance-energy studies (Chou et al., 2019; Kim et al., 2015b; Guliani & Swift, 2019; Prekas et al., 2015), there have also been reports of energy use anomalies when used with different sleep states (Le Sueur & Heiser, 2011).

## 3.2 SYSTEMS SOFTWARE STACKS

As with all experimental efforts, one important step is to control as many external and non-deterministic perturbations as possible. This allows one to gather base line results and gain confidence that measured values are causally related to the change itself and not simply the result of system noise. To address this in Linux, we constructed a set of application specific appliances (Shaffer, 2000). For EbbRT, we take advantage of its library OS framework to construct a set of application specific binaries that links the application code with the OS stack in a single domain of execution.

### 3.2.1 Linux Appliance

Appliances (Shaffer, 2000) are a relatively old idea, often understood as a self-contained system image containing just enough software and operating systems support to run a single application, thereby avoiding running the long list of standard processes that can perturb systems experiments.

We built a set of these Linux appliances for the four workloads listed in table 3.1. 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 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 (Ren et al., 2019). 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.

### 3.2.2 EbbRT Library OS

EbbRT (Schatzberg et al., 2016) is used as a platform to investigate the performance and energy impacts of a specialized OS. EbbRT provides a framework for building per-application library OSes and originally only ran within a virtualized environment through its custom `virtio` device driver. For this experimental study, we ported EbbRT to run baremetal by developing a device driver for the Intel 82599 family of NICs; this port can be found at https://github.com/handong32/EbbRT/tree/baremetalNIC. EbbRT's NIC device driver is written in C++ and totals over 3000 lines of code. It interfaces with a multicore TCP/IP network stack. The device driver programs the NIC using per-core queues and interrupts, maintaining the affinity of TCP connections to their respective cores.

EbbRT consists of specialized library OS components written in C++; all components are multi-core functional and optimized to aggressively use per-core memory and fine grain locking. This also includes its 82599 NIC driver, custom TCP/IP stack, virtual and physical memory allocators which make aggressive use of large pages and pinned memory to avoid page-faults, and generic I/O buffers designed to enable zero-copy application data processing. It is packaged as a library of configurable modules and *gcc-5.3.0*-based tool-chain targeting the base components of the OS.

| 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 3.1:** Workload configurations. The column *Nature* indicates open (OL) -versus-closed (CL) loop nature and *CPU* indicates application work demand.

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 a MOC node. Once booted, OS and application code is executed under a single supervisor privilege domain. Prior work in EbbRT have also demonstrated the benefits of both compile-time and link-time optimization on library OS function dispatching (Schatzberg et al., 2016). Compared to general purpose OSes, non-preemptive processing via a specialized OS and application binary enables library OS components to avoid many checks and streamline execution, this ranges from interrupt dispatch to application logic.

## 3.3 APPLICATION WORKLOADS

In our study, we break the space of network oriented applications down along Closed-versus-Open Loop (Schroeder et al., 2006) and level of complexity in CPU usage. We select a representative benchmark for each quadrant (see fig. 3.1). Below, we expand on the importance of each quadrant and the benchmarks we used. Table 3.1 lists the four network-driven applications that we target and their respective attributes.

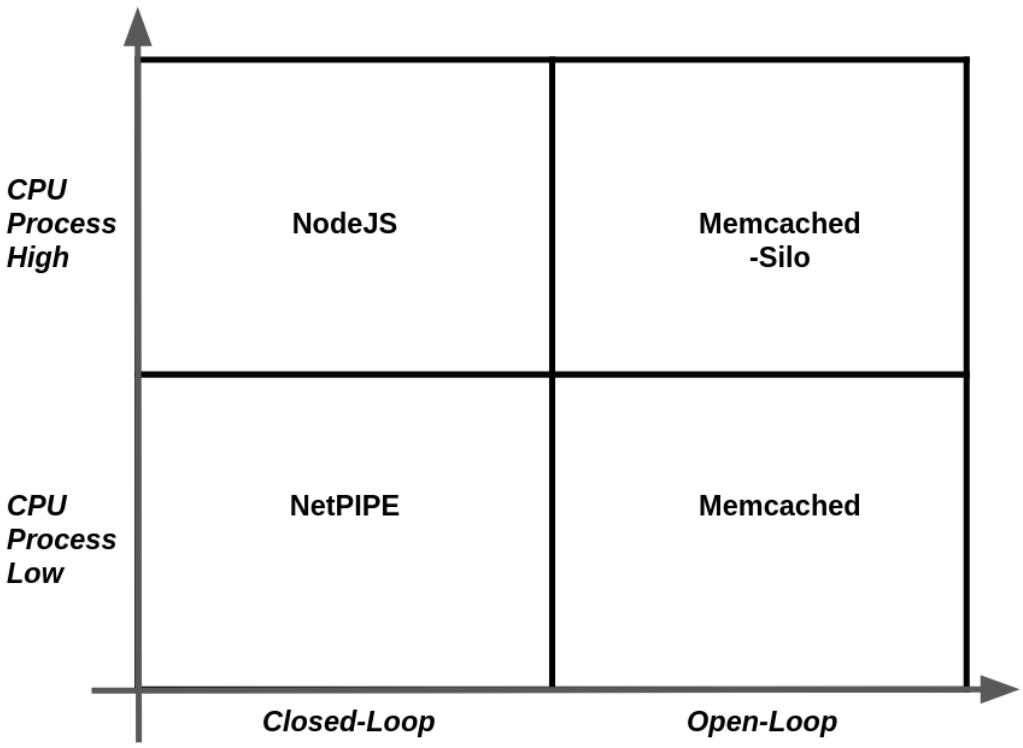


**Figure 3.1:** Graphical breakdown of the workloads and in which quadrant they lie in. CPU processing represents the ratio of application work versus systems work i.e. CPU processing high means more time is spent in application work on a per request basis. Closed loop workloads are all single connection workloads between a single client and server while open loop workloads consists of multiple clients with thousands of connection and performance is typically measured as tail latency at some percentile of requests.

NetPIPE and NodeJS Webservers are single core and TCP connection closed-loop applications due to the fact that their performance (throughput) is dependent on the responsiveness of the constituent server and client components to conduct the synchronous packet request transmit and receive work. In contrast, Memcached and Memcached-silo are open-loop experiments, as their performance boils down to the ability of the server node, running on all physical cores and thousands of TCP connections, to maintain a 99% response tail latency under 500 $\mu s$ under an offered load that is generated by external client nodes. NetPIPE and Memcached are OS-centric workloads by virtue of their computationally light application logic and is heavily dominated by the OS stack work for packet transmissions. In contrast, the NodeJS Webserver and Memcached-silo are application-centric workloads as their runtime and application logic are more computationally intensive in comparison to the OS stack work. Details of the workloads, their deployment and other benchmark information is elaborated below.

### 3.3.1 Closed Loop

#### *3.3.1.1 NetPIPE*

Netpipe (Snell et al., 1996) involves sending messages of identical size between two identically configured systems in both HW and SW for a fixed number of iterations. Given our goal of studying and explaining the implications of the OS on network driven processing our analysis framework, and evaluation, includes closed loop settings.

Figure 3.2 illustrates the deployment of the application between a client and server machine where the same binary is executed on both machines. The benchmark consists of the two machines sending and replying with a payload of the

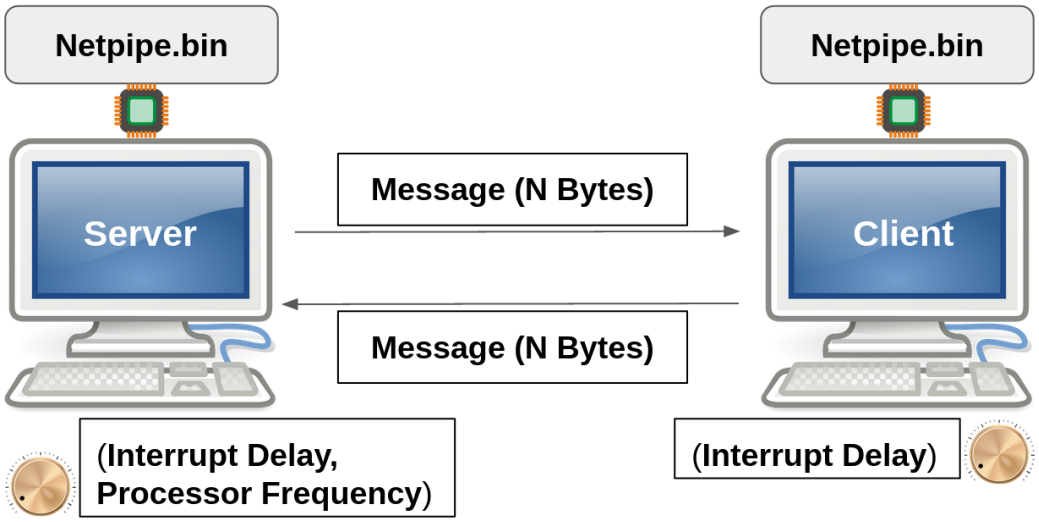


**Figure 3.2:** NetPIPE benchmark.

same bytes back and forth for a fixed number of iterations; the resulting time to do this communication results in a throughput value as a measure of performance. We run NetPIPE in a symmetric configuration where the same interrupt delay value is controlled on both client and server node, this is primarily due to the closed-loop nature of the workload and to explore the tighter integration of interrupt delay values in affecting performance and energy use. On the server side, we further run additional experiments where the processor frequency knob is manually controlled to see if it impacts the performance and energy. NetPIPE is a non processing intensive application as the user level's work only consists of responding back to each payload with a response payload of the same data. NetPIPE allows us to also change the message size, therefore opening up the scope of how the interrupt delay and processor frequency affects different message sizes. The data generated from running this workload is akin to middle-ware and video streaming services commonly found in data centers where payloads of varying sizes are frequently are communicated between different machines.

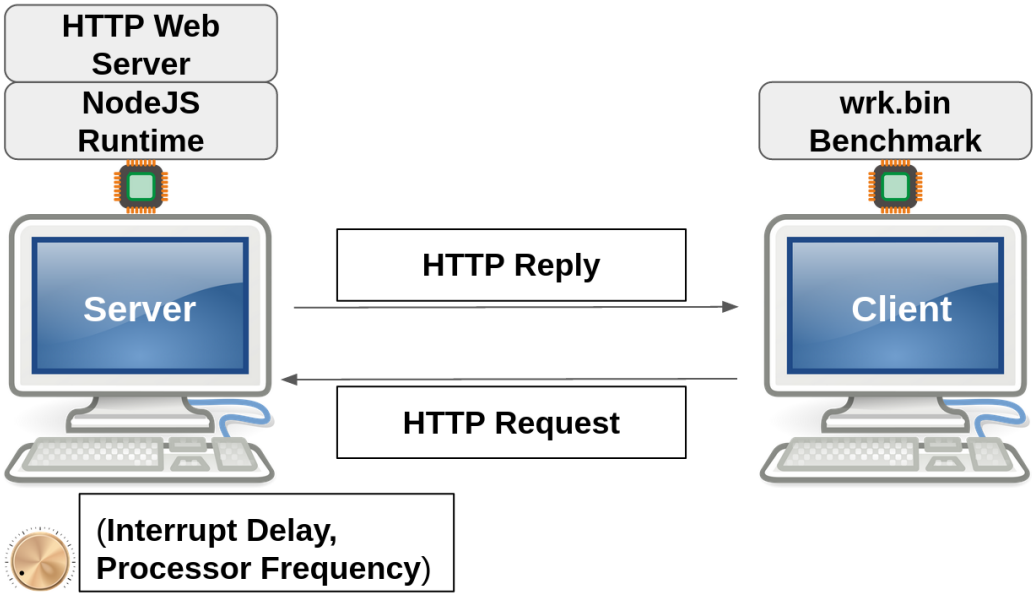


**Figure 3.3:** NodeJS HTTP Web Server benchmark with wrk workload generator.

In the Linux experiments, we run NetPIPE-3.7.1 while EbbRT uses a custom version of NetPIPE's protocol that is ported to its networking interfaces. In our experimental results below, we fix the iteration count at 5000 and show results for a range of message sizes. We found that the 10 GB link is close to saturation when a message of size greater 700 KB is exchanged and both Linux and EbbRT's throughput begin to hit the same plateau. As message size increases in the NetPIPE benchmark, the workload becomes more network bound; therefore a system's network path specialization has less of an impact on performance.

#### *3.3.1.2 NodeJS HTTP Web Server*

Figure 3.3 shows the NodeJS benchmark (Joyent, 2013), which consists of a JavaScript HTTP Webserver running inside a NodeJS runtime. In contrast to NetPIPE, this workload is more CPU intensive as it runs a NodeJS runtime below the HTTP Web Server. The runtime is involved in receiving the HTTP Request packet, which

contains a JavaScript body and must parse it and then pass that packet up into the webserver for processing. The same application flow also happens for every HTTP Reply packet as well. Given that this is not as a symmetric setup as NetPIPE, we controlled interrupt delay and processor frequency only on the server node running the webserver. On the client side, the default policies of interrupt delay and processor frequency was left as is and a single core runs the *wrk-4.0.2* (Glozer, 2014) benchmark to send web requests to the NodeJS server for a fixed period of time. We modified *wrk-4.0.2* to place a fixed request load of 100K. The server responds to each request with a small static payload of size 148 bytes. Linux runs nodejs-0.10.46, and EbbRT runs the same version ported to support bare-metal NodeJS by providing OS interfaces that link with the V8 (Google, 2022) JavaScript engine and libuv (libuv, 2022).

### 3.3.2 Open Loop

#### *3.3.2.1 Memcached*

Figure 3.4 illustrates the deployment of memcached (https://memcached.org, 2020), it is a multi-threaded workload that runs on all 16 cores of one server node. It consists of an unloaded client node running mutilate (J. Leverich, 2022). 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 server. Mutilate (J. Leverich, 2022) is used to generate three different requests-per-second (QPS) rates of 200K, 400K, 600K, etc., for a fixed period of 20 seconds each on a single memcached server, we also pipeline up to four connections to further increase process-

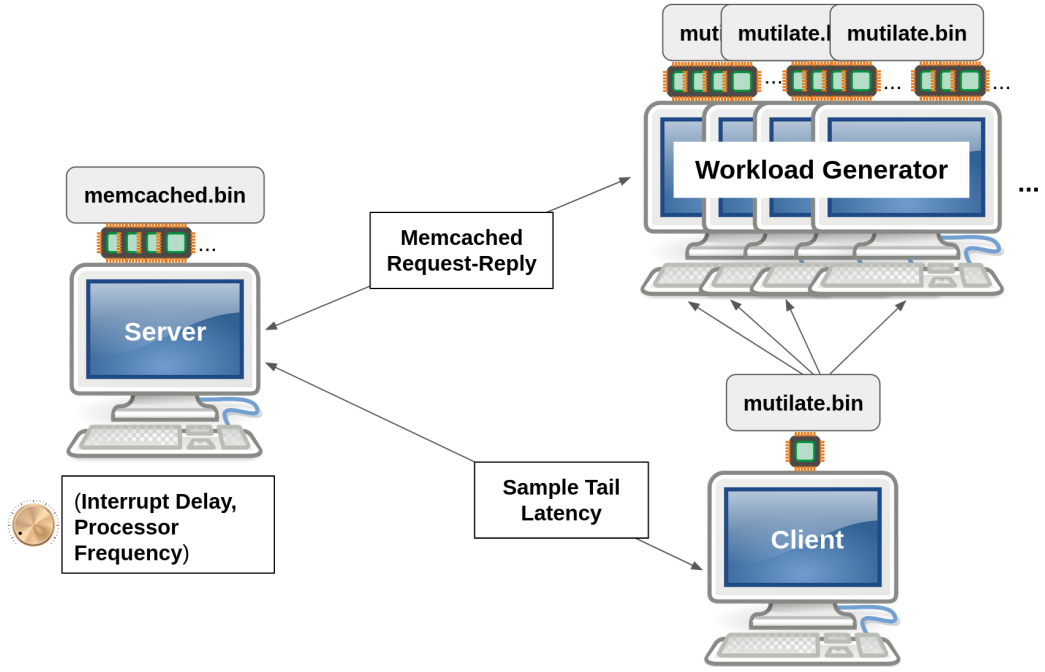


**Figure 3.4:** Memcached benchmark with mutilate workload generator.

ing load. For each experimental run, we manually fix the server's interrupt delay and processor frequency values and use the mutilate clients to generate request loads at specific QPS rates. EbbRT uses a re-implemented version of memcached, written to its interfaces, which supports the standard memcached binary protocol. To alleviate lock contention, an RCU hashtable is used to store key-value pairs. We run a representative load from Facebook (Atikoglu, Berk and Xu, Yuehai and Frachtenberg, Eitan and Jiang, Song and Paleczny, Mike, 2012) (ETC) which represents the highest capacity deployment. It uses 20 - 70 byte keys and 1 byte to 1 KB values and contains 75% GET requests.

For an open-loop workload, we consider it as non processing intensive as 75% of requests are GET which entails reading a in-memory data structure and responding to the request with that payload. Furthermore, memcached protocol is not processing intensive as it mainly deals with maintaining a in-memory lookup

table, so most transactions on the table involve memory copies. In contrast to the workloads in sections §3.3.1, the client machines running mutilate generates floods of requests, typically pipelined up to four per thread, to the memcached server. A single client machine that is not loaded is used to periodically sample tail latency values from the memcached server and the metric of performance used in this case is maintaining 99% tail latency under a specific threshold as the loads that mutilate clients generate increases. Memcached is an application that is often deployed in datacenters as it is very effective at supporting high fan-out requests from millions of end users requesting similar resources and as a result, is often a research topic in optimization (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; Schatzberg et al., 2016; Belay et al., 2014; Prekas et al., 2017).

#### *3.3.2.2 Memcached-Silo*

Memcached-silo (Prekas, 2017; Prekas et al., 2017) is a workload built on top of the normal memcached protocol. It is more computationally complex as it incorporates both latency-sensitive network compute and memory-intensive TPC-C style transaction processing via the Silo database (Tu et al., 2013). The modified memcached-silo server supports memcached binary protocol and as we use mutilate to generate requests, it will travel the same network paths as memcached application. However, a modification was added such that for every request, an extra transaction was initiated on the Silo database and this transaction is more computationally intensive. After the database transaction has been completely,

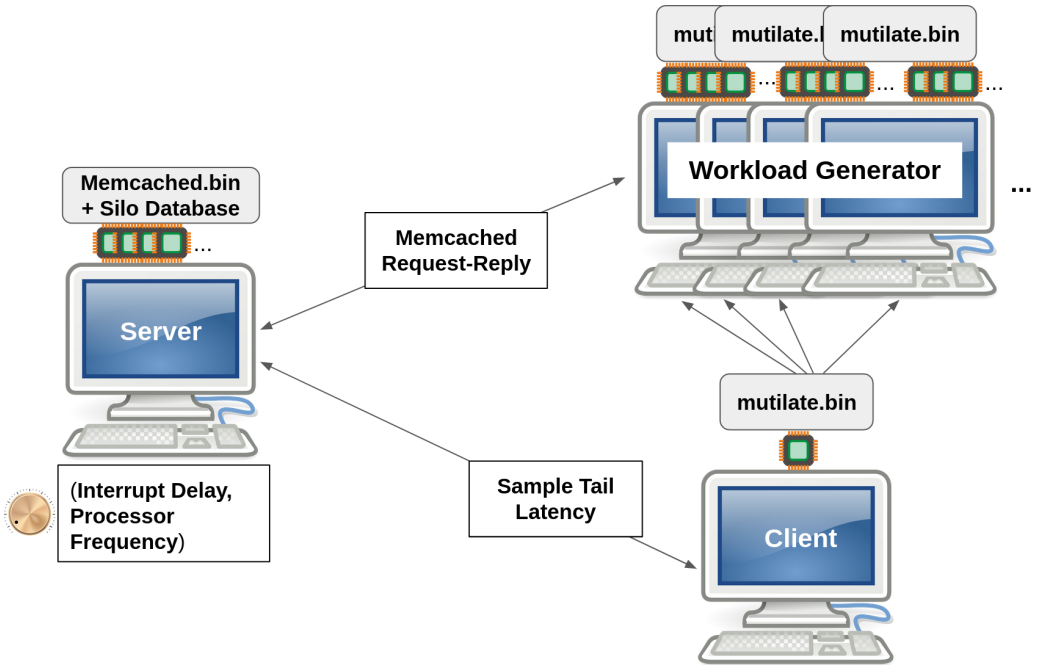


**Figure 3.5:** Memcached-silo benchmark with mutilate workload generator.

then the memcached reply will be sent. We ported the same memcached-silo implementation to EbbRT. The configuration and SLA constraints of memcached-silo follow from those of memcached and as figure 3.5 shows, it is mostly similar to memcached in deployment except the extra Silo database running on the server. Given its computationally heavier nature, we only needed two 16-core nodes at 16 connections per core to saturate our memcached-silo server. This also meant we use lowered QPS rates of 50K, 100K, 200K, etc. to account for increased computational load per request.

# CHAPTER 4

## Break Down of Network Processing in lieu of Analysis and Models

Before delving into the detailed findings and analysis from our experimental data in chapter 5, it is important to establish a baseline understanding of network processing such that one can better reason about the collected data. This chapter tackles this by providing an abstract model in order to decompose the different components of network processing and discusses how they can be implemented in different OS stacks.

Fig. 4.1 illustrates a simple request processing timeline. Although this is drawn and discussed from the perspective of a single core, our analysis and evaluation assumes that multiple cores can be used concurrently to shorten servicing times. From an OS perspective, we break down network driven processing into stages that allows us to organize and reflect the OS and application performance-energy interactions with different offered loads. This model drives the quantitative study and helps in the construction of our analytical models in chapter 6; which can then be used to explain and explore alternatives in both software and hardware.

We decompose the timeline in fig. 4.1 into three main components in the following sections and discuss their opportunities for different performance-energy trade-offs. The *quiescent periods* section expands on OS idling policies when no requests are coming in to be processed and how the implementation of these idling policies differ between Linux and EbbRT. *OS request detection* reveals how interrupts, polling, and hybrid interrupt-polling strategies are implemented in the two OSes and its subsequent impacts on processing efficiency. The *request servicing* section discusses the potential performance and energy benefits of OS path specialization techniques.

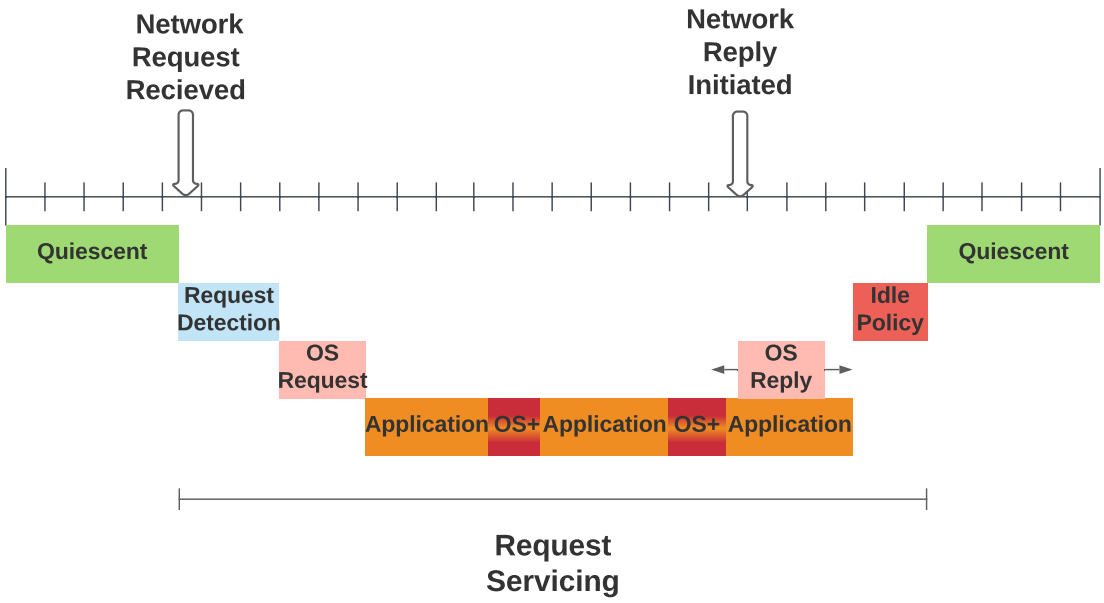


**Figure 4.1:** Application processing request timeline. Quiescent are periods between packet arrivals. Request Servicing includes all software components.

Lastly, the final section discusses how the type of network applications, Open vs Closed-loop (Schroeder et al., 2006) and application-centric vs OS-centric, interplays with all three of the above components to impact the performance-energy trade-offs space.

## 4.1 QUIESCENT PERIODS

Given the packetized and transactional nature of network driven applications, a quiescent period in which no external requests are generated, precedes all network activity on the server. During these periods, an OS typically has a built-in idle policy to reduce energy consumption.

### 4.1.1 Idle Policies

If all processing is complete, no traffic is pending and aggressive polling is not in use, the OS has a policy that selects a hardware sleep state (i.e. Intel c-states (Intel, 2022a)) to halt the core. Various optimizations have been studied (Chou et al., 2016; Meisner & Wenisch, 2012; Le Sueur & Heiser, 2011). These sleep states typically have an associated reduction in static power consumption (Meisner & Wenisch, 2012), where in the extreme, the deepest sleep state can flush micro-architectural state such as caches and power down these hardware structures. However, each sleep state also imposes a progressively larger wake-up latency, which subsequently impacts the execution speed given possible flushing of these performance sensitive states (Zhan et al., 2017). Below, we discuss how these policies can differ between a general purpose and a specialized OS.

#### *4.1.1.1 Linux Idle Policy*

General purpose OSes such as Linux use a dynamic scheduler and idle policy to decide if a core should be halted and to what sleep state (Rafael J. Wysocki, 2018). This policy exploits various system statistics to predict how long the core is likely to be idle. Linux's idle policy is set up such that a high-level algorithm decides which level of sleep state a specific core should go into; this is dependent upon the operating history of the core such as how long did it idle previously and how busy the core currently is. As we run on a Intel x86_64 architecture, once a specific sleep state has been identified, the policy uses Intel specific architecture code to physically `halt` the core. The specific sleep state is selected to save energy while minimizing wake-up latency costs. Overall, this algorithm is a subtle implementation that interacts across many layers of the OS software and device drivers.

#### *4.1.1.2 EbbRT Idle Policy*

In contrast, we use the specialized nature of EbbRT to explore a simple *race-to-halt* policy where the processor is always put into the deepest sleep state, thus ignoring any performance and energy trade-offs with using other sleep states. Originally, EbbRT's run-to-completion processing meant that if there were no further work, pending interrupts or packets dequeued from prior interrupts to process, the event manager would invoke the `halt` instruction to stop the core until another interrupt wakes the core for new work to do. We modify EbbRT to enable the use of sleep states as follows: the event-loop of EbbRT infers that the system is idle and halts the core to enter the deepest sleep state supported by our hardware (Intel C7). This is achieved by adding custom `monitor` and `mwait` instructions in the event loop code prior to calling `halt`. Also, upon waking up from a NIC interrupt, such as receiving a packet, the device driver management code, protocol processing code, and application code are dispatched and run-to-completion on a single event. EbbRT's simple implementation is similar to previous *race-to-halt* strategies that have been implemented in Linux as external policies (Meisner & Wenisch, 2012; Chou et al., 2016). EbbRT's sleep algorithm provide an alternative to examine the performance and energy differences between the two different OS stacks and their interactions with a complex versus simple idle strategy.

## 4.2 OS REQUEST DETECTION

Fundamental to any operating system is how it detects and schedules request processing in response to IO device activity such as packet reception or transmission completion. At the two extremes are interrupt and poll driven IO detection.

### 4.2.1 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) it allows 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 (whe, 2012) or sleep state exit penalties(Rafael J. Wysocki, 2018). Interrupts can also have a negative impact on the instruction efficiency. Induced micro-architecture hazards such as the inability to pre-fetch or speculatively execute across an interrupt can increase the number of cycles required.

### 4.2.2 Poll driven IO

Most modern NICs 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 efficiently 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, and subsequently higher energy consumption. There has been a large body of work to improve the performance of latency sensitive applications through the judicious use of polling (Belay et al., 2014; Peter et al., 2015; Prekas et al., 2017; Ousterhout et al., 2019; Schatzberg et al., 2016; Cadden et al., 2020; Qin et al., 2018; Jeong et al., 2014; whe, 2012; Marinos et al., 2014).

### 4.2.3 Linux NAPI Policy

Linux uses a hybrid policy where its New API (NAPI)(The Linux Foundation, 2022) framework uses interrupts when the load is low and switches to polling when load is high and back to interrupts when load reduces; this poll phase is bounded to avoid starving other devices and software. NAPI is a complicated algorithm as it bounds packet processing through dynamic budgets that are set by Linux's scheduler and is also influenced by the NIC driver's current processing efficiency.

### 4.2.4 EbbRT

When an IO device activity event first occurs in EbbRT, it processes the interrupt in a run-to-completion model. From this, EbbRT implements two different types of request processing for future IO activities: interrupt-driven and poll-based. Similar to EbbRT's idling policy, the interrupt-driven and poll-based approaches detailed below are much simpler compared to Linux's dynamic policies. By implementing these different policies, another goal of this study is to help reveal the contrast between the two OS stacks by presenting a detailed quantification of execution trends and power events in order to uncover how OS functionality impacts overall system execution and what are the possible OS-level adaptations that can achieve even better performance and energy trade-offs.

#### *4.2.4.1 Interrupt-driven*

In this mode, EbbRT uses a configurable constant, set to *64*, for all our experiments at the moment, that is used to control how many packet descriptors[1] can be

[1] A descriptor is a data structure that holds the memory location of the packet data along with packet processing information such as checksums, length, etc.

processed in a single interrupt invocation before returning to the event-loop of the core on which the interrupt was processed. If a total of *64* descriptors are processed or if there are less than *64* descriptors to process then EbbRT will simply halt the processor to the deepest sleep state until the next IO activity wakes the core. We borrowed this value, *64*, from default NIC settings in Linux's device driver and this constant also helps to induce a simple and bounded per-cpu device-level poll. In contrast to Linux's dynamic NAPI budgets, our design is much simpler as the constant is fixed at compile time and does not need additional logic to dynamically adjust its value during runtime.

#### *4.2.4.2 Poll-based*

The simple run-to-completion, and lightweight event-driven execution model of EbbRT also allowed us to also explore the performance-energy trade-offs of using DVFS 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 REQUEST SERVICING

Once the OS detection mechanism identifies the NIC has data to process, several components of the software must be run in accordance with the execution model of the OS. Typically, a network stack parses the packet header and eventually enqueues the payload to the application for processing. OS request servicing distinguishes general purpose systems from specialized systems as general purpose

OSes are designed to support multiple applications, NICs, and different dynamic policies. In contrast, specialized OS's can be compile and runtime specialized for a single request optimized servicing path. Below, we expand on these differences for the request servicing component.

### 4.3.1 Linux Network Processing

Network processing on a general purpose OS, such as Linux, is typically split between two levels of scheduling; 1) interrupt level in which minimal work is done but at highest critical priority and runs-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. 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).

### 4.3.2 EbbRT Network Processing

In contrast to Linux, a specialized OS such as EbbRT sheds much of the above complexity. As it consists of a single application-specific binary that is compile and link-time optimized with the OS code, both OS and application code is executed under a single supervisor privilege domain. This enables the potential for improved performance and energy efficiency through OS specialization that can

void software overheads associated with general purpose OSes.

## 4.4 POTENTIAL PERFORMANCE-ENERGY TRADE-OFFS IN DIFFERENT APPLICATIONS

As discussed in §3.3, network applications tends to fall into two main categories dependent on the nature of the application and its processing. The nature of the application, i.e. **Open or Closed-loop**, drives the length of the *quiescent periods* and different offered loads impacts the time required for *OS request detection* and *request servicing*. The degree of processing, i.e. **application or OS-centric**, is mainly impacted by *request servicing* component as the efficiency of the OS stacks often influences overall performance and energy consumption when servicing network requests. Below, we discuss the broader implications in these interactions between OS stacks and the different application types.

### 4.4.1 Open-Loop

In an open loop scenario, such as Memcached (https://memcached.org, 2020), an external request rate, such as requests-per-second (QPS), induces an inter-arrival gap that will drive the length of the *quiescent periods* – longer at lighter loads (low QPS) and shorter at heavier loads (high QPS). This external arrival rate can be considered independent of the time required to service a request and will impact the ability of idle policies to save energy. Providers will often set a Service-level Agreement (SLA) target, such as some percentage of requests to be completed under a stringent time budget (i.e. 99% tail latency < 500 $\mu s$), for these open-loop applications. There has been a wealth of research in using these SLA *budgets* to save datacenter energy by using different processor frequency settings (Wu et al., 2016;

Hsu et al., 2018; Lo et al., 2014; Hsu et al., 2015; Lo et al., 2015; Barroso, Luiz André and Hölzle, Urs, 2007; Fan et al., 2007). The performance and energy profiles of open-loop applications can be impacted by the efficiency of the OS stacks to process *request detection* and *request servicing* within these SLA budgets. Our goal is to reveal and quantify these OS effects.

### 4.4.2 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 (Barroso & Hoelzle, 2009; Meisner et al., 2011; Lo et al., 2014; Fan et al., 2007; Barroso, Luiz André and Hölzle, Urs, 2007; Barroso et al., 2003). 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. However, depending on the nature of the workload and the quiescent periods, composed of the network transmission times and client servicing time, there can still exist opportunities exploit new energy-performance trade-offs. For example, the use of a dedicated polling loop in *request detection* can be used to eliminate most of the cost of interrupts and OS path specializations can also improve performance and reduce energy during *request servicing* in these synchronous processing loops.

### 4.4.3 OS-centric

Applications such as NetPIPE and Memcached are considered to be OS-centric as the compute portion of the work is minimal compared to the OS work needed to service each request through its network stack. 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. Specialized OS paths can reduce the time spent in *request servicing* due to reduction in architectural hazards associated with interrupts, protection domain crossing, etc. This time reduction potentially increase the utility of using processor energy settings to find optimal settings for different applications.

### 4.4.4 Application-centric

In the case of a service oriented workload that has significant application work such as NodeJS and Memcached-silo, where the fraction of the instructions composed by OS network processing is small in comparison, there is also potential for both improved performance and energy by taking advantage of optimized network paths in a specialized OS. As the system no longer needs to support other processes and multiplex different devices, the entire software stack can be dedicated towards one use-case, therefore more application work can thus be done per instruction during *request servicing* compared to a general purpose OS.

# CHAPTER 5

## Experimental Findings

Below, we present findings and analysis of our static sweep over the space of ITR and DVFS in Linux and EbbRT. Figures 5.2, 5.8, and 5.12 illustrates a summary of the performance and energy trade-off space studied in this thesis; the X and Y axis are not drawn to scale in order to highlight the structures in the data. In these figures, each point plotted on the graphs represent an optimal ITR and DVFS setting that resulted in the lowest energy for the particular observed performance – all other settings resulted in higher energy consumption for the same performance. These points also represent static settings that yield optimal efficiency and represent the Pareto-optimal (Pareto efficiency, 2022) trade-off space in performance and energy. Our analysis reveals that the points composing the Pareto-optimal curves only correspond to 0.04% of all configurations explored.

For each experiment with an ITR and DVFS combination, we repeat it up to 10 times for stability and our gathered statistics show a standard deviation error less than 0.01%. Every marker in the figures shown below represents a single experimental run which is associated with a corresponding per-epoch log containing various statistics collected using our infrastructure. We will use the terms *slowing* or *slow down* of ITR and DVFS as indicators to when we artificially induce packet batching by *increasing* ITR values and lowering a processor's frequency by *decreasing* DVFS values. In the case of closed loop applications, the *performance* metric is defined as the total time (in seconds) to finish the work where lower is better. For open loop applications, *performance* is the defined as the 99% tail latency (in $\mu s$) where lower is also better. Energy is measured in Joules ($J$) from the energy counters in our data collection infrastructure to run a single experiment.

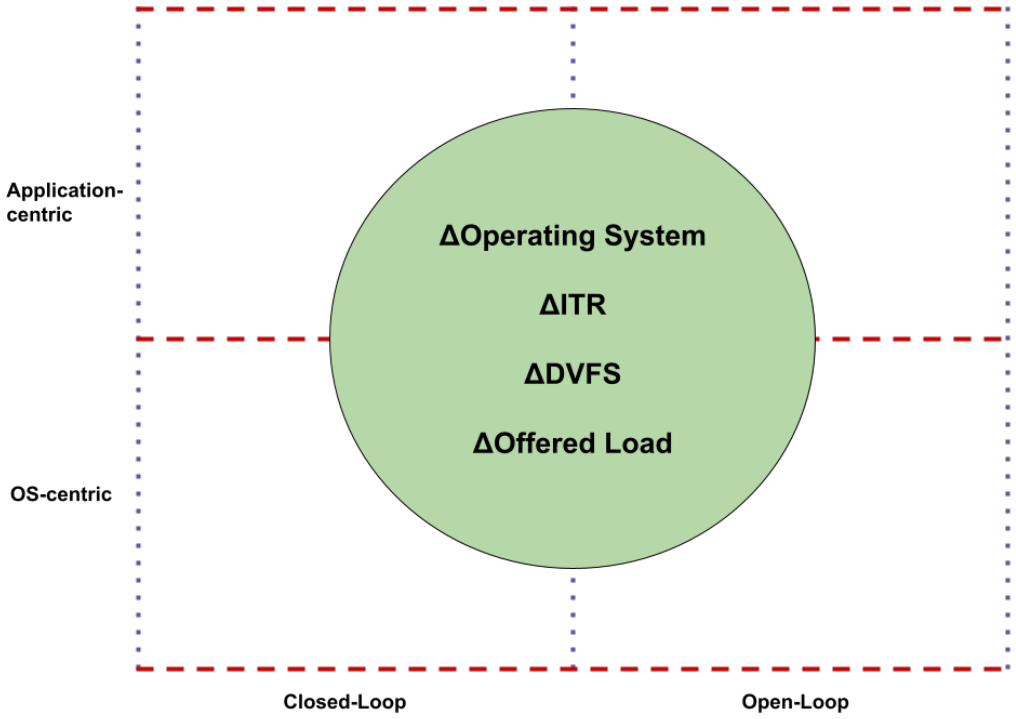


**Figure 5.1:** Break down of OS, ITR, DVFS, and offered load changes across the applications studied.

In order to provide a structured global understanding of our findings, we will decompose the following sections using the same application format as §4.4 (Open vs Closed-loop, application vs OS-centric). Moreover for each section, we expand on its details as a function of when the **OS**, **ITR**, **DVFS**, and **offered load** changes by using detailed plots shown in fig. 5.14, and fig. 5.18 – these figures use all available data in order to demonstrate its fidelity as well as the intrinsic structures within each OS stack. This break down is illustrated in fig. 5.1 and we note that in the sections below, there are often overlaps in these changes of OS, ITR, DVFS, and offered load as the energy and performance trade-offs are typically affected through coordination of multiple of these functions.

We introduce the concept of an OS's *efficiency profile*, which represents its char-

acteristic performance and energy measure, to discuss and highlight their similarities and differences. These profiles are a function of the impact of packet batching, via ITR, and processor energy settings, via DVFS, on its overall behavior, they identify optimal operating points that can be exploited to minimize energy consumption for a given performance target.

In all the figures shown below, we differentiate between Linux and Linux-static, where *Linux* has its dynamic algorithms *enabled* and *Linux-static* where they are *disabled*, and we use our support for static settings instead. We use *Linux-static* as a proxy to reveal underlying performance and energy behavior that is unique to Linux and *EbbRT-static* as a proxy for alternate system structures with optimized network paths.

| | | | NetPIPE | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | NodeJS | | 64 B | | 8 KB | | 64 KB | | 512 KB | |
| | Perf | Energy | Perf | Energy | Perf | Energy | Perf | Energy | Perf | Energy |
| Linux-static | (2, 2.9)<br>7.6 s | (2, 2.9)<br>244 J | (2, 2.9)<br>0.14 s | (4, 2.9)<br>3.1 J | (8, 2.8)<br>0.27 s | (2, 1.3)<br>6.6 J | (8, 2.9)<br>0.84 s | (12, 1.2)<br>20 J | (8, 2.9)<br>4.7 s | (28, 1.2)<br>97 J |
| EbbRT-static | (4, 2.9)<br>5.6 s | (4, 2.5)<br>166 J | (2, 2.3)<br>0.1 s | (6, 2.9)<br>2.4 J | (6, 2.9)<br>0.2 s | (20, 1.7)<br>4.2 J | (6, 1.7)<br>0.71 s | (6, 1.2)<br>13 J | (6, 2.9)<br>4.7 s | (26, 1.2)<br>85 J |
| | Memcached | | | | | | | | | |
| | 200K | | 400K | | 600K | | 1000K | | 1500K | |
| | Perf | Energy | Perf | Energy | Perf | Energy | Perf | Energy | Perf | Energy |
| Linux-static | (2, 2.9)<br>61 $\mu s$ | (350, 1.3)<br>1055 J | (2, 2.9)<br>102 $\mu s$ | 300, 1.3)<br>1167 J | (30, 2.9)<br>102 $\mu s$ | (300, 1.7)<br>1471 J | - | - | - | - |
| EbbRT-static | (2, 2.9)<br>65 $\mu s$ | (400, 1.3)<br>935 J | (2, 1.5)<br>60 $\mu s$ | (300, 1.3)<br>976 J | (2, 1.7)<br>60 $\mu s$ | (400, 1.3)<br>1025 J | (2, 2.5)<br>55 $\mu s$ | (40, 1.3)<br>1160 J | (2, 2.9)<br>57 $\mu s$ | (40, 1.3)<br>1253 J |
| | Memcached-silo | | | | | | | | | |
| | 50K | | 100K | | 200K | | 300K | | | |
| | Perf | Energy | Perf | Energy | Perf | Energy | Perf | Energy | | |
| Linux-static | (20, 2.9)<br>177 $\mu s$ | (100, 1.3)<br>1041 J | (10, 2.9)<br>201 $\mu s$ | (100, 1.7)<br>1345 J | (10, 2.9)<br>306 $\mu s$ | (100, 1.8)<br>2073 J | - | - | | |
| EbbRT-static | (4, 1.3)<br>168 $\mu s$ | (200, 1.3)<br>983 J | (4, 2.9)<br>183 $\mu s$ | (8, 1.3)<br>1034 J | (2, 2.9)<br>228 $\mu s$ | (30, 1.9)<br>1443 J | (2, 2.9)<br>319 $\mu s$ | (200, 2.5)<br>2152 J | | |

**Table 5.1:** Best static setting of ITR and DVFS across all applications and offered loads. For each cell the **(ITR, DVFS)** numbers represents the physical values used to find Perf (Highest performance) and Energy (Lowest energy) and the number below each ITR, DVFS setting is its corresponding performance (seconds for closed-loop applications and microseconds 99% tail latency measurement in open-loop applications) or energy in Joules. ITR units are in $\mu$s and DVFS units are in GHz.

## 5.1 CLOSED-LOOP

### 5.1.1 OS Changes

In fig. 5.2, we can see that different OSes respond uniquely to each closed-loop application. For example in NetPIPE @ 64 KB and NodeJS, EbbRT-static can use the same hardware mechanisms as Linux-static but conduct the same application workload using 2X lower energy while improving overall performance by 2X at the same time.

Though these OS stacks have different absolute values in performance and energy and these values are clearly separate from each other, fig. 5.2 also illustrates that the overall trade-off space is similar and that the intrinsic structure of the OSes response to ITR and DVFS are consistent across these applications even when the offered load varies. In particular, there is a characteristic "V" shape where at the bottom of the "V" shape is a point closest to origin where its configuration of ITR and DVFS yields most of the points with minimal energy used while finishing the work fastest. This shape is unique to closed-loop style applications due to the synchronous nature of its application behavior.

### 5.1.2 ITR Changes

For closed-loop applications, one would ideally like to minimize the amount of batching by artificially setting ITR values to be fast in order to quickly process the requests. This is intrinsic to closed-loop style applications as it is a synchronous process whereby the client depends on the server to respond to its request before issuing the next one. Therefore, we find an opportunity to both improve performance and energy efficiency by purposely lowering the degree of batching via fast

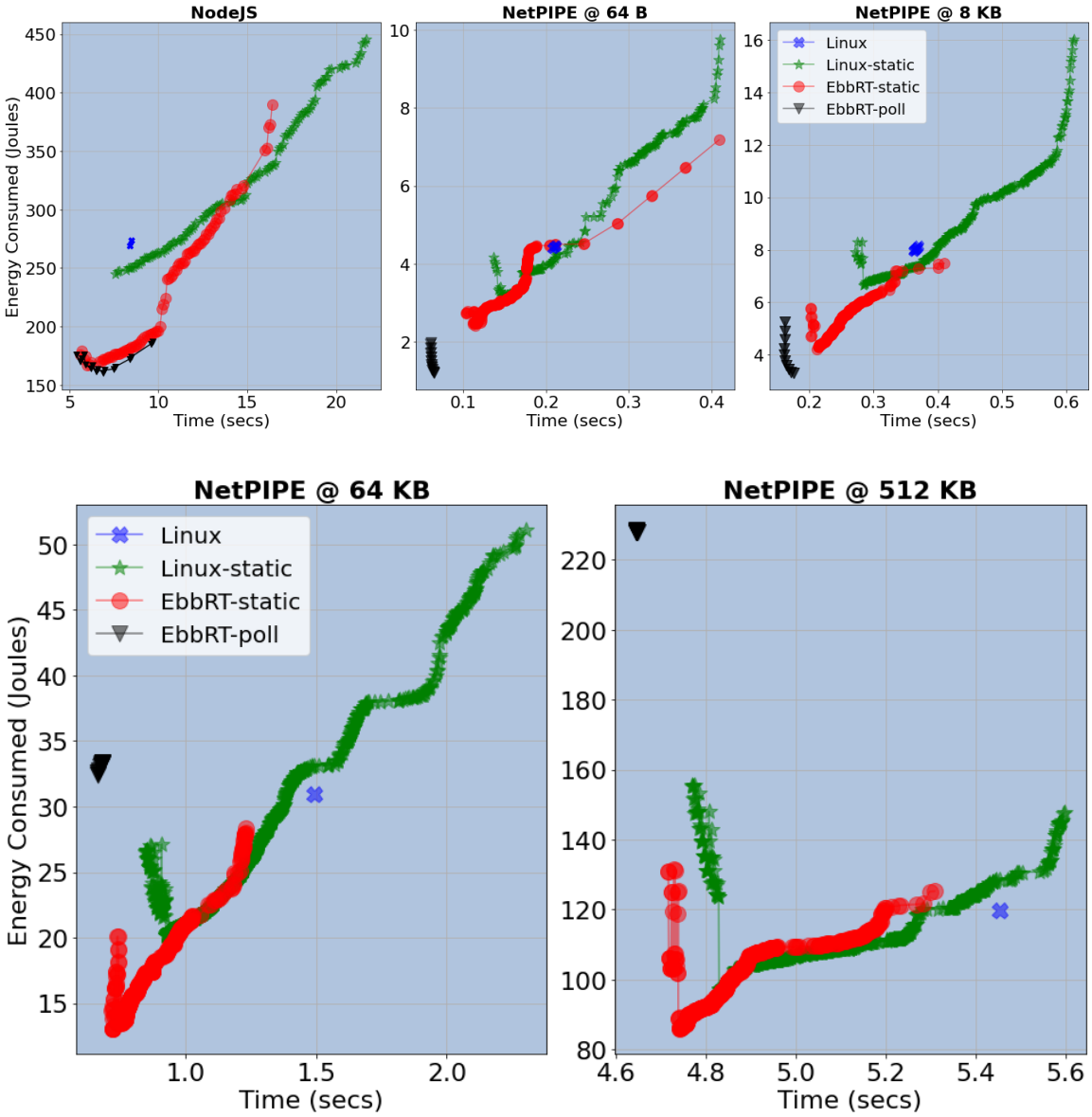


**Figure 5.2:** Pareto-optimal curves of closed loop applications.

interrupt rates. For example in NodeJS, table 5.1 shows that the optimal performance and energy profiles (set of points closest to origin in both OSes) used the lowest possible ITR values between 2-4 $\mu s$. In §5.1.2.1 below, we illustrate details of how ITR can be used to optimize performance of closed-loop applications.

#### 5.1.2.1 *Detailed Finding 1: ITR can be used to induce packet processing stability in order to greatly improve performance.*

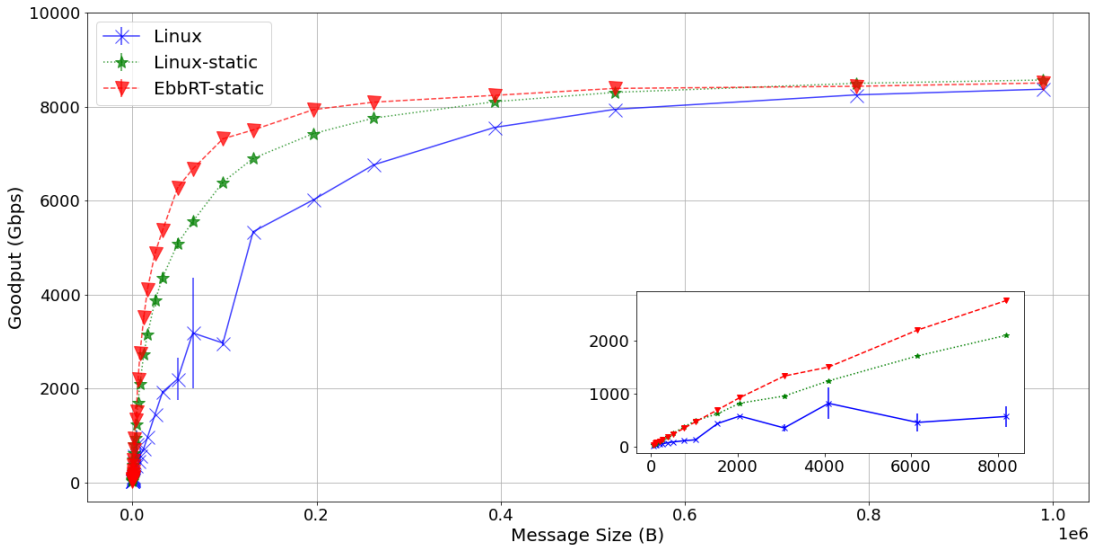


**Figure 5.3:** Throughput measurements for NetPIPE across different message sizes in the three systems studied. The inset zooms in on message sizes between 64 B to 8 KB.

In fig. 5.3, we show the mean throughput generated by the three systems along with its standard deviations for each data point plotted along its error-bars. Linux-static and EbbRT-static both used a fixed ITR value of 10 $\mu s$ in this scenario as an example of its impact on NetPIPE performance as we perform a sweep of *29* different message sizes from 64 Bytes up to 1,000,000 Bytes. At a

message size of 65536 Bytes, we find that Linux-static can improve its throughput over Linux by 1.74X and using an efficient OS such as EbbRT achieved up to 2.1X performance improvements.

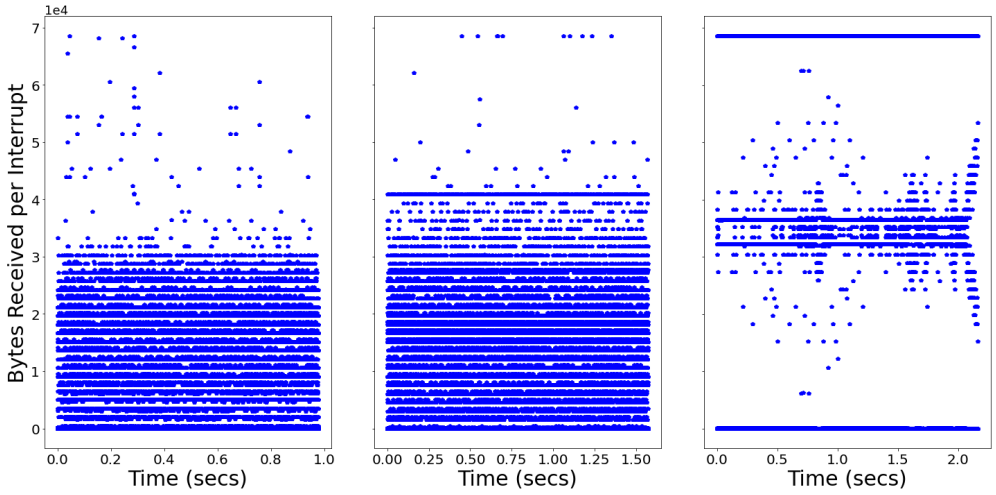


**Figure 5.4:** Three figures showing the bytes received per interrupt for **Linux** with dynamic ITR algorithm enabled when running NetPIPE @ 65536 Bytes. Each figure shows a distinct run of NetPIPE and demonstrates the dynamic behavior of Linux's ITR algorithm on the way packets are being processed (Y-axis) and overall time it takes to run a single experiment (X-axis). Out of the total of 10 different runs, we illustrate these three figures as after examining all 10 collected log datasets, we find performance of Linux with dynamic ITR mainly fluctuates between these three distinct behaviors.

Surprisingly, we find one reason for these performance differences is that Linux's dynamic ITR algorithm induces instability despite a very predictable and static workload where fixed sized message are transmitted between two nodes in a single connection. Examining the results for base Linux running at a message size of 65536 Bytes, we find standard deviations by up to 63% for its throughput measurements. While prior works (Belay et al., 2014; Schatzberg

et al., 2016) have illustrated this behavior in NetPIPE, the novel use of ITR mechanism in this work demonstrates how one can tame these dynamic behaviors in order to maximize performance.

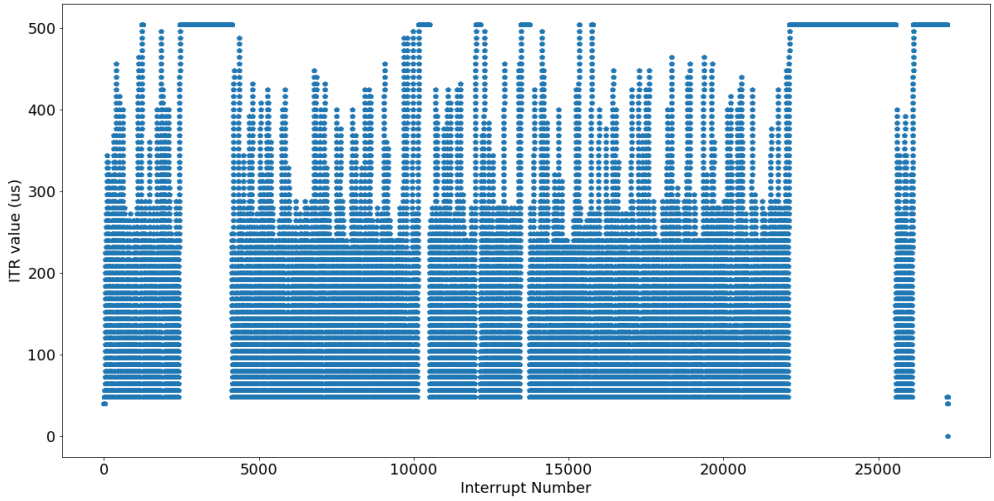


**Figure 5.5:** ITR values set by Linux's dynamic ITR algorithm for a single experimental run of NetPIPE at 64 KB message size.

As an example of the unstable nature induced by Linux's dynamic ITR algorithm, we examined three distinct runs of NetPIPE at 65536 Byte message sizes and plotted the different impacts of the algorithm on bytes processed per interrupt. The three figures shown above illustrate the inherent noise that exist within these dynamic algorithms that can result in dramatically different processing behaviors that and performance differences by 2X between runs of the same application. Furthermore, this can also be seen in fig. 5.5 that illustrates the "noise" created by Linux's dynamic algorithm to tune ITR. This figure demonstrates that this default algorithm is inherently working at the wrong timescale for NetPIPE and it may be better to operate with less dynamic changes.

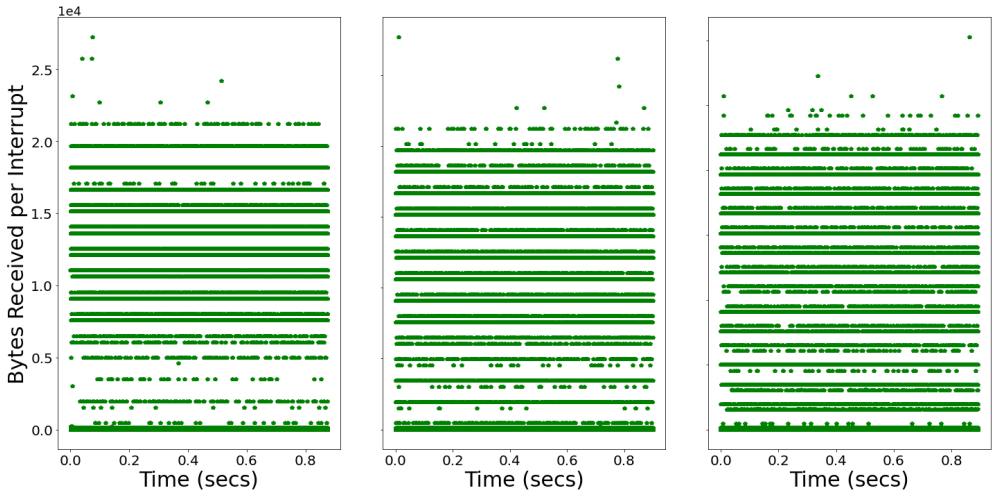


**Figure 5.6:** Three figures showing the bytes received per interrupt for **Linux-static** with ITR value at 10 $\mu s$ for a message size of 65536 Bytes in NetPIPE. The three figures each illustrate a distinct run of NetPIPE.

We illustrate a more stable environment by using a static ITR value and we find it can help to greatly stabilize and improve performance of NetPIPE. Fig. 5.6 above illustrates results from Linux-static and demonstrate how a static ITR mechanism can induce stable and lockstep behavior in packet processing such that closed-loop style workloads, that often have synchronous nature to their processing, can greatly take advantage of such predictability in incoming interrupts for packet processing. Furthermore, having this predictable behavior at a fast ITR value means that per interrupt, smaller portions of the payload are processed as can be seen in Linux-static where the max bytes received per interrupt is at around 2000 bytes while for Linux with dynamic ITR it can fluctuate up to 7000 bytes.

***Contribution:*** *While prior work have only used a static setting of a single ITR value for experimental stability (Yasukata et al., 2016; Peter et al., 2015; Ousterhout et al., 2019); our work demonstrates how this stability can be specialized and exploited for performance improvements.*

### 5.1.3 Offered Load, ITR, and DVFS Changes

However, we also find that as the message size increases in NetPIPE, the overlap between transmission and processing time presents surprising performance improvements and energy efficiency opportunities through the different uses of ITR and DVFS where table 5.1 shows that there exists unique values for NetPIPE across the message sizes. In §5.1.3.1 below, we provide concrete details of the nature of these opportunities to optimize for energy.

#### *5.1.3.1* ***Detailed Finding 2: Combining batching with DVFS to enable energy efficient pacing of packet processing.***

We find that as NetPIPE message sizes increased to 8KB, 64KB, and 512KB, the static ITR values that yielded best energy efficiency also began to increase (up to 28$\mu$s at 512 KB) as shown in table 5.1. A 10 GbE NIC, assuming no network jitter and switching cost, can transmit at an optimal rate of 1250 bytes/$\mu$s. Therefore, we find the ITR value is used to effectively modulate how much payload the OS should process in a fixed quantum. The ITR values which yield best energy efficiency profile is indicating a *sweet spot* in which the OS should pace

packet processing and save energy by sleeping during the its idle periods, and combine with a processor's DVFS setting to lower its overall energy use during these packet processing periods.

We illustrate this behavior in detail in fig. 5.7 below for each NetPIPE message size. We plot the energy consumption on the Y-axis against the static ITR values in our study. We also show three broad regimes that indicate how DVFS impacts the energy use as well. We use DVFS values 1.2 Ghz, 2.2 Ghz, and 2.9 Ghz to represent the slowest processor speeds up to max. In these figures, though one can see similar trends across the message sizes, distinct ITR and DVFS values yields distinct energy trade-offs for both OSes. As shown, the overall lowest energy use in NetPIPE are configurations that used the slowest processor speed setting with specific ITR values.

While one can see the configurations of lowest energy through the lowest points on the Y-axis, we also instrumented **X** markers to indicate the points that resulted in best performance to display the trade-offs between sacrificing performance for energy across the two OSes and as the message size changes which also impacts network processing.

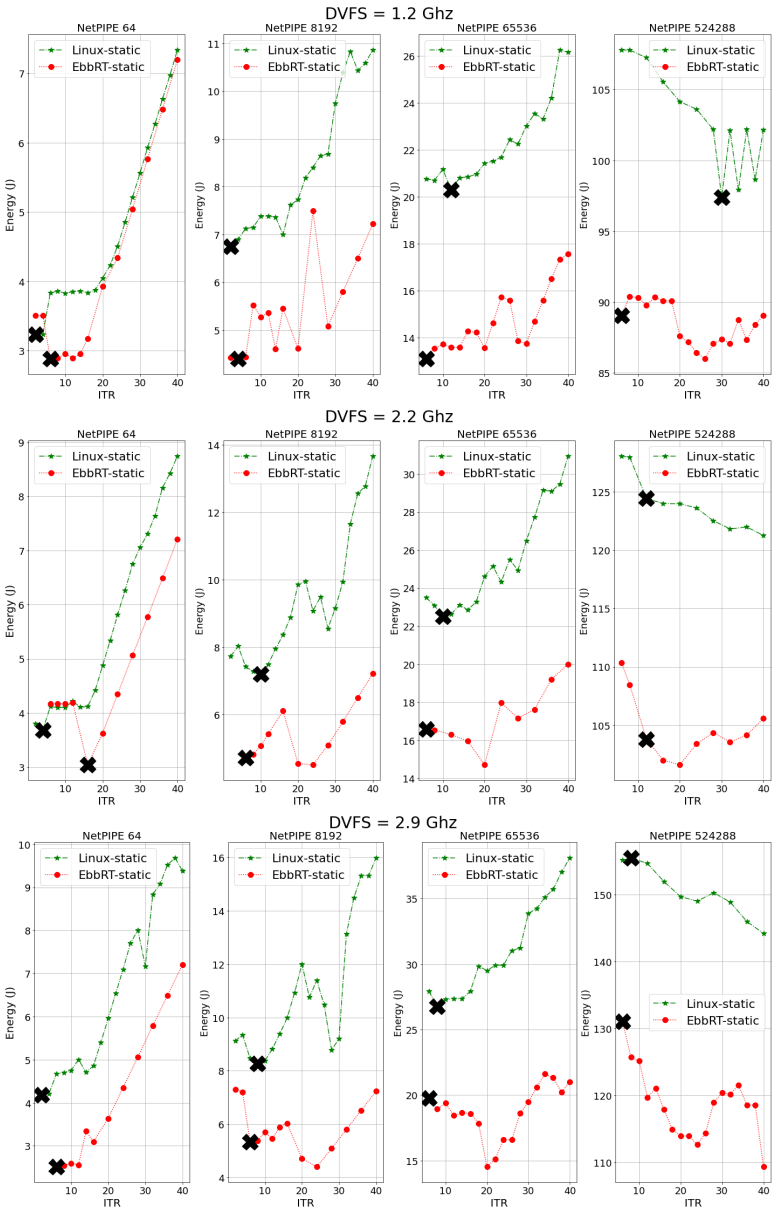


**Figure 5.7:** Three figures showing the changes for energy (J) with different static ITR values used in both EbbRT and Linux across three different DVFS values from slowest (1.2 Ghz) to fastest (2.9 Ghz). Furthermore, the **X** markers indicate configurations that yielded best performance in order to illustrate the performance-energy trade-offs that exist in this application. **Note:** Y-Axis is not scaled to show structure of the two OSes.

***Contribution:*** *For applications that have a predictable and constant data stream, our results suggest that ITR and DVFS can be used to both improve performance and energy for different payload sizes such as video streaming or middle tier services within a data center that exhibit these properties (Barroso & Hoelzle, 2009; Meisner et al., 2011; Lo et al., 2014; Fan et al., 2007; Barroso, Luiz André and Hölzle, Urs, 2007; Barroso et al., 2003).*

## 5.2 OPEN-LOOP

### 5.2.1 OS, Offered Load Changes

For open-loop applications, fig. 5.8 and fig. 5.12 show that: 1) the different OSes also respond uniquely and 2) there are clear performance and energy benefits with using different ITR and DVFS values. As the offered QPS load increases from 200K to 600K, these figures show that the overall energy required to service the load also increases as expected. These performance and energy profiles also behave in a consistent way for the different OS stacks across the offered loads, which suggest using ITR mechanism induces stability such that it can be combined with DVFS to explore new policies that can exploit these trade-offs.

As pointed out by previous works (Wu et al., 2016; Hsu et al., 2018; Lo et al., 2014; Hsu et al., 2015; Kasture et al., 2015; Leverich & Kozyrakis, 2014; Prekas et al., 2017; Asyabi et al., 2020; Zhan et al., 2017; Vamanan et al., 2015; Meisner & Wenisch, 2012; Chou et al., 2016, 2019), in open-loop applications the SLA budgets act as a threshold with which the OS can satisfy requests but still save overall energy use; for example, these works typically use DVFS to slow the processor in order to

slowly process incoming requests in an energy efficient way while still satisfying the SLA.

Instead of the characteristic "V" shape observed in close-loop applications, it has the "L" shape which is unique to open-loop applications. For example in Memcached @ 400K QPS, there are negligible increases in 99% tail latency while using 2X lower energy in EbbRT-static. However for Linux, we can see that its OS stacks does not result in the same level of improvements as a specialized OS and there is a clear horizontal nature to its trade-off space where performance is clearly sacrificed for energy savings.

### 5.2.2 ITR Changes

For these applications, we find controlled batching via ITR can stabilize the tail latency, which results in two effects: 1) enables the server to control per-request response latency at a granularity of $\mu s$ and has the added effect of maximizing energy savings from idle states, and 2) induced batched packet processing can be combined with energy saving benefits of processor energy settings. We elaborate more on this finding in §5.2.2.1. In the extreme of Memcached @ 400K QPS, Linux-static can sacrifice all of its 99% tail latency to be as close to the SLA budget of 500 $\mu s$ as possible in order to save up over 2X in energy over normal Linux. As table 5.1 shows, the optimal energy configurations often used ITR values in the range between 300 - 400 $\mu s$ in order to artificially batch as much as possible without violating the SLA. Furthermore table 5.1 illustrates that a specialized OS can also increase the degree of batching via ITR because its optimized code paths gives more leeway to take advantage of these SLA budgets.

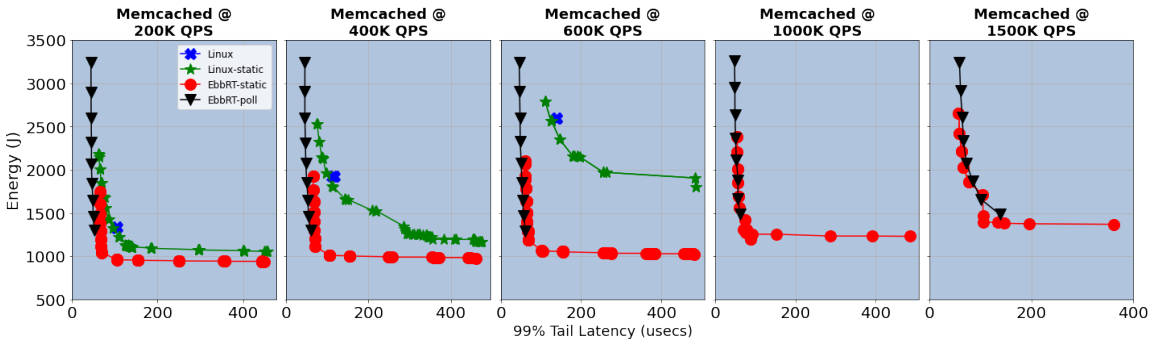


**Figure 5.8:** Pareto-optimal curves of Memcached application. We find that Linux cannot support QPS rates at 1000K and 1500K so therefore the data is not shown.

#### 5.2.2.1 *Detailed Finding 3: Using ITR to stabilize tail latency in open loop applications.*

Fig. 5.9 below shows the effect of batching on 99% latency by illustrating how different static ITR values (on X-Axis) affect the resulting measured latency value (Y-Axis). This figure illustrates that static ITR values stabilizes tail latency of requests in memcached by artificially placing it within certain regimes (i.e. an ITR value of 200 us results in measured 99% tail latency starting at the 200 us range) and the *headroom* between that stable tail latency and overall SLA objective can then be exploited for energy efficiency; this is also dependent on other factors such as an OS's packet processing efficiency, processor energy settings, and policies that govern the use of sleep states. This figure also illustrates when less stringent latency values are considered, i.e. 90% and 50%, there is even more headroom to trade off SLA objective for energy efficiency.

One can also see that having an efficient OS such as EbbRT also causes the latency distribution to be more stable in comparison to Linux in terms of the

vertical spread of measured values. At a light QPS load of 200K, one can see that the datapoints for both OSes are packed together such that ITR largely determines the overall tail latency. As QPS increases, the combined effect of DVFS and ITR on tail latency is magnified such that the vertical spread is more pronounced.

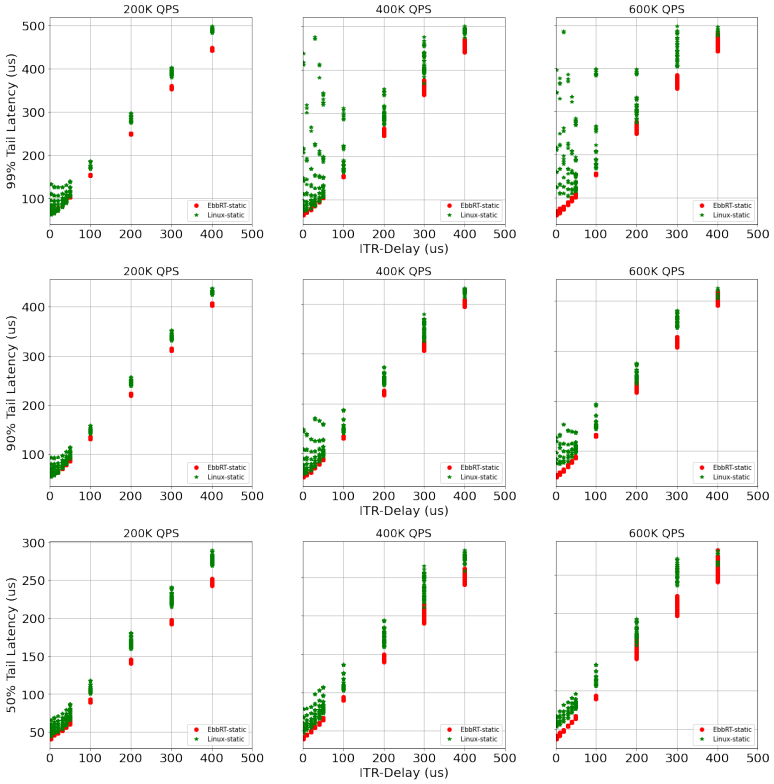


**Figure 5.9:** The X-axis plots the static ITR values explored in both OSes and the Y-axis shows the measured 99%, 90%, and 50% latency in Memcached for the different QPS rates. The range of points along the vertical is indicative of different DVFS explored for each static ITR. This figure illustrates how ITR can be used to induce stability in tail latency measurements even at 99% for a dynamic Memcached workload and the stability is more pronounced in a specialized OS such as EbbRT in comparison to Linux where at fast ITR values, the different DVFS values used causes a larger difference in tail latency measures.

As shown in the figures below, the benefit of slowing down ITR by increasing its value consists of 1) lowering the number of interrupts fired (fig. 5.11), which can help lower overall instruction use and promotes better coalescing of incoming requests, and 2) ensuring a guaranteed period of idle such that the processor can take advantage of with sleep states (fig. 5.10). We find that the total number of interrupts fired across different QPS loads can be lowered by over 90% in Memcached through the use of ITR mechanism.

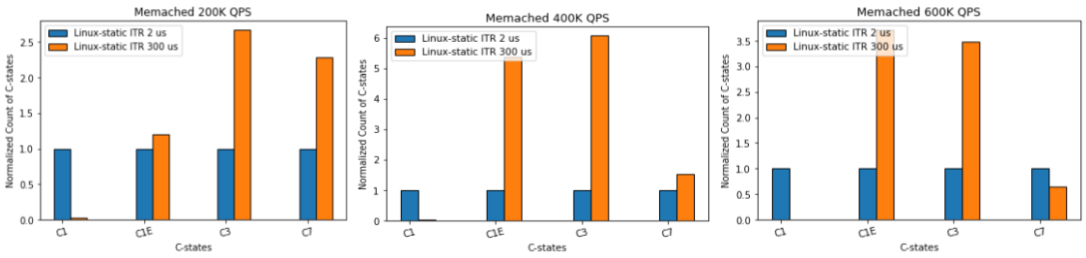


**Figure 5.10:** Using Linux-static as an example, the three figures show C-state counts in Memcached for different QPSes while running at a fixed DVFS of 2.5 Ghz. The X-axis shows for each C-state from C1 to C7 where C1 is the lightest and C7 is heaviest sleep state where architectural state such as caches are completely flushed. We show the count of how many times Linux's idle policy went into each sleep state given two different ITR values at a fast rate of 2 us and a slow rate of 300 us. The Y-axis are normalized against the counts of ITR at 2 us. These figures show that across the QPSes, a fast ITR rate of 2 us typically uses only C1 sleep state as it is the lightest and will be constantly woken up, as ITR increases to 300 us, the heavier sleep states begin to be used more to take advantage of the prolonged idle periods induced by the ITR mechanism.

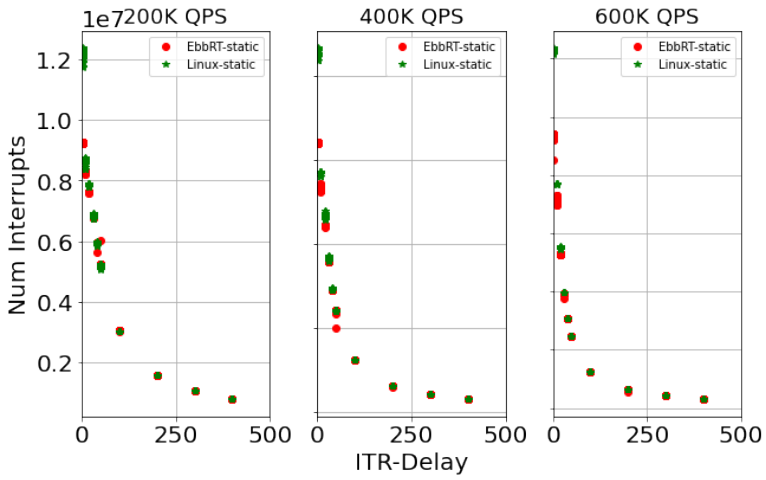


**Figure 5.11:** Static ITR setting impact on total number of interrupts in Memcached.

### 5.2.3 DVFS Changes

In open-loop applications, we find processor energy settings can be combined with batching to take advantage of the prolonged periods between interrupt processing to drastically lower energy use by processing applications slowly; this is in contrast to typical techniques such as run-to-halt where the goal is to finish the work quickly to maximize idle energy states. In table 5.1, one can see DVFS values at a minimum of 1.3 Ghz can be used with a large ITR value of 300-400 $\mu s$ to enable greatest energy savings while maintaining SLA budgets for both Memcached and Memcached-silo in both OSes. Details of this trade-off is discussed in §5.2.3.1 below.

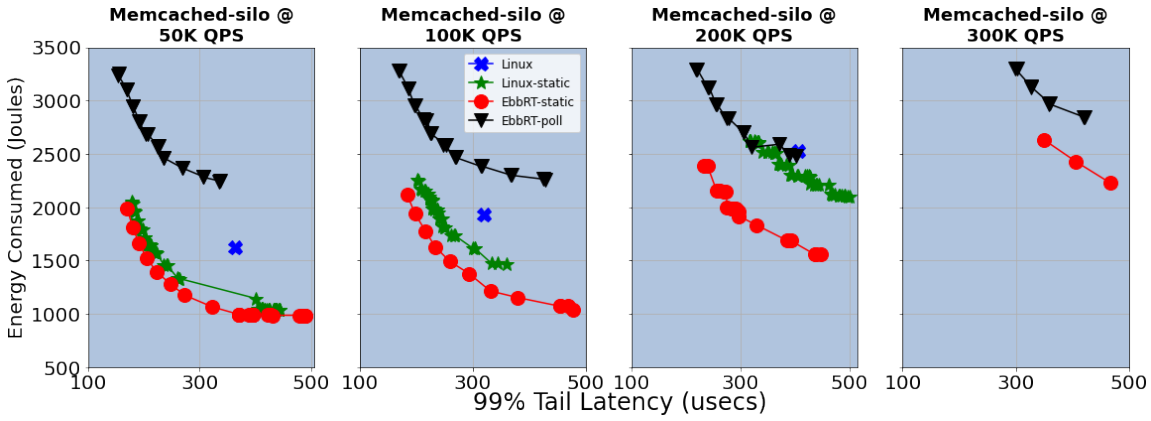


**Figure 5.12:** Pareto-optimal curves of Memcached-silo application.

#### 5.2.3.1 *Detailed Finding 4: Combining DVFS and ITR to lower total energy use.*

In fig. 5.13 below, the X-axis lists the DVFS values used and the Y-axis is the measured energy use of memcached with that DVFS and a range of ITR values. Moreover, to understand the impact of ITR on energy use, the bold lines indicate the mean energy use at fastest ITR value, while dotted lines indicate mean energy use at the slowest ITR value. We see that in both systems, as DVFS is set lower, the impact of ITR on saving energy diminishes (gap between bold and dotted lines shrink as DVFS slows). The effect of slowing down with DVFS results in the lengthening of the time to do application and OS work, therefore reducing opportunities to use ITR to save energy from idle sleep states. Surprisingly, we find this method of slowing application processing with DVFS yielded configurations with minimum energy consumption in contrast to run-to-halt configurations that typically use a fast DVFS to speed up processing in order to maximize idle states. Furthermore, such slow-down is further impacted by SLA requirements as it has stringent time budgets that almost all

requests must adhere to.

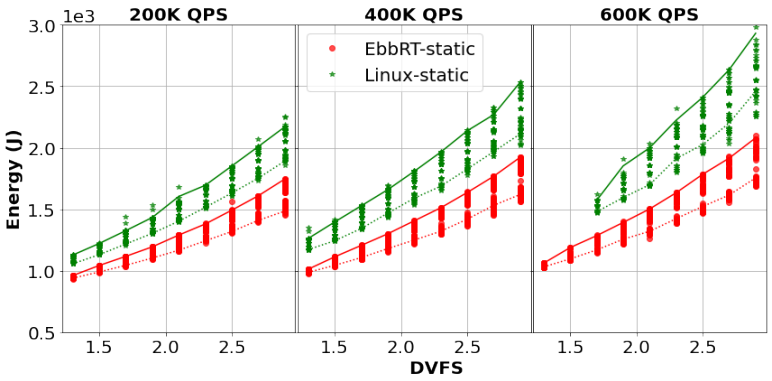


**Figure 5.13:** The X-axis shows for each static DVFS setting and the Y-axis shows the measured total energy use across the two OSes. The vertical span of each DVFS setting is indicative of how different static ITR values impact energy use. The bold lines show the fastest ITR explored and the dotted line show the slowest ITR explored. These lines indicate how within a DVFS value that changing ITR also impacts energy consumption.

However, figure 5.13 also shows that actually combining the effects of both statuc ITR and DVFS together results in overall lowest energy use across both Linux and EbbRT. By using a static ITR setting, this induces stability in application processing as incoming requests arrive in a predictable manner; therefore DVFS can be be finely tuned to find "sweet spots" such that energy is minimized.

***Contribution:*** *While previous work have focused on exploring energy benefits in SLA headroom with DVFS (Wu et al., 2016; Hsu et al., 2018; Lo et al., 2014; Hsu et al., 2015; Lo et al., 2015; Barroso, Luiz André and Hölzle, Urs, 2007; Fan et al., 2007) only, our findings demonstrate even further energy savings by up to 76% with the novel combined use of ITR and DVFS.*

## 5.3 OS-CENTRIC

### 5.3.1 OS, ITR, DVFS Changes

As fig. 5.2 and fig. 5.8 show, OS-centric applications such as NetPIPE and Memcached exhibit a unique performance and energy trade-off even though they display different ITR and DVFS efficiency responses in the shapes of "V" and "L". In particular, we find that in both applications, there is a optimal region on the leftmost side of these figures where performance is best such that it is possible to drastically reduce energy usage with a negligible performance penalty. This finding can be partly attributed to eq. (2.1) where applications that use more memory operations, such as NetPIPE and memcached, are less impacted by performance penalties of DVFS while still attaining its energy savings benefits.

For example in NetPIPE, slowing down DVFS for Linux-static @ 512 KB lowered performance by 16% but used 2X lower energy and EbbRT-static suffered no negligible performance degradation and still managed to lower its energy by up to 2X. Further, table 5.1 also illustrates that both OSes use different combinations ITR and DVFS values to achieve their respective performance and energy improvements; therefore demonstrating the importance that OS software plays in optimizing these goals with ITR and DVFS mechanisms.

For example in Memcached @ 200K for both OSes, the vertical dots on the left in fig. 5.8 indicate a single fast ITR value (around 2 $\mu s$) and the lowest point along that vertical line is caused by the use of DVFS to drastically lower energy by over 2X with negligible performance penalties. As discussed earlier, we find ITR places the

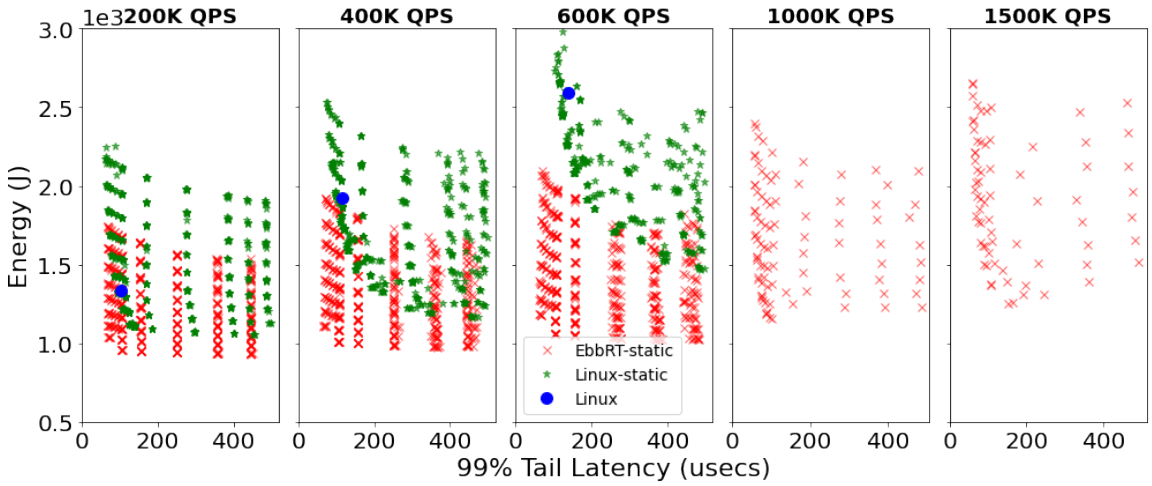


**Figure 5.14:** Each point represents a single experimental run of memcached. Note we don't have Linux results for 1000K and 1500K QPS as the Linux could not support that offered load without violating the SLA objective.

vertical bands by influencing its overall tail latency and DVFS affects the energy consumption within these bands. Moreover, we see this behavior replicated as the QPS loads increase for both OSes. We detail this observation further in §5.3.1.1 below.

#### 5.3.1.1 *Detailed Finding 5: A specialized system has more headroom with DVFS to further reduce energy without sacrificing performance.*

Fig. 5.14 expands on the distinct behaviors from Linux-static and EbbRT-static as ITR and DVFS are statically searched. This figure shows that as QPS increases, EbbRT maintains a relatively *vertical* structure regardless of ITR and DVFS, which implies that its optimized OS structure enables lowering energy use without compromising performance. In contrast, we observe that Linux begins to exhibit more trade-offs between performance and energy as it reached

higher QPSes at 400K and 600K (dots starts curving horizontally). We find that Linux can support a peak QPS load of 800K QPS while EbbRT's more specialized OS paths result in a peak QPS of 2000K; at 1500K QPS, which is around 75% of EbbRT's peak, we begin to see small horizontal curves in its data.

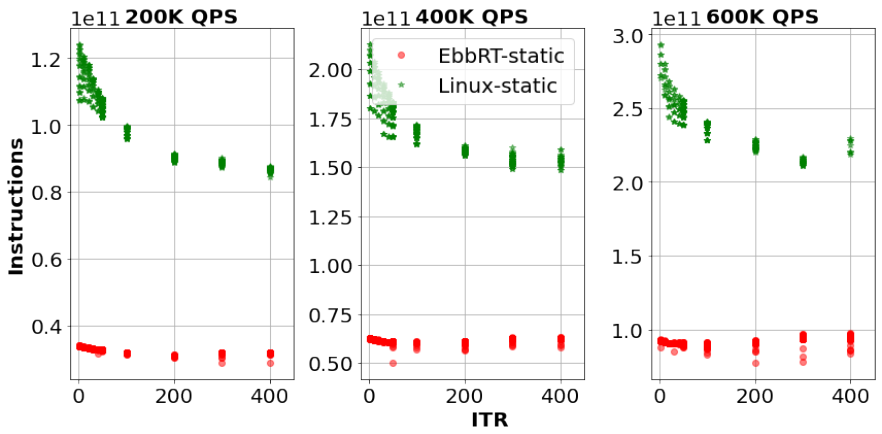


**Figure 5.15:** ITR impact on instruction count in memcached. Not drawn to scale in order to shown structure in data.

To delve more deeply into these observations, fig. 5.15 shows the impact of ITR on the total amount of instructions needed to run a single memcached experiment. First, this figure shows how a slow ITR value can reduce overall instruction usage from batched packet handling by up to 30% in Linux. Next, this figure shows the drastic differences in instruction count between the two OSes as EbbRT uses up to 2.5X fewer instructions to support the same load as Linux. As memcached is not compute heavy as most of its work consists of memory operations, this means that EbbRT with its optimized OS paths that uses less overall instructions can greatly take advantage of energy saving properties of DVFS without sacrificing as much performance.

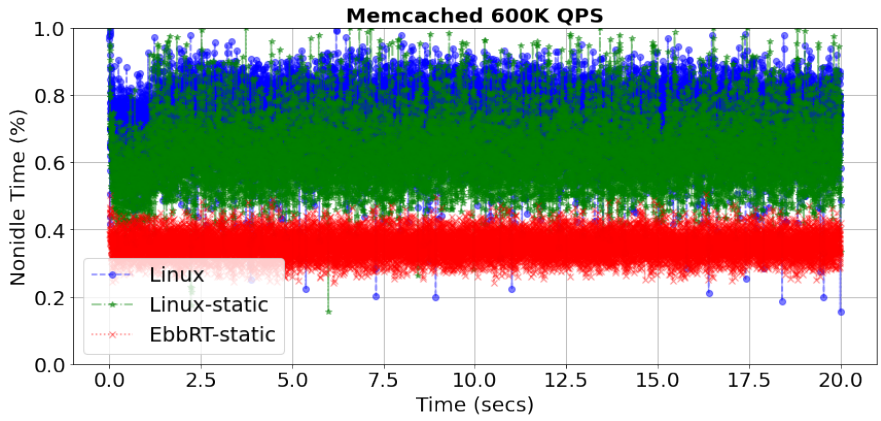


**Figure 5.16:** Timeline plot of non-idle ratio at per-interrupt basis for Linux-static and EbbRT-static that resulted in min energy for memcached @ 600K QPS.

Further, EbbRT's optimized memcached implementation also results in greater opportunities to use idle sleep states and save additional energy. This is shown in fig. 5.16 where each point is computed as the non-idling ratio at the per-epoch rate: we use the difference between the busy cycle count and `rdtsc` (which counts total cycles). This timeline figure demonstrates that a more specialized OS paths can be used to further maximize *run-to-halt* energy benefits (Meisner & Wenisch, 2012; Chou et al., 2016).

***Contribution:*** *These observations demonstrate that a more specialized system contains more headroom with which to exploit performance and energy more aggressively and suggest that different OS designs can result in drastically different displacements for their efficiency profiles.*

### 5.3.2 DVFS Changes

Further, we explore an alternate extension of the observation on DVFS performance and energy efficiency benefits in OS-centric applications by implementing a run-to-completion polling loop on each core of EbbRT. Due to this tight-loop, this alternate configuration of EbbRT will never halt and use idle states. We elaborate on this implementation in §5.3.2.1 below.

#### *5.3.2.1* ***Detailed Finding 6: Polling can be energy efficient.***

As indicated earlier, the use of a single fast ITR value resulted in both best performance and subsequent combining with DVFS also resulted in lowest energy use as well. This prompted us to explore the effects of eliminating interrupts altogether and use a dedicated poll loop. In fig. 5.2 and fig. 5.8, we illustrate that EbbRT-poll was able to achieve both best case latency and competitive lowest energy. We found that by modulating DVFS, EbbRT-poll can be made energy efficient for small payloads under its specialized OS paths - this is in contrast to the normative assumptions of OS poll (whe, 2012; Golestani et al., 2019) where it often trades performance for higher energy use. For example, EbbRT was able to improve tail latency by 27% while using 35% less energy in Memcached. In NetPIPE (Snell et al., 1996), polling can achieve up to 3X better performance while using 4X lower energy as compared to baseline Linux. However, polling with DVFS must be used judiciously as in other application-centric workloads such as Memcached-silo often results in both worst performance and energy use as compared to their interrupt-driven counterparts.

***Contribution:*** *This finding suggests the importance for energy aware OS-level optimizations that can switch between poll and interrupt-driven IO in response to changes in demand. Therefore, OS path specialization techniques can explore the use of polling to achieve both low-latency and energy efficiency with careful use of DVFS in new hybrid policies.*

## 5.4 APPLICATION-CENTRIC

### 5.4.1 OS Changes

Fig. 5.2 and fig. 5.12 demonstrate that in both OSes that as the application work is more processing heavy compared to the OS portion, there is a more horizontal nature to the trend of these efficiency profiles where performance is clearly sacrificed for energy and vice versa. To our surprise, we find that even in application-centric workloads, the specialized OS paths of EbbRT can still induce new performance and energy benefits via new optimizations strategies. We discuss two of these detailed findings below and also discuss their implications on both system design and applicability towards other types of applications. To begin, §5.4.1.1 discusses a new slow-to-stay-busy strategy that we've uncovered in EbbRT that can be applied to fine-tune when interrupts are handled.

#### 5.4.1.1 *Detailed Finding 7: Energy-aware-slow-poll strategy in a run-to-completion OS.*

Previous works have coined terms such as race-to-idle/pace-to-idle Hoffmann (2013) for alternative energy saving strategies. In both NodeJS and NetPIPE with only a small message of 64 B, we find a new strategy which we call the *energy-aware-slow-poll* effect; whose behavior is different from previous works as the system is always kept busy by processing in a slow manner such that new requests are always in the pipeline after the current request is finished.

Fig. 5.17 illustrates that for EbbRT, a slow DVFS can lower the total number of interrupts by up to 90%. The reason for this is that the physical transmission of the OS reply packets by the device driver can occur asynchronously with the unwinding of the stack back to the application and then back down to the network receive function to check for new packets. A slow DVFS 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 many hardware interrupts (fired on packet receive) in order to effectively *energy-aware-slow-poll* and process this new reply packet that has already arrived.

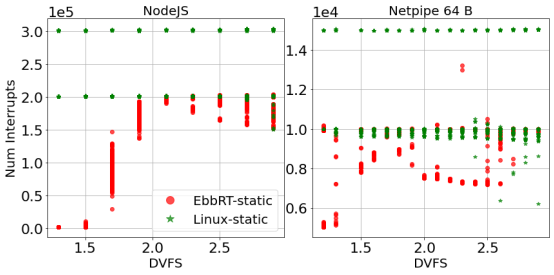


**Figure 5.17:** DVFS impact on number of interrupts in NodeJS and Netpipe 64B.

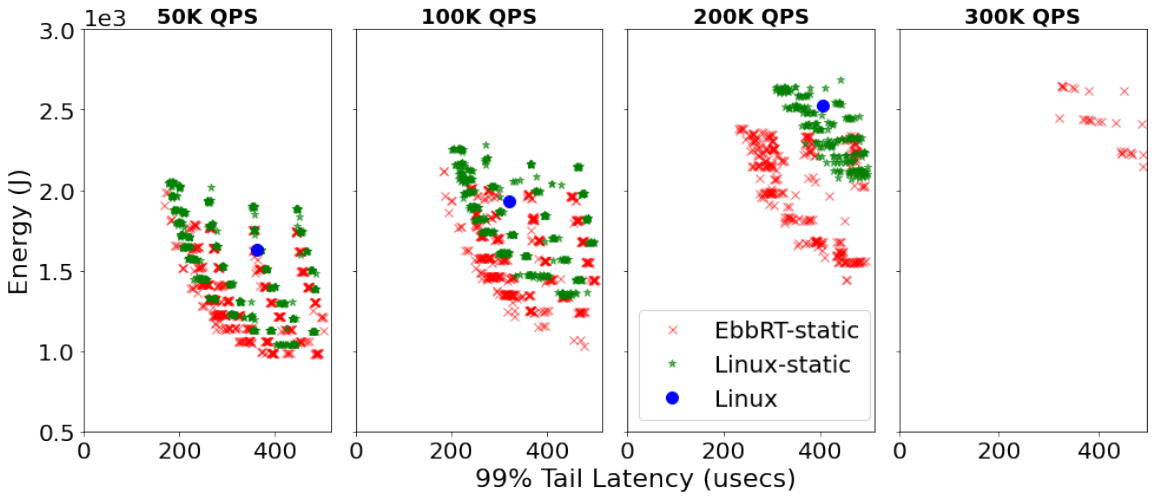


**Figure 5.18:** All memcached-silo results. Note we don't have Linux results for 300K as the Linux could not support that offered load without violating the SLA objective.

***Contribution:*** *We find energy-aware-slow-poll only occurs in EbbRT due to its run-to-completion nature and suggests that for a structurally different OS, new energy saving strategies can be uncovered.*

### 5.4.2 Offered load, ITR, DVFS Changes

In contrast to memcached results, fig. 5.18 shows that as the application work gets larger for each request, the trade-offs of latency and energy become more discernible in both Linux and EbbRT as indicated by the curvature and horizontal pattern of markers. Surprisingly even in a computationally heavy workload under a stringent SLA, it is still possible to use DVFS and ITR to further save energy; in fig. 5.18 at 200K QPS, Linux-tuned improved its tail latency by 21% and energy by 20% over Linux-default, further EbbRT-tuned improved its tail latency by 34%

and energy by 44% over Linux-tuned. In §5.4.2.1 below, we detail how EbbRT's OS path specializations can enable these efficiency gains.

#### 5.4.2.1 *Detailed Finding 8: IPC Benefits Even in Computationally Heavy Applications.*

In fig. 5.19 below, we validate that in a computationally heavy applications such as Memcached-silo, where most of the work is in the application, both Linux and EbbRT used roughly the same number of instructions across all the ITR, DVFS settings. Surprisingly, fig. 5.19 shows that even in a computationally heavy workload, a more specialized system such as EbbRT was able to execute those instructions more efficiently by having higher instructions-per-cycle (IPC) than Linux. We believe this can be attributed to its optimized OS paths and as fig. 5.19 (a) shows that it suffered fewer cache misses than Linux. This IPC benefit further corroborates §5.2.3.1 that a more efficient system can more aggressively slow a processor to save energy; note that for 100K and 200K QPS, EbbRT can use lower DVFS values while not violating SLA compared to Linux whose DVFS value range is more limited.

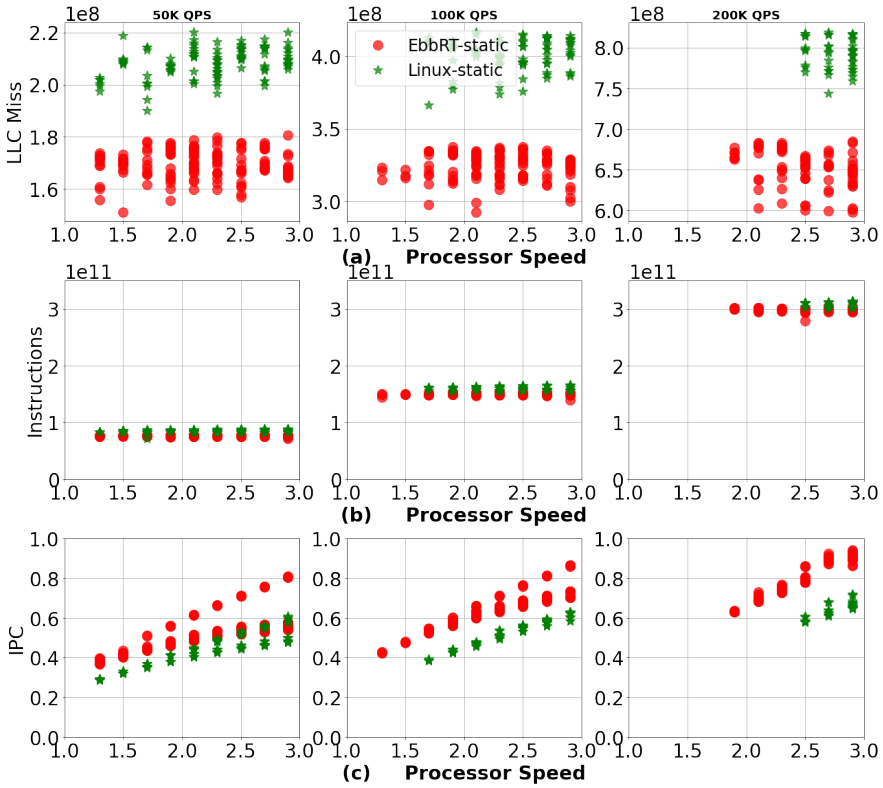


**Figure 5.19:** The figures show collected hardware statistics for Memcached-silo across three QPS values. For consistency, we plot the data from the perspective of different fixed DVFS values. In (a), the Y-axis shows the count of total number of last-level cache misses between the two OSes. In (b), we illustrate that the total number of instructions executed is roughly the same even though the application runs on two different OSes. In (c), our results show that the EbbRT is executing instructions more efficiently than Linux even in a computationally heavy application.

***Contribution:*** *OS path specialization can enable dramatic performance and energy benefits even in application-centric workloads and suggest these adaptions can exhibit similar performance-energy trade-offs even when implemented in general purpose systems.*

## 5.5 SUMMARY OF EXPERIMENTAL FINDINGS TOWARDS BUILDING A MODEL

This chapter began with the the hypothesis that a network service software stack has a characteristic performance and energy efficiency profile that is a function of packet batching (ITR) and processor energy settings (DVFS). This profile identifies optimal operating points that can be exploited to minimize energy consumption for a given performance target. Using *itrLog*, this chapter presented a detailed experimental study of how ITR, DVFS, and OS logic and paths interact together to impact performance and energy trade-offs given different application types and offered loads.

First, the detailed findings described can help pave the way towards new policies that address the importance of energy efficient system designs. For example, the slow-to-stay-busy effect and EbbRT-poll findings reveal how decisions in EbbRT's OS path specialization, such as run-to-completion, can enable novel forms of packet processing that can help achieve even further performance and energy benefits. Moreover, our novel findings on the use of ITR combined with DVFS reveal unique opportunities in performance and energy trade-offs that are dependent on the type of application and particular offered loads that it services.

Second, figures 5.2, 5.8, and 5.12 illustrate that for each of the OS stacks that use a static configuration of ITR and DVFS, there exists unique OS efficiency profiles. While there is a clearly separation between overall performance and energy between the two OSes, it also reveals characteristic shapes that are independent of the OS, and are functions of batching and processor energy settings given a particular application and offered load; further, they share a common and stable structure.

Lastly, this suggests one can treat performance-energy behavior in a general way independent of the choice of OS. This can be done by considering the performance-energy behavior as a parametric family of curves with one parameter set choice denoting EbbRT and another Linux. As the figures show, when OS code paths and other OS characteristics are changed, the underlying parameters also smoothly change. The implication is that the underlying OS response is stable and structured such that one can capture OS agnostic performance and energy profiles using an analytical model in a formal way, which further suggests that controlled learning of policies for OSes can be made feasible.

In chapter 6 below, this thesis details how simplified request processing timeline (fig. 4.1) can be expressed mathematically in order to construct a set of analytical models that can accurately capture the efficiency profiles we measured. These types of models let us evaluate how batching and processor energy settings interact with OS and application request processing, and how this evaluation changes when the OS or offered load changes. Specifically, our models use ITR and and DVFS values as inputs in order to accurately fit the experimental data across the OSes, applications, and offered loads, despite dramatic differences in system structure.

# CHAPTER 6
# Modeling the Experimental Data

The structure revealed by our exhaustive experimental search illustrates the fidelity of our data. This, in turn, suggests that our data can be exploited to create a formal model for analyzing and exploring energy and performance impacts as a function of OS behavior.

We use the timeline in fig. 4.1 as a guide for constructing our model. To summarize, a *quiescent period* in which no requests are present at the server precedes activity. In response to *OS request detection*, the OS detects and schedules processing typically with a combination of interrupt and/or poll-based mechanisms. With new data to process, several components of OS functionality must be run in accordance with the execution model of the OS during *request servicing*. This execution stage can be interleaved between synchronous and asynchronous stages, depending on OS logic (i.e. event-driven or context switching) until a *reply* message is sent. If all processing is complete, the OS can then use an *idle policy* that selects a hardware sleep state to halt the core and save energy.

Guided by this timeline, we then adapt and modify the model's applicability to accurately fit the experimental data. Below, we detail how the applications, the OSes, the different offered loads and the insights gained in chapter 5 all helped to inform the construction of our models that can accurately predict both performance and energy.

## 6.1 OPEN LOOP MODEL

In the open loop applications, we assume a simple model where the offered load is light enough that requests don't batch up in the receive queue and can be treated

independently. We use the SLA objective as 99% of all requests to have a tail latency under 500 us, therefore, we filter out ITR and DVFS settings which violate the SLA in our data set. Below, we model performance as the per-request 99% tail latency value and the energy as the per-request energy consumption. While not detailed in this thesis, we have also validated other tail latency values (i.e. 50%, 75%, 90%, and 95%) and find our models can accurately fit them as well.

**Performance:** The per-request 99% tail latency value can be decomposed into two constituent parts: 1) $t_{interrupt}$, which is the time spent waiting for the NIC interrupt to fire and cost to wake up the processor, and then 2) $t_{work}$, which is the time spent actually processing the request and eventually sending a response. Therefore, we define $\triangle t$ as the time it takes to handle a single request as the following:

$$\triangle t = t_{work} + t_{interrupt}$$

We parameterize $t_{work}$ as a function of DVFS values used in our experiments:

$$t_{\text{work}} = \frac{Z}{DVFS^{1+\alpha}} \tag{6.1}$$

$Z$ and $\alpha$ are parameters that change with respect to both the OS and application request load. In this model, $Z$ acts as a maximum time limit that each request can take (under some SLA target). Influenced by detailed findings in §5.1.2.1 and §5.2.3.1, $\alpha$ represents a system's dependence on DVFS to trade-off performance for energy and how its impact is different dependent on which OS, application, and offered load is used. For example, if $\alpha = -1.0$, then that particular system has no dependence on DVFS and can largely use DVFS to lower energy use without sacrificing performance.

We parameterize $t_{interrupt}$ as a function of ITR values:

$$t_{\text{interrupt}} = \phi * ITR \tag{6.2}$$

Influenced by detailed finding in §5.1.3.1, where ITR is a mechanism that directly influences and stabilizes the measured tail latency value, $\phi$ represents the location in the receive queue where a packet is placed before being processed. For example, if an unlucky packet arrives just as the NIC's $ITR$ value starts counting down, then it will have to artificially wait a full $ITR$ before being processed, thereby delaying overall request processing time.

The predicted performance per request is thus defined as follows:

$$\triangle t = \frac{Z}{DVFS^{1+\alpha}} + (\phi * ITR) \tag{6.3}$$

**Energy:** The total energy consumed is affected by the various power values which in turn depends on the way DVFS is set, and the degree with which ITR is used to induce both prolonged idle periods and the resulting benefits of batched packet handling. Further, we posit that this energy use has a power dependence on DVFS as motivated by eq. (2.1) where $DVFS$ values impacts both a processor's voltage and frequency, therefore, we define $\triangle J$ as the amount of energy it takes to process a single request as:

$$\triangle J = \gamma * (\phi * ITR) * DVFS^{\beta} \tag{6.4}$$

Note that $\phi$ used here is the same variable from eq. (6.2). $\gamma$ (units of watts) acts to convert the interactions of $ITR$ and $DVFS$ and into energy used. The variable $\beta$ acts as a dependence factor on DVFS in a simlar way to $\alpha$ in eq. (6.1). Note, in

our figures we show results after doing a log transformation to eq. (6.4) on both sides to enable linear regressions analysis where $\beta$ and $\gamma$ are free parameters that change dependent on the OS and workload

## 6.2 CLOSED LOOP MODEL

In the closed loop applications, we develop a model that captures its inter-dependency behavior, where the arrival of the next request depends on how fast it takes to service the current request. From a server's perspective, the idle 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.

In this model, we define the predictions on a per-request basis to capture both energy used and total time spent. Broadly, we view a single request in closed loop applications as composing of both a client and server node, which begins from 1) the client sending a request packet to the server, 2) to the time spent to transmit packet to server, 3) and then the server waking the processor up to process the request and sends a response back to the client, and finally 4) the response being transmitted over the wire to the client. We construct two models below, one for predicting performance and another for energy use.

**Performance:** We break the time to process each request, $\triangle t$, into three parts: 1) $t_{work}$ – time spent processing the request and eventually sending a response, 2) $t_{transmit}$ – the transmission time given the request message size and the network link speed, and 3) $t_{interrupt}$ – the effect that ITR and DVFS interactions have on

overall processing time. The value 2 is used to encode the client-server tight-loop nature of closed-loop style applications.

$$\triangle t = 2 * (t_{interrupt} + t_{transmit} + t_{work}) \tag{6.5}$$

We use the same $t_{work}$ variable as defined in eq. (6.1). We compute $t_{transmit}$ using speed of the network (i.e. 10 GbE) and the size of the message sent in the request. Next, we parameterize $t_{interrupt}$ as a function that captures the complex interactions between ITR and DVFS, as informed by §5.1.2.1 where ITR can be used to batch different sized payloads of NetPIPE and for each batching, there is a unique DVFS value that minimizes overall energy use:

$$t_{\text{interrupt}} = ITR^{\gamma * DVFS + \delta}$$

**Energy:** We model per request energy as a simple expansion upon the derivation in eq. (6.5) by adding a set of conversion factors ($CF_a$, $CF_b$, $CF_c$) for each of the three components above in order to convert a measure of time to a measure of energy use. All the conversion factors are in units of Watts (joules/secs). Therefore, we define $\triangle J$ as the amount of energy it takes to process a single request as:

$$\triangle J = 2 * (CF_a * t_{interrupt} + CF_b * t_{transmit} + CF_c * t_{work}) \tag{6.6}$$

## 6.3 MODEL FITTING RESULTS

We inferred the parameters above by minimizing the mean squared error (MSE) loss between the model's calculations and the collected data. Since we wanted to try complex ITR and DVFS dependencies in our equations, we used PyTorch's

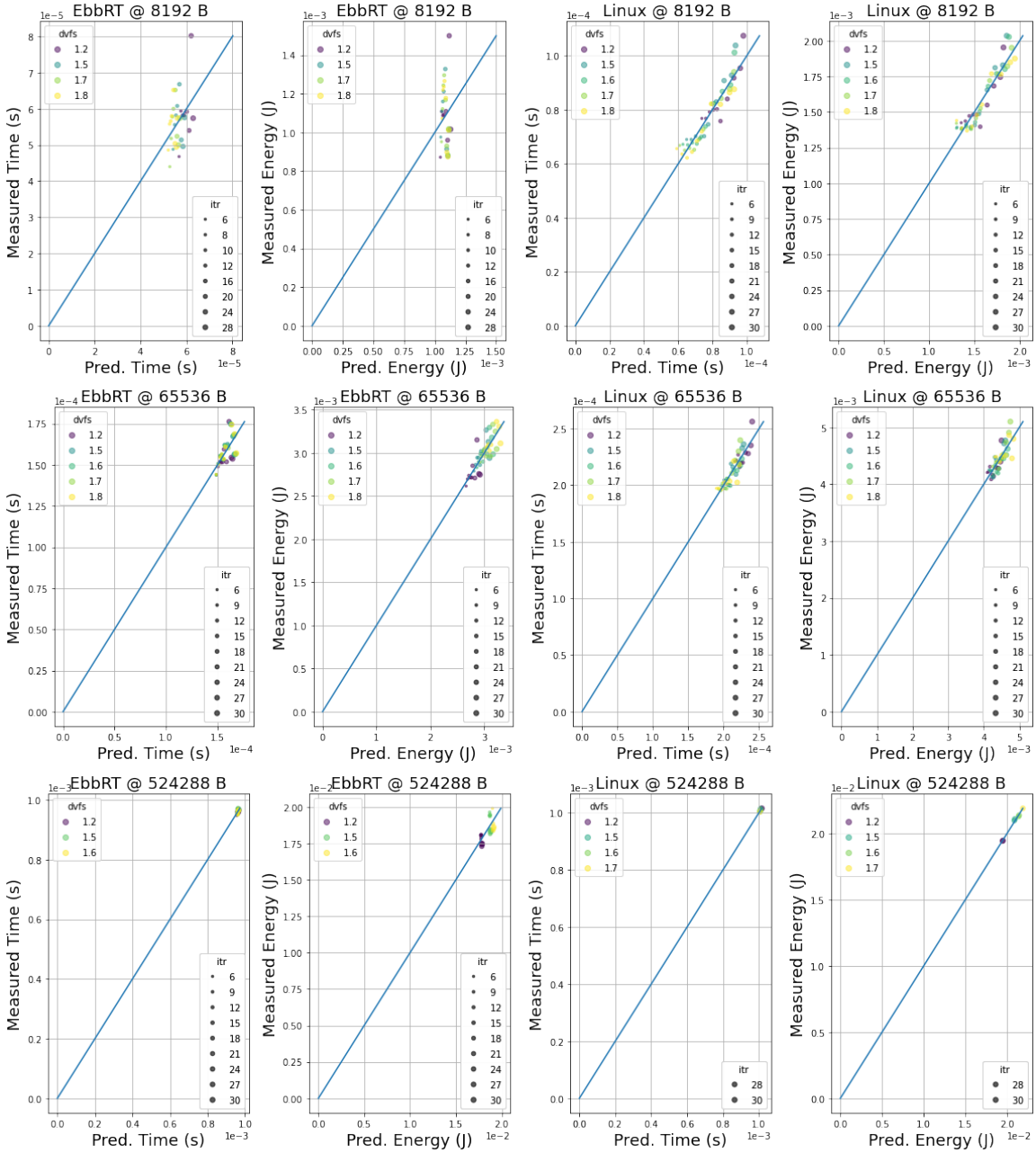


**Figure 6.1:** Prediction of energy and performance using model for Netpipe at different message sizes. The Y-Axis consist of measured values (either performance or energy) and the X-AXis consists of predicted values using the constructed models. We draw diagonal lines and show if the dots (which are measured values) lie on the diagonal line, then it is an accurate fit of the model onto the data.

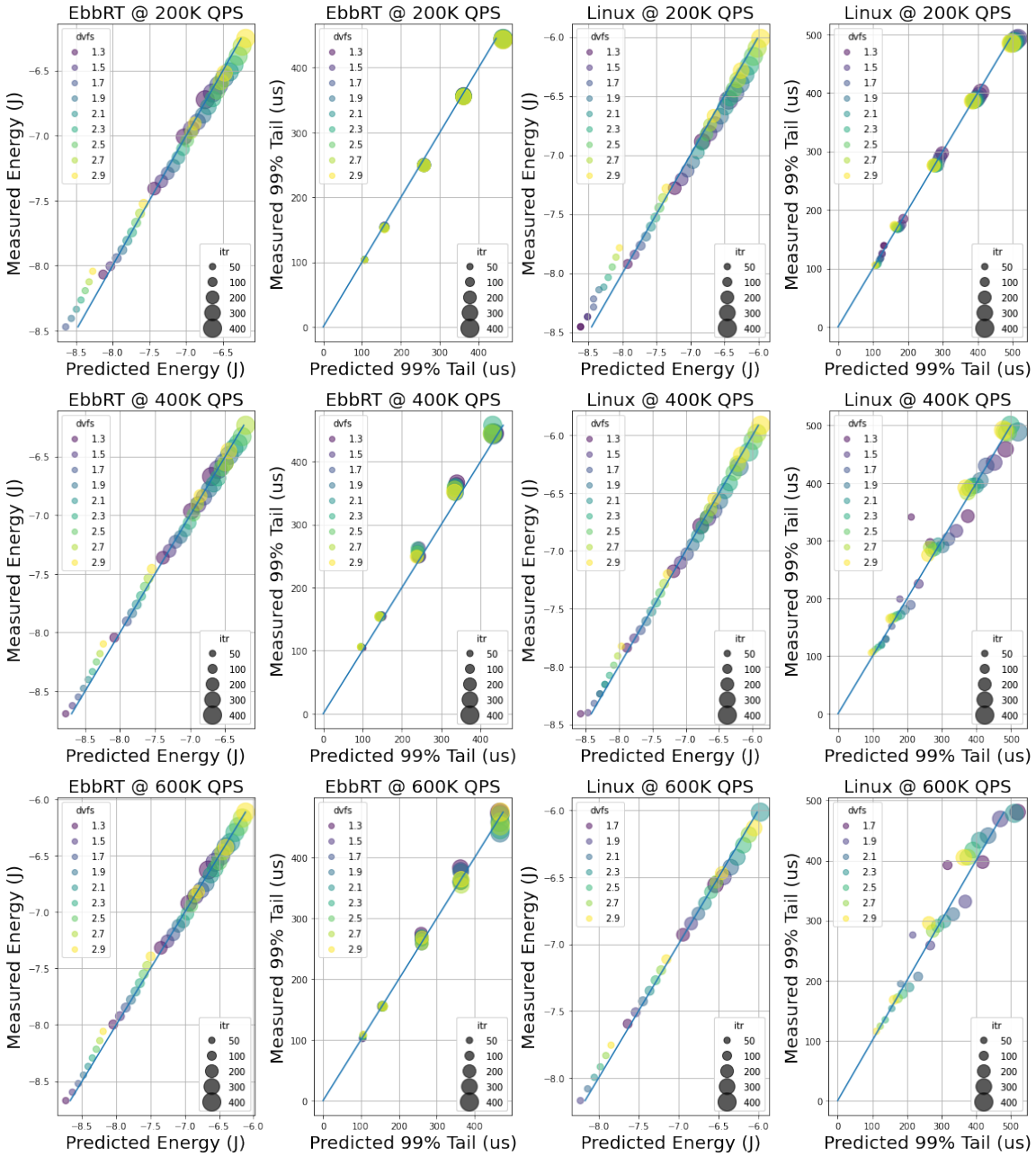


**Figure 6.2:** Prediction of energy and performance using model for Memcached.

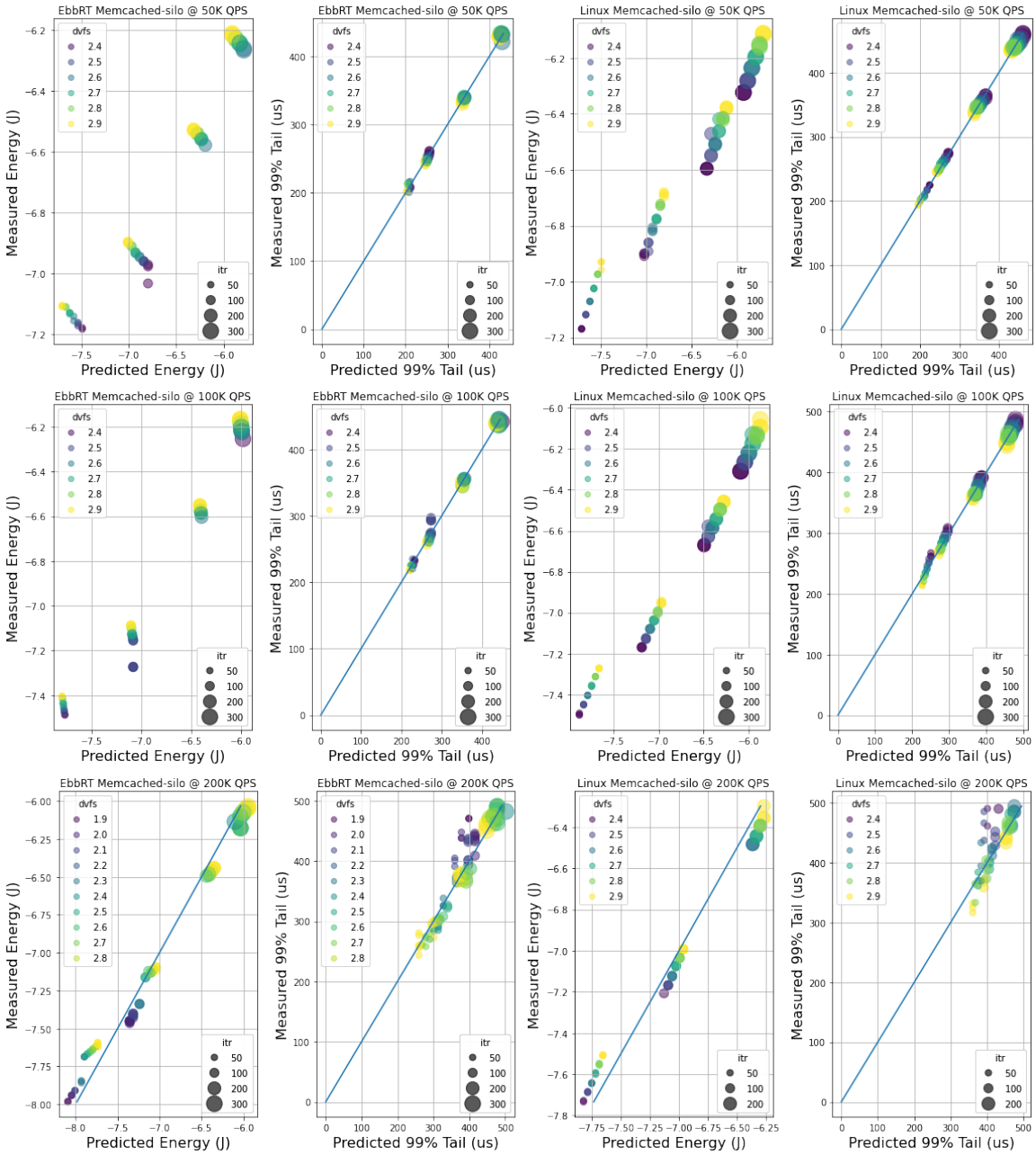


**Figure 6.3:** Prediction of energy and performance using model for Memcached-silo.

| | | Memcached | | | | | |
|---|---|---|---|---|---|---|---|
| | | 200K | | 400K | | 600K | |
| | | Linux | EbbRT | Linux | EbbRT | Linux | EbbRT |
| Tail Latency | $Z$ | 86.3 | 55.88 | 251.67 | 55.63 | 687.97 | 51.72 |
| | $\alpha$ | -0.48 | -0.96 | 0.71 | -1.00 | 1.34 | -1.04 |
| | $\phi$ | 1.09 | 0.98 | 1.08 | 0.98 | 1.02 | 1.03 |
| Energy | $log(\gamma)$ | -12.82 | -12.89 | -12.79 | -12.85 | -12.75 | -12.85 |
| | $\beta$ | 0.71 | 0.68 | 0.76 | 0.66 | 0.89 | 0.68 |

**Table 6.1:** Values for free parameters in memcached at different QPSes from doing fit with Adam optimizer.

auto-differentiation engine to perform gradient descent on the non-convex loss. In particular, the Adam optimizer (pytorch, 2022) was used. Each fitting process was run several times to check the stability of the inferred parameters to not get stuck in local minima. Overall, we find that our models are expressive enough to accurately fit both performance and energy data for both OSes at different offered loads across the set of applications studied.

In fig. 6.1, fig. 6.2, and fig. 6.3 shown above, we demonstrate results of our fitting by plotting the set of energy and performance predictions (X-axis) against their measured values (Y-axis). We plot diagonal lines in the figures below to show where ideal points would lie if our model's calculations are exact. We use two additional visual cues to identify the effects of ITR and DVFS on performance and energy: 1) *slowing down* DVFS is reflected in the color of each marker as going from lighter to darker, and 2) *slowing down* ITR is reflected in size of each marker as going from small to large. Note that the range of ITR and DVFS values are not consistent across applications as they each have particular static settings that produce the best results. Moreover, table 6.3, table 6.1, and table 6.2 detail the values of the free parameters that our model fitting approach eventually settled on, we discuss some of the meanings behind their respective values as well in the sections below.

| | | Memcached-silo | | | | | |
|---|---|---|---|---|---|---|---|
| | | 50K | | 100K | | 200K | |
| | | Linux | EbbRT | Linux | EbbRT | Linux | EbbRT |
| Latency Predict | $Z$ | 403.60 | 225.39 | 324.52 | 245.85 | 429.30 | 364.65 |
| | $\alpha$ | -0.06 | -0.66 | -0.45 | -0.70 | -0.77 | -0.62 |
| | $\phi$ | 0.94 | 0.89 | 0.91 | 0.86 | 0.61 | 0.74 |
| Energy Predict | $log(\gamma)$ | -12.58 | -10.31 | -12.71 | -11.38 | -11.52 | -12.23 |
| | $\beta$ | 1.15 | -1.11 | 1.15 | -0.16 | 0.38 | 0.83 |

**Table 6.2:** Values for free parameters in memcached-silo at different QPSes from doing fit with Adam optimizer.

### 6.3.1 Open Loop Discussion

Fig. 6.2 illustrates one of our open loop model fittings with memcached at 600K QPS. As an example of the accuracy of our fits, the measured mean latency for 600K QPS is 265.5 $\mu$s with standard deviation of 125.2 $\mu$s, and our $\sqrt{MSE}$ on latency is 0.009 $\mu$s. We also do a log transformation for our per-request energy prediction in eq. (6.4) to enable linear regression analysis where $\beta$ and $\gamma$ are free parameters that change dependent on the OS and workload.

#### *6.3.1.1 Memcached*

In our parameter fitting approach, the conversion factor parameter $\gamma$ is consistent in both OSes, indicative of some base cost of running the hardware, and translates to roughly 2.7 Watts from back of the envelope calculations. Surprisingly, we find that the value of $\phi$ is consistently $\sim$1.0 across both OSes, which implies our model is correctly adapting to the 99% tail. We examined other tail latencies such as 50% and found $\phi$ also consistently adapted to a value of $\sim$0.5. Moreover, fig. 6.2 shows the greater effect that DVFS has on energy while ITR has a greater effect on tail latency. Below, we provide more details of the values in the fitted parameters in the context of an OS and its application.

- $Z$: We see that EbbRT's value stays roughly the same while Linux's increases as QPS increases. This value represents some mean time that the system needs to process a request for memcached under the SLA. EbbRT's stability is largely a result of being a more efficient system and the importance of this observation will be tied to the $\alpha$ parameter.

- $\alpha$: This value stays roughly the same at -1.0 for EbbRT while increasing for Linux as QPS increases. The -1 value means that DVFS settings in eq. (6.1) will largely not increase the time to process the request as QPS increases, which demonstrates the ability of a more efficient system to exploit DVFS to save energy without sacrificing performance. However, we see in Linux that as QPS increases, there is a greater dependence on DVFS and its impacts on reducing energy use while maintaining SLA.

- $\phi$: We find that this value is closely correlated with the SLA, which is 99% tail in this case; we explored other tail latencies and saw similar behavior (i.e. 0.5 for 50% tail latency). This implies that given a $t_{work}$ that is stable, then most of the variation in latency is driven by the position of a request in the NIC's receive queue.

- $log(\gamma)$: Our fitting shows that this parameter is a conversation factor that is consistent in both OSes, indicative of some base cost of running the hardware and translates to roughly 2.7 Watts from back of the envelope calculations.

- $\beta$: Scales similarly as $\alpha$ where in Linux as QPS increases, so does $\beta$, while in EbbRT it stays roughly stable. Its used in a similar manner to scale an OS sensitivity to DVFS but with respect to energy instead of time, the results in this table show that at the lowest offered load, a more efficient system will

use less overall energy and will also scale better as load increases.

#### 6.3.1.2 *Memcached-silo*

Fig. 6.3 illustrates an example result of our fitting with Memcached-silo at 200K QPS and table 6.2 lists the values of the free parameters. Given the compute heavier nature of this workload, one can see that the interactions of ITR and DVFS is complex in both Linux and EbbRT. Below, we discuss the values of the free parameters listed in table 6.2.

- $Z$: In contrast to memcached, we see this parameter increasing in both OSes as the QPS increases due to the extra set of application work per request. However, a more efficient system can still maintain a lower $Z$ value as it is faster at processing each request.

- $\alpha$: As mentioned, both OSes will be dependent on DVFS to finish each request while maintaining the SLA objective. Recall that the dependence is defined by $DVFS^{1+\alpha}$, where if $\alpha = -1$ then there no DVFS dependence to affect latency. The table shows that EbbRT has a consistent DVFS dependence regardless of QPS while Linux's fluctuates across the board. This is perhaps indicative of some inherent noise within Linux's data.

- $\phi$: In contrast to memcached, $\phi$ begins to decrease as QPS load increases. We suspect this is due to the SLA objectives where each request is now required to arrive earlier in the receive queue so that there is enough buffer time to process the request and send a response without violating the objective.

- $log(\gamma)$: Similar to memcached, our fitting shows a similar conversion factor for $\gamma$ in both OSes, but the physical value seems to indicate that the base cost

has changed relative to the application.

- $\beta$: In contrast to $\alpha$, $\beta$ is now showing difference behaviors with respect to energy than time, possibly due to the complex interactions of ITR and DVFS to save energy while maintaining SLA.

### 6.3.2 Closed Loop Discussion

Fig. 6.1 illustrates an example result of our model fitting with NetPIPE for messages sized at 8 KB. We find that our model can accurately fit message sizes where there is a discernible trade-off for performance and energy when using ITR and DVFS. For example, fig. 6.1 shows that though our model has a correct fit for Linux, it did not work as well for EbbRT at 8 KB message sizes. This is due to OS path efficiencies of EbbRT where the use of a single fast ITR in EbbRT was able to achieve both best case performance and energy, therefore there is no discernible performance and energy trade-off with using ITR to induce batching behavior.

- $Z$: For a light workload such as NetPIPE, we expect the time to process each request as negligible given it is mainly network driven.

- $\alpha$: Similar to memcached, when the application work is light, the performance of a more efficient OS is largely not affected by a slow DVFS.

- $\gamma * DVFS + \delta$: These two parameters are intertwined in how ITR is affected by DVFS. The $\delta$ is a base cost that is consistent and a smaller $\gamma$ is indicative that the system isn't as affected by DVFS.

- $CF_a, CF_b, CF_c$: These constant factors are relatively stable across both OSes to convert the time to energy.

| | | NetPIPE | | | |
|---|---|---|---|---|---|
| | | 65536 | | 524288 | |
| | | Linux | EbbRT | Linux | EbbRT |
| Performance | $Z$ | 0.000058 | 0.000019 | 0.00003 | 0.000022 |
| | $\alpha$ | -0.23 | -0.85 | -0.10 | -0.66 |
| | $\gamma$ | 0.11 | 0.09 | 0.13 | -0.13 |
| | $\delta$ | 0.69 | 0.60 | 0.66 | 0.68 |
| Energy | $CF_a$ | 16.52 | 16.69 | 19.22 | 14.92 |
| | $CF_b$ | 4.16 | 5.7 | 3.19 | 3.60 |
| | $CF_c$ | 15.27 | 13.4 | 13.38 | 15.41 |

**Table 6.3:** Values for free parameters in NetPIPE fromdoing fit with Adam optimizer.

## 6.4 MODEL LIMITATIONS

Using the experimental data alongside our detailed systems' analysis, we show how properties of the OS and applications induce trade-offs for batching and processor energy settings under different loads. As these two underlying parameters change, the result is a response behavior that is both stable and structured enough to allow us to develop analytical models that capture performance and energy profiles in an OS agnostic way. While our models demonstrate an accurate fit for our data. It is not practical for all but highly constrained static settings. To replicate this approach one needs to exhaustively tune parameters and gather data. However, the accuracy of our model's suggest the viability in statistical approaches to efficiently find ITR and DVFS settings that improve the behavior of a network application. Building on these results, the next chapter describes a framework known as Bayesian optimization that can be used to automatically control batching and processor energy settings for real applications.

# CHAPTER 7

## Tuning with Machine Learning Techniques

While the analytical models discussed in chapter 6 demonstrate that one can precompute ideal batching and processor energy settings for some software stacks; it was only only applied in highly constrained static environments. However, the existence and accuracy of the analytic model's equations suggests the viability of using a black-box learner to exploit these structures. In this chapter, we present an example of this in our use of such a black-box learner to efficiently find batching and energy settings that target performance and energy goals across the applications and OSes. Such a technique can compensate for inaccuracies in our analytical model and the need for exhaustively searched experimental data.

At a high level, given a target *metric* to optimize (performance, energy or a combination of them), we can treat the problem as that of optimizing a function of two variables:

$$\text{itr}_*, \text{DVFS}_* = \text{metric}(\text{itr}, \text{DVFS}) \tag{7.1}$$

There is a large body of work on optimizing numerical functions using iterative techniques, however, in our use case there are some additional requirements:

- The domain consists of two quantities that are ordered but not continuous.
- The only accessible information about the metric is its value evaluated at a fixed ITR and DVFS. This function is considered as black-box as we don't know anything about its derivatives given the non-continuous nature of the domain.

- The process of evaluating this metric at a given ITR and DVFS, is potentially expensive, i.e. time-consuming. Ideally, we would like to find the optimum with as few evaluations as possible.

Given these requirements, we find Bayesian optimization (Frazier, 2018; Garnett, 2022) is a class of techniques that directly addresses these requirements. While Bayesian optimization has a long history, it has also seen a resurgence in interest in the last decade owing to its applications to hyper-parameter tuning for deep neural networks (Turner et al., 2021; Snoek et al., 2012). Below, we briefly describe the core elements of Bayesian optimization.

## 7.1 SUMMARY OF BAYESIAN OPTIMIZATION

To summarize, there is a true function $y = f(\vec{x})$ that maps points in the domain to the objective metric of interest. The core idea of Bayesian optimization is to search for a proxy, $f_\theta$ to this true function $f$ and use it to suggest points that are likely to be optimal.

The process starts by randomly sampling the domain to get a **fixed small number** of measurements $(\vec{x}_i, y_i)$ where $y_i$ is the true objective corresponding to the input $\vec{x}_i$. This data is used to then fit the proxy, $f_\theta$. The key point is that while the proxy will only be an approximation to the true function, it is very cheap to evaluate at any $\vec{x}$. The proxy is most often a Gaussian process that provides well-principled mean and uncertainty estimates. The proxy is then used to identify points that are most likely to lead to an objective value better than the current best observed objective. Once a measurement is made at this new point, it is used to update the parametric fit, $f_\theta$ and the process is repeated till a fixed budget of evaluations is exhausted. In practice, Bayesian optimization leads to very effective

solutions in a small fraction of evaluations.

To evaluate the feasibility of such a search on our problem, we first use the gathered dataset from the experimental study as a hash table to do look-ups for the optimization process. We use an off-the-shelf platform, Ax (Ax:Adaptive Experimentation Platform, 2022; Bakshy et al., 2018), to demonstrate Bayesian optimization's ability to find optimal ITR, DVFS configurations that resulted in lowest energy and best performance across the OSes and applications presented in this thesis. Next, we demonstrate a real-world application by applying it to a dynamic in-memory key-value store application over a 24 hour period.

## 7.2 BAYESIAN OPTIMIZATION APPLIED TO EXPERIMENTAL STUDY

We begin by applying Bayesian optimization to the existing dataset we have collected from our experimental study in chapter 2. The motivation was to verify and demonstrate its feasibility to dynamically exploit structures in the dual use of ITR, DVFS to optimize towards energy and performance in an OS and application agnostic way. In this process, each sample conducted by the Bayesian process results in a look-up of the ITR, DVFS combination from our study and the resultant energy or performance metric. This metric is the *reward* with which the Bayesian process uses to guide its optimization until all evaluation steps are exhausted.

An example of our results is shown in fig. 7.1 where the optimization process requires very few (compared to the total number of points) samples, to find the optimum or a point close to the optimal of lowest energy. Moreover, we find this framework is generic enough to be applicable in two fundamentally different OSes and across all the applications studied as shown in fig. 7.3, fig. 7.4, fig. 7.2. These results also demonstrate that Bayesian optimization can be used to target minimiz-

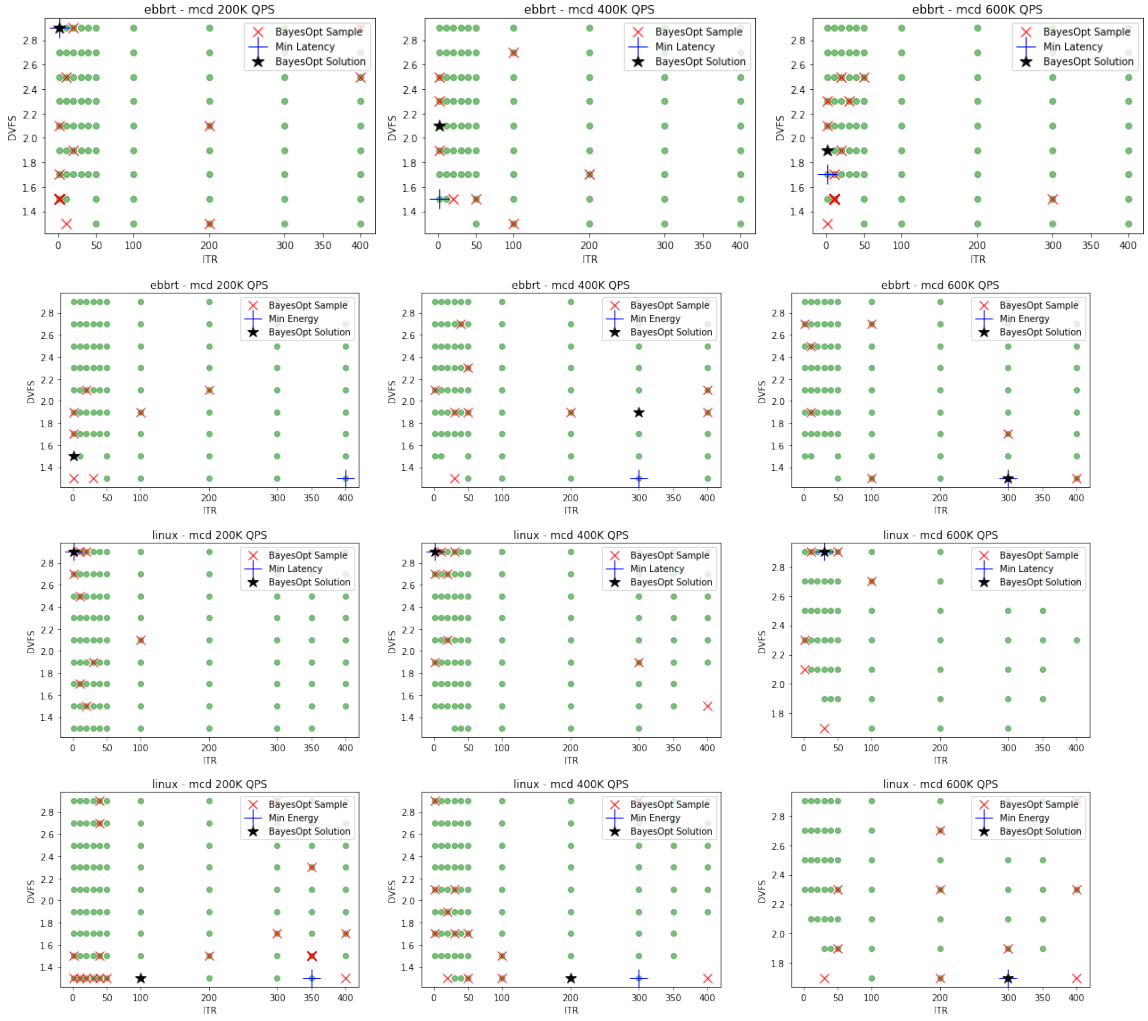


**Figure 7.1:** Bayesian optimization for Memcached for 99% tail latency and energy. The X and Y axis represent unique ITR, DVFS pairs in a single experimental run and is also illustrated by every **O**. We show the samples that the Bayesian process undertook via the **X**. The **+** indicates the best case (performance/energy) ITR, DVFS configuration found by Bayesian optimization and the * is the best case found so far by the exhaustive experimental study search.

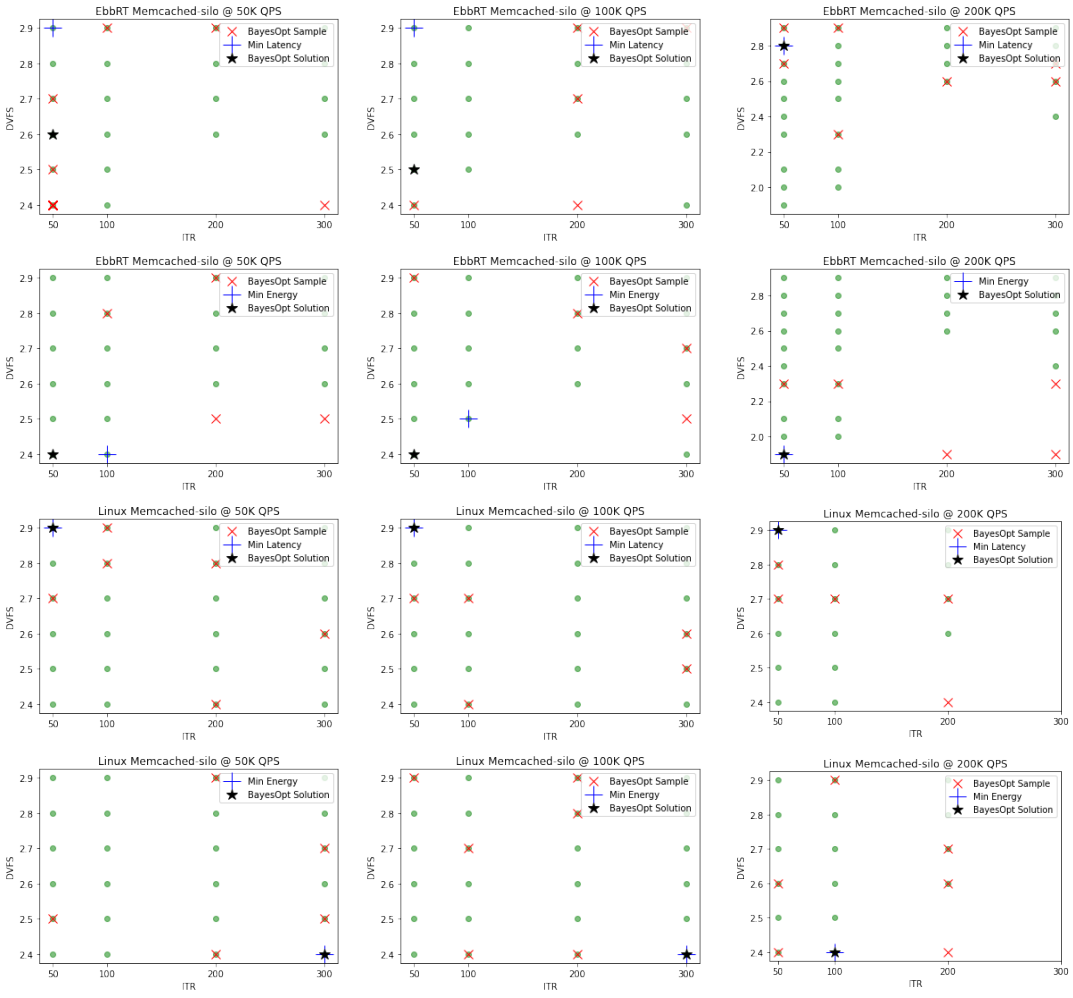


**Figure 7.2:** Bayesian optimization for 99% tail latency and energy in Memcached-silo.

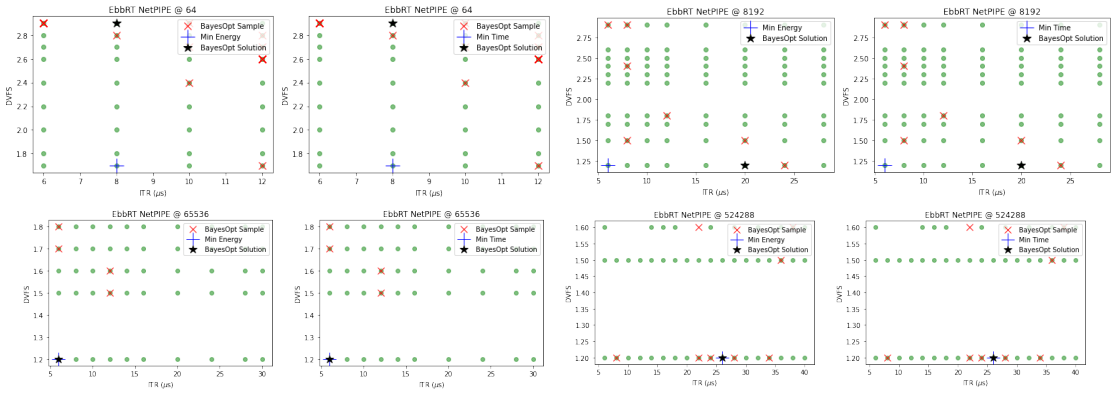


**Figure 7.3:** Bayesian optimization for NetPIPE for performance and energy.

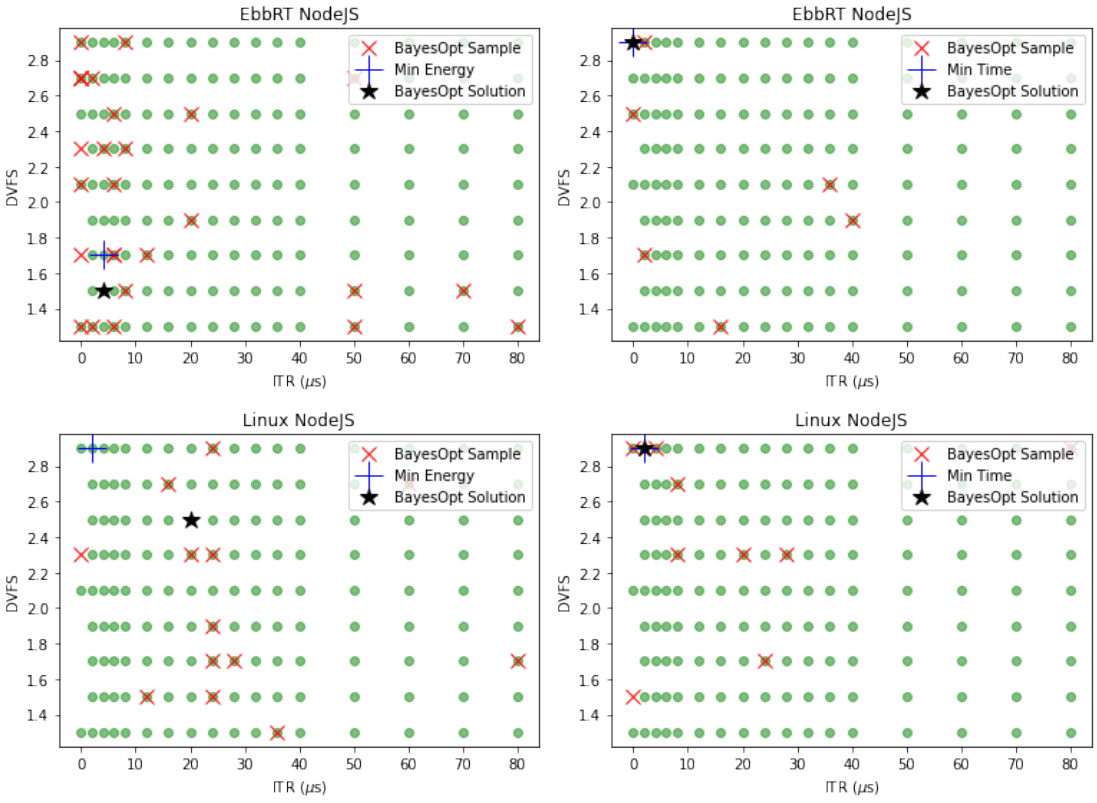


**Figure 7.4:** Bayesian optimization for NodeJS for performance and energy.

ing energy use and also improving performance. Motivated by these results, the next section describes our application in a real-world scenario.

## 7.3 BAYESIAN OPTIMIZATION APPLIED TO REAL-WORLD TRACE DATA

In order to probe the effectiveness of ITR and DVFS; our experimental study, modeling work, and black-box learner approach were applied in highly constrained static environments with only up to 340 combinations of ITR, DVFS values. In these circumstances, it is unrealistic and expensive to conduct such a study on new applications or different request rates to generate a dataset, and then guess, fit and refine the equations, which would then be used to identify a more precise value of ITR and DVFS to minimize latency or energy or some combination of both.

Therefore, to demonstrate the practicality of the black-box learner, this section presents results from applying Bayesian optimization to a publicly available in-memory key-value store workload trace from Twitter called
*cache-trace* (Juncheng Yang, 2020). We were motivated to explore this avenue as recent utilization studies of widely deployed services of in-memory key-value stores (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; Yang et al., 2020; Shashi Madappa, 2012; Daniel Ellis, 2017) reveal that these services often maintain a mean demand curve that changes over the course of hours and up to days in a repetitive way, which suggests that specialization of a single application at a specific offered load can be a realistic form of optimization to exploit the stable regions of these demand curves.

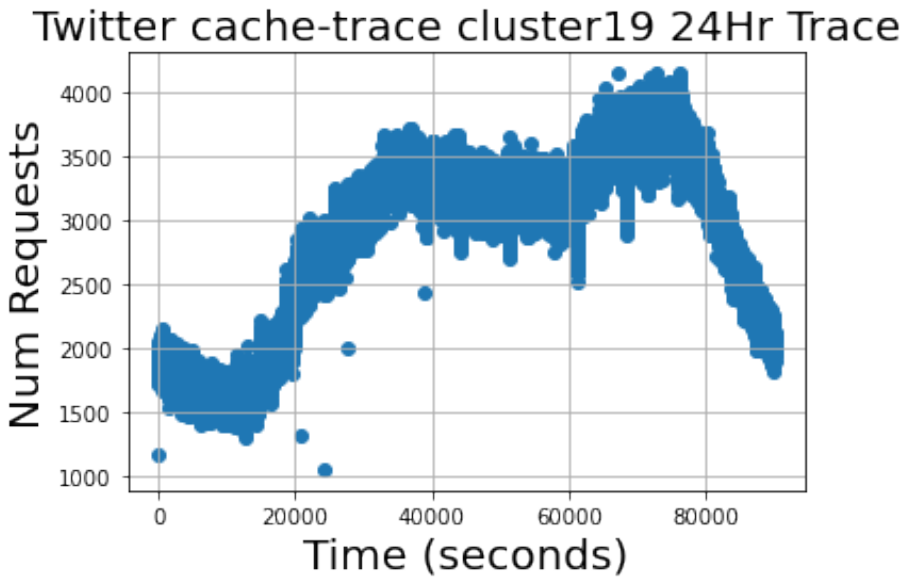


**Figure 7.5:** Raw requests-per-second log from Twitter cache-trace.

### 7.3.1 Experimental Setup

To conduct this experiment, we utilize the same itrLog infrastructure but modified the mutilate workload generator to generate requests from cache-trace instead. From cache-trace, first we selected a single trace log from one of Twitters servers that contained data from seven days and extracted a full 24 hour set of requests rates from that log. The raw request-per-second rate of this 24 hour trace log is shown in fig. 7.5. Next, we clustered the data from this 24 hour trace into 24 bins in order to capture the mean request rate which changes at an hourly basis. We then use the same infrastructure as chapter 3 to generate these mean request rates to our Linux appliance which runs a memcached server and capture its Watt (Joules/second) usage over an entire 24 hour period.

During this period, we trigger the Bayesian optimization process (on a separate machine) at a hourly rate. During this, the optimization process will run for a

number of trials[1] where each trial consists of: 1) setting a specific ITR, DVFS pair, 2) and measuring the resultant energy and latency of the server at the current request rate, and 3) using that measure to refine its optimization criteria.

At the end of the trial runs, we set the memcached server to use the specific ITR, DVFS pair which was determined by Bayesian optimization to be *optimal* to support the current request rate. This process is then triggered again on an hourly basis[2]. In contrast to our study, which was limited to only using up to 340 ITR, DVFS pairs due to experimental scope, we were able to let Bayesian optimization use all possible ITR, DVFS values instead; which consists of a total of *2 million* possible combinations.

### 7.3.2 Reward Function

In each trial of Bayesian optimization, we compute a reward function that is a combination of the resultant energy and latency measure of the memcached server. This reward function is used by the Bayesian process to optimize towards its objective. In this work, we use a simple function that penalizes the reward by the amount of measured Joules that violate the SLA objective in a linear fashion:

$$R = measured\_joules * (measured\_latency - SLA + 1) \tag{7.2}$$

This reward function has the additional benefit of enabling one to target different performance goals, for example, we show results where we changed the SLA objective from 500 $\mu$s to 200 $\mu$s below. In addition, it is also possible to change the $measured\_latency$ from 99% to other percentiles such as 90%, which is also shown

[1] Default of 30

[2] This process runs on a single thread and takes around 5 minutes total, we did not explore additional optimizations and leave that as future work.

below. In both cases, we demonstrate how Bayesian optimization can be used to automatically adapt to these changing requirements.

### 7.3.3 System Configurations

We compare result of applying Bayesian optimization across five different system configurations to get a better sense of its broad applicability.

**Linux:** Operating in its *default* state where the dynamic ITR and DVFS algorithms are enabled. DVFS algorithm is enabled to use the *power* governor (Dominik Brodowski, Nico Golde, Rafael J. Wysocki, Viresh Kumar, 2022) in which the CPU is typically set into the lowest operating frequency while idle and dynamically adjusts as workload changes.

**Linux-BayOp and EbbRT-BayOp:** In these configurations, we run a separate Bayesian process on another machine whose job is to tune ITR, DVFS statically towards a performance or energy goal. Given that the application is a memcached server, our initial target is to minimize overall energy use of the machine while maintaining the SLA objective such that 99% tail latency values are under 500 $\mu$s.

**Linux-DVFS-BayOp:** We only apply Bayesian optimization to optimize for DVFS while running with the dynamic ITR algorithm enabled.

**Linux-ITR-BayOp:** We only apply Bayesian optimization to optimize for ITR while running with the dynamic DVFS *powersave* algorithm enabled.

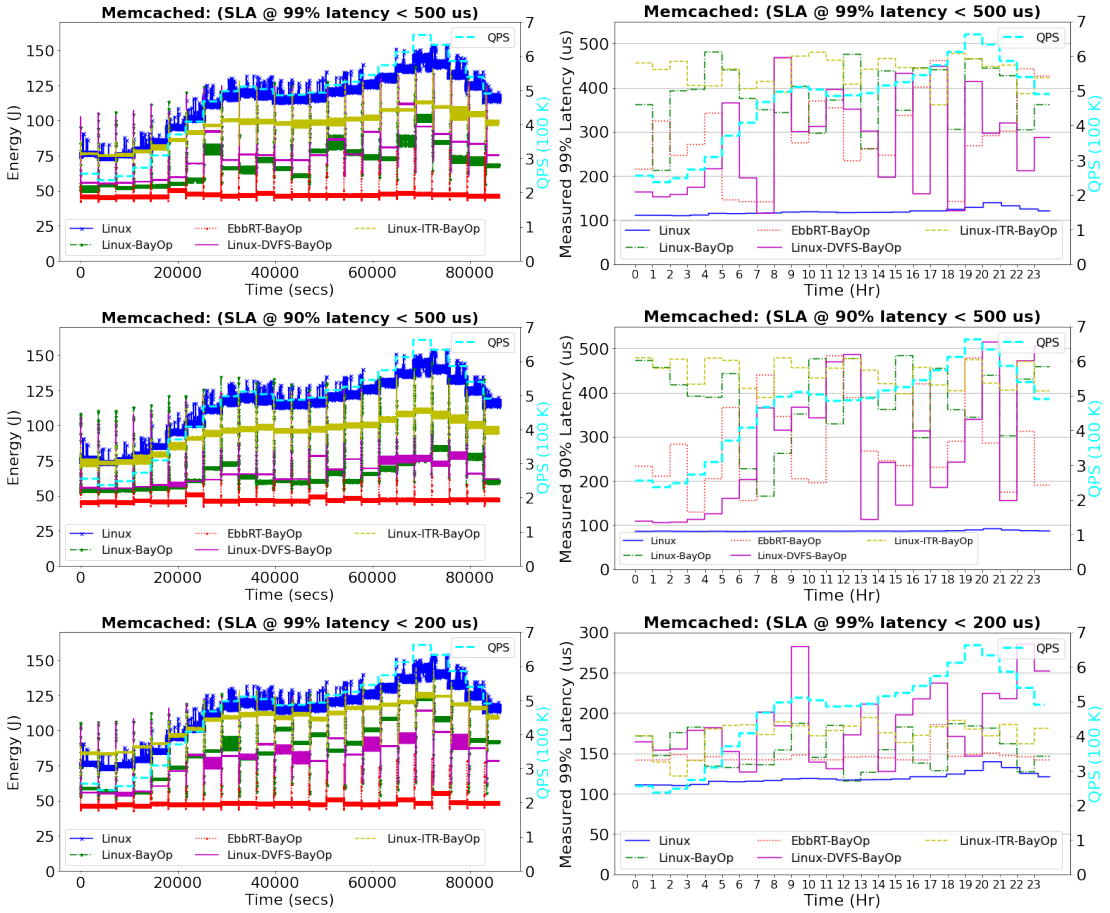


**Figure 7.6:** Bayesian optimization applied to Twitter cache-trace request rates over a 24 hour period. Each row represents a single SLA objective we are targeting and we display the change in energy (Joules) as QPS changes Next to each energy (Joules) figure, we also show the change in measured tail latency as QPS changes.

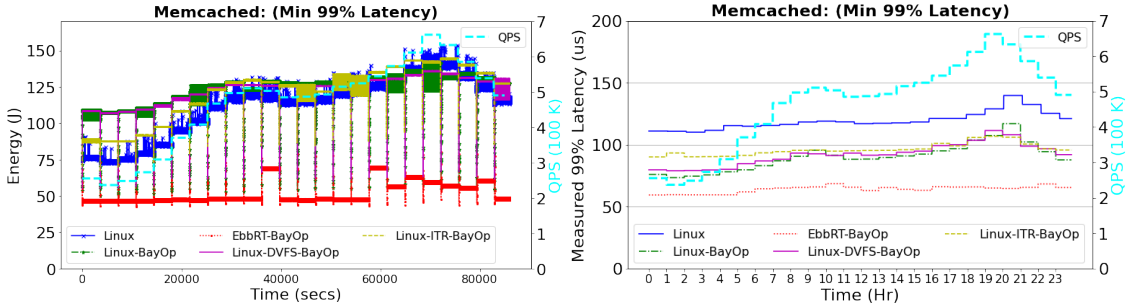


**Figure 7.7:** Bayesian optimization applied to Twitter cache-trace request rates where it used to optimize **only** for minimizing 99% tail latency. We show show the energy (Joules) consumption for this mode of operation.

### 7.3.4 Memcached Results

Fig. 7.6 illustrates examples of how one can utilize Bayesian optimization for the cache-trace data to minimize energy while satisfying the SLA objective. We demonstrate three different SLA objectives: 99% < 500 $\mu$s, 90% < 500 $\mu$s, 99% < 200 $\mu$s. The three figures on the left of fig. 7.6 show the overall energy use; in this figure, we show the change in QPS at an hourly basis on right side and the measured Watt (Joule/sec) use of the memcached server over the 24 hour period. The three figures on the right of fig. 7.6 shows the measured tail latency value during those QPS periods.

The spikes in energy use of *Linux-BayOp*, *EbbRT-BayOp*, *Linux-DVFS-BayOp*, *Linux-ITR-BayOp* in this figure illustrates the process by which Bayesian optimization is running and is dynamically changing ITR, DVFS settings on the server. After this initial energy spike, the system settles to a steady energy consumption state until the next hourly trigger.

In the case of an SLA objective of 99% latency < 500 $\mu$s, we find that *Linux-BayOp* can result in energy savings of up to 50% and by relaxing the SLA objective to 90% < 500 $\mu$s enabled Bayesian optimization to find ITR, DVFS configurations that yielded even more energy savings. To enable these energy savings, one can also see that in fig. 7.6 that while *Linux-BayOp* had higher tail latencies than *Linux*, it stayed within the specified SLA objective.

Fig. 7.6 also demonstrates another capability of our approach by allowing the changing of the SLA objective to be more stringent at 99% < 200 $\mu$s and this figure shows that Bayesian optimization can adapt to this lower tail latency requirement was still being able to save up to 30% more energy. Further, one can also observe that Linux's dynamic algorithms inherently cannot adapt to these changing re-

quirements due to its initial design.

One can also see that comparing *Linux-DVFS-BayOp* and *Linux-ITR-BayOp*, *Linux-DVFS-BayOp* typically results in much larger energy savings than purely modulating ITR. This is evident from the prior finding where DVFS drastically lowers energy use. However, we still see that it is the combined ITR-DVFS (*Linux-BayOp*) approach that yielded most energy savings in comparison. It should also be noted that while for the SLA @ 99% < 200 $\mu$s it seemed that *Linux-DVFS-BayOp* often resulted in lower energy use than *Linux-BayOp*, one can see that these scenarios often results in tail latencies that violate the SLA objective of 200 $\mu$s.

In fig. 7.7 we demonstrate another example of this optimization by changing reward function in eq. (7.2) to focus on minimizing tail latency instead. We find that in almost all cases the Bayesian process was able to find ITR, DVFS regimes that lowered the measured tail latency value by up to 30% but at a higher energy cost of up to 40%. Lastly, fig. 7.6 and fig. 7.7 demonstrate that Bayesian optimization can be applied to two fundamentally different OSes and still find ITR, DVFS settings that minimize latency and energy for both.

### 7.3.5 Memcached-silo Results

We also applied Bayesian optimization to memcached-silo as it is intensive both in computation and memory-use as it is structured such that every request triggers a corresponding set of TPC-C transaction processing logic on a in-memory database Tu et al. (2013).

Likewise, we selected another QPS trace from *cache-trace* that was akin to a more computationally intense behavior. Compared to fig. 7.6, fig. 7.8 shows a lower QPS rate overall. Even in this situation, we still find that for different SLA

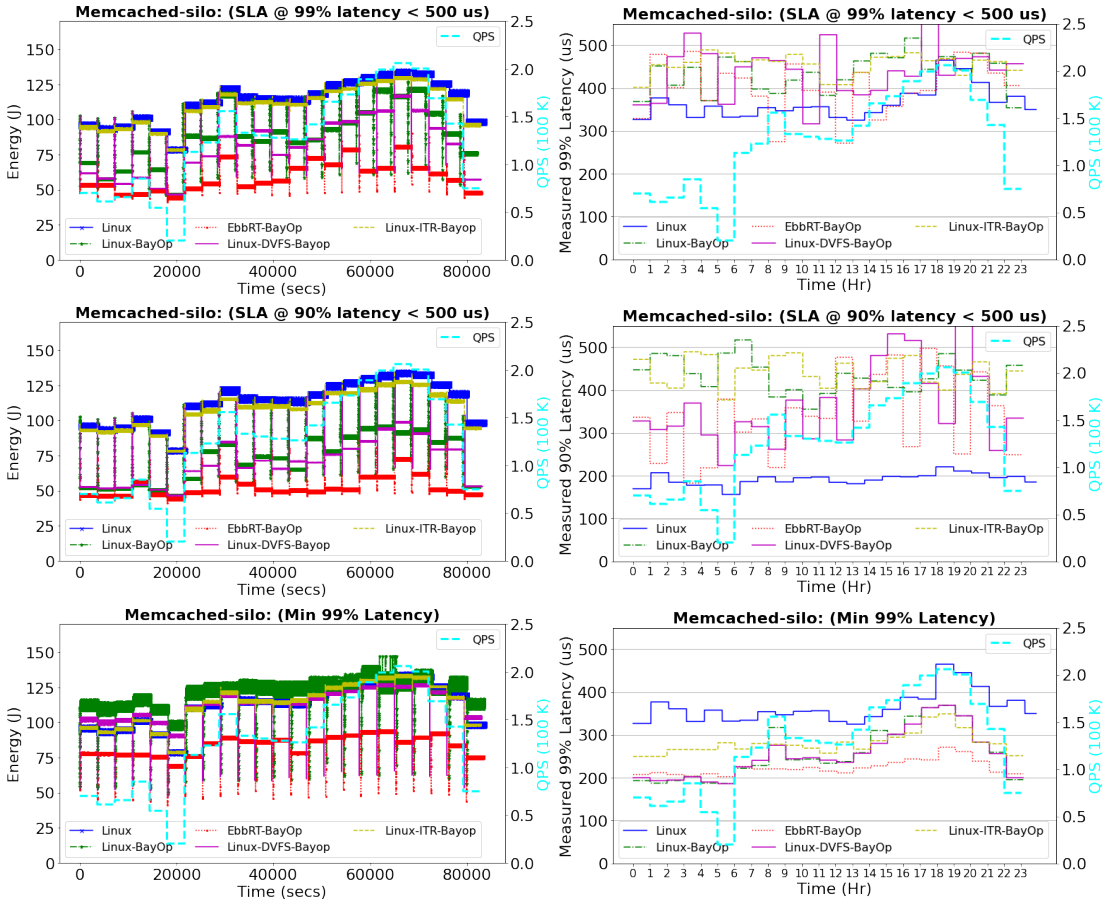


**Figure 7.8:** Bayesian optimization applied to memcached-silo over a 24 hour period. Each row represents a single SLA objective we are targeting and we display the change in energy (Joules) as QPS changes Next to each energy (Joules) figure, we also show the change in measured tail latency as QPS changes.

objectives, applying Bayesian optimization could reduce a server's energy consumption by over 50%. However, we do see that given a more computationally heavy application, the *Linux-ITR-BayOp* optimization did not result in as much benefits compared to memcached as it is no longer network driven. Subsequently, we find that only modulating DVFS via *Linux-DVFS-BayOp* also typically resulted in configurations that yielded best energy savings. Fig. 7.8 also shows in some cases the SLA objective was violated by the settings from Bayesian optimization, which suggest the difficulty in applying this technique in computationally heavier applications. However, this may be redeemed through more trials of Bayesian optimization as well.

### 7.3.6 Bayesian Optimization Implications

Below, we summarize a set of implications of the results of our work in applying Bayesian optimization to support a real world workload trace.

#### *7.3.6.1 Deployment in Datacenters*

Though our experiments were only configured for a single server, the benefits of Bayesian optimization approach suggests one can utilize this implementation on a single server and then build a simple load balancing mechanisms that drives requests at specific QPS rates to subsets of server machines where they have been specially configured with the correct hardware and OS settings to fully exploit the range of its performance and energy possibilities.

Moreover, we also did not optimize the Bayesian process in Ax so it only used a single thread for its work, this process can also be potentially sped up to further maximize its benefits. Overall, this task can also be improved by utilizing the rich

history of datacenter usage data that allows one to simulate and pre-configure sets of nodes with this previously known request rate changes.

#### *7.3.6.2 Reward Function*

Though in this thesis we only illustrate the reward metric of best performance or lowest energy, we believe this metric can be customized and expanded to new combinations of performance and energy or known metrics such as energy-delay-product (Horowitz et al., 1994; Brooks et al., 2000) as proposed by the architecture community.

#### *7.3.6.3 SLA Objectives*

As shown in fig. 7.6, our experimental setup enables one to change the SLA objectives to target different use cases. In these scenarios, a user can select the degree with which they are willing to trade-off performance for energy and vice versa. While previous works in taking advantage of SLA for lower energy have all focused on minizing energy at all costs (Wu et al., 2016; Hsu et al., 2018; Lo et al., 2014; Hsu et al., 2015; Kasture et al., 2015; Leverich & Kozyrakis, 2014; Prekas et al., 2017; Asyabi et al., 2020; Zhan et al., 2017; Vamanan et al., 2015; Meisner & Wenisch, 2012; Chou et al., 2016, 2019), our results demonstrate that this process can be automated and more wide ranging than previously shown.

#### *7.3.6.4 Learning Implicit Hardware Encoding*

We find that to change static DVFS values ultimately resulted in a write to the Intel processor's *IA32_PERF_CTL* MSR register. The exact meaning behind this register in terms of processor frequency and voltage is unfortunately not documented and

changes depending on the architecture of the CPU. The register itself contain a 16-bit encoding which reflects some current *performance* state and Intel provides no guarantee on what this architecture actually is.

Despite this, we found that Bayesian optimization was still able to adapt to changing request rates for different performance and energy goals. This implies that during its sampling stages it was able to discover some structure behind these hidden encodings. This implies that this process can be generic enough to also be applicable to new architectures that support the same mechanism but may have different *meanings* encoded behind those mechanisms.

# CHAPTER 8

## Related Work

Table 8.1 lists major research results in the area of using hardware features to control performance and/or energy. As shown in the table, most research in this area have solely focused on either DVFS or combining it with the power limiting capabilities of RAPL to modulate energy consumption while still meeting performance goals. Other than these two main hardware knobs, systems such as NapSAC (Krioukov et al., 2011), Heracles (Lo et al., 2015), and PowerNap (Meisner et al., 2009) also began to explore more experimental features such as Intel's Cache Allocation Technology (CAT) to explore how cache isolation can improve energy efficiency. These systems also tend to focus on the processing component of their respective applications, in contrast, this thesis presents the novel use of controlling network batching in conjunction with processor energy settings to enact even better performance and energy improvements. Further, these works have all focused on adapting their techniques for a particular set of applications by developing heuristics and systems that carefully control these hardware mechanisms. This thesis presents the novel use of an off-the-shelf machine learning library (Ax:Adaptive Experimentation Platform, 2022) to dynamically tune these mechanisms and adapt to both a diverse of applications and their respective loads.

**Table 8.1:** List of related systems and the hardware parameters explored. **WOL** – Wake-On-Lan capability on certain NICs and an experimental feature. **CAT** – Cache Allocation Technology hardware feature on certain Intel CPUs. **TB** – Turbo-Boost. **CS** – C-states.

| System | DVFS | RAPL | TB | CS | WOL | CAT | DRAM |
|---|---|---|---|---|---|---|---|
| Heracles (Lo et al., 2015) | ✓ | ✓ | | | | ✓ | |
| NapSac (Krioukov et al., 2011) | | | | ✓ | ✓ | | |
| (Fan et al., 2007) | ✓ | | | | ✓ | | |
| (Tolia et al., 2008b) | ✓ | | | | | | |
| Adrenaline (Hsu et al., 2015) | ✓ | | ✓ | | | | |
| SmoothOperator (Hsu et al., 2018) | ✓ | ✓ | | | | | |
| PowerNap (Meisner et al., 2009) | | | | ✓ | ✓ | | ✓ |
| Rubik (Kasture et al., 2015) | ✓ | | | | | | |
| Pegasus (Lo et al., 2014) | | ✓ | | | | | |
| IXCP (Prekas et al., 2015) | ✓ | | ✓ | | | | |
| Dynamo (Wu et al., 2016) | | ✓ | | | | | |
| (Guliani & Swift, 2019) | ✓ | ✓ | | | | | |
| (Meisner et al., 2011) | ✓ | | | ✓ | | | |
| Power Capping (Petoumenos et al., 2015) | ✓ | ✓ | | ✓ | | | |
| (Khan et al., 2018) | | ✓ | | | | | |
| (Zhang & Hoffman, 2015) | | ✓ | | | | | |
| (Lefurgy et al., 2007) | ✓ | | | | | | |

Continued on next page

**Table 8.1 – continued from previous page**

| System | DVFS | RAPL | TB | CS | WOL | CAT | DRAM |
|---|---|---|---|---|---|---|---|
| Pack & Cap (Cochran et al., 2011) | ✓ | | | | | | |
| (Isci et al., 2006) | ✓ | | | | | | |
| (Li & Martinez, 2006) | ✓ | | | | | | |
| (Sasaki et al., 2013) | ✓ | | | | | | |
| (Kanev et al., 2014) | ✓ | | | ✓ | | | |
| TURBO (Wamhoff et al., 2014) | ✓ | | | | | | |
| DreamWeaver (Meisner & Wenisch, 2012) | | | | ✓ | | | |
| (Kim et al., 2008) | ✓ | | | | | | |
| CPU MISER (Ge et al., 2007) | ✓ | | | | | | |
| Green Governors (Spiliopoulos et al., 2011) | ✓ | | | | | | |
| (Kondo et al., 2007) | ✓ | | | | | | |
| (Lee & Kim, 2009) | ✓ | ✓ | | | | | |

This dissertation also falls within a wider space of research on energy proportional computation in datacenters (Barroso, Luiz André and Hölzle, Urs, 2007; Fan et al., 2007; Tolia et al., 2008b). Much of this research stems from the challenges of improving the performance of network-bound datacenter workloads like MapReduce (Chen et al., 2012) and in-memory key-value stores (Lim et al., 2014; Prekas et al., 2017) 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 (Tolia et al., 2008a; Meisner et al., 2009; Krioukov et al., 2011) while simultaneously maintaining the ability to support high-utilization peaks and strict latency constraints (Wu et al., 2016; Hsu et al., 2018; Lo et al., 2014; Hsu et al., 2015; Kasture et al., 2015; Leverich & Kozyrakis, 2014; Prekas et al., 2017; Asyabi et al., 2020; Zhan et al., 2017; Vamanan et al., 2015; Meisner & Wenisch, 2012; Chou et al., 2016, 2019). 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. Furthermore, we demonstrate that is not only possible to do in-situ fine-grained data gathering and analysis of a complete full software stack, but also correlate the fine-grained packet by packet behavior to its impacts on an application's performance and energy profile.

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 (Sasaki et al., 2013; Flautner et al., 2001; Dominik Brodowski, Nico Golde, Rafael J. Wysocki, Viresh Kumar, 2022; Lefurgy et al., 2007; Cochran et al., 2011; Isci et al., 2006; Li & Martinez, 2006; Lee & Kim, 2009; Kim et al., 2008; Ge et al., 2007; Spiliopoulos et al., 2011; Kondo et al., 2007; Le Sueur & Heiser, 2011; Freeh et al., 2007; Elnozahy et al., 2003), processor power limiting mechanisms such as RAPL (Intel, 2022b; Lo et al., 2015; Hsu et al., 2018; Lo et al., 2014; Wu et al., 2016; Guliani & Swift, 2019; Petoumenos et al., 2015), and idle power states (Rafael J. Wysocki, 2018; Asyabi et al., 2020; Chou et al., 2019; Kanev

et al., 2014; Meisner & Wenisch, 2012; Kim et al., 2015b) (c-states) by applying feedback control mechanisms and relying on activity models. The authors of (Lo et al., 2015) and (Guliani & Swift, 2019) 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 (Tolia et al., 2008a; Hwang & Pedram, 2016) 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 (Le Sueur & Heiser, 2011; Meisner et al., 2009), 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 given the range of work on optimizing OS paths for performance, from NIC driver mechanisms (Kaufmann et al., 2016; Pesterev et al., 2012; BeifuSS et al., 2015) to the network stack (Jeong et al., 2014; Marinos et al., 2014; BeifuSS et al., 2015) and the dataplane (Peter et al., 2015; Schatzberg et al., 2016; Ousterhout et al., 2019; Prekas et al., 2017; Belay et al., 2014). Our work was also influenced by previous work in energy efficiency by slowing down both the networking and processor: $\mu$DPM (Chou et al., 2019) is a application-level policy for memcached to

delay request processing and maximize idle periods where deep sleep states can then be utilized, in (Laros et al., 2012) the authors combined bandwidth limiting in Cray clusters and scaling processor frequency to reduce energy use of HPC applications. In contrast to $\mu$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.

There is a synergistic relationship between the kind of system software used to support modern applications, the use of specialized features on modern hardware to support these applications, and the performance and energy trade-offs that exist within this space. Part of this has been explored with recent works on specializing systems towards a single application by using techniques such as library OSes (Belay et al., 2014; Peter et al., 2015; Ousterhout et al., 2019; Schatzberg et al., 2016; Prekas et al., 2017), kernel-bypass (Dragojević et al., 2014; Intel Corporation, 2022; Jeong et al., 2014; Lim et al., 2014; Yang et al., 2021; Yasukata et al., 2016), and unikernels (Madhavapeddy et al., 2013; Antti Kantee, Justin Cormack, 2014; Raza et al., 2019). However, they have typically only focused on performance. One aspect of our work is begin to document energy impact of these alternate system designs as the insights from the EbbRT results are also applicable to other application specific systems developed for accelerating network workloads (Belay et al., 2014; Peter et al., 2015; Prekas et al., 2017; Ousterhout et al., 2019; Antti Kantee, Justin Cormack, 2014; Raza et al., 2019; Madhavapeddy et al., 2013; 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; Qin et al., 2018; Jeong et al., 2014; Marinos et al., 2014; Pesterev et al., 2012; Kaufmann

et al., 2016; Lim et al., 2014; Welsh et al., 2001) as they share similar structural properties such as run-to-completion, event-driven execution model, single execution domain, and compile-time optimization, etc.

Previous work has advocated for full-system and hardware optimizations for energy efficiency (Le Sueur & Heiser, 2011; Meisner et al., 2009). Our work builds on that and asserts that OS path specialization itself plays a critical role in attaining energy efficiency. Furthermore, we take a more holistic view of performance and energy by reasoning about the fine-grained interactions between software and hardware at the per-interrupt level.

Modern hardware components and software stacks expose a large number of parameters that govern internal system operations and interactions. There is a lot of work on defining heuristics to control these parameters (Carroll & Heiser, 2014; Imes et al., 2015; Mishra et al., 2018, 2015; Dong et al., 2021; Kim et al., 2015a). In recent years, there has been an explosion in using ML-based techniques (Wu & Xie, 2023; Ding et al., 2019) to uncover more subtle system heuristics for resource management (Ganapathi et al., 2009; Bitirgen et al., 2008; Chen & John, 2011; Delimitrou & Kozyrakis, 2013, 2014; Dubach et al., 2010; Hoffmann, 2015; Ipek et al., 2008; Mishra et al., 2018, 2015; Oliner et al., 2013; Petrica et al., 2013; Snowdon et al., 2009; Zhu & Reddi, 2013), hardware and system configuration (Ansel et al., 2012; Chen et al., 2015; Deng et al., 2017; Dubach et al., 2010; Ganapathi et al., 2009; Lee & Brooks, 2006; Li & Martinez, 2006; Tomusk et al., 2015; Van Aken et al., 2017; Wu & Lee, 2012; Yigitbasi et al., 2013; Zhou et al., 2016; Zhu et al., 2017), high-performance computing (Lee et al., 2007; Mishra et al., 2015; Ipek et al., 2008; Zhang & Hoffmann, 2016; Ansel et al., 2012; Li & Martinez, 2006; Roy et al., 2021), and data-center-scale applications (Delimitrou & Kozyrakis, 2013, 2014; Yigitbasi

| System | Configurations | Application |
|---|---|---|
| (Bitirgen et al., 2008) | CPU, Cache Coherence | SPEC |
| (Chen & John, 2011) | SMT | SPEC CPU 2006 |
| (Dubach et al., 2010) | LLC | SPEC |
| (Ipek et al., 2008) | DRAM Load/Store | SPEC OpenMP |
| (Choi & Yeung, 2006) | CPU | SPEC |
| (Deng et al., 2017) | NVM | SPEC |
| (Li & Martinez, 2006) | DVFS | HPC |
| (Lee & Brooks, 2010) | Instructions | SPEC |
| (Wu & Lee, 2012) | Instructions | SPEC CPU |
| (Winter et al., 2010) | Branch Predictor, L1 | SPEC |
| NURD (Ding et al., 2022a) | CPU, Memory | Traces |

**Table 8.2:** List of related works of applying ML to automatically configure various configurations. However, all these works were run in a simulator only.

et al., 2013; Chen et al., 2015; Van Aken et al., 2017; Zhu et al., 2017; Tesauro, 2007; Ding et al., 2022a, 2021; Li et al., 2020; Wang et al., 2018; Ding et al., 2022b).

Though ML is a natural solution for domains like image, video, and audio processing, the complexity of computer systems often requires extensive expertise to map systems problems to ML tasks. For example table 8.2 lists recent works of applying ML to problems in the micro-architecture community, however all these works are demonstrated in simulators only. In table 8.3, there have been tremendous amounts of research to apply ML to tune data-center scale applications such as Hadoop, MapReduce, etc, and in many of these cases they target software level configurations only. Furthermore, in all of these works they only demonstrate feasibility on Linux whereas our work demonstrates how fundamental behaviors of these hardware mechanisms are applicable across OSes.

To our knowledge, there have been few publicly available systems datasets for ML (Joseph L. Hellerstein, 2010; Juncheng Yang, 2020; Atikoglu, Berk and Xu, Yuehai and Frachtenberg, Eitan and Jiang, Song and Paleczny, Mike, 2012; Cortez

et al., 2017; Ding et al., 2021). Further, our approach to conducting an experimental study, performing data analysis, and building a model which reflects the applicability of a black-box learner was motivated by the work of (Ding et al., 2019), who advocated that learning for systems should "incorporate the system problem's structure into the learner".

| System | Configurations | Application |
|---|---|---|
| Paragon (Delimitrou & Kozyrakis, 2013) | CPU, Cache, DRAM | Hadoop |
| Quasar Delimitrou & Kozyrakis (2014) | CPU | Hadoop, Memcached, Cassandra |
| CALOREE (Mishra et al., 2018) | CPU | Mobile |
| Carat (Oliner et al., 2013) | App. Energy | Mobile/Embedded |
| (Zhu & Reddi, 2013) | Big/LIttle Cores | Mobile |
| Pack & Cap (Cochran et al., 2011) | DVFS, Threads | PARSEC |
| (Lee et al., 2007) | Working Sets | HPC |
| (Yigitbasi et al., 2013) | Map/Reduce Tasks | MapReduce |
| (Zhang & Hoffmann, 2016) | Thread Migration | HPC |
| TEMM (Zhang et al., 2012) | DVFS | SPEC CPU |
| Siblingrivalry (Ansel et al., 2012) | Cores | HPC |
| (Chen et al., 2015) | Hadoop Config | Hadoop |
| (Van Aken et al., 2017) | DBMS Configs | TPC-C |
| BestConfig (Zhu et al., 2017) | Software Configs | Tomcat, Cassandra, Hive |
| (Ding et al., 2021) | Spark Configs | Spark |
| ALERT (Wan et al., 2020) | RAPL | Speech/Image |
| (Li et al., 2020) | Software Configs | HDFS, Cassandra, MapReduce |
| Cello (Ding et al., 2022b) | Spark Configs | Spark |
| (Wang et al., 2018) | Configs | HDFS, Cassandra, MapReduce |

**Table 8.3:** List of related works of applying ML to automatically configure various configurations for datacenter scale applications. However, most of these works only on software settings only and do not have publicly available datasets. Lastly, they have applied to Linux only.

# CHAPTER 9
# Future Work

In this chapter, we will discuss the details of implications and possible future directions of research that this thesis entails.

## 9.1 PERFORMANCE AND ENERGY STUDY

Linux contains many dynamic policies that have their own objectives and are full of implicit and/or hidden heuristics. Our exhaustive approach to conducting this study creates a methodology that lets us reveal and (re)define the real objectives of the parameters we tackle. We find that careful coordination among different hardware features has substantial benefits and should be used towards common performance and energy objectives in order to explore new trade-offs that achieve even further efficiencies over today's policies; which are often too narrowly focused. We believe that our method and subsequent data analysis approach can be reapplied towards different objectives in any system or domain. In turn, this can better enable developers to target what these built-in policies should be optimal for.

## 9.2 ENERGY REPORTING IN SYSTEMS RESEARCH

Although there have been many recent OS research topics focused on specialization techniques to accelerate network applications (Belay et al., 2014; Peter et al., 2015; Prekas et al., 2017; Ousterhout et al., 2019; Antti Kantee, Justin Cormack, 2014; Raza et al., 2019; Madhavapeddy et al., 2013; 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; Qin et al., 2018; Jeong et al., 2014; Marinos et al., 2014; Pesterev et al., 2012; Kaufmann et al., 2016; Lim et al., 2014; Welsh et al., 2001), we find that their impact on energy is not as clearly understood. By integrating our logging approach at the NIC's device driver level, we demonstrate the feasibility of such a test bed to perform the data driven driven exploration needed for our statically searched results. We believe that the set of tools, *itrLog*, is a step in the right direction to better understanding the complex interactions of modern systems software and its applications. As this tool contains a system and application agnostic way to log fine-grained measurement data, we believe it can be easily expanded upon by other researchers in other domains to better understand and report the performance and energy profiles of their systems and to discover new fine-grained interactions in order to enact similar performance-energy trade-offs.

## 9.3 SPECIALIZED OSES AND NETWORK PATH OPTIMIZATIONS

We demonstrate that there is significant benefit to dedicating OS stack and the hardware not only for workload, but also for specific offered loads. The performance and energy gains by the static configurations opens new opportunities for these optimizing widely deployed cloud services with mean demand curves that are stable across hours and days. While there is an enormous body of work demonstrating the performance advantages of application-specific OSes, our work demonstrate that these systems also offer enormous value in energy efficiency. Moreover, we see that the with simplified OS paths offer more opportunities for load-specific optimizations than general purpose operating systems. As our find-

ings for specialized OSes are generalizable due to observations of efficiency in network and application processing at a more fundamental level, such as IPC, we expect that many other application specific OSes such as unikernels and/or library OSes (Madhavapeddy et al., 2013) should contain similar energy efficiency benefits.

## 9.4 NIC POLLING WITHOUT SLEEP IN SPECIALIZED OS PATHS

The preliminary polling results reveals that there is enormous value to adopting specialized OS paths even in general purpose OSes for both performance and energy efficiency. Specialized systems that use polling to achieving low-latency (Marinos et al., 2014; Belay et al., 2014; Peter et al., 2015; Prekas et al., 2017; Ousterhout et al., 2019; Schatzberg et al., 2016; Cadden et al., 2020; Qin et al., 2018; Jeong et al., 2014; whe, 2012) 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. This also suggest optimizations such as OS path specialization and OS polling can be integrated into existing systems to enable these systems for both high performance and energy efficiency.

Modern processors provide a Monitor Wait (or MWAIT) instruction and it is typically used for power management by allowing a processor to select a specific C-state to enter after a call to HALT. However, a dual use of MWAIT is that can be used to monitor an address. A write to this address will automatically induce a processor wakeup from its halted state. With this capability, it becomes possible to explore an alternative system design where polling can be used in conjunction with sleep states for both performance and energy.

## 9.5 CONFIGURING NICS FOR PERFORMANCE

As an addendum to the work on development of 82599 NIC driver for baremetal EbbRT, it provided us an opportunity to explore how different combinations of NIC settings affected the performance or a real work application such as Memcached. We isolate and focus on the NIC features in EbbRT as its smaller codebase enabled us to experimentally validate the advantages of these features without requiring systemic modifications that impact various subsystems, potentially perturbing experimental conclusions.

Table 9.1 lists such a result and shows the overall peak QPSes achieved while maintaining SLA objectives of 99% tail latency of requests under 500 $\mu s$. We used the same experimental setup and software for running Memcached in EbbRT and we fixed ITR, DVFS to a single value that yield best performance. These results are separated between TSO single and multiple and for each TSO, the combinations of DCA and RSC is toggled and shown. TSO is a common feature and allows the device driver to give the NIC a large payload so that the NIC can automatically break the payload into multiple MTU sized frames with the appropriate Ethernet, TCP/IP header information. To explore impacts of TSO, we modified EbbRT's 82599 device driver to investigate TSO in two manners: 1) always use a **single** descriptor by copying payload to a single buffer, and 2) use **multiple** descriptors (up to max of 40). The motivation behind this is to better understand performance extremities of this feature. DCA is a mechanism where incoming packet data can be written directly into CPU caches via a hardware pre-fetch (Huggahalli et al., 2005; Farshin et al., 2020; Cai et al., 2021). RSC is the NIC's implementation of the large receive offload (Jonathan Corbet, 2007) in hardware.

Table 9.1 demonstrates that toggling these NIC configuration settings has dra-

| TSO | DCA | RSC | Peak QPS (K) |
|---|---|---|---|
| **single** | ✗ | ✗ | 2200 |
| | ✗ | ✓ | 2400 |
| | ✓ | ✗ | 2400 |
| | ✓ | ✓ | 2400 |
| **multiple** | ✗ | ✗ | 1400 |
| | ✗ | ✓ | 1200 |
| | ✓ | ✗ | 1200 |
| | ✓ | ✓ | 1200 |

**Table 9.1:** Measured performance of each NIC feature configuration when running **EbbRT memcached**. TSO is separated into *single* and *multiple* categories, and for the rest of the hardware features (DCA, RSC), this table lists the peak QPS achieved for every combination while maintaining SLA of 99% tail latency < 500 $\mu s$.

matic impact on peak throughput in Memcached by up to 1.71X **without modifications to the application or system code**. This result suggests that previous work in optimizing memcached (Belay et al., 2014; Lim et al., 2014; Prekas et al., 2017; Ousterhout et al., 2019; Ghigoff et al., 2021; Chou et al., 2016; Jin et al., 2017; Han et al., 2012; Yang et al., 2021; Yasukata et al., 2016) can benefit from adopting these hardware features as they share similar system design qualities such as kernel-bypass, no domain-crossings, run-to-completion of every request, etc. Furthermore as fig. 1.3 shows, there are over 5000 possible configurations of the NIC and our results also suggest there is enormous value to exploring how hardware can be configurable dependent on the application and OS for enormous benefits.

## 9.6 EXPANDING ON ITR-DELAY MECHANISM

There are a few advantages and disadvantages to the ITR mechanism. As it can only be set to a maximum of 1024 $\mu$s, it remains to be seen how its benefits fare in open loop workloads that use SLAs in the milliseconds range. Further, as it

interacts with re-transmit timeout policies in the TCP/IP stack; if one delays the interrupt mechanism too long, it could cause redundant retransmissions and decrease overall throughput while increase energy use. An opportunity with ITR is that it can also be modified on a per receive queue basis, which are typically affinitized to each distinct core. Potentially, if different workloads with different SLA objectives are consolidated on a single server node, it could be possible to customize interrupt receive delays on a per application basis while maintaining different SLAs across diverse set of applications.

## 9.7 MODEL EXTENSION

Our modeling work can also be expanded to a full probabilistic analysis using graphical models and an inference engine like Pyro (pyr, 2022). Further, it is also possible to generalize the open loop model across QPS values and/or combine the different models into a central model that captures all the workloads.

# CHAPTER 10
# Conclusion

This dissertation establishes that optimizing performance-energy simultaneously, despite complex OS structure, can be made a well-defined task using existing hardware mechanisms:

1. We design and construct a reproducible and new experimental methodology that enables the impact of different OS designs and implementations on application performance and energy to be studied in a controlled fashion. This methodology captures lossless in-situ fine-grained time series data, reflecting all packet and software interactions.

2. We plan to open source both the experimental infrastructure and the over 5 TB of collected data for existing researchers to study performance and energy impact in OSes.

3. Using our methodology, we are the first to demonstrate that application performance and energy can be significantly optimized by externally controlling (independent of the OS) request batching and processor speed using standard hardware mechanisms. We find one can achieve over 2X performance improvement for existing general purpose systems while using 2X lower energy and can be readily applied to other application domains.

4. We find specialized OS structure enables new and important interplay with energy consumption. Our work is the first to show that OS specialization has a dramatic impact not only on performance but also energy: over 2.5X energy savings in workloads such as Memcached and can sustain over 2X

higher throughput. There are even greater advantages from interactions with external controls as discussed in (2), such as new optimization opportunities using efficient polling leading to over 4X lower energy while improving performance by 3X.

5. We demonstrate results of our experimental study are sufficiently structured such that complex layers of systems software and their interactions in (2) can be characterized with a analytical model to accurately predict application performance and energy.

6. Lastly, we use an off-the-shelf black-box learner to demonstrate how this approach can be applied to save energy in Linux by up to 60% while supporting a real-world in-memory key-value store workload from Twitter by automatically adapting to different request rates and performance and energy goals.

The implications of this dissertation impacts not only OS design and implementation for energy efficiency but also data center orchestration in light of changing traffic patterns:

1. The experimental methodology is open sourced and empowers other developers to experimentally report energy implications on their system, and can also be extended to other domains.

2. Our results suggest that data center applications can significantly lower energy use by introducing simple load balancers that direct traffic to nodes with the correct static configuration. This design is amenable both to open loop applications that have a stable mean demand curve and closed loop applications with periodic packet transactions.

3. We demonstrate that library OSs are strong candidates to serve as tools for aggressive optimization of energy by both simplifying system policies for managing hardware features and enabling new forms of optimizations such as coupling processor frequencies with an OS poll.

4. Our discoveries suggest optimizations such as OS path specialization and OS polling can be integrated into existing systems to enable these systems to not only be performant but also energy efficient.

5. The analytical model results suggests an opportunity to combine low level system metrics with statistical learning techniques such as reinforcement learning to begin building smarter hardware policies within the OS.

6. The results with Bayesian optimization suggest it is practical way to deploy and target application performance and energy goals.

## BIBLIOGRAPHY

(2012). When poll is better than interrupt. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*. San Jose, CA: USENIX Association. https://www.usenix.org/conference/fast12/when-poll-better-interrupt.

(2022). 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.

(2022). MSR Tools. https://github.com/intel/msr-tools.

(2022). pyro.ai. https://pyro.ai/.

Agrawal, Sandeep R and Idicula, Sam and Raghavan, Arun and Vlachos, Evangelos and Govindaraju, Venkatraman and Varadarajan, Venkatanathan and Balkesen, Cagri and Giannikis, Georgios and Roth, Charlie and Agarwal, Nipun and Sedlar, Eric (2017). A Many-core Architecture for In-memory Data Processing. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (pp. 245–258). New York, NY, USA: ACM. http://doi.acm.org/10.1145/3123939.3123985.

Ansel, J., Pacula, M., Wong, Y. L., Chan, C., Olszewski, M., O'Reilly, U.-M., & Amarasinghe, S. (2012). Siblingrivalry: Online autotuning through local competitions. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '12, (p. 91100). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2380403.2380425.

Antti Kantee, Justin Cormack (2014). Rump Kernels: No OS? No Problem! https://www.usenix.org/publications/login/october-2014-vol-39-no-5.

ARM (2023). https://developer.arm.com/documentation/den0013/d/Power-Management.

Asyabi, E., Bestavros, A., Sharafzadeh, E., & Zhu, T. (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*, SoCC '20, (p. 150164). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3419111.3421298.

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*, SIGMETRICS '12, (pp. 53–64). New York, NY, USA: ACM. http://doi.acm.org/10.1145/2254756.2254766.

Ax:Adaptive Experimentation Platform (2022). https://ax.dev/.

Bakshy, E., Dworkin, L., Karrer, B., Kashin, K., Letham, B., Murthy, A., & Singh, S. (2018). Ae: A domain-agnostic platform for adaptive experimentation.

Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., & Ustuner, A. (2006). Thorough static analysis of device drivers. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, (pp. 73–85). New York, NY, USA: ACM. http://doi.acm.org/10.1145/1217935.1217943.

Barbalace, Antonio and Iliopoulos, Anthony and Rauchfuss, Holm and Brasche, Goetz (2017). It's Time to Think About an Operating System for Near Data Processing Architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, (pp. 56–61). New York, NY, USA: ACM. http://doi.acm.org/10.1145/3102980.3102990.

Barroso, L. A., Dean, J., & Hölzle, U. (2003). Web search for a planet: The google cluster architecture. *IEEE Micro*, *23*, 22–28.

Barroso, L. A., & Hoelzle, U. (2009). *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st ed.

Barroso, Luiz André and Hölzle, Urs (2007). The case for energy-proportional computing. *Computer*, *40*(12), 3337.

BeifuSS, A., Raumer, D., Emmerich, P., Runge, T. M., Wohlfart, F., Wolfinger, B. E., & Carle, G. (2015). A study of networking software induced latency. In *2015 International Conference and Workshops on Networked Systems (NetSys)*, (pp. 1–8).

Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., & Bugnion, E. (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*, OSDI'14, (p. 4965). USA: USENIX Association.

Bitirgen, R., Ipek, E., & Martinez, J. F. (2008). Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, (pp. 318–329).

Brewer, T., & Astfalk, G. (1997). The evolution of the hp/convex exemplar. In *Proceedings of the 42Nd IEEE International Computer Conference*, COMPCON '97, (pp. 81–). Washington, DC, USA: IEEE Computer Society. http://dl.acm.org/citation.cfm?id=792770.793731.

Brooks, D. M., Bose, P., Schuster, S. E., Jacobson, H., Kudva, P. N., Buyuktosunoglu, A., Wellman, J.-D., Zyuban, V., Gupta, M., & Cook, P. W. (2000). Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, *20*(6), 2644.

businesswire (2016). http://www.businesswire.com/news/home/20160425005805/en/.

Cadden, J., Unger, T., Awad, Y., Dong, H., Krieger, O., & Appavoo, J. (2020). Seuss: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3342195.3392698.

Cai, Q., Chaudhary, S., Vuppalapati, M., Hwang, J., & Agarwal, R. (2021). Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, (p. 6577). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3452296.3472888.

Carroll, A., & Heiser, G. (2014). Mobile multicores: Use them or waste them. *SIGOPS Oper. Syst. Rev.*, *48*(1), 4448.

Chen, C.-O., Zhuo, Y.-Q., Yeh, C.-C., Lin, C.-M., & Liao, S.-W. (2015). Machine learning-based configuration parameter tuning on hadoop system. In *2015 IEEE International Congress on Big Data*, (pp. 386–392).

Chen, J., & John, L. K. (2011). Predictive coordination of multiple on-chip resources for chip multiprocessors. In *Proceedings of the International Conference on Supercomputing*, ICS '11, (p. 192201). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1995896.1995927.

Chen, Y., Alspaugh, S., Borthakur, D., & Katz, R. (2012). Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys 12, (p. 4356). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2168836.2168842.

Cheriton, D. (1988). The V Distributed System. *Commun. ACM*, *31*(3), 314–333.

Cheriton, D. R., & Duda, K. J. (1994). A caching model of operating system kernel functionality. In *Proceedings of the 1st USENIX Conference on Operating Systems*

*Design and Implementation*, OSDI '94. Berkeley, CA, USA: USENIX Association. http://dl.acm.org/citation.cfm?id=1267638.1267652.

Choi, S., & Yeung, D. (2006). Learning-based smt processor resource distribution via hill-climbing. In *33rd International Symposium on Computer Architecture (ISCA'06)*, (pp. 239–251).

Chou, C., Bhuyan, L. N., & Wong, D. (2019). udpm: Dynamic power management for the microsecond era. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, (pp. 120–132). Los Alamitos, CA, USA: IEEE Computer Society. https://doi.ieeecomputersociety.org/10.1109/HPCA.2019.00032.

Chou, C.-H., Wong, D., & Bhuyan, L. N. (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*, ISLPED '16, (p. 212217). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2934583.2934616.

Cochran, R., Hankendi, C., Coskun, A. K., & Reda, S. (2011). Pack & Cap: Adaptive DVFS and Thread Packing under Power Caps. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (p. 175185). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2155620.2155641.

Corbet, J. (2003). The seq_file Interface. https://www.kernel.org/doc/html/latest/filesystems/seq_file.html.

Cortez, E., Bonde, A., Muzio, A., Russinovich, M., Fontoura, M., & Bianchini, R. (2017). Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (p. 153167). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3132747.3132772.

Daniel Ellis (2017). https://web.archive.org/web/20210205121832/https://redditblog.com/2017/1/17/caching-at-reddit/.

David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., & Le, C. (2010). Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED 10, (p. 189194). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1840845.1840883.

Delimitrou, C., & Kozyrakis, C. (2013). Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (p. 7788). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2451116.2451125.

Delimitrou, C., & Kozyrakis, C. (2014). Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, (p. 127144). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2541940.2541941.

Deng, Z., Zhang, L., Mishra, N., Hoffmann, H., & Chong, F. T. (2017). Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in nvms. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (p. 232244). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3123939.3124548.

Desrochers, S., Paradis, C., & Weaver, V. M. (2016). A validation of dram rapl power measurements. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, (p. 455470). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2989081.2989088.

Ding, Y., Mishra, N., & Hoffmann, H. (2019). Generative and multi-phase learning for computer systems optimization. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, (p. 3952). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3307650.3326633.

Ding, Y., Pervaiz, A., Carbin, M., & Hoffmann, H. (2021). Generalizable and interpretable learning for configuration extrapolation. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, (p. 728740). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3468264.3468603.

Ding, Y., Rao, A., Song, H., Willett, R., & Hoffmann, H. (2022a). Nurd: Negative-unlabeled learning for online datacenter straggler prediction. https://arxiv.org/abs/2203.08339.

Ding, Y., Renda, A., Pervaiz, A., Carbin, M., & Hoffmann, H. (2022b). Cello: Efficient computer systems optimization with predictive early termination and censored regression. https://arxiv.org/abs/2204.04831.

Dominik Brodowski, Nico Golde, Rafael J. Wysocki, Viresh Kumar (2022). CPU frequency and voltage scaling code in the Linux(TM) kernel. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

Dong, H., Arora, S., Awad, Y., Unger, T., Krieger, O., & Appavoo, J. (2021). Slowing down for performance and energy: An os-centric study in network driven workloads. https://arxiv.org/abs/2112.07010. https://arxiv.org/abs/2112.07010.

Dragojević, A., Narayanan, D., Castro, M., & Hodson, O. (2014). Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, (pp. 401–414). Seattle, WA: USENIX Association. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi{ć}.

Dubach, C., Jones, T. M., Bonilla, E. V., & O'Boyle, M. F. (2010). A predictive model for dynamic microarchitectural adaptivity control. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, (pp. 485–496).

Elnozahy, M., Kistler, M., & Rajamony, R. (2003). Energy conservation policies for web servers. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, (p. 8). USA: USENIX Association.

Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., & Burger, D. (2011). Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (p. 365376). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2000064.2000108.

Fan, X., Weber, W.-D., & Barroso, L. A. (2007). Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA 07, (p. 1323). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1250662.1250665.

Farshin, A., Roozbeh, A., Jr., G. Q. M., & Kostić, D. (2020). Reexamining direct cache access to optimize i/o intensive applications for multi-hundred-gigabit networks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, (pp. 673–689). USENIX Association. https://www.usenix.org/conference/atc20/presentation/farshin.

Field Programmable Gate Arrays (FPGAs) (2022). https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html.

Flautner, K., Reinhardt, S., & Mudge, T. (2001). Automatic performance setting for dynamic voltage scaling. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, MobiCom '01, (p. 260271). New York, NY,

USA: Association for Computing Machinery. https://doi.org/10.1145/381677.381702.

Frazier, P. I. (2018). A tutorial on bayesian optimization. https://arxiv.org/abs/1807.02811.

Freeh, V. W., Bletsch, T. K., & Rawson, F. L. (2007). Scaling and packing on a chip multiprocessor. In *2007 IEEE International Parallel and Distributed Processing Symposium*, (pp. 1–8).

Gamsa, B., Krieger, O., Appavoo, J., & Stumm, M. (1999). Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, (pp. 87–100). Berkeley, CA, USA: USENIX Association. http://dl.acm.org/citation.cfm?id=296806.296814.

Ganapathi, A., Datta, K., Fox, A., & Patterson, D. (2009). A case for machine learning to optimize multicore performance. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, (p. 1). USA: USENIX Association.

Ganapathy, V., Renzelmann, M. J., Balakrishnan, A., Swift, M. M., & Jha, S. (2008). The design and implementation of microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, (pp. 168–178). New York, NY, USA: ACM. http://doi.acm.org/10.1145/1346281.1346303.

Garnett, R. (2022). *Bayesian Optimization*. Cambridge University Press. In preparation.

Ge, R., Feng, X., Feng, W.-c., & Cameron, K. W. (2007). Cpu miser: A performance-directed, run-time system for power-aware clusters. In *2007 International Conference on Parallel Processing (ICPP 2007)*, (pp. 18–18).

Ghigoff, Y., Sopena, J., Lazri, K., Blin, A., & Muller, G. (2021). BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, (pp. 487–501). USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/ghigoff.

Glozer, W. (2014). wrk: Modern HTTP benchmarking tool. https://github.com/wg/wrk.

Golestani, H., Mirhosseini, A., & Wenisch, T. F. (2019). Software data planes: You can't always spin to win. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, (p. 337350). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3357223.3362737.

Google (2022). V8 JavaScript Engine. http://code.google.com/p/v8/.

Graphics Processing Units (2022). http://www.nvidia.com/object/what-is-gpu-computing.html.

Grindley, R., Abdelrahman, T., Brown, S., Caranci, S., DeVries, D., Gamsa, B., Grbic, A., Gusat, M., Ho, R., Krieger, O., Lemieux, G., Loveless, K., Manjikian, N., McHardy, P., Srbljic, S., Stumm, M., Vranesic, Z., & Zilic, Z. (2000). The numachine multiprocessor. In *Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, ICPP '00, (pp. 487–). Washington, DC, USA: IEEE Computer Society. http://dl.acm.org/citation.cfm?id=850941.852940.

Gu, B., Yoon, A. S., Bae, D.-H., Jo, I., Lee, J., Yoon, J., Kang, J.-U., Kwon, M., Yoon, C., Cho, S., Jeong, J., & Chang, D. (2016). Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (pp. 153–165). Piscataway, NJ, USA: IEEE Press. https://doi.org/10.1109/ISCA.2016.23.

Guliani, A., & Swift, M. M. (2019). Per-application power delivery. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3302424.3303981.

Gupta, U., Kim, Y. G., Lee, S., Tse, J., Lee, H.-H. S., Wei, G.-Y., Brooks, D., & Wu, C.-J. (2020). Chasing carbon: The elusive environmental footprint of computing.

Han, S., Marshall, S., Chun, B.-G., & Ratnasamy, S. (2012). Megapipe: A new programming interface for scalable network i/o. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, (p. 135148). USA: USENIX Association.

Hanford, N., Ahuja, V., Farrens, M. K., Tierney, B., & Ghosal, D. (2018). A survey of end-system optimizations for high-speed networks. *ACM Comput. Surv.*, *51*(3).

Hoffmann, H. (2013). Racing and pacing to idle: An evaluation of heuristics for energy-aware resource allocation. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2525526.2525854.

Hoffmann, H. (2015). Jouleguard: Energy guarantees for approximate applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, (p. 198214). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2815400.2815403.

Horowitz, M., Indermaur, T., & Gonzalez, R. (1994). Low-power digital design. In *Proceedings of 1994 IEEE Symposium on Low Power Electronics*, (pp. 8–11).

Hsu, C., Zhang, Y., Laurenzano, M. A., Meisner, D., Wenisch, T., Mars, J., Tang, L., & Dreslinski, R. G. (2015). Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, (pp. 271–282).

Hsu, C.-H., Deng, Q., Mars, J., & Tang, L. (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*, ASPLOS '18, (pp. 535–548). New York, NY, USA: ACM. http://doi.acm.org/10.1145/3173162.3173190.

https://memcached.org (2020). Memcached. https://github.com/memcached/memcached.

Huggahalli, R., Iyer, R., & Tetrick, S. (2005). Direct cache access for high bandwidth network i/o. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, ISCA '05, (p. 5059). USA: IEEE Computer Society. https://doi.org/10.1109/ISCA.2005.23.

Hwang, I., & Pedram, M. (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), 2103–2116.

Imes, C., Kim, D. K., Maggio, M., & Hoffmann, H. (2015). Poet: a portable approach to minimizing energy under soft real-time constraints. In *2015 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, (pp. 75–86). Los Alamitos, CA, USA: IEEE Computer Society. https://doi.ieeecomputersociety.org/10.1109/RTAS.2015.7108419.

Intel (2021). 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.

Intel (2022a). Intel 64 and IA-32 Architectures Software Developers Manual Volume. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/.

Intel (2022b). Intel 64 and IA-32 Architectures Software Developers 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.

Intel (2022c). Intel 64 and IA-32 Architectures Software Developers 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.

Intel 82599 10 Gigabit Ethernet Controller: Datasheet (2022). https://www.intel.com/content/www/us/en/embedded/products/networking/82599-10-gbe-controller-datasheet.html.

Intel Corporation (2022). Intel DPDK: Data Plane Development Kit. http://dpdk.org.

Ipek, E., Mutlu, O., Martínez, J. F., & Caruana, R. (2008). Self-optimizing memory controllers: A reinforcement learning approach. *SIGARCH Comput. Archit. News*, *36*(3), 3950.

Isci, C., Buyuktosunoglu, A., Cher, C.-Y., Bose, P., & Martonosi, M. (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, (p. 347358). USA: IEEE Computer Society. https://doi.org/10.1109/MICRO.2006.8.

J. Leverich (2022). Mutilate: high performance memcached load generator. https://github.com/leverich/mutilate.

Jeong, E., Wood, S., Jamshed, M., Jeong, H., Ihm, S., Han, D., & Park, K. (2014). mtcp: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, (pp. 489–502). Seattle, WA: USENIX Association. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong.

Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., & Stoica, I. (2017). Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (p. 121136). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3132747.3132764.

Jonathan Corbet (2007). Large receive offload. https://lwn.net/Articles/243949/.

Joseph L. Hellerstein (2010). Google Cluster Data . https://ai.googleblog.com/2010/01/google-cluster-data.html.

Joyent (2013). Node.js. https://github.com/nodejs/node/tree/cc56c62ed879ad4f93b1fdab3235c43e60f48b7e.

Juncheng Yang, R. V., Yao Yue (2020). A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association. https://www.usenix.org/conference/osdi20/presentation/yang.

Kadav, A., & Swift, M. M. (2012). Understanding modern device drivers. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (pp. 87–98). New York, NY, USA: ACM. http://doi.acm.org/10.1145/2150976.2150987.

Kan Liang, Andi Kleen, and Jesse Brandenburg (2010). Improve Network Performance By Setting Per-queue Interrupt Moderation In Linux. https://01.org/linux-interrupt-moderation.

Kanev, S., Hazelwood, K., Wei, G.-Y., & Brooks, D. (2014). Tradeoffs between power management and tail latency in warehouse-scale applications. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, (pp. 31–40).

Kasture, H., Bartolini, D. B., Beckmann, N., & Sanchez, D. (2015). Rubik: Fast analytical power management for latency-critical systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, (pp. 598–610).

Kaufmann, A., Peter, S., Sharma, N. K., Anderson, T., & Krishnamurthy, A. (2016). High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 16, (p. 6781). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2872362.2872367.

Khan, K. N., Hirki, M., Niemi, T., Nurminen, J. K., & Ou, Z. (2018). Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, *3*(2).

Kim, D. H., Imes, C., & Hoffmann, H. (2015a). Racing and pacing to idle: Theoretical and empirical analysis of energy optimization heuristics. In *2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications*, (pp. 78–85).

Kim, D. H. K., Imes, C., & Hoffmann, H. (2015b). 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, (p. 7885). USA: IEEE Computer Society. https://doi.org/10.1109/CPSNA.2015.23.

Kim, W., Gupta, M. S., Wei, G.-Y., & Brooks, D. (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*, (pp. 123–134).

Kondo, M., Sasaki, H., & Nakamura, H. (2007). Improving fairness, throughput and energy-efficiency on a chip multiprocessor through dvfs. *SIGARCH Comput. Archit. News*, *35*(1), 3138.

Krioukov, A., Mohan, P., Alspaugh, S., Keys, L., Culler, D., & Katz, R. (2011). Napsac: Design and implementation of a power-proportional web cluster. *SIGCOMM Comput. Commun. Rev.*, *41*(1), 102108.

Krste Asanovi (????). FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. FAST 2014, year=2014.

Kuskin, J., Ofelt, D., Heinrich, M., Heinlein, J., Simoni, R., Gharachorloo, K., Chapin, J., Nakahira, D., Baxter, J., Horowitz, M., Gupta, A., Rosenblum, M., & Hennessy, J. (1994). The stanford flash multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, ISCA '94, (pp. 302–313). Los Alamitos, CA, USA: IEEE Computer Society Press. http://dx.doi.org/10.1145/191995.192056.

Laros, J. H., Pedretti, K. T., Kelly, S. M., Shu, W., & Vaughan, C. T. (2012). Energy based performance tuning for large scale high performance computing systems. HPC '12. San Diego, CA, USA: Society for Computer Simulation International.

Laudon, J., & Lenoski, D. (1997). The sgi origin: A ccnuma highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, (pp. 241–251). New York, NY, USA: ACM. http://doi.acm.org/10.1145/264107.264206.

Le Sueur, E., & Heiser, G. (2011). Slow down or sleep, that is the question. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, (p. 16). USA: USENIX Association.

Lee, B. C., & Brooks, D. (2010). Applied inference: Case studies in microarchitectural design. *ACM Trans. Archit. Code Optim.*, *7*(2).

Lee, B. C., & Brooks, D. M. (2006). Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (p. 185194). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1168857.1168881.

Lee, B. C., Brooks, D. M., de Supinski, B. R., Schulz, M., Singh, K., & McKee, S. A. (2007). Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, (p. 249258). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1229428.1229479.

Lee, J., & Kim, N. S. (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*, DAC '09, (p. 4750). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1629911.1629926.

Lefurgy, C., Wang, X., & Ware, M. (2007). Server-level power control. In *Fourth International Conference on Autonomic Computing (ICAC'07)*, (pp. 4–4).

LeVasseur, J., Uhlig, V., Stoess, J., & Götz, S. (2004). Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (pp. 2–2). Berkeley, CA, USA: USENIX Association. http://dl.acm.org/citation.cfm?id=1251254.1251256.

Leverich, J., & Kozyrakis, C. (2014). Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2592798.2592821.

Li, C., Wang, S., Hoffmann, H., & Lu, S. (2020). Statically inferring performance properties of software configurations. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3342195.3387520.

Li, J., & Martinez, J. (2006). Dynamic power-performance adaptation of parallel computation on chip multiprocessors. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, (pp. 77–87).

libuv (2022). http://libuv.org.

Lim, H., Han, D., Andersen, D. G., & Kaminsky, M. (2014). MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, (pp. 429–444). Seattle, WA: USENIX Association. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim.

Lo, D., Cheng, L., Govindaraju, R., Barroso, L. A., & Kozyrakis, C. (2014). Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (p. 301312). IEEE Press.

Lo, D., Cheng, L., Govindaraju, R., Ranganathan, P., & Kozyrakis, C. (2015). Heracles: Improving resource efficiency at scale. *SIGARCH Comput. Archit. News*, *43*(3S), 450462.

Lovett, T., & Clapp, R. (1996). Sting: A cc-numa computer system for the commercial marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, (pp. 308–317). New York, NY, USA: ACM. http://doi.acm.org/10.1145/232973.233006.

Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., & Crowcroft, J. (2013). Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, (pp. 461–472). New York, NY, USA: ACM. http://doi.acm.org/10.1145/2451116.2451167.

Marinos, I., Watson, R. N., & Handley, M. (2014). Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, (pp. 175–186). New York, NY, USA: ACM. http://doi.acm.org/10.1145/2619239.2626311.

Mark Silberstein (2017). Accelerators in data centers: the systems perspective. https://www.sigarch.org/accelerators-in-data-centers-the-systems-perspective/.

Mass Open Cloud (2022). https://massopen.cloud/.

Meisner, D., Gold, B. T., & Wenisch, T. F. (2009). Powernap: Eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, (p. 205216). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1508244.1508269.

Meisner, D., Sadler, C. M., Barroso, L. A., Weber, W.-D., & Wenisch, T. F. (2011). Power management of online data-intensive services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, (p. 319330). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2000064.2000103.

Meisner, D., & Wenisch, T. F. (2012). Dreamweaver: Architectural support for deep sleep. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, (p. 313324). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2150976.2151009.

Mellanox (2022). https://community.mellanox.com/s/article/understanding-interrupt-moderation.

Mellanox Innova SmartNIC (2016). http://www.businesswire.com/news/home/20160615005424/en/.

Mishra, N., Imes, C., Lafferty, J. D., & Hoffmann, H. (2018). Caloree: Learning control for predictable latency and low energy. *SIGPLAN Not.*, *53*(2), 184198.

Mishra, N., Zhang, H., Lafferty, J. D., & Hoffmann, H. (2015). A probabilistic graphical model-based approach for minimizing energy under performance constraints. *SIGPLAN Not.*, *50*(4), 267281.

Nicola Jones (2020). How to stop data centres from gobbling up the worlds electricity. https://www.nature.com/articles/d41586-018-06610-y.

Oliner, A. J., Iyer, A. P., Stoica, I., Lagerspetz, E., & Tarkoma, S. (2013). Carat: Collaborative energy diagnosis for mobile devices. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2517351.2517354.

Ousterhout, A., Fried, J., Behrens, J., Belay, A., & Balakrishnan, H. (2019). Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, NSDI'19, (p. 361377). USA: USENIX Association.

Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., & Welch, B. B. (1988). The Sprite Network Operating System. *Computer*, *21*(2), 23–36.

Pareto efficiency (2022). https://en.wikipedia.org/wiki/Pareto_efficiency.

Perez, M. (1995). Scalable hardware evolves, but what about network OS? PCWeek 1995.

Pesterev, A., Strauss, J., Zeldovich, N., & Morris, R. T. (2012). Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys 12, (p. 337350). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2168836.2168870.

Peter, S., Li, J., Zhang, I., Ports, D. R. K., Woos, D., Krishnamurthy, A., Anderson, T., & Roscoe, T. (2015). Arrakis: The operating system is the control plane. *ACM Trans. Comput. Syst.*, *33*(4).

Petoumenos, P., Mukhanov, L., Wang, Z., Leather, H., & Nikolopoulos, D. S. (2015). Power capping: What works, what does not. In *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, (pp. 525–534).

Petrica, P., Izraelevitz, A. M., Albonesi, D. H., & Shoemaker, C. A. (2013). Flicker: A dynamically adaptive architecture for power limited multicore systems. *SIGARCH Comput. Archit. News*, *41*(3), 1323.

Pike, R. (2000). Systems software research is irrelevant.

Pike, R., Presotto, D., Thompson, K., & Trickey, H. (1990). Plan 9 from bell labs. In *In Proceedings of the Summer 1990 UKUUG Conference*, (pp. 1–9).

ploty (2022). Dash Overview. https://plotly.com/dash/.

Prekas, G. (2017). https://github.com/ix-project/servers/tree/master.

Prekas, G., Kogias, M., & Bugnion, E. (2017). Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP 17, (p. 325341). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3132747.3132780.

Prekas, G., Primorac, M., Belay, A., Kozyrakis, C., & Bugnion, E. (2015). Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, (p. 342355). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2806777.2806848.

Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J.-Y., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P. Y., & Burger, D. (2014). A reconfigurable fabric for accelerating large-scale datacenter services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (pp. 13–24). Piscataway, NJ, USA: IEEE Press. http://dl.acm.org/citation.cfm?id=2665671.2665678.

pytorch (2022). Adam. https://pytorch.org/docs/stable/generated/torch.optim.Adam.html.

Qin, H., Li, Q., Speiser, J., Kraft, P., & Ousterhout, J. (2018). Arachne: Core-aware thread management. In *13th USENIX Symposium on Operating Systems Design*

*and Implementation (OSDI 18)*, (pp. 145–160). Carlsbad, CA: USENIX Association. https://www.usenix.org/conference/osdi18/presentation/qin.

R. Balasubramonian and J. Chang and T. Manning and J. H. Moreno and R. Murphy and R. Nair and S. Swanson (2014). Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro*, *34*(4), 36–42.

Rafael J. Wysocki (2018). CPU Idle Time Management. https://www.kernel.org/doc/html/v5.0/admin-guide/pm/cpuidle.html.

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)*, (pp. 385–398). Lombard, IL: USENIX. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

Raza, A., Sohal, P., Cadden, J., Appavoo, J., Drepper, U., Jones, R., Krieger, O., Mancuso, R., & Woodman, L. (2019). Unikernels: The next stage of linuxs dominance. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS 19, (p. 713). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3317550.3321445.

Ren, X. J., Rodrigues, K., Chen, L., Vega, C., Stumm, M., & Yuan, D. (2019). An analysis of performance evolution of linux's core operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, (p. 554569). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3341301.3359640.

Renzelmann, M. J., & Swift, M. M. (2009). Decaf: Moving device drivers to a modern language. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, (pp. 14–14). Berkeley, CA, USA: USENIX Association. http://dl.acm.org/citation.cfm?id=1855807.1855821.

Roy, R. B., Patel, T., Gadepally, V., & Tiwari, D. (2021). Bliss: Auto-tuning complex applications using a pool of diverse lightweight learning models. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, (p. 12801295). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3453483.3454109.

Ryzhyk, L., Chubb, P., Kuz, I., & Heiser, G. (2009). Dingo: Taming device drivers. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, (pp. 275–288). New York, NY, USA: ACM. http://doi.acm.org/10.1145/1519065.1519095.

Ryzhyk, L., Walker, A., Keys, J., Legg, A., Raghunath, A., Stumm, M., & Vij, M. (2014). User-guided device driver synthesis. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, (pp. 661–676). Berkeley, CA, USA: USENIX Association. http://dl.acm.org/citation.cfm?id=2685048.2685101.

Ryzhyk, L., Zhu, Y., & Heiser, G. (2010). The case for active device drivers. In *Proceedings of the First ACM Asia-pacific Workshop on Workshop on Systems*, APSys '10, (pp. 25–30). New York, NY, USA: ACM. http://doi.acm.org/10.1145/1851276.1851283.

Sasaki, H., Imamura, S., & Inoue, K. (2013). Coordinated power-performance optimization in manycores. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, (p. 5162). IEEE Press.

Schatzberg, D., Cadden, J., Dong, H., Krieger, O., & Appavoo, J. (2016). Ebbrt: A framework for building per-application library operating systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, (pp. 671–688). GA: USENIX Association. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/schatzberg.

Schroeder, B., Wierman, A., & Harchol-Balter, M. (2006). Open versus closed: A cautionary tale. In *Proceedings of the 3rd Conference on Networked Systems Design and Implementation - Volume 3*, NSDI'06, (p. 18). USA: USENIX Association.

Schüpbach, A., Baumann, A., Roscoe, T., & Peter, S. (2011). A declarative language approach to device configuration. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, (pp. 119–132). New York, NY, USA: ACM. http://doi.acm.org/10.1145/1950365.1950382.

Semiconductor Industry Association (2015). 2015 International Technology Roadmap for Semiconductors (ITRS).

Shaffer, M. W. (2000). A linux appliance construction set. In *14th Systems Administration Conference (LISA 2000)*. New Orleans, LA: USENIX Association. https://www.usenix.org/conference/lisa-2000/linux-appliance-construction-set.

Shashi Madappa (2012). https://netflixtechblog.com/ephemeral-volatile-caching-in-the-cloud-8eba7b124589.

Snell, Q. O., Mikler, A. R., & Gustafson, J. L. (1996). Netpipe: A Network Protocol Independent Performance Evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*.

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. https://arxiv.org/abs/1206.2944.

Snowdon, D. C., Le Sueur, E., Petters, S. M., & Heiser, G. (2009). Koala: A platform for os-level power management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, (p. 289302). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1519065.1519097.

Spiliopoulos, V., Kaxiras, S., & Keramidas, G. (2011). Green governors: A framework for continuously adaptive dvfs. In *Proceedings of the 2011 International Green Computing Conference and Workshops*, IGCC '11, (p. 18). USA: IEEE Computer Society. https://doi.org/10.1109/IGCC.2011.6008552.

Strubell, E., Ganesh, A., & McCallum, A. (2019). Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, (pp. 3645–3650). Florence, Italy: Association for Computational Linguistics. https://www.aclweb.org/anthology/P19-1355.

Tang, C., Yu, K., Veeraraghavan, K., Kaldor, J., Michelson, S., Kooburat, T., Anbudurai, A., Clark, M., Gogia, K., Cheng, L., Christensen, B., Gartrell, A., Khutornenko, M., Kulkarni, S., Pawlowski, M., Pelkonen, T., Rodrigues, A., Tibrewal, R., Venkatesan, V., & Zhang, P. (2020). Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, (pp. 787–803). USENIX Association. https://www.usenix.org/conference/osdi20/presentation/tang.

Tesauro, G. (2007). Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing*, *11*(1), 22–30.

The Linux Foundation (2022). napi. https://wiki.linuxfoundation.org/networking/napi.

Tolia, N., Wang, Z., Marwah, M., Bash, C., Ranganathan, P., & Zhu, X. (2008a). Delivering energy proportionality with non energy-proportional systems: Optimizing the ensemble. In *Proceedings of the 2008 Conference on Power Aware Computing and Systems*, HotPower'08, (p. 2). USA: USENIX Association.

Tolia, N., Wang, Z., Marwah, M., Bash, C., Ranganathan, P., & Zhu, X. (2008b). Delivering energy proportionality with non energy-proportional systems—optimizing the ensemble. In *Workshop on Power Aware Computing and Systems (HotPower 08)*. San Diego, CA: USENIX Association. https://www.usenix.org/conference/hotpower-08/delivering-energy-proportionality-non-energy-proportional-systems{\T1\textemdash}optimizing.

Tom Herbert, Willem de Bruijn (2022). Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

Tomusk, E., Dubach, C., & Oboyle, M. (2015). Four metrics to evaluate heterogeneous multicores. *ACM Trans. Archit. Code Optim.*, *12*(4).

Tu, S., Zheng, W., Kohler, E., Liskov, B., & Madden, S. (2013). Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP 13, (p. 1832). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2517349.2522713.

Turner, R., Eriksson, D., McCourt, M., Kiili, J., Laaksonen, E., Xu, Z., & Guyon, I. (2021). Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. https://arxiv.org/abs/2104.10201.

Vamanan, B., Sohail, H. B., Hasan, J., & Vijaykumar, T. N. (2015). Timetrader: Exploiting latency tail to save datacenter energy for online search. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, (p. 585597). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2830772.2830779.

Van Aken, D., Pavlo, A., Gordon, G. J., & Zhang, B. (2017). Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, (p. 10091024). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3035918.3064029.

Verghese, B., Devine, S., Gupta, A., & Rosenblum, M. (1996). Operating system support for improving data locality on cc-numa compute servers. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, (pp. 279–289). New York, NY, USA: ACM. http://doi.acm.org/10.1145/237090.237205.

Wamhoff, J.-T., Diestelhorst, S., Fetzer, C., Marlier, P., Felber, P., & Dice, D. (2014). The turbo diaries: application-controlled frequency scaling explained. In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference*, (pp. 193–204). USENIX Association.

Wan, C., Santriaji, M. H., Rogers, E., Hoffmann, H., Maire, M., & Lu, S. (2020). Alert: Accurate learning for energy and timeliness. In A. Gavrilovska, & E. Zadok (Eds.) *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, (pp. 353–369). USENIX Association. https://www.usenix.org/conference/atc20/presentation/wan.

Wang, S., Li, C., Hoffmann, H., Lu, S., Sentosa, W., & Kistijantoro, A. I. (2018). *Understanding and Auto-Adjusting Performance-Sensitive Configurations*, (p. 154168). New York, NY, USA: Association for Computing Machinery.

Welsh, M., Culler, D., & Brewer, E. (2001). Seda: An architecture for well-conditioned, scalable internet services. *SIGOPS Oper. Syst. Rev.*, *35*(5), 230243.

Winter, J. A., Albonesi, D. H., & Shoemaker, C. A. (2010). Scalable thread scheduling and global power management for heterogeneous many-core architectures. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, (pp. 29–39).

Wu, N., & Xie, Y. (2023). A survey of machine learning for computer architecture and systems. *ACM Computing Surveys*, *55*(3), 1–39.

Wu, Q., Deng, Q., Ganesh, L., Hsu, C.-H., Jin, Y., Kumar, S., Li, B., Meza, J., & Song, Y. J. (2016). Dynamo: Facebook's data center-wide power management system. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, (pp. 469–480). Piscataway, NJ, USA: IEEE Press. https://doi.org/10.1109/ISCA.2016.48.

Wu, W., & Lee, B. C. (2012). Inferred models for dynamic and sparse hardware-software spaces. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, (pp. 413–424).

Yang, J., Yue, Y., & Rashmi, K. V. (2020). A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, (pp. 191–208). USENIX Association. https://www.usenix.org/conference/osdi20/presentation/yang.

Yang, J., Yue, Y., & Vinayak, R. (2021). Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, (pp. 503–518). USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/yang-juncheng.

Yasukata, K., Honda, M., Santry, D., & Eggert, L. (2016). StackMap: Low-Latency networking with the OS stack and dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, (pp. 43–56). Denver, CO: USENIX Association. https://www.usenix.org/conference/atc16/technical-sessions/presentation/yasukata.

Yigitbasi, N., Willke, T. L., Liao, G., & Epema, D. (2013). Towards machine learning-based auto-tuning of mapreduce. In *2013 IEEE 21st International Symposium*

*on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, (pp. 11–20).

Young, M., Tevanian, A., Rashid, R., Golub, D., & Eppinger, J. (1987). The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, (pp. 63–76). New York, NY, USA: ACM. http://doi.acm.org/10.1145/41457.37507.

Zhan, X., Azimi, R., Kanev, S., Brooks, D., & Reda, S. (2017). Carb: A c-state power management arbiter for latency-critical workloads. *IEEE Computer Architecture Letters*, *16*(1), 6–9.

Zhang, H., & Hoffman, H. (2015). A quantitative evaluation of the rapl power control system. *Feedback Computing*.

Zhang, H., & Hoffmann, H. (2016). Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, (p. 545559). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2872362.2872375.

Zhang, X., Zhong, R., Dwarkadas, S., & Shen, K. (2012). A flexible framework for throttling-enabled multicore management (temm). In *2012 41st International Conference on Parallel Processing*, (pp. 389–398).

Zhou, Y., Hoffmann, H., & Wentzlaff, D. (2016). Cash: Supporting iaas customers with a sub-core configurable architecture. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, (pp. 682–694).

Zhu, Y., Liu, J., Guo, M., Bao, Y., Ma, W., Liu, Z., Song, K., & Yang, Y. (2017). Bestconfig: Tapping the performance potential of systems via automatic configuration tuning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, (p. 338350). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3127479.3128605.

Zhu, Y., & Reddi, V. J. (2013). High-performance and energy-efficient mobile web browsing on big/little systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, (pp. 13–24).

**CURRICULUM VITAE**

# Han Dong

January 10, 2023


111 Cummington Mall
Boston, MA 02215
(617) 353-8919
handong@bu.edu

Work office
Boston University
Boston, MA 02215
handong@bu.edu


**Academic Training:**

| | |
|---|---|
| 01/2023(expected) | PhD Boston University, Boston, MA; Computer Science |
| 05/2013 | BS University of Maryland Baltimore County, Baltimore, MD; Computer Science |
| 05/2010 | BS University of Maryland Baltimore County, Baltimore, MD; Computer Science |

**Doctoral Research:**

| | |
|---|---|
| **Title**: | A Data-Driven Study of Operating System Energy-Performance Trade-offs towards System Self Optimization |
| **Thesis advisor**: | Jonathan Appavoo, PhD |
| **Defense date**: | October 17, 2022 |
| **Summary**: | Despite growing complexity in software and hardware, my thesis research demonstrates that taking a disciplined data-driven approach to understanding performance-energy trade-offs in OSes can shed better light on how to formulate ML tasks towards systems self-optimization. |

**Original, Peer Reviewed Publications (newest first):**

1. James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: skip redundant paths to make serverless fast. *In Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 115. https://doi.org/10.1145/3342195.3392698.

2. Dan Schatzberg and James Cadden and Han Dong and Orran Krieger and Jonathan Appavoo. *EbbRT: A Framework for Building Per-Application Library*

*Operating Systems*. 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), 2016. 978-1-931971-33-1. 671–688.