Energy-efficient Data-intensive Computing with a Fast Array of Wimpy Nodes

Vijay R. Vasudevan

CMU-CS-11-131
October 2011

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David G. Andersen, Chair
Luiz A. Barroso, Google
Gregory R. Ganger
Garth A. Gibson
Michael E. Kaminsky, Intel Labs

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by the U.S. Army Research Office under grant number W911NF-09-1-0273, the National Science Foundation under grant numbers ANI-0331653, CNS-0546551, CNS-0619525, CNS-0716287, and CCF-0964474, Intel, by gifts from Network Appliance and Google, and through fellowships from APC by Schneider Electric, Facebook, and Yahoo! Inc.

Keywords: Energy Efficiency, Low Power, Cluster Computing, Flash

For my family by blood, my family by sweat, and my family by tears.

Abstract

Large-scale data-intensive computing systems have become a critical foundation for Internet-scale services. Their widespread growth during the past decade has raised datacenter energy demand and created an increasingly large financial burden and scaling challenge: Peak energy requirements today are a significant cost of provisioning and operating datacenters. In this thesis, we propose to reduce the peak energy consumption of datacenters by using a FAWN: A Fast Array of Wimpy Nodes. FAWN is an approach to building datacenter server clusters using low-cost, low-power servers that are individually optimized for energy efficiency rather than raw performance alone. FAWN systems, however, have a different set of resource constraints than traditional systems that can prevent existing software from reaping the improved energy efficiency benefits FAWN systems can provide.

This dissertation describes the principles behind FAWN and the software techniques necessary to unlock its energy efficiency potential. First, we present a deep study into building FAWN-KV, a distributed, log-structured key-value storage system designed for an early FAWN prototype. Second, we present a broader classification and workload analysis showing when FAWN can be more energy-efficient and under what workload conditions a FAWN cluster would perform poorly in comparison to a smaller number of high-speed systems. Last, we describe modern trends that portend a narrowing gap between CPU and I/O capability and highlight the challenges endemic to all future balanced systems. Using FAWN as an early example, we demonstrate that pervasive use of "vector interfaces" throughout distributed storage systems can improve throughput by an order of magnitude and eliminate the redundant work found in many data-intensive workloads.

Acknowledgments

I would not be where I am today without the mentorship of my advisor, David Andersen. Successfully convincing him to work with me five years ago remains a shock, and I spent most of my time trying to make sure he didn't regret the choice. Not only is he a fantastically creative thinker and teacher, he is an exceptionally fun person to be around: our research meetings would sometimes get hijacked by topics ranging from food, health or exercise to the latest new technology gadget he was tinkering with. He demonstrated first-hand how curiosity, intuition, and determination are the agents by which we solve big problems. If I am at all successful in my professional life, I can without hesitation attribute it to the experience I have had working with Dave.

This dissertation would not be what it is without the efforts of my mentor at Intel, Michael Kaminsky. He helped push my research forward by asking the right questions and connecting me with people and resources needed to fully examine the answers to those questions. He was an expert in keeping the big picture in mind, both in research and in life.

Garth Gibson was an endless source of advice, experience, and interesting research problems. Our work on Incast would not have gotten as far it did without his efforts. He always looked out for me as much as his own students, and for that I am eternally grateful.

Greg Ganger cracked me up as much as he made me think hard about my work (and he made me laugh *a lot*). He knew when to keep things light and when to push me and my colleagues hard. The success of his PDL leadership provided me with wonderful opportunities and experiences during retreats, visit days, and my job search.

Luiz Barroso was kind enough to provide feedback on my research on FAWN, helping to shape the direction and context of the latter part of my dissertation. If I were to have even a fraction of the influence he has had on the community, I will be proud, and I consider myself lucky to be joining a company he has helped build so successfully.

My early collaborations with Srini Seshan and Hui Zhang provided me with innumerable examples of how to think clearly and ask the right questions. I am indebted to my

undergraduate advisor, Ken Goldberg, for providing me a unique opportunity to explore what research is about, having good taste in problems, and, with Ron Alterovitz, encouraging me to pursue a Ph.D. I also owe a debt of gratitude to Scott Shenker and Ion Stoica for inspiring me to pursue graduate studies in distributed systems and networking, a field never lacking in hard, rewarding problems to solve.

This dissertation was aided by the support, effort, and discussions I had with Michael Kozuch and Babu Pillai. Their continual search for deep understanding of energy efficiency and balance helped shape a significant portion of the thesis work and ensured that the underlying principles were described more precisely. My summer internship with Sudipta Sengupta and Jin Li taught me a lot about the impact of research on real products, and I am sure these lessons will come in handy very soon.

The Parallel Data Lab provided a rich, stimulating environment in which to discuss everything in large-scale systems. The alumni and PDL consortium members were never hesitant to offer their mentorship and advice to graduate students young and old. Sneaking away from PDL retreat walks to play some tennis with Julio Lopez and Iulian Moraru provided some well-needed exercise as a reprieve from the grueling but educational experiences the PDL retreats offered us.

A large part of our department's incredible culture was the social atmosphere supported by FreeCSD and Dec5. Although my participation in both organizations was meager by comparison to others, I got to know most of the students in the department through these organizations. I particularly enjoyed Michael Tschantz's rants about funding and TG sponsorship. The departments would not function at all without the heavy lifting done by Deborah Cavlovich and Catherine Copetas, among numerous others. Angie Miller, Karen Lindenfelser, and Joan Digney managed to put up with my requests for help whenever I needed it; for that I owe them an unpayable debt of gratitude. I learned a tremendous amount from Michael Abd-El-Malek. He showed me by example how to critique and deliver a good talk, and provided me sage advice during my job hunt and decision process. I look forward to continued interaction with him and other PDL alumni over the next few years.

During my early years, I was fortunate to play ultimate frisbee with the likes of Vyas Sekar, Gaurav Veda, Amit Manjhi, Debabrata Dash, and many others. If they were still around to play frisbee with, I'd probably be a good fifteen pounds lighter. The incredible group of systems and networking students and faculty fostered a great community, one that I'll miss dearly. Tuesday seminar was always a useful sounding board in which to bounce off crazy ideas. Avijit Kumar and Pratyusa Manadhata were my officemates in Wean Hall and provided early guidance during my formative graduate student years. Varun Gupta and Mike Dinitz were always willing to teach me about systems theory in a way I could understand, though I am sure they were dumbing it down beyond what they could bear.

Kanat, Tunji, Fahad, and Sam, you were always patient in explaining to me complex ideas, and I hope to speak to others with such clarity in my daily conversations. I looked up to both David Brumley and Bruce Maggs during my first few years, both for the wisdom they imparted on me as well as for how much they put into the ice hockey team we briefly played on together. Jayant and Akshay: who would have thought that we would all be in the same doctoral program back when we were kids? I look forward to seeing all of you and the usual crew during our Thanksgiving and Christmas reunions. To Lawrence Tan, Hiral Shah, Christina Ooi, Alex Gartrell, Alex Katkova, Nathan Wan, Reinhard Munz, Jack Ferris, and Timmy Zhu: your intelligence, exuberance, and work ethic will take you far, if they haven't already. I now feel a pang of regret for not attending CMU as an undergrad.

To Bin, Hyeontaek, Hrishi and Wyatt: I am continually impressed with how you managed to do anything with the FAWN-KV codebase. The fact you have all done amazing research on top of it is a testament to your patience as well as your skills. Wittawat and Lin could always be found working hard at odd hours, and whenever I wanted to try a new restaurant, they would never hesitate to join me. Lin: one of these days Swapnil will eventually say yes to having coffee at Tazza—it's a graduation requirement for him. Whenever I contemplated doing anything outside of work, Elie Krevat was the one constant saying "Let's do it!". I won't forget softball and our 9am Guitar Hero session after pulling an all-nighter for the Incast project deadline (which surprisingly was my first ever).

Jiri Simsa showed me how to balance opposites: he could work hard and play hard, be serious and horse around, see the big picture and dive into details. He was as honest with his emotions as he was generous with his time; we would frequently commiserate with each other during the dour times of grad school but celebrate during the more frequent good times. Jiri, I'm glad you have put up with me for these past several years and there's a good chance we may work together again. Swapnil Patil and Dan Wentlandt were the first graduate students I met at CMU during visit day. Dan persuaded me both to be more social and to work with Dave; unfortunately only one of those really stuck as well as he might hope. Swapnil offered me a place to stay and showed me that the graduate student life wasn't as bad as they say. He always was willing to patiently listen to my bad ideas and occasionally help me turn them into much better ones. I have him to thank for exploring more of the wonderful coffee locations in Pittsburgh and I hope to do the same in other coffee cities around the world.

Jason Franklin was my officemate of the past two years, FAWN colleague, and certainly the most ambitious graduate student I've known. Always abreast of new restaurants to try and connoisseur of all things high quality, I'll miss our daily office conversations, though I hope to continue them in hacker coffee shops in San Francisco (or on your yacht when you've made it big). Milo Polte was my housemate during the last four years and made

coming home from work fun, whether it was cooking a vegetarian Bittman recipe, playing Team Fortress 2, or mixing together an amazing pre-prohibition era cocktail. Milo knew more about computer science, history, literature, politics, and philosophy than anyone else I knew and could wield his knowledge with razor sharp wit and timing. I am sure our paths will cross in the near future and I look forward to when that happens.

Amar Phanishayee can claim many firsts: the first prospective I met at CMU visit day, my first housemate, and the first co-author on a paper with me. I owe most of my positive experiences at CMU in no small part to his presence and interaction, and I feel bad that I learned a lot more from him than he did from me. I'll miss our late-night hacking sessions near Incast and FAWN deadlines and the long lunch discussions about how to change the world; with any luck, we'll get to work together soon and do exactly that. Speaking of world-changing events, you are lucky to have found Laura and I see a bright future for you both.

Jen, Parthi, Kevin and Beverly, I still have fond memories of our road trip in 2008 and I can't wait to do it again, maybe on the east coast this time. Jen: I always laugh thinking about those Wimbledon finals and how we ended up reading about the final result, even though you accidentally found out hours earlier. Parthi, our trip to Barcelona was full of strange events but we managed to survive and have a great time. Kevin frequently provided me opportunities to hang around Berkeley and San Francisco when I was visiting home, and I intend to do so even more now that I'll be back.

It is hard to put into words how much Beverly Wang has meant to me during these past few years. Her warm heart and caring defies what is possible to transmit from such a long distance. A day without her interaction never feels complete. I have changed significantly since college, and I can say without pause that I am a better person today because of her. With each passing day she becomes more special to me, and I hope there are many more days ahead.

My family has been a life-long inspiration and source of support and encouragement over the past several years. My mother was the voice of clarity when I needed to make tough decisions and she encouraged me to finish my dissertation when I felt like relaxing; I might not be done now if not for her. I was inspired by my brother Gautam to go into computer science so that I could make video games like him, though I inadvertently ended up completing a dissertation in distributed systems and networking like my father. As they say, the apple never falls far from the tree.

The absurd length of this acknowledgements section shows the degree of support and richness of environment I've had over the past several years. To all of you, named and unnamed, I graciously thank you and hope I have done you proud.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem and Scope

Energy has become an increasingly large financial burden and scaling challenge for computing. With the increasing demand for and scale of Data-Intensive Scalable Computing (DISC) [32], the power and cooling costs of running large data centers are a significant fraction of the total cost of their ownership, provisioning, and operation [24]. On a smaller scale, power and cooling are serious impediments to the achievable density in data centers [109]: companies frequently run out of power before they exhaust rack space.

Today's DISC systems are primarily designed to access large amounts of data stored on terabytes to petabytes of storage. Examples of DISC systems include those being built by Google, Microsoft, Yahoo!, Amazon.com, and many others. These systems often span the globe with multiple datacenters, each consisting of tens of thousands of individual servers built from commodity components. The peak power provisioned for each datacenter can reach tens of megawatts, which has motivated the deployment of datacenters near abundant access to cheap energy [72, 123].

Given the degree to which today's largest datacenters are affected by energy, we propose to reduce the energy consumed by large-scale computing by building datacenters using a FAWN: A Fast Array of Wimpy Nodes. FAWN is an approach to building datacenter clusters using low-cost, low-power hardware devices that are individually optimized for energy efficiency (in terms of work done per joule, or equivalently, performance per watt) rather than raw performance alone. The abundant parallelism found in many data-intensive workloads allows a FAWN system to use many more individually wimpier components in parallel to complete a task while reducing the overall energy used to do the work.

The FAWN approach balances the I/O gap between processing and storage, but chooses a specific balance that optimizes for energy efficiency. Over the last several years, we have built several different FAWN instantiations by using off-the-shelf hardware consisting of embedded/low-power processors paired with consumer-class flash storage, which have hardware characteristics that most existing datacenter software systems are unable to take full advantage of.

This dissertation describes how to adapt software to this new environment, and how doing so can significantly improve energy efficiency for a subset of datacenter workloads. Specifically, FAWN can improve energy efficiency for data-intensive workloads that are easily partitioned, parallelizable, and require computations (both simple and complex) across petabytes of data that tend to be more I/O-bound than CPU-bound on traditional brawny systems. In contrast, traditional HPC and transaction-processing systems perform complex computations and synchronization on small amounts of data for which FAWN is less effective and are workloads we do not consider.

1.2 Thesis Statement

This dissertation claims that *the FAWN approach can drastically improve energy efficiency for data-intensive computing systems, but achieving its potential requires revisiting and jointly changing the entire compute and storage stack.*

Successfully deploying FAWN requires more than simply choosing a different hardware platform; it requires the careful design and implementation of software techniques optimized for the different balance, constraints, and distributed system challenges resulting from FAWN system properties. FAWN nodes have a different balance of I/O capability to CPU and a reduced memory capacity per core. Because FAWN nodes are individually less capable, more individual nodes are required to complete the same amount of work. These different properties often prevent existing software from taking full advantage of the improved efficiency FAWN systems can provide, but we show that properly architecting software for FAWN systems can improve datacenter energy efficiency by an order of magnitude.

1.3 Thesis Overview

To substantiate the claim made in the thesis statement, this dissertation describes the design, implementation, and evaluation of several systems built specifically with FAWN clus-

ters in mind and analyzes why existing software can fail to attain the efficiency potential of FAWN. In Chapter 2, we introduce the background concepts and trends motivating the FAWN approach and the key unique combination of constraints of FAWN systems to which systems software must adapt. In Chapter 3, we describe the foundation of related work upon which this dissertation builds.

Chapter 4 describes our in-depth study into building a distributed key-value storage system on an early FAWN prototype system, demonstrating the benefits of re-thinking the design of the higher-level distributed system and local storage techniques targeted for the FAWN environment. In Chapter 5 we then provide a broader classification and workload analysis for other data-intensive workloads to understand when FAWN can be more energy-efficient and why software can fall short of capitalizing on its benefits.

Chapter 6 then delves more deeply into FAWN-KV running on modern FAWN hardware and state-of-the-art solid state devices, showing how a dramatic shift in balance between CPU and I/O highlights the need to re-think and re-design the lower-layer operating system implementations and interfaces to I/O. In Chapter 7, we propose the pervasive use of vector interfaces to storage and RPC systems as a technique to overcome this shrinking I/O gap. We then follow with a proposal for using vector interfaces throughout operating systems and applications in Chapter 8. Finally, we conclude with future work and a summary of the work in Chapter 9.

1.4 Contributions

This dissertation makes the following contributions:

- The principles and trends underpinning energy-efficient cluster design using a Fast Array of Wimpy Nodes;
- a capital and power cost analysis showing when FAWN systems are preferable for a given dataset size and query rate;
- the first design and implementation of a distributed key-value storage system on flash storage (FAWN-KV);
- the design for a memory-efficient hash table targeted towards wimpy processors and low flash read-amplification;

- an analysis of several classes of data-intensive workloads on wimpy and traditional systems demonstrating where and why FAWN can be more effective than traditional systems;
- measurements showing the deficiencies of the Linux I/O stack for modern high speed flash solid state drives and an interrupt mitigation scheme for the SATA driver to batch I/O completions;
- the design, implementation and evaluation of a key-value storage server architecture using vector interfaces to RPC and storage that provides 1.6M key-value lookups per second over the network;
- and the principles of applying vector interfaces to eliminate redundant execution in operating systems running on balanced systems.

The FAWN-KV software developed in this thesis as well as the interrupt mitigation SATA implementation are available for download at `http://github.com/vrv/`.

Chapter 2

Background Concepts and Trends

2.1 Datacenter Energy Efficiency

The tremendous growth of cloud computing and large-scale data analytics highlights the importance of reducing data center energy consumption. Cluster distributed systems consisting of tens to hundreds of thousands of machines are becoming more prevalent each year, and the financial burden imposed by datacenter power and cooling requirements increases the total cost of ownership for datacenters. At today's energy prices, the direct cost to power a datacenter server is a modest fraction (perhaps 10%) of the total cost of ownership (TCO) of the server [24], but the proportion of a server's TCO related to *total* energy use, such as cooling and infrastructure costs, has become high enough to warrant concern from the large companies that operate these datacenters. For example, assuming a cost of $10 for power and cooling infrastructure per Watt of delivered power [24], a 200W server would require $2000 of initial investment, equal to or higher than the cost of the server itself. If this cost is amortized over four server generations, power and cooling costs would comprise 20% of TCO.

At the warehouse computing scale (hundreds of thousands of machines in a single location), the components of overall energy use are widespread, ranging from the energy use of the server itself to its supporting infrastructure such as battery backup systems, power supplies, and higher-level energy management infrastructure. Because datacenter floor space is often an important cost concern, many datacenter designs increase the compute and storage density of their datacenter. But the density of the datacenters that house the machines has traditionally been limited by the ability to supply and cool 10–20 kW of power per rack and up to 10–20 MW per datacenter [72]. Datacenters today are designed with a maximum

power draw of 50 MW [72], or the equivalent of nearly 40,000 residential homes. Such high power density requires dedicated electrical substations to feed them, further driving up the datacenter TCO.

Datacenter lifetimes span multiple generations of server hardware to amortize the initial cost of construction, whereas the average server's lifetime is on the order of three to four years [24]. The infrastructure required to support a datacenter must plan for a *peak capacity* that projects for future growth over a long duration. The designed peak power draw of the datacenter then informs the design and subsequent cost of building the datacenter. Not surprisingly, building a datacenter to support fifteen years of growth requires both high upfront costs and complex, long-term business projections that can have significant impact on profitability. As a result, reducing peak power requirements can substantially reduce datacenter cost.

The peak power of a datacenter is determined by the aggregate power draw of all server components at full load. Assuming that the amount of work to be done in a datacenter is fixed, one way to reduce the peak power draw of a datacenter is by improving *energy efficiency*. We define energy efficiency as the amount of work done per Joule of energy, or equivalently, the performance per Watt. By improving the energy efficiency of a datacenter, we can reduce the amount of energy required to perform a defined amount of work.

2.2 FAWN: A Fast Array of Wimpy Nodes

To reduce peak power and its related costs, we introduce an approach to building clusters using low-power, low-speed nodes at large scale which we call **FAWN: A Fast Array of Wimpy Nodes** [17]. The central observation of this idea is that efficient data-intensive clusters must be both balanced in their CPU and I/O-capabilities (i.e., not wasting the resources of the CPU, memory, storage, or network), and also efficient in the amount of work done per Joule of energy, because balance alone does not necessarily imply energy efficiency.

A FAWN cluster is composed of more nodes than a traditional cluster because each FAWN node is individually slower. Our initial prototype FAWN node from 2007 used an embedded 500MHz processor paired with CompactFlash storage, which was significantly slower per-node than a multi-GHz multi-core server system balanced with multiple disks available then. Today, a FAWN node may use a small number of cores operating at 1.6GHz, or roughly half the clockspeed of high-speed servers.

A FAWN must use more wimpy nodes in parallel to replace each traditional, "brawny" system [64] to achieve the same level of performance. As a result, the FAWN approach

might fail to improve energy efficiency for workloads that cannot be easily parallelized or whose computation requires a serialized component (because of Amdahl's Law [12]). Fortunately, many (but not all [23, 64]) DISC workloads are parallelizable because of the data-oriented nature of the workloads. The FAWN approach can work well for these types of workloads, which we focus on throughout this dissertation.

2.3 Reducing Energy in Datacenters

There are several viable approaches to reducing energy consumption and costs in datacenters, FAWN being the focus in this dissertation. In this section, we highlight two other approaches and how they contrast and complement the FAWN approach.

Reducing PUE Over the last several years, many of the largest datacenter builders have largely focused on reducing a datacenter's *Power Usage Effectiveness*, or PUE. PUE is the ratio of total power draw to aggregate server power draw. In 2009, the Energy Star program estimated that the average datacenter PUE was 1.91–for every watt of power delivered to a server, the datacenter infrastructure required another 0.91 watts to deliver the power and remove the heat generated by the server [126].

While the industry average PUE remains relatively high, state-of-the-art datacenters have been built that reduce the PUE to about 1.1, so that only an additional 10% of power is used to deliver power to servers.[1] Providing this low of a PUE has required innovation in battery backup systems, efficient power supplies, voltage regulators, and novel cooling infrastructures. The EPA PUE study suggested that PUE was inversely correlated with size: the largest datacenter providers had the lowest PUE. Thus, if state-of-the-art techniques are employed, PUE can be reduced to a negligible fraction of TCO.

Moreover, improving PUE is orthogonal to reducing peak power, because improving PUE mostly requires focusing on the infrastructure surrounding the servers that do the real work. Less focus has been placed on the peak power draw of a datacenter, which still remains a major factor in determining the capital and operational costs of running a datacenter today.

Improving Proportionality Datacenters must be provisioned for peak power, but their *average* utilization is often far below peak—anywhere from 5% [77] for smaller datacen-

[1]The PUE metric does not capture additional efficiency losses when distributing the power to the individual components inside the server, which can add another 10 or 20% power overhead.

ters to 30% at the low end for larger datacenters [23]. Innovations in cluster scheduling and other techniques fueled by cost pressures have enabled state-of-the-art datacenters to operate at upwards of 70% of peak cluster utilization.

Ideally, a datacenter would use only a proportional fraction of power when not fully utilized (e.g., operating at 20% utilization should require only 20% of the cluster's peak power draw), a feature termed "energy proportionality" [23].

Unfortunately, individual servers have often not been energy proportional because of their high fixed power draw even when idle: servers can consume 20-50% of their peak power at 0% utilization. While server proportionality has improved, it is only a part of the equation: when considering the datacenter as a whole, one must factor in the energy proportionality of other components such as power supply, distribution, and cooling, which are also far from energy proportional [24].

Achieving energy proportionality in a datacenter thus may require "ensemble-level techniques" [128], such as turning portions of a datacenter off completely [13]. This can be challenging because workload variance in a datacenter can be quite high, and opportunities to go into deep sleep states are few and far between [23, 90], while "wake-up" or VM migration penalties can make these techniques less energy-efficient. Also, VM migration may not apply for some applications, e.g., if datasets are held entirely in DRAM to guarantee fast response times. Finally, companies may be loathe to turn expensive machines off completely because they fear increasing failure rates caused by repeated power-cycling, instead preferring to find additional tasks to occupy spare capacity.

As Figure 2.1 shows, improvements to hardware proportionality or ensemble-level techniques to do so should apply equally to FAWN. Improving energy proportionality is therefore a complementary approach to FAWN. In other words, energy proportionality techniques reduce the day-to-day operational cost of a datacenter, whereas FAWN's lower peak power requirements reduce initial capital cost and worst-case operational cost. We note that proportionality can additionally reduce peak power requirements by 30% for real datacenter workloads because not all machines will be simultaneously operating at full capacity [50].

2.4 FAWN Principles

To understand why FAWN can be fundamentally more energy-efficient for serving massive-scale I/O and data-intensive workloads, we explain how fundamental trends in computing and technology make FAWN the optimal choice for energy efficiency.

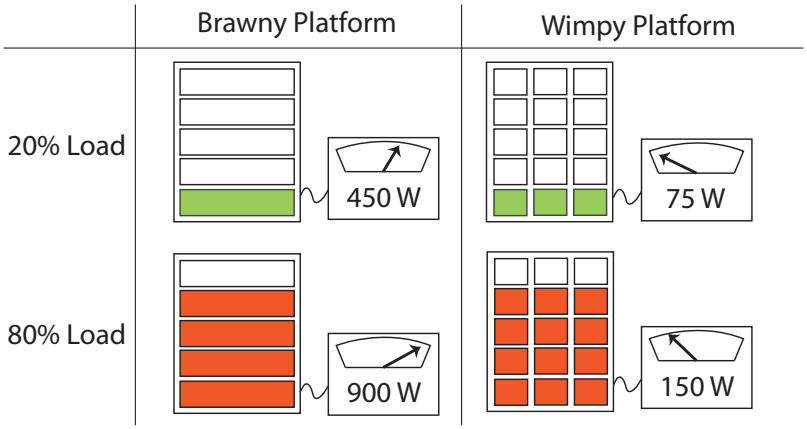


Figure 2.1: Energy proportionality is orthogonal to energy efficiency. Here, the wimpy platform is six times more efficient in performing the same amount of work regardless of load. This assumes that the proportionality of both platforms is identical, which may not be true in practice.

2.4.1 Defining Energy Efficiency

Before outlining FAWN trends and principles, we must first define how to measure energy efficiency. Several such definitions exist, but the one we focus on is "work done per Joule". For large-scale cluster computing applications that are consuming a significant fraction of energy in datacenters worldwide, "work done per Joule" is a useful metric to optimize: it relies on being able to parallelize workloads, which is often explicitly provided by data-intensive computing models such as MapReduce [42] and Dryad [67] that harness data-parallelism.

More specifically, when the amount of work is fixed but parallelizable, one can use a larger number of slower machines yet still finish the work in the same amount of time, e.g., ten nodes running at one-tenth the speed of a traditional node. If the aggregate power used by those ten nodes is less than that used by the traditional node, then the ten-node solution is more energy-efficient.

Other communities, such as in low-power VLSI design, have defined efficiency using the "energy-delay product," which multiplies the amount of energy to do an amount of work with the time it takes to do that amount of work. This penalizes solutions that reduce the amount of energy by reducing performance for energy efficiency gains. Our goal of meeting the same throughput or latency targets through increased cluster parallelism, however, also

optimizes for energy-delay product. Others have gone further by proposing using "energy $delay^2$" to further penalize solutions that simply reduce voltage at the expense of performance; FAWN does not work well in optimizing this latter metric.

One additional metric we do not study in detail is the cost of software development. As we will repeatedly show in this dissertation, software may not run well "out-of-the-box" on wimpy hardware for a number of reasons, requiring additional development time to either rewrite from scratch or tune/optimize appropriately. When calculating the cost of transitioning a portion of a cluster to the wimpy platform, energy costs, capital costs, and software development costs will all play a factor. For the purposes of narrowing the research, however, we focus only on energy efficiency and software design techniques that work well for balanced system designs, though it is likely that software development costs will necessarily work in favor of "brawnier" platforms [64] because they offer more computational headroom.

2.4.2 CPU Trends

FAWN is inspired by several fundamental trends in energy efficiency for CPUs, memory, and storage.

Increasing CPU-I/O Gap: Over the last several decades, the gap between CPU performance and I/O bandwidth has continually grown. For data-intensive computing workloads, storage, network, and memory bandwidth bottlenecks often cause low CPU utilization.

FAWN Approach: To efficiently run I/O-bound data-intensive, computationally simple applications, FAWN uses processors that are more energy efficient in instructions per Joule while maintaining relatively high performance. Typically, we target FAWN systems that operate at a third to half of the speed of the fastest available systems. This reduced processor speed then benefits from a second trend:

CPU power consumption grows super-linearly with speed. Operating processors at higher frequency requires more energy, and techniques to mask the CPU-memory bottleneck at higher speeds come at the cost of energy efficiency. CMOS transistor switching (dynamic) power can be calculated using the formula $P = CV^2f$, where C is the capacitance of the transistor, V is the operating voltage, and f is the frequency. According to this equation,

increasing frequency alone should only linearly increase power consumption[2], but high-speed processors have typically been designed to hide the latency between the processor and various levels of memory. Because CPUs have traditionally grown faster in speed than the memory hierarchy, operating at higher frequency did not provide real performance improvements unless the processor avoided memory stalls.

Techniques to avoid memory stalls, such as complex branch prediction logic, speculative execution, out-of-order execution and increasing the amount of on-chip caching, all require additional processor die area; modern processors dedicate as much as half their die to L2 and L3 caches [65]. These additional features on high-speed processors are designed to provide the processor with data, but they do not increase the speed of performing computations themselves. The additional die area contributes to static and dynamic power consumption and additional power circuitry, making faster CPUs less energy efficient. For example, the Intel Xeon 5080 operates 2 cores at 3.73GHz, contains 373 million transistors on the die and has a max thermal dissipated power of 135W. The Intel Atom Z500 operates a single 32-bit core at 800MHz, contains 47 million transistors, and has a max thermal dissipated power of just 0.65W.

FAWN Approach: A FAWN cluster's simpler CPUs dedicate more transistors to basic operations. These CPUs execute significantly more *instructions per Joule* than their faster counterparts (Figure 2.2): multi-GHz superscalar quad-core processors can execute up to 500 million instructions per Joule, assuming all cores are active and avoid stalls or mispredictions. Lower-frequency in-order CPUs, in contrast, can provide nearly 2 billion instructions per Joule—four times more efficient while still running at $1/3$rd the frequency.

Implications: FAWN systems therefore choose simpler processor designs whose single-core speed is close to those of low-end server processors; processors that are too slow can make software development difficult [64], and unavoidable fixed costs (e.g., power supply, I/O controllers, networking) can eliminate the benefits of extremely slow but energy-efficient processors.

[2]Similarly, decreasing frequency should only linearly decrease power consumption. A primary energy-saving benefit of Dynamic Voltage and Frequency Scaling (DVFS) for CPUs was its ability to reduce voltage as it reduced frequency [133] to see superlinear energy savings when operating more slowly, but retaining the capability to operate at full speed when needed. Unfortunately, modern CPUs already operate near minimum voltage at the highest frequencies, and various other factors (such as static power consumption and dynamic power range) have limited or erased many of the energy efficiency benefits of DVFS today [124]. In fact, today most processors operate most efficiently at either full capacity, or when completely idle due to processor C-states, motivating approaches that attempt to "race-to-idle" (`http://www.lesswatts.org`).

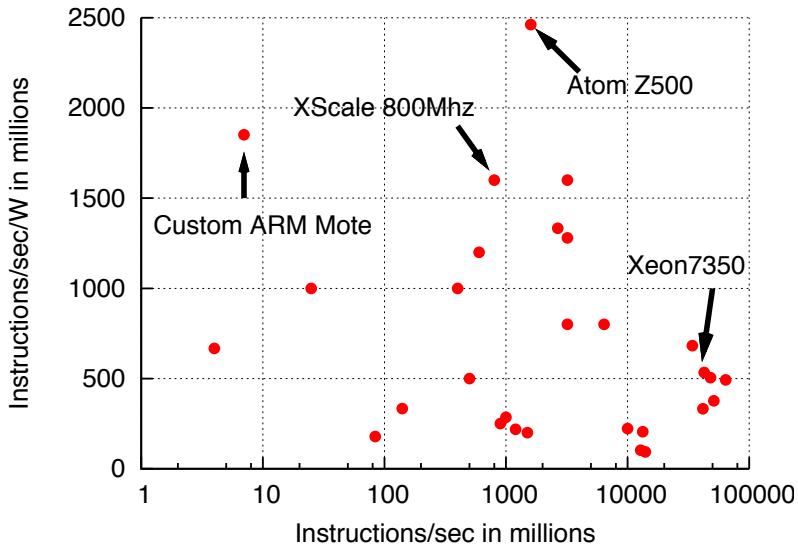


Figure 2.2: Max speed (MIPS) vs. Instruction efficiency (MIPS/W) in log-log scale. Numbers gathered from publicly-available spec sheets and manufacturer product websites.

2.4.3 Memory Trends

The previous section examined the trends that cause CPU power to increase drastically with an increase in sequential execution speed. In pursuit of a balanced system, one must ask the same question of memory as well.

Understanding DRAM power draw. DRAM has, at a high level, three major categories of power draw:

Idle/Refresh power draw: DRAM stores bits in capacitors; the charge in those capacitors leaks away and must be periodically refreshed (the act of reading the DRAM cells implicitly refreshes the contents). As a result, simply storing data in DRAM requires non-negligible power.

Precharge and read power: The power consumed inside the DRAM chip. When reading a few bits of data from DRAM, a larger line of cells is actually precharged and read by the sense amplifiers. As a result, random accesses to small amounts of data in DRAM are less energy-efficient than large sequential reads.

Memory bus power: A significant fraction of the total memory system power draw—perhaps up to 40%—is required for transmitting read data over the memory bus back to the CPU or DRAM controller.

Designers can somewhat improve the efficiency of DRAM (in bits read per Joule) by clocking it more slowly, for some of the same reasons mentioned for CPUs. In addition, both DRAM access latency and power grow with the distance between the CPU (or memory controller) and the DRAM: without additional amplifiers, latency increases quadratically with trace length, and power increases at least linearly.

This effect creates an intriguing tension for system designers: Increasing the amount of memory per CPU simultaneously increases the power cost to access a bit of data. To add more memory to a system, desktops and servers use a bus-based topology that can handle a larger number of DRAM chips; these buses have longer traces and lose signal with each additional tap. In contrast, the low-power DRAM used in embedded systems (cellphones, etc.), LPDDR, uses a point-to-point topology with shorter traces, limiting the number of memory chips that can be connected to a single CPU, but reducing substantially the power needed to access that memory.

Implications: Energy-efficient wimpy systems are therefore likely to contain *less memory per core* than comparable brawny systems. Further exacerbating this challenge is that each FAWN node in a cluster will often have duplicated data structures held in memory, such as operating system and application data structures that must be replicated. If splitting a unit of work into smaller units does not proportionally reduce the amount of memory required to process that work, the *usable* memory of each FAWN node shrinks even further in comparison to a brawny platform.

As we show throughout this work, programming for FAWN nodes therefore requires careful attention to memory use, and reduces the likelihood that traditional software systems will work well on FAWN systems out of the box.

2.4.4 Storage Power Trends

The energy draw of magnetic platter-based storage is related to several device characteristics, such as storage bit density, capacity, throughput, and latency. Spinning the platter at faster speeds will improve throughput and seek times, but requires more power because of the additional rotational energy and air resistance. Capacity increases follow bit density improvements and also increase with larger platter sizes, but air resistance increases quadratically with larger platter sizes, so larger platters also require more power to operate.

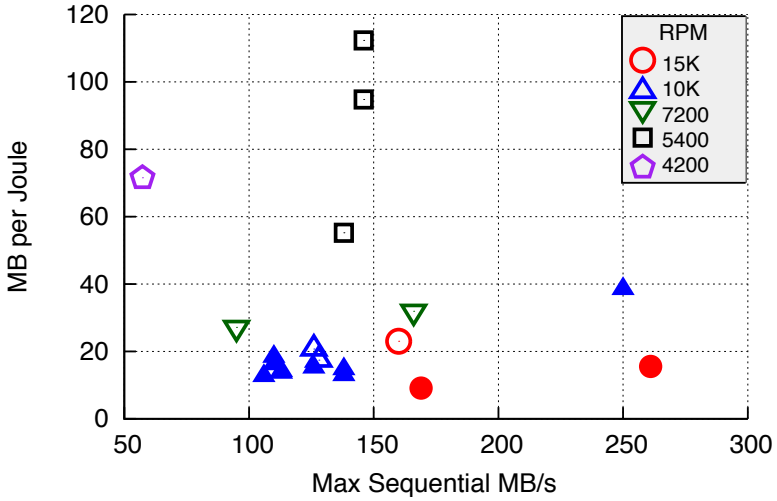


Figure 2.3: Power increases with rotational speed and platter size. **Solid shapes are 3.5" disks and outlines are 2.5" disks.** Speed and power numbers acquired from product specification sheets.

Figure 2.3 demonstrates this tradeoff by plotting the efficiency versus speed for several modern hard drives, including enterprise, mobile, desktop, and "Green" products.[3]

The fastest drives spin at between 10-15K RPM, but they have a relatively low energy efficiency as measured by MB per Joule of max sustained sequential data transfer. The 2.5" disk drives are nearly always more energy efficient than the 3.5" disk drives. The most efficient drives are 2.5" disk drives running at 5400 RPM. Energy efficiency therefore comes at the cost of per-device storage capacity for magnetic hard drives. Our preliminary investigations into flash storage power trends indicate that the number of IOPS provided by the device scales roughly linearly with the power consumed by the device, likely because these devices increase performance through chip parallelism instead of by increasing the speed of a single component. Table 2.1 summarizes the tradeoff between hard disk size, speed, and comparison to flash technology.

Implications: Energy-efficient clusters constrained by storage capacity requirements will continue to use 2.5" disk drives because they provide the lowest cost per bit, but flash storage

[3]The figure uses sequential throughput numbers from vendor spec sheets in 2010, which are often best-case outer-track numbers. The absolute numbers are therefore somewhat higher than what one would expect in typical use, but the relative performance comparison is likely accurate.

Relation to	Platter Radius (R)	Rotational Speed (S)	Advantage (Disk vs. Flash)
Energy per Bit*	$\sim R^5$	$\sim S^3$	Disk
Capacity	R^2	–	Disk
Max Throughput	R	S	Flash
Mean Access Time	R	$1/S$	Flash

Table 2.1: Hard disk power relationships. *Drag forces increase power proportional to $velocity^3$ and area, so energy-per-bit increases by approximately R^5 and S^3.

will continue to make in-roads in the datacenter, particularly for the remote small object retrieval systems that many large services rely on today. Our work on FAWN focuses mostly on pairing wimpy platforms with flash storage and other non-volatile memories, but we do advocate using efficient magnetic disks when storage capacity is the leading cost [17].

2.4.5 Fixed Power Costs

Non-CPU components such as memory, motherboards, and power supplies have begun to dominate energy consumption [23], requiring that all components be scaled back with demand. As a result, running a modern system at 20% of its capacity may still consume over 50% of its peak power [128]. Despite improved power scaling technology, entire systems remain most energy-efficient when operating at peak utilization. Given the difficulty of scaling all system components, we must therefore consider "constant factors" for power when calculating a system's instruction efficiency. Figure 2.4 plots the same data from Figure 2.2 but adds a fixed 0.1W cost for system components such as Ethernet. Because powering even 10Mbps Ethernet dwarfs the power consumption of the tiny sensor-type processors that consume only micro-Watts of power, total system efficiency drops significantly. The best operating point exists in the middle of the curve, where the fixed costs are amortized while still providing energy efficiency.

Low-power processors are often more energy proportional due to reduced static power leakage, but if they surrounded by components that are not proportional or efficient, their whole system efficiency can suffer. In Chapter 5.1.2, we provide a real world benchmark example of how fixed power costs can significantly diminish the efficiency benefits of having an efficient processor, particularly at low utilization.

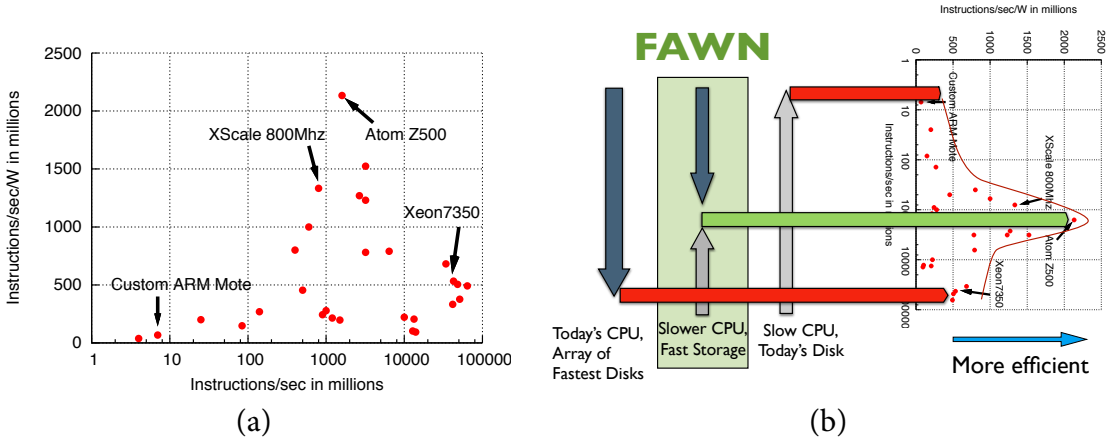


Figure 2.4: (a) Processor efficiency when adding fixed 0.1W system overhead. (b) A FAWN system chooses the point in the curve where each individual node is balanced and efficient.

2.4.6 System Balance

Balanced systems ensure that provisioned resources are not wasted [111], but the envelope of the points in Figure 2.4a illustrates that balance alone does not imply efficiency: Even a balanced system can be inefficient if the amount of energy to operate at a higher speed is disproportionately high.

Figure 2.4b takes the speed vs. efficiency graph for processors in Figure 2.4a and shows where several different "balanced" systems operate in the curve. The FAWN approach chooses a cluster configuration where each node is individually optimized for energy efficiency, focusing on finding both a balanced and efficient point in the curve.

While the specific point on the curve that optimizes for energy efficiency will change over time, the trends described previously suggest that the general shape of the curve should hold for the foreseeable future, with expected shifts and continued improvements in overall efficiency [75]. Other individual components that require a superlinear increase in power to increase speed will reduce efficiency for faster systems, while higher fixed power costs will push the optimal point towards brawnier systems.

2.5 Analysis of FAWN Total Cost of Ownership

Improving energy efficiency at a significantly higher *total* cost would erase the efficiency benefits FAWN can provide. To address this, here we provide an analytical, back-of-the-envelope calculation to better understand when the FAWN approach is likely to cost less than traditional architectures. We examine this question in detail by comparing the three-year total cost of ownership (TCO) for six systems: three "traditional" servers using magnetic disks, flash SSDs, and DRAM; and three hypothetical FAWN-like systems using the same storage technologies. We define the 3-year total cost of ownership (TCO) as the sum of the capital cost and the 3-year power cost at 10 cents per kWh.

We study a theoretical 2009 FAWN node using a low-power CPU that consumes 10–20 W and costs ∼$150 in volume. We in turn give the benefit of the doubt to the server systems we compare against—we assume a 2 TB disk exists that serves 300 queries/sec at 10 W.

Our results indicate that both FAWN and traditional systems have their place—but for the small random access workloads we study, traditional systems are surprisingly absent from much of the solution space, in favor of FAWN nodes using either disks, flash, or DRAM.

Key to the analysis is a question: *why does a cluster need nodes?* The answer is, of course, for both storage space and query rate. Storing a DS gigabyte dataset supporting a query rate QR requires N nodes:

$$N = max\left(\frac{DS}{\frac{gb}{node}}, \frac{QR}{\frac{qr}{node}}\right)$$

For large datasets with low query rates, the number of nodes required is dominated by the storage capacity per node: thus, the important metric is the total cost per GB for an individual node. Conversely, for small datasets with high query rates, the per-node query capacity dictates the number of nodes: the dominant metric is queries per second per dollar. Between these extremes, systems must provide the best tradeoff between per-node storage capacity, query rate, and power cost.

Table 2.2 shows these cost and speculative performance statistics for several candidate systems circa 2009; while the numbers are outdated, the general trends should still apply. The "traditional" nodes use 200 W servers that cost $1,000 each. *Traditional+Disk* pairs a single server with five 2 TB high-speed (10,000 RPM) disks capable of 300 queries/sec, each disk consuming 10 W. *Traditional+SSD* uses two PCI-E Fusion-IO 80 GB flash SSDs, each also consuming about 10 W (Cost: $3k). *Traditional+DRAM* uses eight 8 GB server-quality

System	Cost	W	QPS	$\frac{\text{Queries}}{\text{Joule}}$	$\frac{\text{GB}}{\text{Watt}}$	$\frac{\text{TCO}}{\text{GB}}$	$\frac{\text{TCO}}{\text{QPS}}$
Traditionals:							
5-2TB Disks	$2K	250	1500	6	40	0.26	1.77
160GB PCIe SSD	$8K	220	200K	909	0.72	53	0.04
64GB DRAM	$3K	280	1M	3.5K	0.23	59	0.004
FAWNs:							
2TB Disk	$350	20	250	12.5	100	0.20	1.61
32GB SSD	$500	15	35K	2.3K	2.1	16.9	0.015
2GB DRAM	$250	15	100K	6.6K	0.13	134	0.003

Table 2.2: Traditional and FAWN node statistics circa 2009.

DRAM modules, each consuming 10 W. *FAWN+Disk* nodes use one 2 TB 7200 RPM disk: FAWN nodes have fewer connectors available on the board. *FAWN+SSD* uses one 32 GB Intel SATA flash SSD capable of 35,000 random reads/sec [102] and consuming 2 W ($400). *FAWN+DRAM* uses a single 2 GB, slower DRAM module, also consuming 2 W.

Figure 2.5 shows which base system has the lowest cost for a particular dataset size and query rate, with dataset sizes between 100 GB and 10 PB and query rates between 100 K and 1 billion per second.

Large Datasets, Low Query Rates: *FAWN+Disk* has the lowest total cost per GB. While not shown on our graph, a traditional system wins for exabyte-sized workloads if it can be configured with sufficient disks per node (over 50), though packing 50 disks per machine poses reliability challenges.

Small Datasets, High Query Rates: *FAWN + DRAM* costs the fewest dollars per query rate, keeping in mind that we do *not* examine workloads that fit entirely in L2 cache on a traditional node. We assume the FAWN nodes can only be configured with 2 GB of DRAM per node, so for larger datasets, a traditional DRAM system provides a high query rate and requires fewer nodes to store the same amount of data (64 GB vs 2 GB per node).

Middle Range: *FAWN+SSD*s provide the best balance of storage capacity, query rate, and total cost. If SSD cost per GB improves relative to magnetic disks, this combination is likely to continue expanding into the range served by *FAWN+Disk*; if the SSD cost per performance ratio improves relative to DRAM, so will it reach into DRAM territory. It is therefore conceivable that *FAWN+SSD* could become the dominant architecture for many random-access workloads.

Are traditional systems obsolete? We emphasize that this analysis applies only to small, random access workloads. Sequential-read workloads are similar, but the constants de-

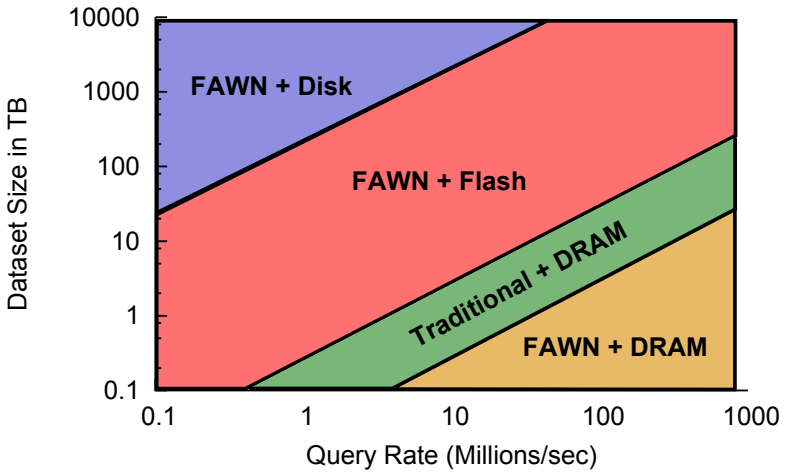


Figure 2.5: Solution space for lowest 3-year TCO as a function of dataset size and query rate.

pend strongly on the per-byte processing required. Traditional cluster architectures may retain a place for some CPU-bound and memory-hungry workloads, but we do note that architectures such as IBM's BlueGene successfully apply large numbers of low-power, efficient processors to many supercomputing applications—but they augment their wimpy processors with custom floating point units to do so.

Our definition of "total cost of ownership" ignores several important costs: In comparison to traditional architectures, FAWN should reduce power and cooling infrastructure costs, but may increase network-related hardware and power costs due to the need for more switches. In the above analysis, the direct cost of energy contributes roughly 10% to TCO and therefore reflects heavily upon the reduced capital cost of commercially-available FAWN nodes rather than direct energy savings. The reduction of peak power by roughly a factor of 3–4, however, would significantly reduce the energy infrastructure costs, which comprise perhaps another 20% of the TCO.

Our current hardware prototype improves work done per volume, thus reducing costs associated with datacenter rack or floor space. Finally, our analysis assumes that cluster software developers can engineer away the human costs of management—an optimistic assumption for all architectures. We similarly ignore issues such as ease of programming, though we selected an x86-based wimpy platform for ease of development.

A full, contemporary analysis of TCO would produce a different result influenced by external market factors, current costs of compute and storage technologies, additional software developer cost, and other non-technology related costs such as energy generation and real estate prices. The above analysis simply attempts to tease out the relevant factors and design choices available to system designers when choosing node architectures and how metrics such as query performance per watt and gigabytes of storage per dollar factor into the optimal node choice for a random-access key-value cluster.

Chapter 3

Related Work

This thesis work touches on numerous topics in systems, architecture, and networking and builds upon a broad set of research spanning topics from balanced computing, energy efficient architecture, distributed storage systems and databases, and uses of non-volatile memory technologies. In this section we summarize the prior work that this dissertation builds upon.

3.1 Balanced Computing

A cluster with balanced systems contains nodes that use most of their capability. An unbalanced system dedicates too many resources to speeding up one component of a system when other components are a bottleneck for a given workload. All systems inevitably have some bottleneck, but the large degree to which I/O-bound workloads running on high-speed traditional systems are out of balance is one motivation for finding better system balance. In this light, FAWN follows in a long tradition of ensuring that systems are balanced in the presence of scaling challenges and of designing systems to cope with the performance challenges created by hardware architectures.

JouleSort [111] was an important first step in identifying the imbalance of traditional systems for I/O-bound workloads, focusing on large-scale sort. Its authors developed a SATA disk-based "balanced" system coupled with a low-power (34 W) CPU that significantly out-performed prior systems in terms of records sorted per joule. As we show in Chapter 5, developing a balanced system for different sized sorts can produce very different outcomes, and that the *energy-efficient* optimal system balance has recently shifted

significantly with the growing popularity and affordability of flash disks combined with innovations in low-power processor design and manufacturing.

Over the last five years, there have been several different proposals for using low-power processors for datacenter workloads to reduce energy consumption [33, 39, 47, 59, 84, 93, 127]. Next, we describe some of these proposals and discuss how this thesis builds upon or differs from this broad set of work.

In late 2007, Adrian Cockcroft from Netflix, Inc. described the concept of "Millicomputing" [39], or building clusters using computers that consume less than 1 Watt for highly-partitionable workloads. His proposal argued for the use of mobile or embedded processors, such as ARM architecture or system-on-chip processors from Freescale, Marvell and Gumstix, combined with microSDHC flash. One focus of Millicomputing was on packaging: how do you build individual millicomputers and how do you assemble them together into a cluster? Our contemporaneous initial work on FAWN explored real infrastructure uses of embedded processors combined with flash, focusing on key-value storage. We discovered that finding the right balance of hardware for a given workload was non-trivial. For example, when developing for the Gumstix platform, we found that it only supported PIO mode access to the flash interface and therefore could not sustain more than 50 random IOPS from a flash device capable of thousands of IOPS. Cockcroft also described software challenges from having nodes with with only 128–256MB of DRAM. The work described in Chapter 4 uses the best off-the-shelf hardware we could find after a several month search in 2007 and 2008, and further highlights how software must be redesigned to fully take advantage of balanced hardware. Furthermore, this thesis covers specific examples where a difference in DRAM sizes produces a non-linear shift in efficiency depending on working set sizes and per-node memory capacity, and contributes some techniques to reducing memory overhead for key-value storage indexing.

The Gordon [33] hardware cluster architecture pairs an array of flash chips and DRAM with low-power CPUs for low-power data intensive computing. The authors provided a simulation-based analysis of the design space for finding the most efficient per-node configuration for a few different workloads, finding that combining flash with low-power nodes like the Intel Atom could be significantly more energy efficient in terms of performance per watt than an Intel Core2 based system. A primary focus of their work was on developing a Flash Translation Layer (FTL) suitable for pairing a single CPU with several raw flash chips, and simulations on general system traces indicated that this pairing could improve energy efficiency by up to a factor of 2.5. FAWN similarly leverages commodity low-power CPUs and flash storage, but focuses on a cluster key-value application, and this thesis contributes software modifications necessary to achieve the potential of the underlying hardware and enabling good performance on flash regardless of FTL implementation.

CEMS [59], AmdahlBlades [127], and Microblades [84] also leverage low-cost, low-power commodity components as a building block for datacenter systems, similarly arguing that this architecture can provide the highest work done per dollar and work done per Joule using simulations, benchmarks, and analysis. Lim et al. provide a quantitative analysis isolating processor choice from other system choices, finding that the low-cost low-power designs can be the best choice for a variety of warehouse computing workloads and describe options to package and cool individual nodes into a cluster, similar in spirit to (but different in outcome from) the Millicomputing architecture.

Microsoft began exploring the use of a large cluster of low-power systems called Marlowe [93], focusing on the very low-power sleep states provided by the Intel Atom chipset (between 2–4 W) to turn off machines and migrate workloads during idle periods and low utilization, initially targeting the Hotmail service.

In the last two years, commercial "wimpy node" products have emerged with significant adoption. Several ultra-low power server systems have become commercially available, with companies such as SeaMicro, Marvell, Calxeda, and ZT Systems producing low-power datacenter computing systems based on Intel Atom and ARM platforms. SeaMicro, for example, has produced a 768-node Intel Atom 1.66GHz server in use (as of 2011) by services such as The Mozilla Foundation for their web hosting needs and eHarmony for MapReduce [42] analysis using Hadoop [8]. Tilera, a producer of custom 100-core processors, has worked with Facebook engineers to demonstrate that using low-power, low-speed (866MHz) many-core systems can significantly improve energy efficiency for use in `memcached` clusters [27].

Considerable prior work has examined ways to tackle the "memory wall." The Intelligent RAM (IRAM) project combined CPUs and memory into a single unit, with a particular focus on energy efficiency [29]. An IRAM-based CPU could use a quarter of the power of a conventional system to serve the same workload, reducing total system energy consumption to 40%. Early IRAM systems placed an array of CPUs with an array of DRAM chips on a single die [74] specifically for massively-parallel embedded systems, noting that programming their message-oriented I/O model "requires significant software involvement". These designs hark back to the early Transputer [136] and ZMOB [78] proposals for system-on-chip and array processors, respectively, each focusing on the microarchitectural ways these processors are composed together. FAWN takes a thematically similar view—placing smaller processors very near flash—but with a significantly different realization, one that uses more loosely coupled systems (though the cost benefits of tight integration are numerous [5, 117]). Similar efforts, such as the Active Disk project [110], focused on harnessing computation close to disks. Schlosser et al. proposed obtaining similar benefits from coupling MEMS-based storage systems with CPUs [115].

3.2 Flash in Databases and Filesystems

Much prior work has examined the use of flash in databases, such as how database data structures and algorithms can be modified to account for flash storage strengths and weaknesses [81, 82, 96, 98, 129]. Some of this work has concluded that NAND flash might be appropriate in "read-mostly, transaction-like workloads", but that flash was a poor fit for high-update-rate databases [96]. This earlier work, along with FlashDB [98] and FD-Trees [82], like ours also noted the benefits of a log structure on flash; however, in their environments, using a log-structured approach slowed query performance by an unacceptable degree. Prior work in sensor networks [40, 87] has employed flash in resource-constrained sensor applications to provide energy-efficient filesystems and single node object stores.

Several filesystems are specialized for use on flash. Most are partially log-structured [113], such as the popular JFFS2 (Journaling Flash File System) for Linux. Our observations about flash's performance characteristics follow a long line of research [49, 96, 98, 102, 138]. Past solutions to these problems include the eNVy filesystem's use of battery-backed SRAM to buffer copy-on-write log updates for high performance [137], followed closely by purely flash-based log-structured filesystems [73]. Others have explored ways to expose flash to applications using memory-like interfaces but internally using log-structured techniques [22].

3.3 High-throughput Distributed Storage and Analysis

Recent work such as Hadoop or MapReduce [42] running on GFS [57] has examined techniques for scalable, high-throughput computing on massive datasets. More specialized examples include SQL-centric options such as the massively parallel data-mining appliances from Netezza [99], AsterData [4], and others [1, 3, 9].

Designing software for FAWN shares common traits with designing software for distributed storage and analysis appliances. Developers can spend time optimizing software layers to take advantage of specific hardware configurations. The constraints of FAWN nodes require the use of memory-efficient data structures and algorithms and specialized use of flash devices, topics which this thesis explores with specific hardware and workloads in mind. Some of these techniques would apply equally well to other appliances such as WAN accelerators [14] and deduplication [43].

Related cluster and wide-area hash table-like services include Distributed data structure (DDS) [58], a persistent data management layer designed to simplify cluster-based Internet services. Myriad distributed key-value storage systems such as memcached [91], Dy-

namo [46], Voldemort [103], Cassandra [6], and Riak [2] have been developed over the last several years. Our work in Chapter 4 is motivated by systems like memcached, Dynamo and Voldemort, while Riak's BitCask storage system uses the log-structured key-value technique we proposed in our original FAWN-KV work [17]. Our key-value work throughout the dissertation focuses mostly on identifying ways that software can fully take advantage of the balanced hardware it runs on, whereas systems like Dynamo target eventually-consistent, wide-area environments whose goal is to maintain high-availability for writes. Systems such as Boxwood [85] focus on the higher level primitives necessary for managing storage clusters. Our focus in this area is on the lower-layer architectural and data-storage functionality.

3.4 "Low and slow" High Performance Computing Systems

High performance computing clusters such as IBM's BlueGene place thousands of compute nodes in a single facility, an environment where "low-power design is the key enabler" [55]. Their need for high performance, energy efficiency, and reliability for supercomputing workloads required a high degree of system integration. The BlueGene/L system turned to a system-on-chip (SoC) design using a low-power embedded PowerPC core. These cores are connected using a network specialized for supercomputing workloads: they contain hardware support for small messages and collective communication patterns common in these environments.

The FAWN approach conceptually extends the low-power core design of modern supercomputers to warehouse-scale, data-intensive systems and focuses on workloads more commonly found in datacenters than supercomputing environments. We note, however, that many of the same ideas apply to both environments, and that recent work has proposed using the BlueGene architecture for datacenters [19].

3.5 Energy Proportionality

Barroso and Hölzle have rallied the industry for improvements in energy proportionality (Section 2.3) for compute systems [23]. Energy efficiency and energy proportionality are orthogonal concepts, but both are desired in "heterogeneous" or "asymmetric" core designs [51, 94]. Asymmetric processors pair a few high performance cores with many more low-performance cores to provide a tunable knob to trade between performance and efficiency. Systems using asymmetric cores can use the low-performance, high-efficiency cores

at low load and the high-performance cores when computational performance is needed. Alternatively, sequential parts of code can be executed using the high-speed cores and parallel parts can be executed using the efficient slower cores. Research in this area primarily focuses on scheduling work across these cores to optimize for performance, power, or to remain within a particular power profile.

Unfortunately, CPUs are only one consumer of energy within an entire system, and similar techniques to trade lower performance for higher efficiency need to apply to the memory, storage, and power supply components of a system. For example, recent work has identified that idle low-power modes for cores achieve proportionality for the CPU, but similar mechanisms need to exist for shared caches and memory components [90]. This work also showed that for a particular datacenter workload, batching queries to create periods of full system idleness (which could then use PowerNap [89] to save energy) did not provide an effective power vs. latency tradeoff. In contrast, Chapter 7 finds that batching *similar* work together can improve overall system efficiency if appropriate system interfaces are provided, though its benefits stem from eliminating redundant work rather than exploiting periods of idleness.

Challen and Hempstead recently proposed the idea of power-agile systems, heterogeneous devices that combine multiple types of processors, memory, and storage systems each offering a different optimal power-envelope. These systems can then be dynamically configured based on workload characteristics to choose the optimal types of each subsystem to improve whole system efficiency [36].

A final set of research in this area examines the storage system: how and when to put disks to sleep. We believe that the FAWN approach complements them well. Hibernator [139], for instance, focuses on large but low-rate OLTP database workloads (a few hundred queries/sec). Ganesh et al. proposed using a log-structured filesystem so that a striping system could perfectly predict which disks must be awake for writing [54]. Finally, Pergamum [122] used nodes much like our wimpy nodes to attach to spun-down disks for archival storage purposes, noting that the wimpy nodes consume much less power when asleep. The system achieved low power, though its throughput was limited by the wimpy nodes' Ethernet.

3.6 FAWN Workload Studies

Several recent studies have explored applying the FAWN approach to different workloads [37, 68, 79]. For example, Lang et al. [79] studied using wimpy nodes for database workloads, noting that the serialized components of many query-processing workloads, or the lack of

scale out, make FAWN systems less energy-efficient than their brawny counterparts. As we articulated in the previous section, the FAWN approach willingly trades sequential performance for efficiency, and existing database workloads with a large serial component are not well-suited to the FAWN approach.

Reddi et al. [68] studied the use of FAWN systems for web search workloads (which include CPU-intensive components found in machine learning algorithms), demonstrating that search can be up to five times more efficient using Intel Atoms compared to Intel Xeons but suffers from poor QoS guarantees under overload situations. This study also argues for several microarchitectural changes to the design of the wimpy cores, such as increasing the size of the cache hierarchy, that can substantially improve efficiency without increasing cost or power.

Similar to the work presented in this dissertation, Facebook and Tilera's study of a many-core key-value system using low-power cores shows that with software modifications to the memcached layer, a 512-core 866MHz Tilera system could improve efficiency by a factor of three over a server-class system. Their workload is similar to the FAWN-KV work presented in this dissertation, but differs in their lack of data persistence, avoiding the traditional I/O storage stack because their requests are served only from DRAM. Nonetheless, their work corroborates our thesis statement that changes to software are required to make the best use of the FAWN approach, and that doing so can substantially improve energy efficiency.

Chapter 4

FAWN-KV: A Distributed Key-value Store for FAWN and Flash

The FAWN-KV distributed key-value storage system is one we designed and implemented to help answer the question: How does a FAWN approach change the way distributed systems software is built? Specifically, how do the constraints the FAWN approach imposes on system architects require them to rethink and redesign the infrastructure software running on this platform?

To look at a concrete, large-scale data-intensive application, we focus our attention first on high-performance key-value storage systems, which are growing in both size and importance; they now are critical parts of major Internet services such as Amazon (Dynamo [46]), LinkedIn (Voldemort [103]), and Facebook (`memcached` [91]).

The workloads these systems support share several characteristics: they are I/O, not computation, intensive, requiring random access over large datasets; they are massively parallel, with thousands of concurrent, mostly-independent operations; their high load requires large clusters to support them; and the size of objects stored is typically small, e.g., 1 KB values for thumbnail images, 100s of bytes for wall feed or stream posts, Twitter messages, etc.

The clusters that serve these workloads must provide both high performance and low cost operation. Unfortunately, small-object random-access workloads are particularly ill-served by conventional disk-based or memory-based clusters. The poor seek performance of disks makes disk-based systems inefficient in terms of both performance and performance per watt. High performance DRAM-based clusters, storing terabytes or petabytes of

data, are both expensive and consume a surprising amount of power—two 2 GB DIMMs consume as much energy as a 1 TB disk.

The workloads for which key-value systems are built for are often both random I/O-bound and embarrassingly parallel—the lowest hanging fruit and most applicable target for FAWN. We therefore choose this small-object, random-access workload as the first distributed system built using a FAWN architecture. For this workload, we pair low-power, efficient embedded CPUs with flash storage to provide efficient, fast, and cost-effective access to large, random-access data. Flash is significantly faster than disk, much cheaper than the equivalent amount of DRAM, and consumes less power than either.

Several key constraints make deploying existing software on this type of FAWN hardware challenging:

1. Individual FAWN nodes have a lower memory capacity per core. Our 2007-era FAWN nodes contain only 256MB of DRAM per node, an order of magnitude less DRAM than a traditional server configuration from the same era.

2. Flash is relatively poor for small random writes because flash technology cannot update in place and must perform block erasures and rewrite existing data along with new updates.

3. More nodes in a FAWN cluster leads to more frequent failures than a traditional cluster with fewer nodes. FAWN software designs must therefore accommodate this higher rate of failure with efficient failover and replication mechanisms.

FAWN-KV is a system designed to deal with these challenges by conserving memory use, avoiding random writes on flash and using most of the available I/O capability of each individual wimpy node, and together harnessing the aggregate performance of each node while being robust to individual node failures. In the next section, we detail the design and implementation of various components within FAWN-KV, including our memory-efficient in-memory hash index, our log-structured key-value local storage system called FAWN-DS, and our local storage software mechanisms for restoring replication of data efficiently on failures and arrivals into the system.

4.1 Design and Implementation

Figure 4.1 gives an overview of the entire FAWN-KV system. Client requests enter the system at one of several *front-ends*. The front-end nodes forward the request to the *back-end* FAWN-KV node responsible for serving that particular key. The back-end node serves

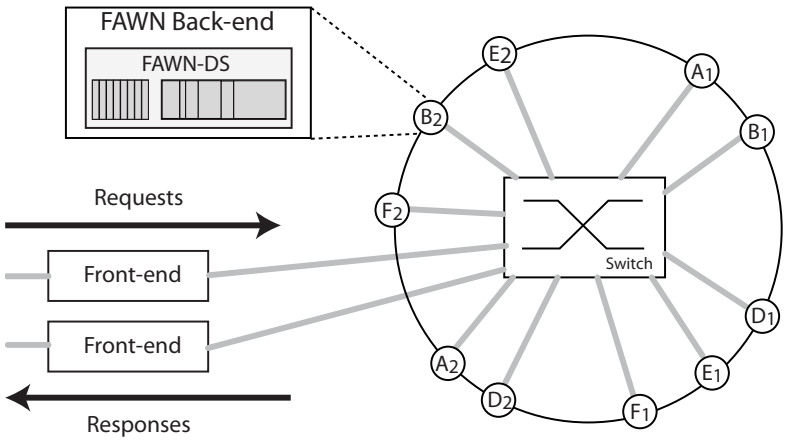


Figure 4.1: FAWN-KV Architecture.

the request from its FAWN-DS datastore and returns the result to the front-end (which in turn replies to the client). Writes proceed similarly.

The large number of back-end FAWN-KV storage nodes are organized into a ring using consistent hashing. As in systems such as Chord [121], keys are mapped to the node that follows the key in the ring (its *successor*). To balance load and reduce failover times, each *physical node* joins the ring as a small number (V) of *virtual nodes*, each virtual node representing a *virtual ID* ("*VID*") in the ring space. Each physical node is thus responsible for V different (non-contiguous) key ranges. The data associated with each virtual ID is stored on flash using FAWN-DS.

We describe the design and implementation details of the relevant FAWN-KV's system components from the bottom up: We begin with a brief overview of flash storage (Section 4.1.1), and then describe the per-node FAWN-DS datastore (Section 4.1.2) and its internal and external interface and design.

4.1.1 Understanding Flash Storage

Flash provides a non-volatile memory store with several significant benefits over typical magnetic hard disks for random-access, read-intensive workloads—but it also introduces several challenges. Three characteristics of flash underlie the design of the FAWN-DS system described throughout this section:

1. **Fast random reads**: ($\ll$ 1 ms), up to 175 times faster than random reads on magnetic disk [96, 102].

2. **Efficient I/O**: Many flash devices consume less than one Watt even under heavy load, whereas mechanical disks can consume over 10 W at load. Flash alone is over two orders of magnitude more efficient than mechanical disks in terms of random seeks per Joule.

3. **Slow random writes**: Small writes on flash are expensive. At the flash chip level, updating a single page requires first erasing an entire erase block (128 KB–256 KB) of pages, and then writing the modified block in its entirety. As a result, updating a single byte of data is as expensive as writing an entire block of pages [98].

Modern devices improve random write performance using write buffering and preemptive block erasure implemented in the device firmware and Flash Translation Layer. These techniques typically improve performance for short bursts of random writes, but recent studies show that sustained random writes still perform poorly on these devices [102].

These performance problems have motivated log-structured techniques for flash filesystems and data structures [69, 97, 98]. These same considerations inform the design of FAWN's node storage management system, described next.

4.1.2 FAWN-DS: The FAWN Data Store

FAWN-DS is a log-structured key-value store. Each store contains values for the key range associated with one virtual ID. It acts to clients like a disk-based hash table that supports `Store`, `Lookup`, and `Delete`.[1]

FAWN-DS is designed specifically to perform well on flash storage and to operate within the constrained DRAM available on wimpy nodes: all writes to the datastore are sequential, and reads typically require a single random access. To provide this property, FAWN-DS maintains an in-DRAM hash table (Hash Index) that maps keys to an offset in the append-only Data Log on flash (Figure 4.3a). This log-structured design is similar to several append-only filesystems [57, 107], which avoid random seeks on magnetic disks for writes.

[1]We differentiate datastore from database to emphasize that we do not provide a transactional or relational interface.

```
/* KEY = 0x93df7317294b99e3e049, 16 index bits */
INDEX = KEY & 0xffff; /* = 0xe049; */
KEYFRAG = (KEY >> 16) & 0x7fff; /* = 0x19e3; */
for i = 0 to NUM_HASHES do
  bucket = hash[i](INDEX);
  if bucket.valid && bucket.keyfrag==KEYFRAG &&
    readKey(bucket.offset)==KEY then
    return bucket;
  end if
  {Check next chain element...}
end for
return NOT_FOUND;
```

Figure 4.2: Pseudocode for hash bucket lookup in FAWN-DS.

Mapping a Key to a Value. FAWN-DS uses an in-memory (DRAM) Hash Index to map 160-bit keys to a value stored in the Data Log. It stores only a fragment of the actual key in memory to find a location in the log; it then reads the full key (and the value) from the log and verifies that the key it read was, in fact, the correct key. This design trades a small and configurable chance of requiring multiple reads from flash (we set it to roughly 1 in 16,384 accesses) for drastically reduced memory requirements (only six bytes of DRAM per key-value pair).

Figure 4.2 shows the pseudocode that implements this design for `Lookup`. FAWN-DS extracts two fields from the 160-bit key: the i low order bits of the key (the *index bits*) and the next 15 low order bits (the *key fragment*). FAWN-DS uses the index bits to select a bucket from the Hash Index, which contains 2^i hash buckets. Each bucket is only six bytes: a 15-bit key fragment, a valid bit, and a 4-byte pointer to the location in the Data Log where the full entry is stored.

Lookup proceeds, then, by locating a bucket using the index bits and comparing the key against the key fragment. If the fragments do not match, FAWN-DS uses hash chaining to continue searching the hash table. Once it finds a matching key fragment, FAWN-DS reads the record off of the flash. If the stored full key in the on-flash record matches the desired lookup key, the operation is complete. Otherwise, FAWN-DS resumes its hash chaining search of the in-memory hash table and searches additional records. With the 15-bit key fragment, only 1 in 32,768 retrievals from the flash will be incorrect and require fetching an additional record, a resulting read-amplification factor of less than 1.0001.

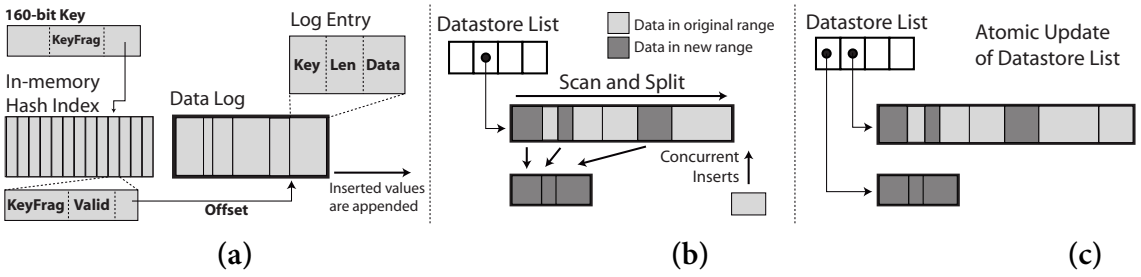


Figure 4.3: (a) FAWN-DS appends writes to the end of the Data Log. (b) Split requires a sequential scan of the data region, transferring out-of-range entries to the new store. (c) After scan is complete, the datastore list is atomically updated to add the new store. Compaction of the original store will clean up out-of-range entries.

The constants involved (15 bits of key fragment, 4 bytes of log pointer) target the 2007 prototype FAWN nodes described in Section 4.2. A typical object size is between 256 B to 1 KB, and the nodes have 256 MB of DRAM and approximately 4 GB of flash storage. Because each node is responsible for V key ranges (each of which has its own datastore file), a single physical node can address 4 GB * V bytes of data. Expanding the in-memory storage to 7 bytes per entry would permit FAWN-DS to address 1 TB of data per key range. While some additional optimizations are possible, such as rounding the size of objects stored in flash or reducing the number of bits used for the key fragment (and thus incurring, e.g., a 1-in-1000 chance of having to do two reads from flash), the current design works well for the target key-value workloads we study.

Several subsequent hashtable designs have chosen an even more aggressive space-time tradeoff building off our design (e.g., FlashStore [44], SILT [83]) to further reduce memory consumption. These techniques add per-hash computations, bloom filters, and compression to do so. On the generation of wimpy nodes we use in this work, hashing alone already consumes 25-40% of available CPU cycles at full load, so adding computational complexity was not worth the memory consumption tradeoff for this platform.

Reconstruction. Using this design, the Data Log contains all the information necessary to reconstruct the Hash Index from the log alone. As an optimization, FAWN-DS can periodically checkpoint the index by writing the Hash Index and a pointer to the last log entry to flash. After a failure, FAWN-DS uses the checkpoint as a starting point to reconstruct the in-memory Hash Index quickly.

Virtual IDs and Semi-random Writes. A physical node has a separate FAWN-DS datastore file for each of its virtual IDs, and FAWN-DS appends new or updated data items to the appropriate datastore. Sequentially appending to a small number of files is termed *semi-random writes*. Prior work by Nath and Gibbons observed that with many flash devices, these semi-random writes are nearly as fast as a single sequential append [97]. We take advantage of this property to retain fast write performance while allowing key ranges to be stored in independent files to speed the maintenance operations described below. We show in Section 4.2 that these semi-random writes perform sufficiently well.

Basic functions: Store, Lookup, Delete

`Store` appends an entry to the log, updates the corresponding hash table entry to point to this offset within the Data Log, and sets the valid bit to true. If the key written already existed, the old value is now *orphaned* (no hash entry points to it) for later garbage collection.

`Lookup` retrieves the hash entry containing the offset, indexes into the Data Log, and returns the data blob.

`Delete` invalidates the hash entry corresponding to the key by clearing the valid flag and writing a *Delete entry* to the end of the data file. The delete entry is necessary for fault-tolerance—the invalidated hash table entry is not immediately committed to non-volatile storage to avoid random writes, so a failure following a delete requires a log to ensure that recovery will delete the entry upon reconstruction. Because of its log structure, FAWN-DS deletes are similar to store operations with 0-byte values. Deletes do not immediately reclaim space and require compaction to perform garbage collection. This design defers the cost of a random write to a later sequential write operation.

Maintenance: Split, Merge, Compact

Inserting a new virtual node into the ring causes one key range to split into two, with the new virtual node gaining responsibility for the first part of it. Nodes handling these *VIDs* must therefore `Split` their datastore into two datastores, one for each key range. When a virtual node departs the system, two adjacent key ranges must similarly `Merge` into a single datastore. In addition, a virtual node must periodically `Compact` its datastores to clean up stale or orphaned entries created by `Split`, `Store`, and `Delete`.

The design of FAWN-DS ensures that these maintenance functions work well on flash, requiring only scans of one datastore and sequential writes into another. We briefly discuss each operation in turn.

`Split` parses the Data Log sequentially, writing each entry in a new datastore if its key falls in the new datastore's range. `Merge` writes every log entry from one datastore into the other datastore; because the key ranges are independent, it does so as an append. Split and Merge propagate delete entries into the new datastore.

`Compact` cleans up entries in a datastore, similar to garbage collection in a log-structured filesystem. It skips entries that fall outside of the datastore's key range, which may be left-over after a split. It also skips orphaned entries that no in-memory hash table entry points to, and then skips any delete entries corresponding to those entries. It writes all other valid entries into the output datastore.

Concurrent Maintenance and Operation

All FAWN-DS maintenance functions allow concurrent read and write access to the datastore. `Stores` and `Deletes` only modify hash table entries and write to the end of the log.

The maintenance operations (`Split`, `Merge`, and `Compact`) sequentially parse the Data Log, which may be growing due to deletes and stores. Because the log is append-only, a log entry once parsed will never be changed. These operations each create one new output datastore logfile. The maintenance operations therefore run until they reach the end of the log, and then briefly lock the datastore, ensure that all values flushed to the old log have been processed, update the FAWN-DS datastore list to point to the newly created log, and release the lock (Figure 4.3c). The lock must be held while writing in-flight appends to the log and updating datastore list pointers, which typically takes on the order of microseconds at the end of a `Split` or `Merge` (Section 4.2.1).

4.2 Evaluation

We begin by characterizing the I/O performance of a wimpy node. From this baseline, we then evaluate how well FAWN-DS performs on this same node, finding that its performance is similar to the node's baseline I/O capability. To further illustrate the advantages of FAWN-DS's design, we compare its performance to an implementation using the general-purpose Berkeley DB, which is not optimized for use on flash. We finish the evaluation of the single-node FAWN-DS design by measuring the impact of the background failure operations triggered in FAWN-DS on foreground query latency.

Seq. Read	*Rand Read*	*Seq. Write*	*Rand. Write*
28.5 MB/s	1424 QPS	24 MB/s	110 QPS

Table 4.1: Baseline CompactFlash statistics for 1 KB entries. QPS = Queries/second.

Evaluation Hardware: Our FAWN cluster has 21 back-end nodes built from commodity PCEngine Alix 3c2 devices, commonly used for thin-clients, kiosks, network firewalls, wireless routers, and other embedded applications. These devices have a single-core 500 MHz AMD Geode LX processor, 256 MB DDR SDRAM operating at 400 MHz, and 100 Mbit/s Ethernet. Each node contains one 4 GB Sandisk Extreme IV CompactFlash device. A node consumes 3 W when idle and a maximum of 6 W when deliberately using 100% CPU, network and flash. The nodes are connected to each other and to a 27 W Intel Atom-based front-end node using two 16-port Netgear GS116 GigE Ethernet switches.

Evaluation Workload: FAWN-KV targets read-intensive, small object workloads for which key-value systems are often used. The exact object sizes are, of course, application dependent. In our evaluation, we show query performance for 256 byte and 1 KB values. We select these sizes as proxies for small text posts, user reviews or status messages, image thumbnails, and so on. They represent a quite challenging regime for conventional disk-bound systems, and stress the limited memory and CPU of our wimpy nodes.

4.2.1 Individual Node Performance

We benchmark the I/O capability of the FAWN nodes using iozone [66] and Flexible I/O tester [7]. The flash is formatted with the ext2 filesystem and mounted with the `noatime` option to prevent random writes for file access [96]. These tests read and write 1 KB entries, the lowest record size available in iozone. The filesystem I/O performance using a 3.5 GB file is shown in Table 4.1.

FAWN-DS Single Node Local Benchmarks

Lookup Speed: This test shows the query throughput achieved by a local client issuing queries for randomly distributed, existing keys on a single node. We report the average of three runs (the standard deviations were below 5%). Table 4.2 shows FAWN-DS 1 KB and 256 byte random read queries/sec as a function of the DS size. If the datastore fits in the

DS Size	*1 KB Rand Read* (in queries/sec)	*256 B Rand Read* (in queries/sec)
10 KB	72352	85012
125 MB	51968	65412
250 MB	6824	5902
500 MB	2016	2449
1 GB	1595	1964
2 GB	1446	1613
3.5 GB	1150	1298

Table 4.2: Local random read performance of FAWN-DS.

buffer cache, the node locally retrieves 50–85 thousand queries per second. As the datastore exceeds the 256 MB of RAM available on the nodes, a larger fraction of requests go to flash.

FAWN-DS imposes modest overhead from hash lookups, data copies, and key comparisons, and it must read slightly more data than the iozone tests (each stored entry has a header). The resulting query throughput, however, remains high: tests reading a 3.5 GB datastore using 1 KB values achieved 1,150 queries/sec compared to 1,424 queries/sec from the filesystem. Using the 256 byte entries that we focus on below achieved 1,298 queries/sec from a 3.5 GB datastore. By comparison, the raw filesystem achieved 1,454 random 256 byte reads per second using Flexible I/O.

Bulk `store` Speed: The log structure of FAWN-DS ensures that data insertion is entirely sequential. As a consequence, inserting two million entries of 1 KB each (2 GB total) into a *single* FAWN-DS log sustains an insert rate of 23.2 MB/s (or nearly 24,000 entries per second), which is 96% of the raw speed that the flash can be written through the filesystem.

`Put` Speed: In FAWN-KV, each FAWN node has $R * V$ FAWN-DS files: each virtual ID adds one primary data range, plus an additional $R - 1$ replicated ranges. A node receiving puts for different ranges will concurrently append to a small number of files ("semi-random writes"). Good semi-random write performance is central to FAWN-DS's per-range data layout that enables single-pass maintenance operations. We therefore evaluate its performance using five flash-based storage devices.

Semi-random performance varies widely by device. Figure 4.4 shows the aggregate write performance obtained when inserting 2GB of data into FAWN-DS using five different flash drives as the data is inserted into an increasing number of datastore files. All SATA-

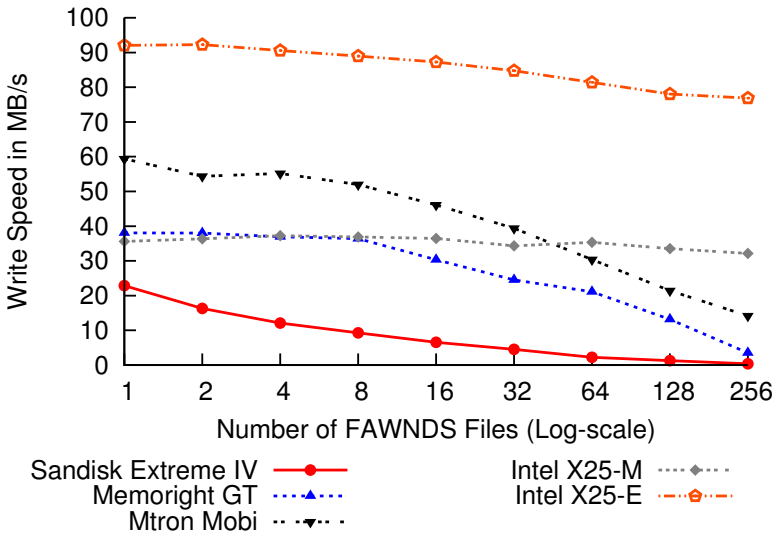


Figure 4.4: Sequentially writing to multiple FAWN-DS files results in semi-random writes.

based flash drives measured below use an Intel Atom-based chipset because the Alix3c2 lacks a SATA port. The relatively low-performance CompactFlash write speed slows with an increasing number of files.

The 2008 Intel X25-M and X25-E, which use log-structured writing and preemptive block erasure, retain high performance with up to 256 concurrent semi-random writes for the 2 GB of data we inserted; both the Mtron Mobi and Memoright GT drop in performance as the number of files increases. The key take-away from this evaluation is that flash devices are *capable* of handling the FAWN-DS write workload extremely well—but a system designer must exercise care in selecting devices that actually do so.

Comparison with BerkeleyDB

To understand the benefit of FAWN-DS's log structure, we compare with a general purpose disk-based database that is *not* optimized for flash. BerkeleyDB provides a simple put/get interface, can be used without heavy-weight transactions or rollback, and performs well versus other memory or disk-based databases. We configured BerkeleyDB using both its default settings and using the reference guide suggestions for flash-based operation [28]. The best performance we achieved required 6 hours (B-Tree) and 27 hours (Hash) to insert

seven million, 200 byte entries to create a 1.5 GB database. This corresponds to an insert rate of 0.07 MB/s.

The problem was, of course, small writes: When the BDB store was larger than the available RAM on the nodes ($<$ 256 MB), both the B-Tree and Hash implementations had to flush pages to disk, causing many writes that were much smaller than the size of an erase block.

That comparing FAWN-DS and BDB seems unfair is exactly the point: even a well-understood, high-performance database will perform poorly when its write pattern has not been specifically optimized to flash's characteristics. We evaluated BDB on top of NILFS2 [100], a log-structured Linux filesystem for block devices, to understand whether log-structured writing could turn the random writes into sequential writes. Unfortunately, this combination was not suitable because of the amount of metadata created for small writes for use in filesystem checkpointing and rollback, features not needed for FAWN-KV—writing 200 MB worth of 256 B key-value pairs generated 3.5 GB of metadata. Other existing Linux log-structured flash filesystems, such as JFFS2 [69], are designed to work on raw flash, but modern SSDs, compact flash and SD cards all include a Flash Translation Layer that hides the raw flash chips. While future improvements to filesystems can speed up naive DB performance on flash, the pure log structure of FAWN-DS remains necessary even if we could use a more conventional backend: it provides the basis for replication and consistency across an array of nodes.

Read-intensive vs. Write-intensive Workloads

Most read-intensive workloads have at least some writes. For example, Facebook's memcached workloads have a 1:6 ratio of application-level puts to gets [70]. We therefore measured the aggregate query rate as the fraction of puts ranged from 0 (all gets) to 1 (all puts) on a single node (Figure 4.5).

FAWN-DS can handle more puts per second than gets because of its log structure. Even though semi-random write performance across eight files on our CompactFlash devices is worse than purely sequential writes, it still achieves higher throughput than pure random reads.

When the put-ratio is low, the query rate is limited by the get requests. As the ratio of puts to gets increases, the faster puts significantly increase the aggregate query rate. On the other hand, a pure write workload that updates a small subset of keys would require frequent cleaning. In our current environment and implementation, both read and write rates slow to about 700–1000 queries/sec during compaction, bottlenecked by increased

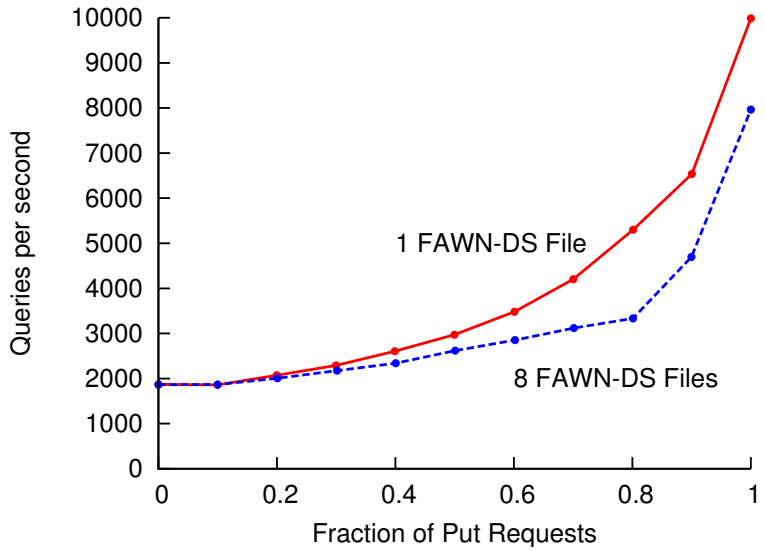


Figure 4.5: FAWN supports both read- and write-intensive workloads. Small writes are cheaper than random reads due to the FAWN-DS log structure.

thread switching and system call overheads of the cleaning thread. Last, because deletes are effectively 0-byte value puts, delete-heavy workloads are similar to insert workloads that update a small set of keys frequently. In the next section, we mostly evaluate read-intensive workloads because it represents the target workloads for which FAWN-KV is designed.

Impact of Failures/Arrivals on Query Latency

Figure 4.6 shows the distribution of query latency for three workloads: a pure get workload issuing gets at the highest sustainable rate (Max Load), a 500 requests per second workload with a concurrent Split (Split-Low Load), and a 1500 requests per second workload with a Split (Split-High Load).

Accesses that hit buffer cache are returned in 300 μs including processing and network latency. When the accesses go to flash, the median response time is 800 μs because the access time of the CompactFlash device is approximately 500 μs. Even during a split, the median response time remains under 1 ms. The median latency increases with load, so the max load, get-only workload has a slightly higher median latency than the split workloads that have a slowly lower external query load.

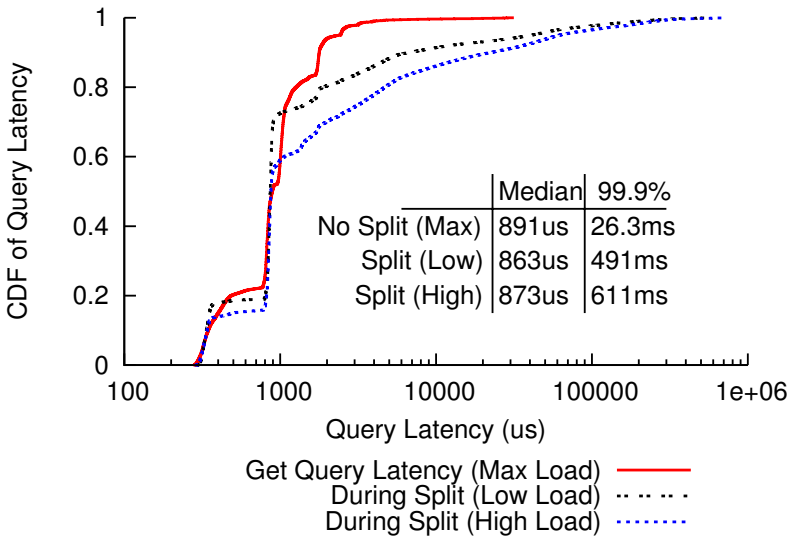


Figure 4.6: Query latency CDF for normal and split workloads.

Many key-value systems care about 99.9th percentile latency guarantees as well as fast average-case performance. During normal operation, request latency is very low: 99.9% of requests take under 26.3 ms, and 90% take under 2 ms. During a split with low external query load, 10% of requests experience a query latency above 10 ms. The 99.9%-ile response time during the low-activity split is 491 ms. For a high-rate request workload, the incoming request rate is occasionally higher than can be serviced during the split. Incoming requests are buffered and experience additional queuing delay: the 99.9%-ile response time is 611 ms. Fortunately, these worst-case response times are still on the same order as those worst-case times seen in production key-value systems [46], but they are still much higher than anticipated from our design.

High latency behavior due to flash garbage collection. We initially attributed the high latency (even at low external load) to the need to atomically lock the datastore at the end of a split. However, we measured the lock duration to be between 5–50 microseconds, which doesn't explain the spikes of hundreds of milliseconds we observe in Figure 4.6.

Our investigation into this problem identified that this high latency behavior can be attributed to the background garbage collection algorithms implemented in the flash translation layer in the flash device. The sequential writes caused by a split, merge or rewrite in FAWN-DS can trigger the block erasure operation on the CompactFlash, which contains

only one programmable flash chip. During this operation, all read requests to the device are stalled waiting for the device to complete the operation. While individual block erasures often only take 2 ms, the algorithm used on this flash device performs a bulk block erasure lasting *hundreds* of milliseconds, causing the 99.9%ile latency behavior of key-value requests to skyrocket. With larger values (1KB), query latency during Split increases further due to a lack of flash device parallelism—a large write to the device blocks concurrent independent reads, resulting in poor worst-case performance.

Modern SSDs support and *require* request parallelism to achieve high flash drive performance by packaging many flash chips together in one SSD [102]; these devices could greatly reduce the effect of background operations on query latency by using algorithms that perform writes or block erasures on only a subset of the flash chips inside the SSD. Unfortunately, our experience with many modern SSDs shows that they all perform bulk block erasures that block access to the entire device, producing the same tens to hundreds of milliseconds of delays we see with our 2007-era CompactFlash device.

We hypothesize that these hundreds of millisecond worst-case latencies occur due to engineering decisions made by Flash Translation Layer designers. Lazy garbage collection algorithms require erasing blocks in bulk, trading higher worst-case latency for higher peak write throughput. This decision is similar to the firmware scheduling policies used in hard disks to perform thermal calibration, which also blocks access to external requests until the calibration is complete. In the case of hard drives, the latency impact of thermal calibration was eliminated by distributing the calibrations during periods of light I/O. This improvement was important for media applications that needed more predictable streaming performance [116] from hard drives. We imagine that flash vendors will follow a similar path as customers call for more stringent and predictable latencies from enterprise flash systems.

One way to reduce this worst-case latency behavior is to use an m-of-n striping algorithm, where an object is erasure-coded such that access to m flash chips allows the full object to be reconstructed. The flash translation layer can then use an algorithm that performs bulk block erasures on no more than $n-m$ flash chips to ensure that reads for an individual object are never blocked. Our lack of access to the FTL on modern commercially-available SSDs prevents us from evaluating this space-QoS tradeoff.

Chapter 5

Workload Analysis for FAWN Systems

Through a deep study into a key-value storage workload, FAWN-KV demonstrated the potential of the FAWN approach and described a set of techniques to achieve the hardware's full potential. In this section, we explore the use of FAWN for a variety of other workloads: I/O-throughput bound, memory-bound, and CPU-bound. To a first order, we find that FAWN can be several times more efficient than traditional systems for I/O-bound workloads, and on par with or more efficient for some memory and CPU-limited applications.

Our experiences highlight several challenges to achieving the potential energy efficiency benefits of the FAWN approach for these workloads. Whereas the previous chapter described how rethinking the application software stack could harness the underlying capability of the wimpy node approach, this section demonstrates why existing software may not run as well on FAWN nodes which have limited resources (e.g., memory capacity, CPU cache sizes), and highlights specific challenges for new algorithms and optimizations.

A secondary contribution of this work is an evaluation of more modern FAWN prototypes than in the previous section, showing that many existing low-power hardware platforms have high fixed power costs that diminish the potential efficiency returns.

5.1 Workloads

In this section, we describe under what conditions a FAWN approach can provide higher energy efficiency, and where traditional architectures can be as efficient, or in some cases, more energy-efficient than low-power systems. For each case, we identify characteristics

of the workload, software, or hardware that contribute to results of the energy efficiency comparison.

5.1.1 Taxonomy

Warehouse-scale datacenters run a large and varying set of workloads. While they typically must process large amounts of data, the amount of per-byte processing required can vary significantly. As a result, it is worth investigating the applicability of the FAWN approach to a wide set of workloads that differ in their computational requirements.

We begin with a broad classification of the types of workloads found in data-intensive computing often found in large-scale datacenter deployments:

1. Memory/CPU-bound workloads
2. I/O-bound workloads
3. Latency-sensitive, but non-parallelizable workloads
4. Large, memory-hungry workloads

The first category includes CPU and memory-bound workloads, where the running time is limited by the speed of the CPU or memory system. In other words, these are workloads where the storage I/O subsystem is not fully saturated. This also assumes that the broader applications represented by these workloads can overlap computation and I/O such that improving the speed of the I/O system would not dramatically improve overall performance.

The second class of workloads, I/O-bound workloads, have running times that are determined primarily by the speed of the I/O devices (typically disks for data-intensive workloads). I/O-bound workloads can be either seek- or scan-bound as described earlier in Chapter 4 and related work [17]. Typically, these workloads represent the low-hanging fruit for the FAWN approach because traditional systems with complex, high-speed processors usually target more computational workloads for which they have been tuned more aggressively. Because we have already covered the seek-bound workload extensively, we discuss one other example of an I/O-bound workload: performing energy-efficient, large sorts as demonstrated in JouleSort [111].

Third, latency-sensitive workloads require fast responses times to provide, for example, an acceptable user-experience; anything too slow (e.g., more than 150ms) impairs the

quality of service unacceptably. Key to this workload definition is the inability to shard or parallelize the unit of work that carries a strict latency bound.

Finally, large, memory-hungry workloads frequently access data that can reside within the memory of traditional servers (on the order of a few to 10s of gigabytes per machine today). As we describe in Section 5.1.5, the data structure created in `grep` when searching for millions of short phrases requires several gigabytes of memory and is accessed randomly. This causes frequent swapping on FAWN nodes with limited memory, but fits entirely within DRAM on modern server configurations.

5.1.2 Memory-bound Workloads

We begin by exploring some worst-case workloads expected to be *more* energy-efficient on traditional, high-power, high-speed systems than low-power, low-speed systems.

Cache-bound Microbenchmark

Workload description: We created a synthetic memory-bound benchmark that takes advantage of out-of-order execution and large caches. This benchmark repeatedly performs a matrix transpose multiplication, reading the matrix and vector data from memory and writing the result to memory. We chose matrix transpose specifically because it exhibits increasingly poor memory locality as the size of the problem increases. The matrix data is in row-major format, which means that the transpose operation cannot sequentially stream data from memory. Each column of the matrix is physically separated in memory, requiring strided access and incurring more frequent cache evictions when the matrix does not fit entirely in cache.

The vector multiplications are data-independent to benefit from instruction reordering and pipelining, further biasing the workload in favor of modern high-speed, complex processors. We spawn as many parallel, independent incarnations of the benchmark as there are cores on the system to measure peak performance. We ran the benchmark with various input matrix sizes and estimate the metric of performance, FLOPS (floating point operations per second) as the number of multiply operations performed.[1]

Evaluation hardware: In this experiment, we compare an Intel Core i7-Desktop, our traditional system proxy, to an Intel Atom chipset, our more modern FAWN instantiation.

[1]Comparing the FLOPS numbers here to those found in other CPU-intensive benchmarks such as in the Green500 competition will underestimate the actual computational capabilities of the platforms we measured, because this benchmark primarily measures memory I/O, not peak floating point operation performance.

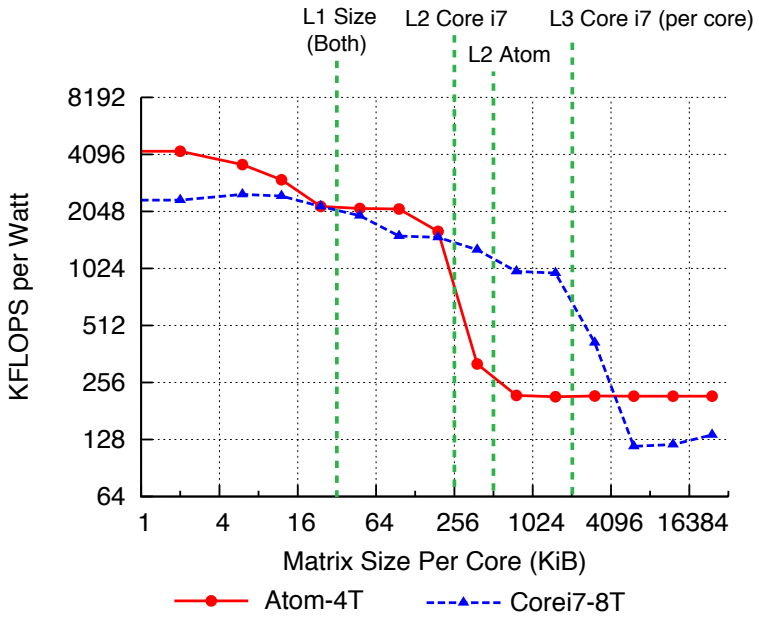


Figure 5.1: Efficiency vs. Matrix Size. Green vertical lines show cache sizes of each processor.

The i7-Desktop operates 4 cores at a max of 2.8GHz, though we used the Linux CPU ondemand scheduler to choose the appropriate speed for each workload. The i7 860 has a 32 KB L1 cache and a 256 KB L2 cache *per core*, and also has an 8 MB L3 cache shared across all 4 cores. We enabled two-way Hyper-threading (Simultaneous Multi-Threading) so that the system exposed 8 "processors" to the operating system; disabling Hyper-threading did not improve performance for this benchmark. We attached one X25-E and one 2 GB DRAM DIMM to the traditional system further reduce power over a fully I/O-balanced system. At idle, the power consumed by the machine was 40 W and at full load reached 130 W.

The Atom's processor cores each have a 24 KB L1 data cache and a 512 KB L2 cache. Two-way hyper-threading was enabled, exposing 4 "processors" to the OS. At idle, the Atom system consumed 18 W and at full load would reach 29 W.

Results: Figure 5.1 shows the energy efficiency (in KFLOPS/W) of our matrix multiply benchmark as a function of the size of the matrix being multiplied. When the matrix fits in the L1 data cache of both the i7-Desktop and the Atom, the Atom is roughly twice as efficient as the i7-Desktop. As the matrix size exceeds the L1 data cache, most memory

accesses hit in L2 cache, and the efficiency drops by nearly a factor of two for both systems, with the Atom retaining higher energy efficiency.

The i7-Desktop's efficiency drops even further as the matrix size exhausts the 256 KB of L2 cache per core and accesses hit in L3. As the matrix size overflows the L2 cache on the Atom, most accesses then fall to DRAM and efficiency remains flat thereafter. Meanwhile, the matrix size continues to fit within the 8 MB L3 cache of the i7. Once the matrix grows beyond 8 MB, most of its accesses then hit in DRAM, and its energy efficiency drops below that of the Atom.

When the working set of the benchmark fits in the same level caches of each architecture, the Atom is up to twice as energy-efficient in KFLOPS/W as the i7-Desktop. However, when the workload fits in the L2/L3 cache of the i7-Desktop but exhausts the Atom's on-die cache, the i7-Desktop is considerably more efficient by up to a factor of four.

In other words, workloads that are cache-resident on a traditional system but not on a FAWN can be more efficient on the traditional system simply because of the amount of cache available on traditional systems. The incongruity of cache size between wimpy and brawny processors is one possible reason why existing software may not run as efficiently on FAWN systems, particularly when the software has not been carefully tuned to work on both types of platforms. Others have noted this similar discontinuity, arguing that wimpy processors should often have a larger cache hierarchy to be more efficient for some workloads [68]. While this would improve the energy efficiency comparison between FAWN and traditional systems, doing so could increase capital costs beyond the efficiency benefit it would provide.

Running below peak load: The above experiment used OpenMP to run multiple threads simultaneously, eight threads on the i7-Desktop and four threads on the Atom. Running multiple threads is required to fully tax the CPU and memory systems of each node. We also ran the same experiment on each system with one thread, to see how efficiency scales with load. Figure 5.2 shows that with one thread, the i7-Desktop is more efficient regardless of the size of the matrix.

This can be explained by *fixed power costs*. The i7-Desktop running one thread consumed 70 W (versus 40 W at idle), and the Atom running one thread consumed 20 W (versus 18 W at idle). The Atom platform we evaluated therefore pays a large penalty for not operating at full capacity. Its energy-proportionality is much worse than that of the i7-Desktop. Because the Atom was, at best, only twice as energy efficient as the i7-Desktop for this worst-case workload at 100% load, the inefficient chipset's power overhead dominates the CPU power and reduces the energy efficiency at low-load significantly. In the case of

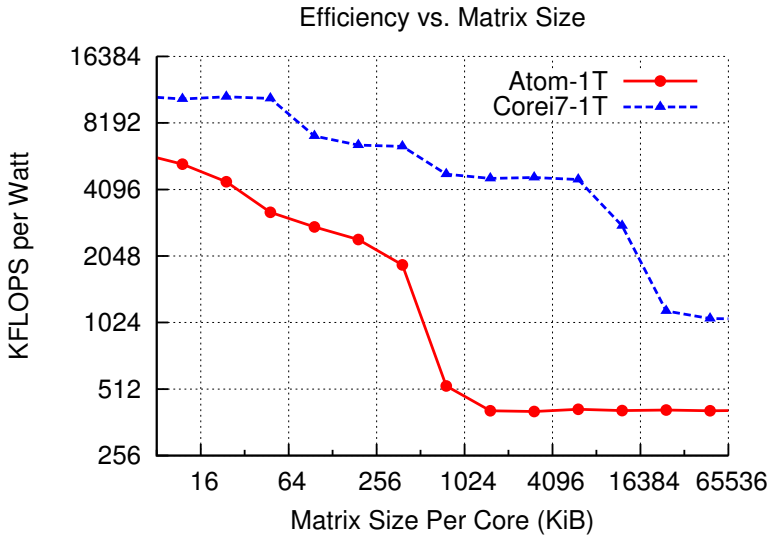


Figure 5.2: Efficiency vs. Matrix Size, Single Thread

our particular system, many of the fixed energy costs are due to non-"server" components: the GPU and video display circuitry, extra USB ports, and so on. Some components, however, such as the Ethernet port, cannot be eliminated. These same factors preclude the use of extremely low-power CPUs, as discussed in Chapter 2.4.5. Similar fixed power overheads have been noted by other researchers exploring the use of the FAWN approach to different workloads [68, 79].

5.1.3 CPU-bound Workloads

The memory-bound workloads in the previous section required frequent memory accesses per computation across a large dataset. Next, we look at a CPU-intensive task: cryptographic computation. This further isolates the comparison betwen wimpy and brawny *processors*, because the input and outputs are small and do not stress the storage I/O system at all.

Table 5.1 shows several assembly-optimized OpenSSL speed benchmarks on the i7-Desktop and Atom systems described above. On SHA-1 workloads, we find that the Atom-based platform is slightly more efficient in terms of work done per Joule than the i7-Desktop architecture, and for RSA sign/verify, the reverse is true.

Workload	i7-Desktop	Atom
SHA-1		
MB/s	360	107
Watts	75	19.1
MB/J	4.8	5.6
SHA-1 multi-process		
MB/s	1187	259
Watts	117	20.7
MB/J	10.1	12.51
RSA		
Sign/s	8748	1173.5
Verify/s	170248	21279.0
Watts	124	21.0
Sign/J	70.6	55.9
Verify/J	1373	1013

Table 5.1: Encryption Speed and Efficiency

This flip in efficiency appears to be due to the optimization choices made in the assembly code versions of the algorithms. The OpenSSL "C" implementations of both SHA-1 and RSA are both more efficient on the Atom; we hypothesize that the asm version is tuned for high-performance CPUs. The SHA-1 assembly implementation, in contrast, was recently changed to use instructions that also work well on the Atom, and so its efficiency again exceeds that of the i7-Desktop. These results suggest that, first, CPU-bound operations can be as or more efficient on low-power processors, and second, they underscore that nothing comes for free: code must sometimes be tweaked, or even rewritten, to run efficiently on these different architectures.

5.1.4 Scan-bound Workloads: JouleSort

The above microbenchmark described a tightly controlled cache size-bound experiment showing that differences in cache sizes can significantly impact energy efficiency comparisons. But these discontinuities appear in more real world macrobenchmarks as well. More specifically, in this section we look at sorting many small records and describe our experiences competing for the 2010 10GB JouleSort competition. Our best system consists of a machine with a low-power server processor and five flash drives, sorting the 10GB dataset

in 21.2 seconds (±0.227s) seconds with an average power of 104.9W (±0.8W). This system sorts the 10GB dataset using only 2228 Joules (±12 J), providing 44884 (±248) sorted records per Joule.

Our entry for the 10GB competition tried to use the most energy-efficient platform we could find that could hold the dataset in memory to enable a one-pass sort. We decided to use a one-pass sort on this hardware over a two-pass sort on more energy efficient hardware (such as Intel Atom-based boards) after experimenting with several energy efficient hardware platforms that were unable to address enough memory to hold the 10GB dataset in memory. The low-power platforms we tested suffered from either a lack of I/O capability or high, relative fixed power costs, both stemming from design decisions made by hardware vendors rather than being informed by fundamental properties of energy and computing.

Hardware: Our system uses an Intel Xeon L3426 1.86GHz quad-core processor (with two hyperthreads per core, TurboBoost-enabled) paired with 12GB of DDR3-1066 DRAM (2 DIMMS were 4GB modules and the other 2 DIMMS were 2GB modules). The mainboard is a development board from 2009 based on an Intel 3420 chipset (to the best of our knowledge, this confers no specific power advantage compared to off-the-shelf versions of the board such as the Supermicro X8SIL-F or Intel S3420GPV Server Board), and we used a Prolimatech "Megahalems" fanless heatsink for the processor.

For storage, we use four SATA-based Intel X25-E flash drives (three had a 32GB capacity and one had 64GB), and one PCIe-based Fusion-io ioDrive (80GB). We use a 300W standard ATX power supply (FSP300) with a built-in and enabled cooling fan.

The storage devices were configured as follows: one small partition of a 32GB X25-E contained the OS. The other three X25-Es, the leftover portions of the OS disk, and the Fusion-IO (partitioned into three 10GB partitions) were arranged in a single partition software RAID-0 configuration. Both the input and output file were located in a single directory within this partition. We used a Fusion-io in addition to 4 X25-Es because the SATA bus exists on the DMI bus with a bandwidth limitation of 10Gbps in theory and slightly less in practice. The Fusion-io was in a PCIe slot that is independent of the DMI bus and had a much higher bandwidth to the processor and memory system. Using both types of devices together therefore allowed us to more easily saturate the I/O and CPU capabilities of our system.

System power and software: The total power consumption of the system peaks at about 116 W during the experiment, but as mentioned below, averages about 105W over the duration of the sort runs. While we do not have individual power numbers for each component during the experiment, the {processor, DRAM, motherboard, power supply} combination

	Time (s)	Power (W)	Energy (J)	SRecs/J
Run 1	21.278	105.4	2242.5	44593
Run 2	21.333	104.1	2219.8	45049
Run 3	21.286	104.9	2232.6	44791
Run 4	21.454	104.1	2233.7	44769
Run 5	20.854	106.0	2211.5	45218
Avg	21.241	104.9	**2228.0**	**44884**
Error	0.227	0.849	12.273	247.564

Table 5.2: Summary of JouleSort Experiment Results.

consumes about 31 W at idle, the Fusion-io adds 6W at idle, and each X25-E adds about 1W to the idle power consumption for a total of 43 W at idle with all components attached.

All of our results are using Ubuntu Linux version 9.04 with kernel version 2.6.28 for driver compatibility with the Fusion-io device. We used ext4 with journaling disabled on the RAID-0 device. We use the `gensort` utility provided by the competition organizers (`http://sortbenchmark.org`) to create the 10^8 100-byte records and use `valsort` to validate our final output file. For sorting, we used a trial version of NSort software (`http://www.ordinal.com`).

Results: Our results are summarized in the Table 5.2. Our system improves upon the January 2010 Daytona winner by nearly a factor of two, and also improves upon the January 2010 Indy winner by 26% [26]. The January 2010 Indy winner group's more recent entry closes this gap to 5% for the Indy designation and 12% for the Daytona designation.

We log the statistics provided by NSort for comparison with [41]. Table 5.3 summarizes the information (Utilization measured out of a total of 800% and bandwidth measured in terms of MB/s for reading and writing the data).

Experiences: Our submission used a server-class system as opposed to a low-power component system like the Intel Atom. The dominating factor in this choice was the ability of our server system to hold the entire 10GB dataset in DRAM to enable a one-pass sort—in this case, the energy efficiency benefits of performing a one-pass sort outweighed the hardware-based energy efficiency of low-power platforms that must perform a two-pass sort. Our submission tried to use the most energy-efficient platform we could find that allowed for a one-pass sort, and this turned out to use the low-frequency Xeon platform described above. Below, we describe some details about what other systems we tried before settling on the entry system described above.

	In CPU Util	Out CPU Util	Input BW (MB/s)	Output BW (MB/s)
Run 1	343	628	973.71	1062
Run 2	339	651	953.29	1074
Run 3	339	613	971.82	1056
Run 4	336	622	975.61	1050
Run 5	343	646	976.56	1081
Avg	340	632	970.198	1065
Error	3	16.078	9.626	12.759

Table 5.3: JouleSort CPU and bandwidth statistics.

Alternative Platforms: We tried several alternative low-power configurations based on the Intel Atom as part of our research into the FAWN approach [17]. In particular, we began with the Zotac Ion board based on an Intel Dual-core Atom 330 (also used by Beckmann et. al) paired with 4 Intel X25-E drives. Without any special software tweaking, we were able to get approximately 35000 SRecs/J at an average power of about 33W. We also tried to use the NVidia GPU available on the Ion to do a portion of the sorting, but found that the I/O was the major bottleneck regardless.

We also experimented with a single core Atom board by Advantech paired with 1 X25-E, and a dual-core Atom Pineview development board with two X25-Es. These boards were both lower power than the Zotac Ion—the Pineview board moved from a three-chip to a two-chip solution, placing the graphics and memory controllers on-die, thus reducing chipset power slightly. We also tried attaching a Fusion-io board to a dual-core Atom system, but because the Fusion-io currently requires significant host processing and memory, the Atom could not saturate the capabilities of the drive and so was not currently a good fit.

5.1.5 Limitations

FAWN and other low-power many-core cluster architectures may be unsuited for some datacenter workloads. These workloads can be broadly classified into two categories: latency-sensitive, non-parallelizable workloads and memory-hungry workloads.

Latency-sensitive, non-parallelizable

As mentioned previously, the FAWN approach of reducing speed for increased energy efficiency relies on the ability to parallelize workloads into smaller discrete chunks, using more nodes in parallel to meet performance goals; this is also known as the scale-out approach using strong scaling. Unfortunately, not all workloads in data-intensive computing are currently amenable to this type of parallelism.

Consider a workload that requires encrypting a 64 MB chunk of data within 1 second, and assume that a traditional node can optimally encrypt at 100 MB/sec and a FAWN node at 20 MB/sec. If the encryption cannot be parallelized, the FAWN node will not encrypt data fast enough to meet the strict deadline of 1 second, whereas the traditional node would succeed. Note that if the fastest system available was insufficient to meet a particular latency deadline, parallelizing the workload here would no longer be optional for either architecture. Thus, the move to many-core architectures (with individual core speed reaching a plateau) poses a similar challenge of requiring application parallelism.[2]

Memory-hungry workloads

Workloads that demand large amounts of memory per process are another difficult target for FAWN architectures, because the memory capacity per node in FAWN is typically an order of magnitude lower than a traditional system is typically configured with.

To better understand this class of workload, we examined a specific workload derived from a machine learning application that takes a massive-data approach to semi-supervised, automated learning of word classification. The problem reduces to counting the number of times each phrase, from a set of thousands to millions of phrases, occurs in a massive corpus of sentences extracted from the Web. FAWN converts a formerly I/O-bound problem into a memory size-bound problem, which requires new algorithms and implementations to work well. For example, the 2007 FAWN prototype using Alix3c2 nodes can `grep` for a single pattern at 25 MB/sec, close to the maximum rate the CompactFlash device can provide. However, searching for thousands or millions of phrases with the naive Aho-Corasick algorithm in `grep` requires building a DFA data structure that requires several gigabytes of memory. Although this structure fit in the memory of conventional architectures equipped

[2]Indeed, this challenge is apparent to the designers of next-generation cryptographic algorithms: Several of the entrants to the NIST SHA-3 secure hash competition include a hash-tree mode for fast, parallel cryptographic hashing. The need for parallel core algorithms continues to grow as multi- and many-core approaches find increased success. We believe this general need for parallel algorithms will help make a FAWN many-core approach even more feasible.

with 8–16 GB of DRAM, it quickly exhausted the 256 MB of DRAM on each individual FAWN node.

To enable this search to function on a node with tight memory constraints, colleagues optimized the search using a rolling hash function and large bloom filter to provide a one-sided error grep (false positive but no false negatives) that achieves roughly twice the energy efficiency (bytes per second per Watt) as a conventional node [95].

However, this improved efficiency came at the cost of considerable implementation effort. Our broad experience with using FAWN for different applications suggests that efficiently using FAWN nodes for some scan-based workloads will require the development of easy-to-use frameworks that provide common, heavily-optimized data reduction operations (e.g., grep, multi-word grep, etc.) as primitives. This represents an exciting avenue of future work: while speeding up hardware is difficult, programmers have long excelled at finding ways to optimize CPU-bound problems.

An interesting consequence of this optimization was that the same techniques to allow the problem to fit in DRAM on a FAWN node drastically improved cache performance on more conventional architectures: Others were able to apply these techniques to double the speed of virus scanning on desktop machines [35]. As we detail below, we find that scaling up is often as important as scaling out when the goal is to improve energy efficiency.

5.2 Lessons from Workload Analysis

In this section, we summarize some of the lessons we have learned about applying FAWN to a broader set of workloads. We break down these lessons into two different categories: hardware challenges and software challenges.

Hardware Challenges

Many of today's hardware platforms appear capable of further improvements to energy efficiency, but are currently limited in practice due to several factors, many of which are simply due to choices made by hardware vendors of low-power platforms:

High idle/fixed cost power: The boards we have used all idled at 15-20W even though their peak is only about 10-15W higher. Fixed costs affect both traditional processors and low-power CPUs alike, but the proportionally higher fixed-cost to peak-power ratio on available Atom platforms diminishes some of the benefits of the low-power processor. In

practice, this means that the best FAWN systems target the lowest-speed processor that is not dominated by fixed power costs, precisely the point illustrated by Figure 2.4.

IO and bus limitations: When exploring the sort benchmark, we found it difficult to find systems that provided sufficient I/O to saturate the processor. Most Atom boards we examined provided only two SATA drive connectors. While some exist that contain up to six ports, they were connected to the CPU over a bandwidth-limited DMI bus; this bus provides 10Gbps in each direction, which can support only four X25-E SSDs reading at 250MB/second. These limitations may reflect the fact that these processors are not aimed at the server market in which I/O typically receives more emphasis.

The market for ultra low power server systems has greatly expanded over the last several years, with companies such as SeaMicro, Marvell, Calxeda and ZT Systems all producing low-power datacenter computing platforms. We expect many of the non-fundamental hardware challenges to disappear as competition drives further innovation, but many of the fundamental challenges (e.g., the unavoidable fixed power costs posed by having an onboard Ethernet chip or I/O controllers) will always play a large role in determining the most efficient balance and absolute speed of CPU and I/O on an energy-efficient platform.

Software Challenges

The central theme of this thesis is that existing software often does not run as well on FAWN node platforms. When deploying out-of-the-box software on FAWN and finding poor efficiency results, it is critically important to identify precisely the characteristics of the workload or the software that reduce efficiency. For example, many applications are becoming increasingly memory hungry as server-class hardware makes more memory per node available. As we have shown, the working set size of a cache- or memory-bound application can be an important factor in the FAWN vs. traditional comparison. If these applications cannot reduce their working set size, this is a fundamental limitation that FAWN systems cannot overcome. Fortunately, many algorithmic changes to software can improve memory efficiency to the point where the application's performance on a FAWN platform significantly increases. This emphasizes the thesis statement that writing efficient software on top of efficient hardware has a large role in improving energy efficiency and making the most of the FAWN approach.

Memory efficiency is not the only software challenge to overcome when considering FAWN systems. By shrinking the CPU–I/O gap, more balanced systems may become CPU-bound when processing I/O by exposing previously unimportant design and implementation inefficiencies. As we highlight in more detail in the next chapter, we have observed that the Linux block layer—designed and optimized for rotating media—imposes high per-

request overhead that makes it difficult to saturate a modern flash drive using a single or dual-core Atom processor. Similarly, we have identified that for some sequential I/O workloads microbenchmarks like `dd` or `fio`, some of the FAWN prototypes we have used are unable to read or write *sequentially* to the full capability of the attached SSDs, despite there being no obvious expected software or hardware bottlenecks.

Software optimization back in vogue: Software efficiency was once a community focus: ekeing every last drop of performance or resource from a system was a laudable goal. With the rapid growth of data-intensive computing and a reliance on Moore's law, today's developers are less likely to optimize resource utilization, instead focusing on scalability at the detriment of node efficiency [18]. Instead, the focus has been on scalability, reliability, manageability, and programmability of clusters. With a FAWN-like architecture, each node has fewer resources, making the job of the programmers harder. Our prior work has shown that the limited amount of memory per node has required the design of new algorithms [95] and careful balance of performance and memory footprint for in-memory hashtables [17]. These difficulties are compounded by the higher expected node count in FAWN architectures—not only does resource utilization become more important, these architectures will *further* stress scalability, reliability, and manageability.

Chapter 6

The Shrinking I/O Gap

Up to this point, we have examined application-level redesigns to understand how FAWN applies to different classes of datacenter workloads. In this part of the dissertation, we provide a measurement and analysis of Linux I/O stack performance on a modern FAWN system, demonstrating the "shrinking I/O gap" that future balanced systems will encounter.

6.1 Background

FAWN and other balanced system approaches have taken advantage of the previously *increasing* CPU-I/O gap prevalent throughout the 1990s and much of the 2000s. But two specific trends over the last few years have dramatically changed the landscape of system balance.

First, individual processor core speeds have hit a ceiling, with aggregate performance now being provided by using multiple cores in parallel. Figure 6.1 illustrates that the ratio of individual core speed to the speed of random I/O from storage is shrinking dramatically using historical data over the past 25 years. For example, the Nehalem "Westmere" series Intel Xeon Server processors offered today operate at a maximum speed of 3.46GHz per core, offering up to six cores and twelve hyperthreads per socket, and the "SandyBridge" processor line similarly operates between 2.5 and 3.4GHz, with core counts increasing to ten per socket. Individual core frequency has plateaued, while microarchitectural changes improve cycles per instruction performance over previous generations, but which only increases single-threaded performance by 10-30% (See [15]) rather than the trend of doubling single-threaded performance every few years.

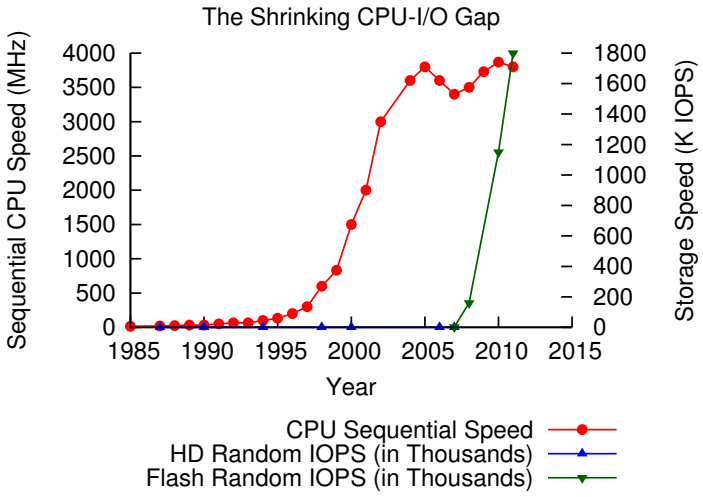


Figure 6.1: The Shrinking I/O gap trend. CPU numbers collected from `http://cpudb.stanford.edu`, showing the fastest (in terms of clock rate) Intel processor released every year. Hard drive seek time numbers from [52], flash IOPS based on public spec sheets.

Second, new non-volatile memories such as flash (with PCM or memristor in the future) have already begun making inroads in real systems, with access latencies and performance significantly faster than disks. The green line in Figure 6.1 shows data collected or measured about sequential and random I/O performance from disks over the past 25 years.[1] Random I/O performance from flash alone has gone from roughly 2500 random I/Os per second from CompactFlash platforms many years ago to over 1,000,000 IOPS per device [34], an improvement of over two orders of magnitude. Non-volatile memories capable of nanosecond access latencies and increased device parallelism will further increase this performance. This narrowing gap between core frequency and I/O speed means that the number of cycles needed to perform I/O may need to improve as the access latencies and IOPS throughputs of these new I/O devices continue to improve.

We term the combination of these two trends as the *shrinking I/O-gap*: As storage gets faster and core speeds stay flat, there is a pressing need to improve IOPS per core, or the processor efficiency of performing I/O.

[1]CPU numbers collected from `http://cpudb.stanford.edu`, showing the fastest clock rate for an Intel processor released every year. Hard drive seek time numbers come from an IBM GPFS Whitepaper [52]. Flash IOPS are derived from public specification sheets.

Mapping FAWN challenges to future balanced systems. The shrinking I/O gap is a trend that all systems are likely to experience. Because FAWN specifically targets slower frequency cores, we can measure the software impact of the shrinking I/O gap by assembling FAWN nodes with very fast SSDs on a hardware platform that should theoretically saturate the I/O system.

FAWN can therefore be thought of as an instrument that allows us to measure and experiment with existing systems to identify bottlenecks that future balanced systems will encounter a few years down the road. In the next part of the dissertation, we demonstrate the impact of the shrinking I/O gap through an analysis of low-level storage and operating system stack bottlenecks on a FAWN platform that provides more IOPS from storage than the system software stack can handle. In Chapter 7, we use the lessons from this measurement study to motivate the need for vector interfaces to storage and RPC, which we examine in greater detail using a novel SSD emulator platform that provides us greater control over the I/O interfaces than off-the-shelf hardware currently provides.

6.2 Measuring I/O Efficiency

In this section we perform several experiments to measure the I/O efficiency (in IOPS/core) of the existing Linux I/O stack. We focus particularly on random reads, a decision made for several reasons. First, random *writes* exhibit worst-case performance from today's flash devices, though this is slowly changing as the devices use more log-structured techniques underneath. Second, random I/O stresses the operating system more than sequential I/O, and we assume non-volatile memories like flash will predominantly be used first for random I/O because of their much better random I/O performance (flash random IOPS per dollar is higher than for disks).

To obtain a baseline measure for IOPS/core, we use a microbenchmark based on Flexible I/O Tester [7]. There are a few important parameters for this benchmark. First, we use the asynchronous libaio interface and set the I/O depth to 64 because modern flash devices require several outstanding I/Os to saturate the internal parallelism exposed by the device [102]. We could also use synchronous I/O, but we must run multiple threads of execution to achieve the same degree of I/O parallelism. Next, we read from the device in direct mode, bypassing the filesystem layer and its associated caching. While most applications will interact with the filesystem, direct I/O is the only way to issue 512 byte reads to the device, which best stresses the efficiency of the I/O stack from the block layer down through the device; going through the filesystem would require 4KB page access.

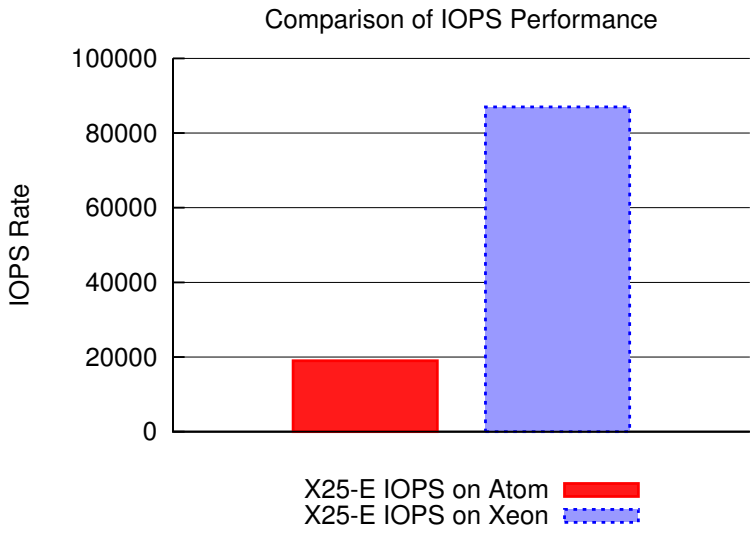


Figure 6.2: Number of random read IOPS provided by pairing an Atom and a Xeon CPU with a single X25 flash device. These experiments are run without the blk_iopoll NAPI polling setting enabled on the Atom.

Experimental Setup: For the rest of this chapter, we perform most of our measurements on Linux 2.6.36 kernels except where noted, and the storage device used is an Intel X25-based SATA drive. To stress I/O efficiency, we close the gap between the CPU and I/O by using a dual-core 1.6GHz Intel Atom D510 system. We also use a Core i7 2.8GHz system to provide a point of comparison to the Atom.

Figure 6.2 shows the result of running the Flexible I/O tester microbenchmark when pairing both the Intel Atom and the Intel Core i7 (Xeon) with the X25. As with this graph and in the rest of the paper, these experiments are run for a minimum of 30 seconds, and we take the final IOPS number reported by fio; our results show $<$5% variance between repeated runs so we omit error bars. The combination of the Intel Atom with the X25 provides only 19,000 IOPS. When pairing the X25 with the Xeon platform, however, we achieved 87,000 IOPS. This suggests that the device itself is capable of this I/O rate, and thus the bottleneck on the Atom platform is likely the processor (as subsequent experiments will corroborate).

Accurately measuring the IOPS/core for the Xeon platform would require attaching many more flash devices to the point where the Xeon becomes CPU-bottlenecked. Because we are CPU limited on the Atom already, improving the number of IOPS on this set of

Atom and flash hardware will increase the I/O efficiency as measured by IOPS/core. Thus, we investigate ways to improve the IOPS rate provided by this Atom + X25 configuration, although our experience has shown that some of these modifications would also apply to balanced brawny platforms as well.

6.2.1 Asynchronous Devices and I/O Paths

To better understand the limitations and bottlenecks that prevent higher IOPS throughput, we first profile the distribution of time spent in various locations in the I/O stack. Because storage devices today are asynchronous, there are two sequential paths to measure and analyze: the issuing path and the interrupt path.

The Issuing Path: When fio makes an I/O request direct to the drive, the code must traverse several parts of the I/O stack: the VFS layer, the block layer (including the I/O scheduling layer), and the device driver layer, which in the case of SATA-based devices is the SCSI/ATA layer.

The Interrupt Path: When the device completes a requested I/O, it will raise an interrupt to signal the operating system that the I/O has finished transferring data from the disk to kernel memory. In the hardware interrupt context, the operating system turns off the hardware interrupt line and then schedules further processing in the software interrupt (softIRQ) context, which completes some initial processing. It then schedules an I/O completion function at the block layer, which cleans up data structures and executes more callbacks at higher layers to notify the application of the completed I/O.

6.3 Analysis of Time Spent in the I/O Stack

Critical to understanding the bottlenecks in these two paths is measuring the time it takes to traverse portions of each path. Although this would normally require kernel modifications to add tracing and logging data, recent Linux kernels automatically provide hooks for tracing various events in the I/O stack.

`blktrace` is a user-level tracing program that can collect these trace events and log them to a file. Specifically, `blktrace` can be run in a separate process during microbenchmarks. Running `blktrace` has a small performance hit on the microbenchmark aggregate because it requires writing in large batches to a file. But the trace results can accurately capture the *minimum* costs of traversing the I/O stack, which is necessary to understand baseline I/O stack overhead.

```
Dev    CPU     SeqNum Timestamp    pid   Act .  Extra Info
8,0    3       89     0.000703200  1432  Q   R  143356401 + 1 [fio]
8,0    3       90     0.000705432  1432  G   R  143356401 + 1 [fio]
8,0    3       91     0.000707520  1432  P   N  [fio]
8,0    3       92     0.000708593  1432  I   R  143356401 + 1 [fio]
8,0    1       17     0.000713951     0  C   R  132397149 + 1 [0]
8,0    3       93     0.000715258  1432  U   N  [fio] 2
8,0    3       94     0.000723261  1432  D   R  143356401 + 1 [fio]
8,0    3       95     0.000754803  1432  Q   R  218920021 + 1 [fio]
```

Figure 6.3: Example `blktrace` output. Records which CPU each event is executed on, timestamp to nanosecond granularity, type of block activity, and other information, e.g., block number and process ID.

Figure 6.3 shows an example subset of output from a run of Flexible I/O tester with `blktrace` running in the background. This subset shows a read for block 143356401 issued on CPU 3, shortly after followed by another read for block 218920021. Note that the fifth entry shows that a completion for a different I/O was executed simultaneously on another CPU, demonstrating that the two paths of I/O can technically be handled in parallel (though a completion must execute on the same CPU on which it was issued). In most of our experiments, however, we enable CPU affinity for issuing/completing requests to simplify the analysis of results. Overlapping I/O on multiple processors makes it difficult to measure IOPS/core.

The important fields we use from this output are the CPU field, the timestamp field, and the Action field. The action field values describe the type of operation within the I/O stack:

- **Q** – Block request arrives at the block layer
- **G** – Request structure is allocated
- **P** – Queue is "plugged" to allow more I/Os to be issued to the queue for batching.
- **I** – Request is inserted into the I/O scheduler.
- **U** – Request queue is unplugged
- **D** – Request is sent to the underlying device driver (in this case, SCSI/ATA).
- **C** – The corresponding I/O is completed.

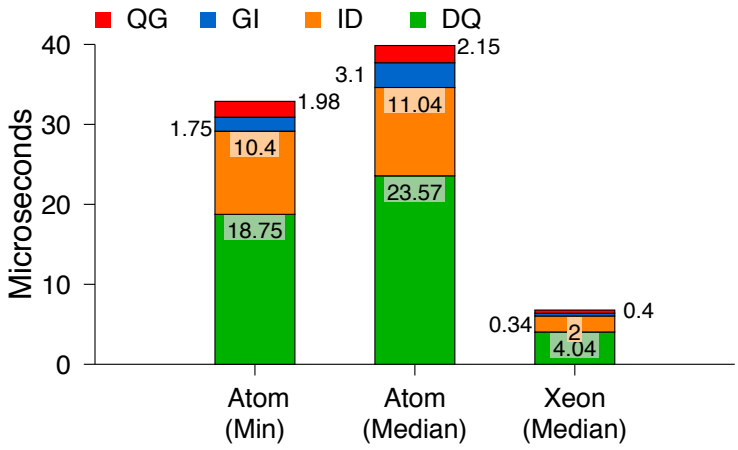


Figure 6.4: Breakdown of I/O time in the Kernel for Stock Kernel configuration.

Many of the subsequent graphs are generated by parsing the output of `blktrace` above. We begin by measuring the minimum and median time it takes for an I/O to transition from the various states above on the issuing path. More specifically, we measure the time spent between Q and G (QG), G and I (GI), I and D (ID), and D back to Q (DQ). For example, the time spent in ID represents how long a request remains in the queue of the I/O scheduler before being issued to the device driver. Time spent in DQ measures how long a request takes to travel through the device driver to when the next I/O arrives at the block layer.

6.3.1 The Issuing Path

Figure 6.4 shows the breakdown of time spent in these phases of the issuing path of the I/O stack. The Atom (Min) bar shows the minimum time to transition through each phase. For example, the time spent in ID, or between when the I/O is queued in the I/O scheduler to when it is issued to the device driver, took at least 10.4 microseconds. The Atom (Median) bar shows the median time to traverse these phases, and is not significantly worse than the minimum. We then plot the median I/O breakdown when measured on the Xeon platform on the far right.

There are several important takeaways from this graph. First, the total time to issue an I/O from the Q state to the next Q state is a minimum of 33 microseconds and a median of about 40 microseconds. Because this process is not pipelined, a single core can only issue

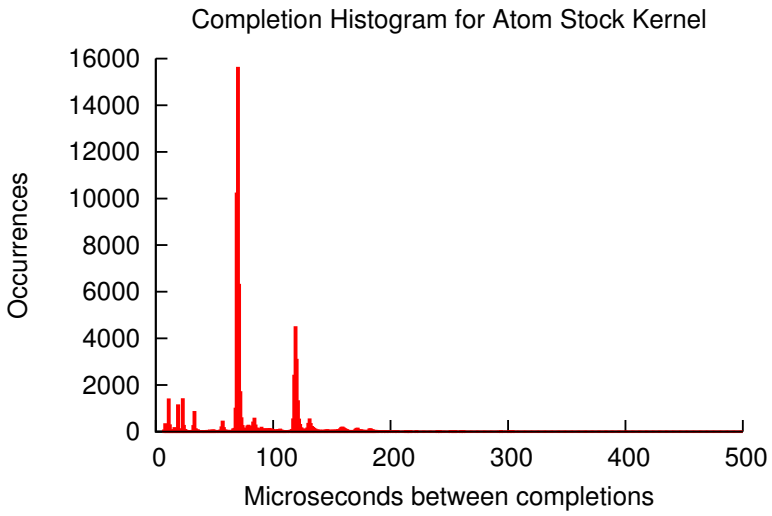


Figure 6.5: Histogram distribution of time between two consecutive completion events in microseconds.

as many I/Os as it can repeatedly traverse this entire path, so the IOPS/core on the issuing path is 1/40 microseconds, or about 25,000 IOPS issued per core. In contrast, the Xeon platform can go through the entire I/O stack in just 6.7 microseconds, so it is capable of performing up to 150,000 IOPS issued per core.

Another interesting result is that a majority of time is spent in the ID and DQ phases: the time an I/O sits in the I/O scheduler queue is significant, and the time spent in the device driver to when the next I/O begins its processing is even longer.

The last important takeaway is that the proportional importance of each phase is identical on both the Atom and the Xeon. On both platforms, the DQ phase is roughly 60% of the entire I/O stack time and the ID stack takes roughly 30% of the time. This suggests that nothing specific to the Atom platform is making any of these phases proportionally worse than on the Xeon platform (i.e., L2 cache overflows are not a large factor even though the two platforms have different L2 cache sizes).

Block Size	IOPS	Interrupt Rate	MB/s
512B	17K	17K	8.5
1KB	17K	17K	17
2KB	17K	17K	34
4KB	17K	17K	70
8KB	17K	17K	140
16KB	15.4K	15.1K	240

Table 6.1: IOPS, Interrupt rate and Bandwidth as a function of the block size for a random access microbenchmark on the Atom + X25 platform.

6.3.2 The Interrupt Path

The IOPS issued per core of the issuing path does not perfectly match the IOPS measured in Figure 6.2, suggesting that the issuing path may not be the bottleneck. Thus, in this section we briefly explore the interrupt path to see if it limits performance.

`blktrace` only tracks one event on the interrupt path: when an I/O finally completes in the block layer. Figure 6.5 shows the distribution of time between consecutive completions based on this coarse-grained information. The distribution contains two strong peaks, one at about 60 microseconds between completions and the other at about 120 microseconds. Note that the *minimum* time it takes to traverse from completion to completion is only about 6 microseconds: the time to handle the entire interrupt stack, such as resource cleanup and callback calling, is quite short in the best case.

Taking the inverse of the average completion time, the system can perform about 16667 completions per second per core. Because the minimum value is significantly faster than the median value, we see a slightly higher average number of completions per second when running experiments.

Comparison with fio

Recall that Figure 6.2 shows the Atom capable of about 19,000 IOPS. We therefore suspect the limit in performance is due to the slower interrupt path. As another indication that the interrupt path may be a bottleneck, Table 6.1 shows the results of an experiment that varies the block size of random I/O through the block layer, measuring IOPS, interrupt rate, and total data throughput. For data access up to 8KB, the IOPS obtained from the Atom

Block Size	IOPS	Interrupt Rate	MB/s
512B	78K	75K	38
1KB	72K	70K	70
2KB	60K	58K	120
4KB	41.5K	40.6K	166
8KB	26K	25.5K	208
16KB	14.6K	14.5K	233

Table 6.2: IOPS, Interrupt rate and Bandwidth as a function of the block size for a random access microbenchmark on the Xeon + X25 platform.

remained at 17,000, with the interrupt rate roughly matching it. It begins to drop at a block size of 16KB because of the limited sequential bandwidth of the drive.

In comparison, Table 6.2 shows the same experiment but run on the Xeon platform. Note that the IOPS and interrupt rate consistently drop as we increase the block size, suggesting that the limit in performance for a block size of 1KB and above is not the interrupt rate, since the system is capable of handling higher interrupt rates. The IOPS numbers for a block size of 512 bytes are lower than the numbers obtained in Figure 6.2 because these experiments were run using a slightly different benchmarking tool that is not as optimized as the fio benchmark used before.

6.3.3 Improving the Interrupt Path

To eliminate the interrupt bottleneck, we explore mechanisms to mitigate the number of interrupts handled by the operating system. We describe two such approaches to interrupt mitigation.

The NAPI Approach: A general interface developed for interrupt mitigation in the Linux kernel is the "New API" interface (NAPI). NAPI allows the OS to switch between interrupt-driven processing and spin-loop polling based on the load generated by the device. At high load, NAPI causes the system to switch to polling mode, which is more efficient because there is plenty of work available. At low load, the normal interrupt-driven mode is sufficient because the frequency of interrupts is low.

NAPI was originally developed for network cards, but since Linux Kernel version 2.6.28, NAPI support for block devices has been added (termed blk-iopoll). This mimics the gen-

eral framework of NAPI and has shown improvements to IOPS performance and CPU utilization for traditional systems [21]. The blk-iopoll system relies on the block device supporting Native Command Queuing (NCQ) to allow the operating system to queue up to 31 outstanding commands to the device at once. If the number of I/Os retrieved on an initial interrupt is high, the blk-iopoll system remains in polling mode to check if more commands will be completed soon.

Unfortunately, even with this change, the number of interrupts handled does not decrease. On fast, multi-core machines, a single fast core can queue several commands to the device before the device completes one of the requests and interrupts the system, so that multiple commands can be completed for only one interrupt. However, on a slower system such as the Intel Atom, the OS can only queue a few commands to the device before the first one completes. Switching from interrupt mode to polling therefore performs additional work to handle only one command, resulting in lower performance.

The Timer-based Approach An alternative approach to NAPI be taken by using event- and timer-based logic [20] to decide when to actually service requests from a network device [112], giving more control to the OS to decide when to perform device-related work.

Specifically, we have modified the blk-iopoll system to defer the completion of a command on an interrupt for a configurable duration. During this deferral, several more commands can be both issued and completed by the device and other OS work can be attended to. This deferral requires a later timer interrupt to finally complete all available commands. In essence, we allow one block device interrupt to trigger a series of timer interrupts, equally spaced, to increase the number of commands completed per interrupt. This deferral introduces a higher cost at low load where efficiency is less of a concern, is just as efficient as spin-loop polling at high-load (depending on how the OS schedules the work), and avoids the cost of switching between interrupt and polling mode frequently during medium load. This technique does not change any of the higher layer processing of the entire interrupt stack; it simply delays the handling of the *software interrupt* processing and higher layers by a specified amount on an initial hardware interrupt.

Figure 6.6 shows the distribution of inter-completion duration when using this patch. The first obvious point is that the histogram is highly clustered around six to eight microseconds. Six microseconds was the minimum inter-completion duration without the patch, representing a situation where a single interrupt happened to trigger two completions in a row. This modification specifically attempts to recreate more of those opportunities, and the results show that our technique is successful at doing so.

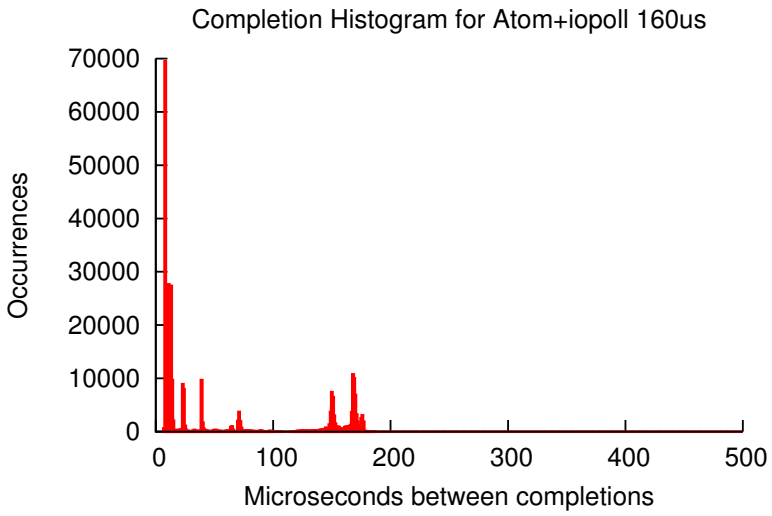


Figure 6.6: Histogram distribution of time between two consecutive completion events in microseconds with interrupt batching enabled. Defer parameter used in this experiment was 160 microseconds.

Our implementation requires one I/O to complete before we can defer the processing of further I/Os, causing a bump in the histogram at 160 microseconds. The time until the immediately next completion thus depends on the configuration parameter of how much to delay. Because we used 160 microseconds as this parameter, we occasionally create one inter-completion duration of 160 microseconds for many subsequent inter-completion durations of 6-8 microseconds.

The histogram alone does not indicate whether the interrupt path has been improved because the long 160 microsecond delays can adversely impact the IOPS rate. However, when we measure the IOPS rate after implementing this interrupt patch, we improve performance to about 25,000 IOPS when using the fio microbenchmark. While this did increase the IOPS rate, suggesting that indeed the interrupt path is a bottleneck, it only increases performance by a few thousand IOPS.

Recall that the median time it takes to traverse the issuing path of the I/O stack is 40 microseconds, meaning that a single core can issue only 25,000 requests per second to storage. While the interrupt mitigation scheme eliminated the interrupt path bottleneck, the new bottleneck is on the issuing path.

6.3.4 Improving the Issuing Path

We revisit Figure 6.4 to identify opportunities for improvement on the issuing path. As mentioned previously, the time spent in the I/O scheduler and the time spent in the device driver are the two major components of time spent on the issuing path. The biggest opportunity for improvement exists in the device driver code. However, the device driver code is complex and contains a lot of layering to translate block commands to the ATA commands issued to the device. Instead, we look at the I/O scheduler to identify whether there are any obvious improvements that can be made.

The default I/O scheduler used in Linux is CFQ: Completely Fair Queuing. Many have argued that the "noop" scheduler is better for SSDs because it does not need to worry about any of the seek or rotational penalties associated with disks. We re-run the fio benchmark and use `blktrace` to measure the time spent in the I/O stack. Figure 6.7 shows the comparison between the stock kernel (which uses CFQ) and our optimized kernel (which uses noop in combination with the kernel patch to mitigate interrupts). As expected, the QG, GI, and DQ phases are roughly the same between the two kernels. However, the ID phase, which is the time spent in the I/O scheduler, is dramatically different: moving to the noop scheduler results in a decrease of 8 microseconds in the I/O scheduler portion of the stack, or 33% of the time spent in the CFQ scheduler.

The overall time spent in the kernel therefore reduces from about 40 microseconds to 32 microseconds, so a single Atom core can traverse the issuing stack fast enough to provide about 30,000 IOPS. Indeed, our fio benchmark with these kernel modifications provides 29,649 IOPS.

One interesting result is that changing to the noop scheduler on a stock kernel may not yield significantly better performance because the bottleneck initially exists on the interrupt path. This highlights the importance of measuring and understanding the location of the bottleneck on a given platform.

Identifying the next opportunity: The right bar also shows that an even more significant amount of time is spent in the DQ phase, or the device driver part of the I/O stack. Improving this layer would take significantly more effort because it requires invasive code changes, but could greatly improve performance.

Unfortunately, the returns begin to diminish once this large bottleneck is removed. For example, reducing the DQ portion of the stack to about 3 microseconds would decrease the overall issuing path time to about 10 microseconds, providing potentially 100,000 IOPS, which is a significant increase over the existing result. At this point, however, improving any one portion of the stack then requires improving all portions of the stack equally to

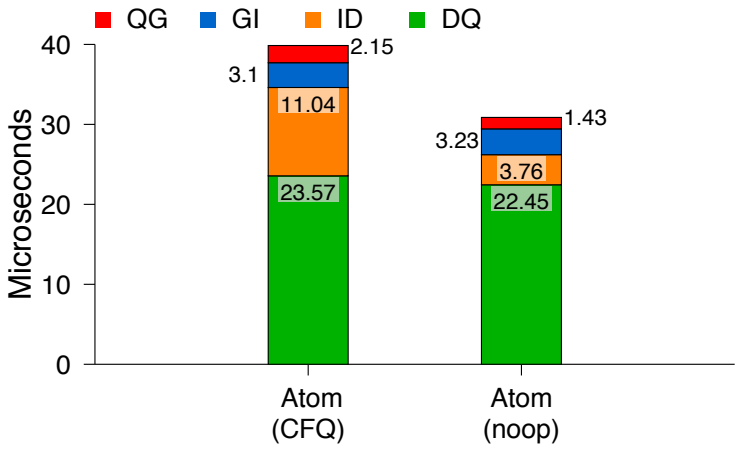


Figure 6.7: Breakdown of I/O stack time for original stock kernel and optimized kernel. Optimized kernel uses interrupt mitigation patch and uses the noop I/O scheduler instead of CFQ.

see significant improvements in performance. These changes will inevitably require significant software engineering effort. Furthermore, it is possible that other portions of the interrupt stack may begin to limit performance. For example, although the minimum inter-completion duration is 6 microseconds (which suggests approximately 150,000 completions per second is possible), this is only achievable by introducing the delay of hundreds of microseconds to batch completions together.

6.3.5 Summary

This part of the dissertation showed evidence of the shrinking I/O gap through an analysis of random I/O performance in the Linux I/O stack on a modern FAWN system. After identifying numerous initial bottlenecks, we improved random I/O performance by 50% through small modifications to parameters in the system combined with an interrupt coalescing algorithm for these flash SSDs.

Unfortunately, this approach does not demonstrate a *scalable* solution as the I/O gap continues to shrink. This motivates the need for novel ways to address the shrinking I/O gap through different interfaces and implementations that scale with increases in I/O performance and increasing core parallelism.

Chapter 7

Vector Interfaces to Storage and RPC

Given the difficulty of optimizing the existing I/O stack of operating systems to deal with the shrinking I/O gap, in this chapter we describe a solution to this problem by describing, implementing and evaluating the use of vector interfaces to storage and RPC systems.

Fast non-volatile memories (NVMs) are poised to change the landscape of data-intensive computing. Flash technology today is capable of delivering 6GB/s sequential throughput and just over 1 million I/Os per second from a single device [53]. But the emergence of these fast NVMs has created a painful gap between the performance the devices can deliver and the performance that application developers can extract from them.

As we show in this chapter, an implementation of a networked key-value storage server designed for the prior generation of flash storage only sustains 112,000 key-value queries per second (QPS) on a server with a storage device capable of 1.8 million QPS. This large performance gap exists for a number of reasons, including Ethernet and storage interrupt overhead, system call overhead, poor cache behavior, and so on. Unfortunately, the bottlenecks that create this gap span application code, middleware, numerous parts of the operating system kernel, and the storage device interface. Solutions that do not address all of these bottlenecks will fail to substantially narrow this performance gap.

To address bottlenecks across the entire stack, we advocate for pervasive use of "vector interfaces" in networked systems. At a high level, vector interfaces express work in groups of similar but independent units whose computation can be shared or amortized across the vector of work. Vector interfaces to storage submit multiple independent I/Os in one large batch (normally issued independently); vector interfaces to RPC coalesce multiple separate RPCs into one large RPC to reduce per-message overhead and send fewer packets over the network.

We focus our attention on one particular compelling use case, distributed key-value storage backed by fast NVM (e.g., solid state drives), to illustrate how applications can best use vector interfaces. We demonstrate that using vector interfaces improves throughput by an order-of-magnitude for our networked key-value storage system, allowing a single server to deliver 1.6 million key-value storage requests per second at a median latency below one millisecond, providing over 90% of the throughput capabilities of the underlying NVM device.

The contributions of this work are as follows:

- We demonstrate that end-to-end use of vector interfaces can deliver the performance potential of an NVM device capable of 1.8 million queries/second;
- We describe the latency versus throughput tradeoffs introduced by using vector interfaces and how systems should dynamically choose vector widths based on load;
- We describe how using vector interfaces exclusively on a storage server can increase *client* system efficiency;
- We provide guidance to system designers as to when vector interfaces should be used and when they are not effective.

7.1 The Shrinking CPU-I/O Gap

Fast non-volatile memories provide several benefits over traditional magnetic storage and DRAM systems. In contrast to magnetic disks, they provide several orders of magnitude lower access latency and higher small random I/O performance. They are therefore well suited to key-value storage workloads, which typically exhibit small random access patterns across a relatively small dataset. In response, researchers have developed several key-value storage systems specifically for flash [14, 17, 44, 45].

Solid state drive (SSD) performance has increased dramatically over the last few years (recall Figure 6.1). SATA-based flash SSDs have been capable of between 50,000 and 100,000 small (512-byte) random reads per second since 2009 [130], and enterprise-level PCIe-based SSDs advertise random read performance of approximately 1.2 million I/Os per second [53]. Prototype PCIe NVM platforms have demonstrated similar IOPS throughput with latencies of 40 microseconds for phase-change memory [11] and 10 microseconds for NVM emulators [34]. Meanwhile, CPU core sequential speeds have plateaued in recent

years, with aggregate system performance achieved through using multiple cores in parallel.

Given this dramatic improvement in throughput and latency, what changes are needed to allow key-value storage systems to take advantage of these improvements?

We answer this question in this section by measuring the performance of FAWN-KV (Chapter 4) running on a hardware emulator for next-generation non-volatile memories. The PCIe-based emulator is capable of 1.8 million IOPS, reflecting a modest but not implausible advance over the 1.1 million IOPS available off-the-shelf today from companies such as FusionIO. FAWN-KV was designed for a hardware architecture combining flash devices from a prior generation of CompactFlash or SATA-based SSDs with low-power processors such as the Intel Atom. The software has therefore already been optimized to minimize memory consumption and CPU overheads and take advantage of SSDs capable of nearly 100,000 IOPS.

First, we describe the non-volatile memory platform and provide baseline numbers to provide bounds for throughput and latency. Then, we demonstrate the best tuning of FAWN-KV on the same hardware platform, highlighting over an order of magnitude gap in throughput between device capability and measured throughput.

7.1.1 NVM Platform and Baseline

We evaluate three different system configurations to understand where the performance is limited (see Figure 7.1). In the *networked system evaluation*, client machines send requests at high rate to a "backend" storage node. The backend node translates these requests into reads to the PCIe SSD emulator, waits for the results, and sends replies back to the clients. The *backend datastore evaluation* measures only the backend storage node and local datastore software. Finally, the *storage-only evaluation* omits the datastore software and sends storage queries from a stripped-down benchmark application designed to elicit the highest possible performance from the SSD emulator.

The backend storage node is a typical fast server machine with two 4-core Intel Xeon X5570 CPUs operating at 2.933 GHz with hyperthreading disabled. It uses an Intel X58 chipset, contains 12GB of DRAM, and is attached to the network using an Intel 82575EB 1Gbps on-board network interface.

The memory-mapped, direct-access SSD emulator does not require system calls, avoiding the associated high overhead for reads and writes; instead, the userspace software interface provides `read()`- and `write()`-like interfaces. These interfaces translate reads and writes into memory copies and commands issued directly to the PCIe-attached SSD em-

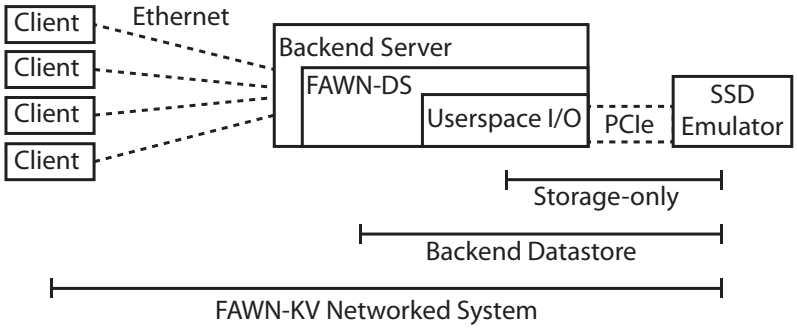


Figure 7.1: Benchmark Scenarios

ulator. The device uses four independent DMA engines for read and write commands, requiring that at least 4 different threads simultaneously access the device for full performance.

Our aim is to use the SSD emulator to understand how applications and system software must adapt in order to support the tremendous recent advances in storage speeds. We do not use the emulator to emulate a specific device technology with particular latencies, wear-out, or access peculiarities (such as erase blocks). We instead use a storage layer, FAWN-DS, which has already been optimized to write sequentially using a log-structured layout. We believe that the performance-related aspects of our work will generalize to future NVMs such as phase-change memory (PCM), but device-type-specific layout optimizations beyond the scope of this work will likely be necessary to make the best use of an individual technology.

The SSD emulator provides 1.8 million I/O operations per second (IOPS) when accessed directly via a simple test program (leftmost bar in Figure 7.2). Microbenchmarks using FAWN-DS, our log-structured, local key-value datastore application, achieve similar performance. We measure the throughput of looking up random key-value pairs with 512B values. Each lookup requires a hash computation, a memory index lookup, and a single read from the underlying storage device. These results show that a *local* key-value datastore can saturate the SSD emulator, despite having to perform hashing and index lookups. A single key-value pair retrieval takes 10 microseconds, on par with the fastest NVM emulator platforms available today [34].

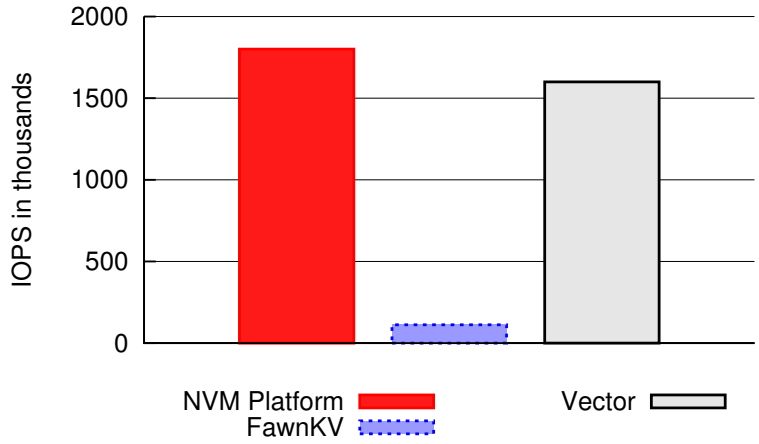


Figure 7.2: Local benchmark demonstrates that the NVM platform is capable of 1.8M IOPS, while FAWN-KV achieves an order-of-magnitude worse throughput. Using vector interfaces provides approximately 90% of device capability for network key-value queries.

7.1.2 FAWN-KV Networked System Benchmark

The FAWN-KV benchmark is an end-to-end benchmark. The server communicates with 25 open-loop, rate-limited client load generators (enough clients to ensure they are not the bottleneck). The middle bar in Figure 7.2 shows that the networked system is an order of magnitude slower than the purely local datastore, achieving only 112,000 IOPS. Even a highly optimized networked key-value storage system designed for current SSDs cannot take advantage of the performance capabilities of next generation SSD systems.

Understanding this large performance gap requires breaking down the components of FAWN-KV. FAWN-KV builds three layers on top of FAWN-DS: 1) networking, 2) RPC, and 3) the associated queues and threads to make parallel use of flash using a staged execution model similar to SEDA [134]. These additional components are responsible for the significantly reduced throughput since FAWN-DS alone can saturate the device capability; we further tested FAWN-KV using a "null" storage backend that returned a dummy value immediately, which only modestly improved throughput.

FAWN-KV uses Apache Thrift [10] for cross-language serialization and RPC. Each key-value request from the client requires protocol serialization and packet transmission; the backend receives the request and incurs a network interrupt, kernel IP and TCP processing, and protocol deserialization before it reaches the application layer; these steps must

be repeated for each response as well. These per-request computations are one source of additional overhead.

To amortize the cost of these per-request computations, the rest of this chapter describes the implementation, evaluation and tradeoffs of using vector interfaces to storage and RPC systems. As we build up to in this chapter, a system design that pervasively uses vector interfaces improves performance from 112,000 IOPS to 1.6M IOPS, shown as the rightmost bar in Figure 7.2, a factor of 14 improvement in performance that achieves nearly 90% of the capability of the NVM platform.

7.2 Vector Interfaces

Given that a state-of-the-art cluster key-value store cannot provide millions of key-value lookups per second (despite underlying storage hardware that can), there are three mostly-orthogonal paths to improving its throughput: Improving sequential code speed through faster CPUs or code optimization; embracing parallelism by doling out requests to the increasing number of cores available in modern CPUs; and identifying and eliminating redundant work that can be shared across requests.

In this work, we focus on only the last approach. Sequential core speeds show no signs of leaping forward as they once did, and sequential code optimization is ultimately limited by Amdahl's law [34]—and, from our own experience, optimizing code that spans the entire height of the kernel I/O stack is a painful task that we would prefer to leave to others. Although improving parallelism is a ripe area of study today, we avoid this path as well for two reasons: First, we are interested in improving system *efficiency* as well as raw throughput; one of the most important metrics we examine is IOPS per core, which parallelism alone does not address. Second, and more importantly, the vector-based interfaces we propose can themselves degrade into parallel execution, whereas a parallel approach may not necessarily yield an efficient vector execution.

Vector interfaces group together similar but independent requests whose execution can be shared across the vector of work. Doing so allows a system to eliminate redundant work found across similar requests and amortize the per-request overheads found throughout the stack. By eliminating redundant work and amortizing costs of request execution, we can significantly improve throughput as measured by IOPS as well as efficiency as measured by IOPS/core.

Vector interfaces also provide an easy way to trade latency for improved throughput and efficiency by varying the width of the vector. Moving to storage devices that support higher throughput can be as simple as increasing the vector width at the cost of higher latency.

We focus on two types of explicit vector interfaces in this work: 1) **Vector RPC interfaces**, aggregating multiple individual, similar RPCs into one request, and 2) **Vector storage interfaces**, or vector-batched issuing of I/O to storage devices.

7.2.1 Vector RPC Interfaces

Vector RPC interfaces batch individual RPCs of the same type into one large RPC request. For example, `memcached` provides programmers a multiget interface and a multiget network RPC. A single application client can use the multiget *interface* to issue several requests in parallel to the `memcached` cluster, improving performance and reducing overall latency. The multiget *RPC* packs multiple key-value get requests into one RPC, reducing the number of RPCs between an application client and a single `memcached` server, resulting in fewer network interrupts and system calls, and reduced RPC processing overhead.

7.2.2 Vector Storage Interfaces

Vector storage interfaces specify vectors of file descriptors, buffers, lengths, and offsets to traditional interfaces such as `read()` or `write()`. They differ from the current "scatter gather I/O" interfaces `readv()` and `writev()`, which read or write only sequentially from or to a single file descriptor into several different buffers, whereas vector storage interfaces can read or write from random locations in multiple file descriptors.

Our proposed vector storage interfaces resemble the `readx()`/`writex()` POSIX extension interfaces, which were designed to improve the efficiency of distributed storage clusters in high-performance computing. The differences are twofold: First, gaining the efficiency we seek requires pushing the vector grouping as far down the storage stack as possible—in our case, to the storage device itself. Second, we emphasize and evaluate the synergistic benefits of comprehensive vectorization: as we show in our evaluation, combining network and storage vectorization provides a large boost in throughput without imposing extra latency for batching requests together (the price of batching, once paid, is paid for all subsequent vector interfaces).

Storage devices today read and write individual sectors at a time. A future device supporting multiread or multiwrite takes a vector of I/O requests as one command. In addition to saving memory (by avoiding duplicated request headers and structure allocation) and

CPU time (to initialize those structures, put more individual items on lists, etc.), a major benefit of these vector interfaces is in drastically reducing the number of storage interrupts. A multi-I/O operation causes at most one interrupt per vector. Interrupt mitigation features found on modern network interface cards share these benefits, but multi-I/O provides several better properties: First, there are no heuristics for how long to wait before interrupting; the device knows exactly how many requests are being worked on and interrupts exactly when they are complete. Second, it is less likely to unintentionally delay delivery of a message needed for application progress—because the application itself determined which requests to batch.

7.3 Vector Interfaces to Storage

We begin by describing the implementation and benefit of vector interfaces to storage, showing that they can help systems match the potential throughput of high-speed SSDs for both asynchronous and synchronous access.

7.3.1 Vector Interface to Device

A storage device supporting a vector interface must implement a single I/O storage command containing multiple, individual and independent requests to storage.

Implementation Details. Current I/O stacks do not contain commands to submit multiple I/Os to a device in a single command. Although hosts can send up to 31 outstanding I/Os to SATA devices using Native Command Queuing (NCQ), these devices process each I/O independently and generate a separate interrupt for each submitted I/O.

Instead, we access the non-volatile memory emulator platform described above using a direct userspace interface. The userspace software interface provides `read()`- and `write()`-like interfaces similar to POSIX. A read or write command prepares an I/O structure containing a buffer, offset, and length as arguments, appends that I/O structure to a submission queue, and "rings a doorbell" using DMA commands to notify the device that a request is ready to be processed. The device processes the request and uses DMA to transfer the result back, similarly ringing a doorbell to signal the host that the request is complete and available in a completion queue.

Our proposed `read_vec()` and `write_vec()` interfaces take multiple buffers, offsets, and lengths as arguments and issue the requests to the device in one large batch. When

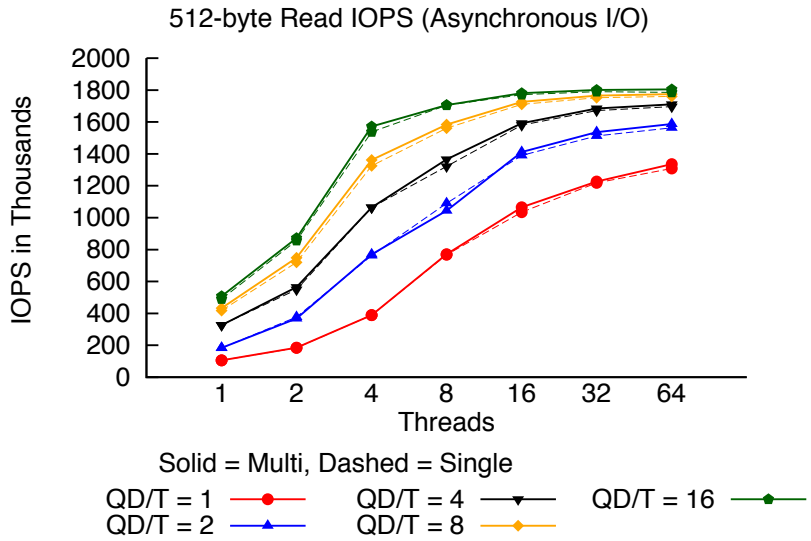


Figure 7.3: 512 Byte read throughput comparison between "single I/O" interface (dashed line) and "multi-I/O" interface (solid line).

delivered in a large batch, the software running on the emulator processes each individual command in submitted order, transfers the results into the host's userspace memory using DMA, and generates a single interrupt to the host only after all of the commands in the vector are complete.

Benchmark: The benchmark in this section measures the raw capability of the emulator platform using asynchronous I/O, which allows a single thread to submit enough outstanding I/Os to keep the device busy. To understand its capabilities, we vary queue depth, thread count, and vector width, whose definitions are as follows:

1. **Queue depth per thread (QD/T)**: The number of asynchronous, outstanding I/O requests sent by one thread. For single-I/O, an interrupt is generated for each individual request, and for each response, one new I/O can be issued.

2. **Thread count**: The number of independent user threads issuing I/Os to the device.

3. **Storage vector width**: The number of I/Os for which one interrupt notifies the completion of the entire vector. For our multi-I/O experiments in this section, the vector width is always set to half of the QD/T value to ensure the device is busy processing commands while new requests are generated.

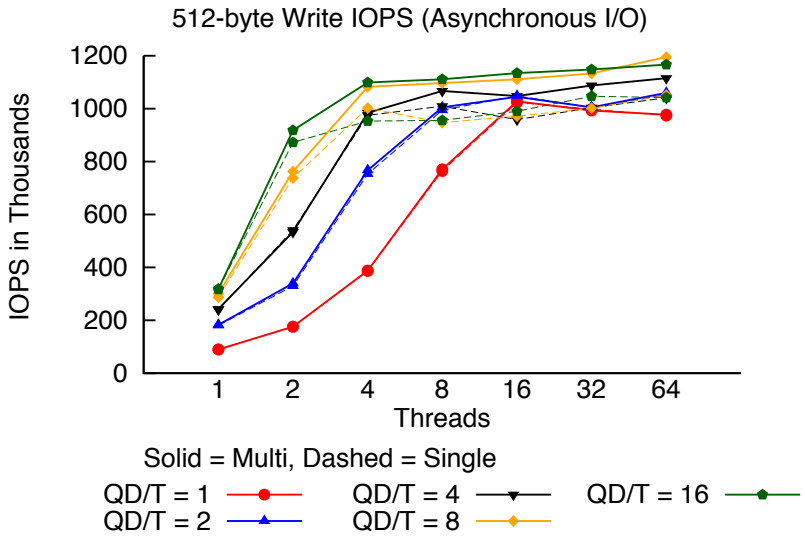


Figure 7.4: Throughput of 512 B single-I/O vs. multi-I/O writes.

Figures 7.3 and 7.4 compare the performance of vector and single read for a variety of thread counts and queue depth/multiread settings. Error bars are omitted because the variance is under 5% across runs. The solid line shows the experiment where vector interfaces are used, and the dashed line shows when a single I/O is posted at a time (and one interrupt is returned for each).

The emulator platform contains 4 DMA engines. Peak throughput requires at least 4 independent threads—each uses one DMA channel regardless of the number of asynchronous I/Os it posts at once. Saturating the device IOPS further requires maintaining a high effective queue depth, either by having a high queue depth per thread value or by spawning many independent threads. Prior work has demonstrated that maintaining an I/O queue depth of 10 is required to saturate the capabilities of current SSDs [102]; our results suggest that this number will continue to increase for next generation SSDs.

Multiread and single I/O read throughput are similar because the read throughput is limited by the hardware DMA capability, not the CPU. In contrast, multiwrite improves performance over single writes, particularly at high thread counts and high queue depth. For example, for a queue depth of 16, single I/O write performance remains between 900K and 1M IOPS, whereas multiwrite performance reaches approximately 1.2M IOPS, an improvement of nearly 20%.

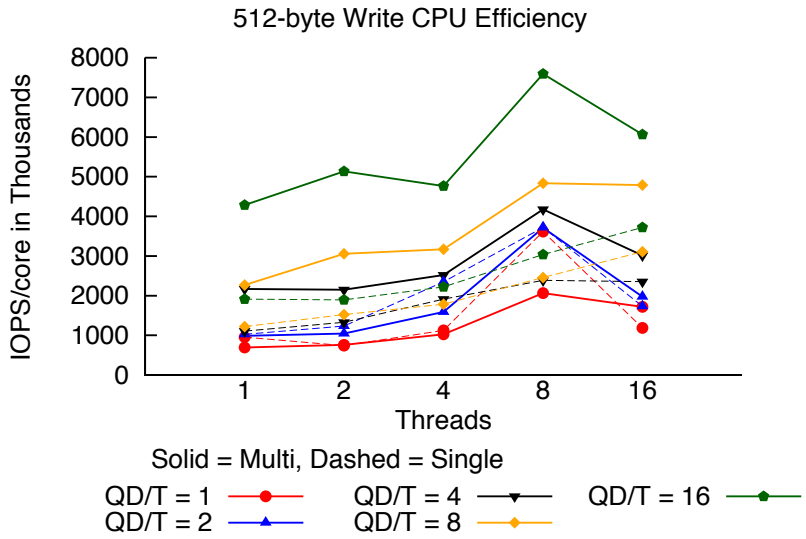


Figure 7.5: Measurement of I/O efficiency in IOPS per core. A multi-I/O interface can reduce overall CPU load by a factor of three.

Raw throughput, however, is a potentially misleading indicator of the benefits of vector interfaces. CPU efficiency, measured by IOPS/core, paints a very different picture. We calculate efficiency by dividing the IOPS performance by CPU utilization reported in `/proc/stat`.

Figure 7.5 shows that at high vector widths, multiread and multiwrite are between 2–3x more efficient than using single I/O. A large vector width reduces substantially the number of interrupts per I/O, allowing the CPU to devote less time to interrupt handling. As we demonstrate in more detail later, this reduced overhead makes many more cycles available to application-level code: The overall system performance improves significantly more than the throughput-only microbenchmark might suggest.

7.3.2 Vector Interfaces to Key-value Storage

Next, we describe using vector interfaces to access a local key-value datastore on the back-end node.

The log-structured FAWN-DS key-value datastore exports a synchronous `put(string key, string value)`/`get(string key)` interface. We added `get(vector<string>`

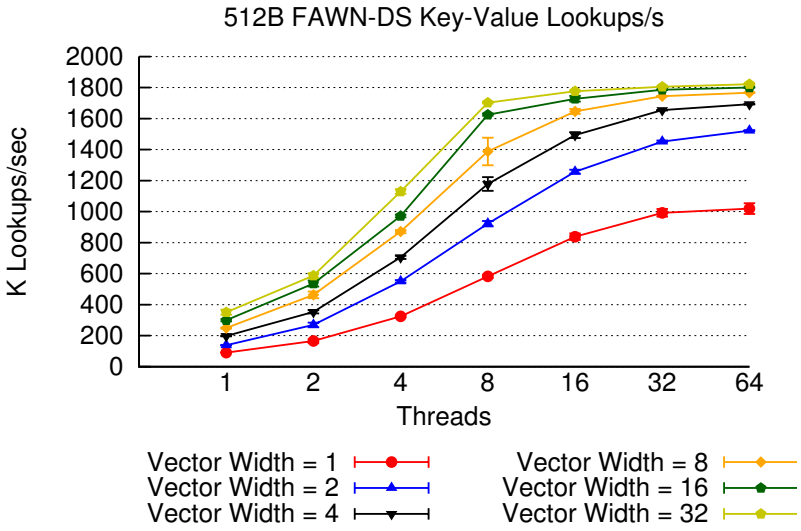


Figure 7.6: 512B synchronous read throughput through FAWN-DS using synchronous I/O interface. Multi-I/O allows applications using synchronous I/O to maintain a high queue depth at the device without requiring hundreds of threads.

`key)` which returns a vector of key-value results. For each lookup, FAWN-DS hashes the key, looks up an entry in a hashtable, and `read()`s from storage using the offset from the hashtable entry.

The vector version requires taking in a vector of keys and calculating a vector of potential offsets from the hashtable in order to issue a `multiread()` for all keys. One feature of FAWN-DS complicates vectorization: To conserve memory, it stores only a portion of the key in the hashtable, so there is a small chance of retrieving the wrong item when the key fragments match but the full keys do not. As a result, the keys being looked up may each require a different number of `read()` calls (though most complete with only one).

The vector version of `get()` must inspect the multiread result to identify whether all entries have been retrieved; for entries that failed, vector `get()` tries again, starting from the last hash index searched. Vector support required significant changes to the code structure to manage this state, but the changes added fewer than 100 additional LoC.

Benchmarking multiget in FAWN-DS: One important difference between the FAWN-DS benchmark and the earlier device microbenchmark is that the FAWN-DS API is *syn-*

chronous, not asynchronous. This means that when FAWN-DS reads from the storage device, the reading thread blocks until the I/O (or I/Os, in the case of multiread) complete. Many applications program to the synchronous I/O model, though some systems support native asynchronous I/O though `libaio`-like interfaces.

Figure 7.6 shows the performance of 512B key-value lookups with an increasing number of threads. Single I/O submission reaches a maximum of about one million key-value lookups per second at 32 threads, whereas multireads of size 16 or 32 can provide approximately 1.8 million key-value lookups per second at 32 threads. In contrast to the asynchronous I/O direct device benchmark, which showed no benefit for reads, multiread doubles key-value lookup throughput. For synchronous I/O, using vector interfaces allows an application to issue multiple I/Os per thread to maintain the high effective queue depth required to saturate the device, whereas the single I/O interface can only maintain as many outstanding I/Os as there are threads available. Beyond 64 threads, single I/O performance drops.

We note that an application capable of issuing multireads should theoretically be structured to also perform asynchronous I/O as the application does not need to issue I/Os one a time. However, the simplicity of synchronous I/O is appealing to many developers, and providing a "bulk I/O" interface can achieve many of the benefits of asynchronous I/O for developers issuing multiple independent reads (e.g., in a for loop). Steere's "dynamic sets" [120] proposed unordered aggregate I/O interfaces for such set iterator patterns, showing that the system can choose an optimal pattern of access that ordered access to I/O would prevent.

Vector interfaces and latency: Vector batching has a complex effect on latency. Batching waits for multiple requests before beginning execution. It similarly does not complete until the last request in the batch completes. Both of these effects add latency. On the other hand, it reduces the amount of work to be done by eliminating redundant work. It also achieves higher throughput, and so on a busy server reduces the time that requests spend waiting for others to complete. These effects reduce latency, particularly at high load.

Figure 7.7 shows the how batching modestly increases latency at low load. The bottom left point in the graph shows the minimum latency for retrieving a key-value pair using the NVM platform, which is approximately 10μs. Doubling the vector width does not double the latency for the entire batch at low load. For example, for one thread, a vector width of 32 I/Os returns in 80μs, which is only a factor of eight worse than for a vector width of 1. In fact, regardless of the number of threads, the vector width of 32 always has 8x the latency of a single request. The next section evaluates latency versus (high) load in the networked

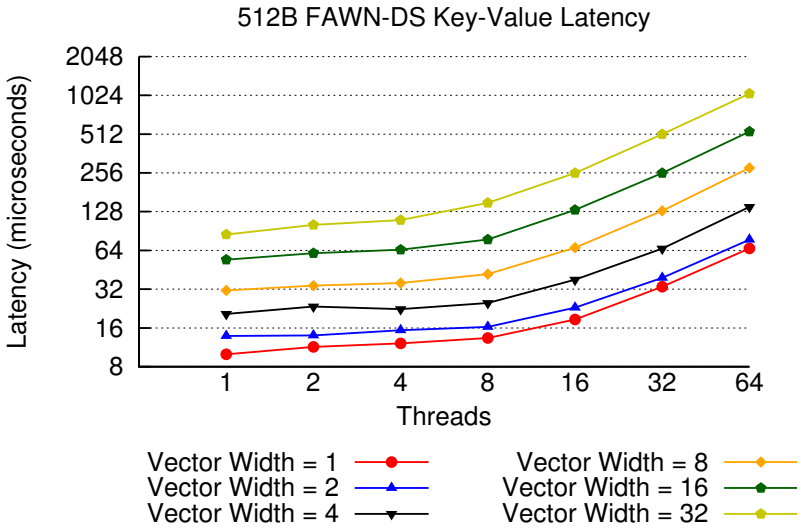


Figure 7.7: Latency of vector I/O as a function of vector width and thread count. High vector widths can hide latency: a vector width of 32 has only eight times higher latency than a vector width of 1.

system, where increases in vector width significantly increase throughput with almost no additional increase in latency.

7.4 Vector Interfaces to Networked Key-Value Storage

Exporting vector interfaces to non-volatile memory devices and to local key-value storage systems provides several throughput, latency, and efficiency benefits. Next, we examine vector interfaces to RPCs and their interaction with the vector interfaces provided by the storage device and key-value storage APIs.

7.4.1 Experimental Setup

The experiments in this section all involve queries made over the network. A cluster of Intel Atom-based machines generates the query workload, using as many nodes as needed to ensure that the backend node, not the load generators, are the bottleneck. The experiments described here use between 20 and 30 query generation nodes. Queries are sent in an *open*

loop: The query generators issue key-value queries at a specific rate regardless of the rate of replies from the backend.[1] We run each experiment for approximately 20 seconds, measuring the throughput of key-value queries at the backend and logging the end-to-end latency of each key-value request at every client in order to report the median latency.

The FAWN-KV software running on the backend node uses 8–16 independent threads to post I/Os to the NVM emulator, which is enough to saturate the DMA engines of the platform.

Asynchrony: The benchmark utility issues asynchronous key-value requests because synchronous requests would be bottlenecked by the end-to-end network latency of our environment. Although some applications may desire a synchronous interface, an asynchronous interface can benchmark the limits of the backend system using a relatively small number of clients and threads.

Request Size: This evaluation inserts and retrieves 4-byte values (prior sections used 512 byte values), because transferring larger values over the network quickly saturates the server's 1Gbps network. Internally, the server still reads 512 bytes per request from storage. This change will slightly over-state the relative benefits of vector RPC (below): *Requests* are unchanged, internal storage I/Os are unchanged, but the amount of time spent copying data into responses—something not helped greatly by network vectorization—will decrease. We believe, however, that these effects are modest.

Multiget and Multiread: Our evaluation in this section measures the impact of multiread, multiget, and their combination when applied to FAWN-KV running on top of the emulator platform with multiread support. We use the term *multiread* to describe vector I/Os to the storage device, and we use the term *multiget* to describe vector RPCs over the network. In figures, we refer to a multiget vector width of N as "GN" and a multiread vector width of M as "RM". We use multiread as a proxy for multiwrite, and multiget as a proxy for other types of RPCs such as multiput.

Reporting individual points: When comparing maximum throughput values from different lines, we compare the maximum throughput achievable at a median latency below 1ms and 99%-ile latency below 10ms. This definition chooses the highest throughput value that does not suffer from extreme latency variability near the knee of the throughput vs. latency curves.

[1]The benchmark client uses three threads: one fills a token bucket at the specified rate, another removes tokens to issue *asynchronous* requests to the backend device; and the final thread receives and discards the responses.

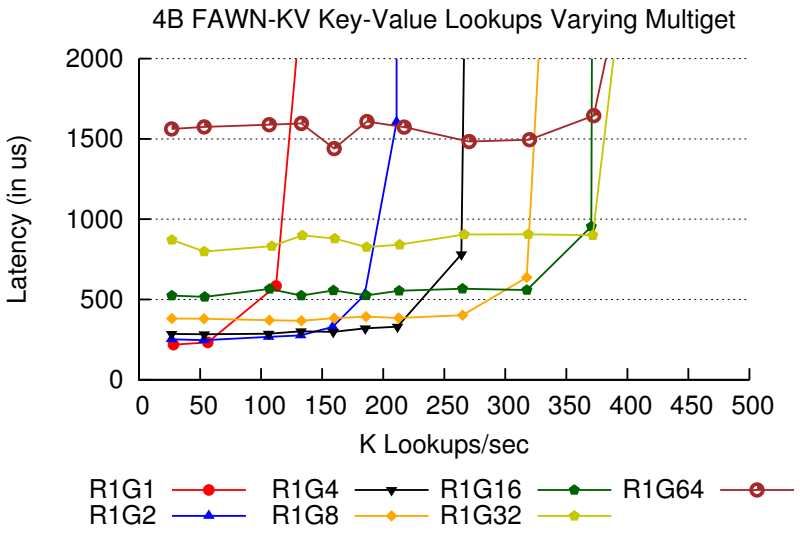


Figure 7.8: Networked 4-byte key-value lookup throughput vs. latency as a function of the multiget width.

7.4.2 Results

We begin by measuring baseline throughput and latency without vectors. The R1G1 line in Figure 7.8 shows the throughput versus latency as the query load increases. Because the measurement is open-loop, latency increases as the system approaches full capacity.

At low load, the median latency is 220μs. As shown previously, the NVM device access time at low load is only 10μs, so most of this latency is due to network latency, kernel, RPC, and FAWN-KV application processing. Without vector interfaces, the system delivers 112K key-value lookup/sec at a median latency of 500μs.

Multiget (Vector RPC) Alone

Network key-value throughput using standard storage and RPC interfaces is over an order of magnitude lower than device capability and eight times lower than local I/O performance. At this point, the overhead from I/O, local datastore lookups, RPC processing, and network I/O has greatly exceeded the CPU available on the backend node. We begin by examining how much the Vector RPC interface can help reduce this overhead.

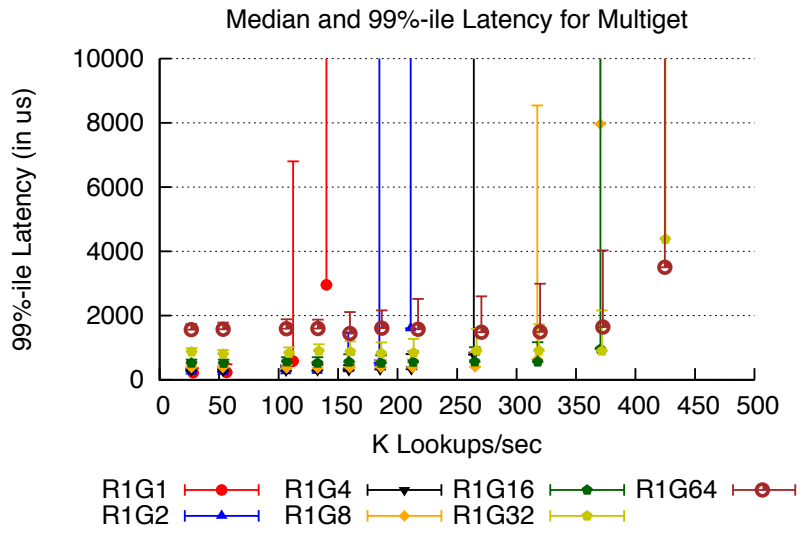


Figure 7.9: Variability between median and 99%-ile latency when using multiget. Top bar above point (median) depicts 99%ile latency. Variability increases slightly with throughput.

The remaining lines in Figure 7.8 show load versus latency for larger multiget widths. Multiget improves the throughput (at reasonable latency) from 112K key-value lookups/sec to 370K for a multiget width of 16. Increasing the vector width further increases latency without improving throughput.

Vector RPCs increase peak throughput more than they increase latency. For a multiget width of 4 (G4), the latency at 50,000 key-value lookups/sec is 300μs compared to 220μs for single get. But the maximum throughput achieved for G4 is roughly 212,000 lookups/second, over twice the throughput of G1 at a latency of just 340μs. The additional latency comes from 1) the time to assemble a batch of key-value requests on the client, 2) the time to enqueue and dequeue all requests and responses in a batch, and 3) the response time for all I/Os in a batch to return from the NVM platform.

99%-ile Latency: While median latency is a good indicator of typical performance, users of key-value systems may prefer a bound on 99%-ile latency. Figure 7.9 shows the difference between median and 99%-ile latency as a function of throughput when using multiget, with connecting lines omitted for clarity. The difference between median and 99%-ile (for a given multiget width) grows slowly as throughput increases. For most points below the knee of that curve, the difference is often only 20% at low load and no more than

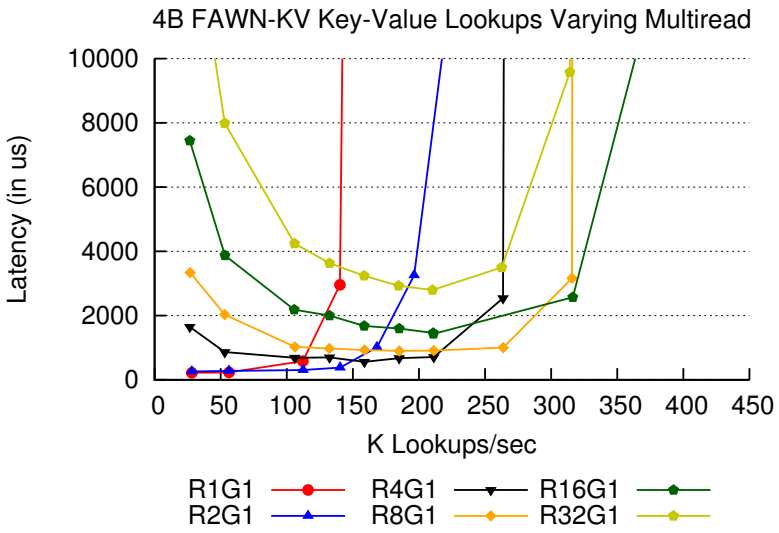


Figure 7.10: Networked 4-byte key-value lookup throughput vs. latency as a function of the multiread vector width. Multiread can create batches of network activity which reduce interrupt and packet rate to improve performance.

a factor of two at high load. Our data show similar results when varying multiread or when combining multiget and multiread together.

Despite the $2\times$ throughput improvement compared to single get throughput, throughput is still far below the 1.8M key-value lookups that the device can provide. Although multiget improves performance, per-RPC processing is not the only bottleneck in the system.

Multiread (Vector Storage) Alone

Next, we hold the multiget width at 1 and vary the multiread width to understand to what degree multiread can improve performance without multiget. Varying multiread alone is useful for environments where only local changes to the storage server are possible.

Figure 7.10 plots throughput versus latency for multiread widths of 1, 2, 4, ..., 32. Grouping 8 reads together (from different RPCs) improves throughput from 112K to 264K key-value lookups/sec. Multiread widths beyond 8 do not increase throughput without a corresponding large increase in median latency.

The low-load latency behavior is different for multiread than for multiget. At very low load, latency scales linearly with the multiread width, drops rapidly as offered load increases, and then increases as the system approaches peak capacity. This behavior is caused by the queue structure that collects similar RPC requests together: Using a multiread width N, the first key-value RPC that arrives into an idle system must wait for the arrival of $N-1$ other key-value requests from the network before the batch of I/Os are issued to the device. At low load, these $N-1$ other requests enter the system slowly and as load increases, request inter-arrival time shortens and reduces the time the first request in the queue must wait.

Multiread's implicit benefits: To our surprise, multiread both improves the efficiency of client network performance—even without multiget—and increases cache efficiency on the backend. We term these two improvements as *implicit benefits*, in contrast with the explicit design goals of vector interfaces: multiread explicitly reduces the total number of commands sent to the storage device, and correspondingly the number of interrupts received; multi*get* explicitly reduces the number of `send()` and `recv()` system calls, serialization and deserialization overhead for individual RPC messages, and number of packets (when Nagle's algorithm is disabled).

Improved client efficiency: A multiread storage I/O contains multiple key-value responses. Because we use a one-thread-per-connection model, these responses are destined to the same client, and so the RPC handling thread sends these responses closely in time. The client receives a burst of several responses in one packet (due to interrupt coalescing support on the network card), reducing significantly the number of ACK packets sent back to the server. This improves the efficiency and performance of the server because it incurs fewer network interrupts, but this behavior also improves efficiency on clients. In Figure 7.10, the backend server is the bottleneck, but we have also found that enabling multiread on the server can improve performance when the client CPU is bottlenecked by network processing.

Improved cache efficiency: A by-product of organizing requests into queues for multiread is that it improves the backend's cache efficiency, and hence sequential performance. On each get request, the servicing thread performs a `read()`, processes the get RPC, and inserts the request into a queue, repeating N times in a row while keeping requisite data structures and code in cache. When the thread issues the multiread request, the hardware interrupt generated will cause the processor's cache to flush, but many of the structures needed for performing I/O are no longer needed. This better cache locality increases the instructions per cycle and contributes partially to the improvement that multiread provides over single reads in the networked server evaluation.

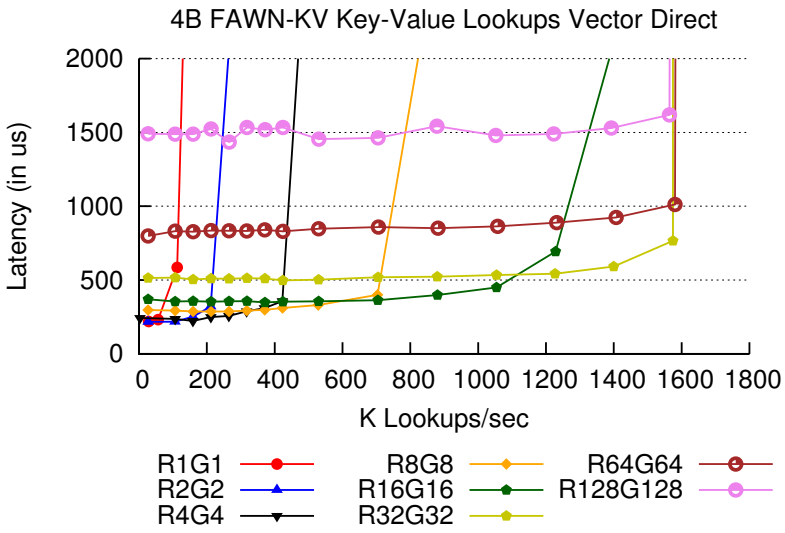


Figure 7.11: Throughput vs. latency for matched vector widths. Vector interfaces enable a single server to provide 1.6M key-value lookups per second at a latency below 1ms.

7.4.3 Combining Vector Interfaces

Despite their benefits, multiread and multiget in isolation cannot achieve full system performance of 1.8M key-value lookups per second. Multiread improves networked key-value retrieval by roughly 2x by reducing the storage interrupt load, freeing the CPU for the costly network and RPC processing. Multiget provides a similar 2x improvement by reducing the number of packets sent across the network and the RPC processing overhead. In this section, we show that the two combine synergistically to increase throughput by more than a factor of 10.

When the multiget width is less than the multiread width, a queue structure is required to collect key-value requests from multiple RPCs. We call this implementation the "intermediate queue" pattern. If the multiget width and multiread width are the same, no intermediate queues are necessary, and the backend can execute a multiread directly—we call this implementation the "direct" pattern.

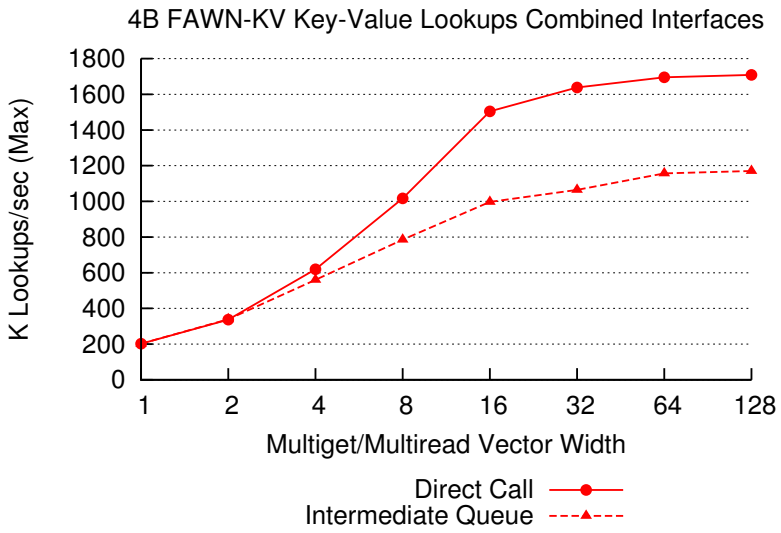


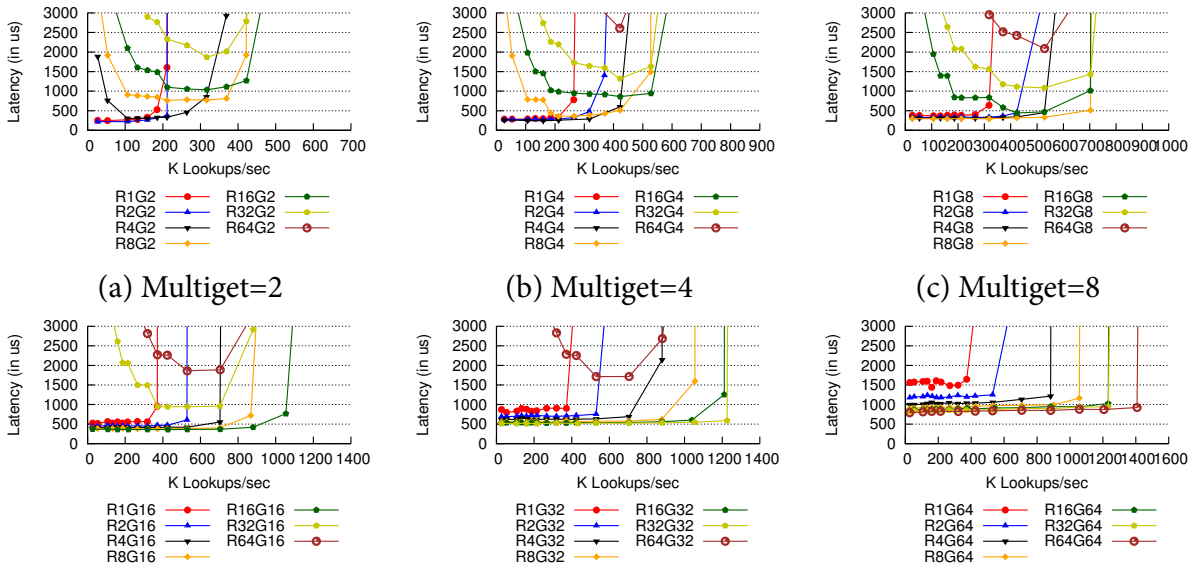
Figure 7.12: Throughput increases as storage and multiget widths increase. Using an intermediate queue between the RPC and storage layer significantly degrades performance at large vector widths.

Matching Multiread and Multiget Widths

The combinations of parameters R and G are numerous; thus, we begin by investigating the performance when the multiget width equals the multiread width. In this case, we use the "direct" pattern that omits intermediate queues.

Figure 7.11 shows the latency versus throughput for equal vector widths using the direct pattern. Combining multiread and multiget provides up to maximum of 1.6M key-value lookups per second from the backend, nearly saturating the SSD emulator, and does so without increasing median latency beyond 1ms. Achieving this only requires vector widths of 32. Because no intermediate queues are required, the latency behavior at low load is identical to that of just using multigets.

The seemingly small overheads of enqueuing and dequeuing requests in the intermediate queue significantly reduce performance at high load. Figure 7.12 shows the peak query throughput with increasing multiget/multiread vector width. The dashed line depicts the performance using an intermediate queue, while the direct call is shown as the solid line. With a queue, the overhead of inserting and removing each get RPC's data limits the per-



(a) Multiget=2 (b) Multiget=4 (c) Multiget=8

(d) Multiget=16 (e) Multiget=32 (f) Multiget=64

Figure 7.13: 4-byte FAWN-KV key-value lookup latency behavior for various settings of multiread and multiget, organized by holding the multiget width constant. The best choice of multiread width depends on both load and the multiget width.

formance to 1.2M key-value IOPS, whereas the direct call implementation performs 1.7M IOPS at peak.

Unequal Vector Widths

Is it ever advisable to set the size of multiget and multiread widths differently? We consider scenarios where the multiget width is fixed at different values as the independent variable and show how varying the multiread width affects throughput and latency. Figure 7.13 shows the throughput versus latency curves for various fixed multiget widths as we change the multiread width. When the multiget width is less than the multiread width, we use an intermediate queue to collect requests together. When the multiget width is greater than the multiread width, we use the direct implementation but issue reads to the device in sizes of the specified multiread width.

For low multiget widths: At low load it is advisable to keep the multiread parameter low. Waiting for the multiread queue to fill up at low load creates long queuing delays

as shown previously. As load increases, however, it is beneficial to increase the multiread width higher than the multiget width because it allows the system to achieve higher rates than otherwise possible. Thus, the best strategy is to scale the multiread width higher as load increases to get the best tradeoff of throughput and latency.

For high multiget widths: Regardless of load, it is always best to match RPC and multiread widths. For example, Figure 7.13(e) shows the results for a multiget width of 32. When multiread is low, issuing each request serially and taking an interrupt for each get request in the multiget batch increases latency significantly. At low load, using a low multiread width nearly doubles median latency, while at high load it is necessary to have a high multiread width to achieve higher throughput. The queue structure required when multiread is greater than multiget, though, reduces the performance benefits having a high multiread width can provide.

In summary, vector interfaces used in isolation improve throughput by amortizing the cost of RPCs and reducing interrupts and packet processing, but still provide only a small fraction of the underlying storage platform's throughput capabilities. By carefully combining both types of vector interfaces, however, we have shown both that such a system is capable of 90% of optimal throughput and also how to set the widths to achieve the best throughput and latency tradeoffs.

7.5 Discussion

7.5.1 When to Use Vector Interfaces

Using vector interfaces at the RPC and storage layers can significantly improve performance for a simple key-value storage system. In this section, we ask: when are vector interfaces generally useful and when should they be avoided?

The principal scenarios where vector interfaces are useful share three properties: The services expose a narrow set of interfaces, exhibiting a high degree of "operator redundancy"; the work being batched together shares common work; and the requests in an initial vector follow a similar path to ensure that vectors are propagated together throughout the distributed system.

Narrow Interfaces: Key-value and other storage systems often export a small number of external interfaces to clients. Under high load, any such system will be frequently invoking the same operators, providing the opportunity to eliminate redundant work found across these similar operations. If the interface is not narrow but the operator mix is skewed to-

wards a common set, then operator redundancy will be high enough that vector interfaces can provide a benefit.

Similarity: The computational similarity found among get requests in a multiget interface allows us to amortize and eliminate redundant work common across independent executions of those requests. StagedDB took advantage of query similarity to improve database performance by eliminating redundant table scans and re-using common data and code references [62, 63]. In contrast, consider a vector "SELECT" interface for SQL used to batch completely different select queries. If the queries in a vector do not operate on the same table or process the resulting data similarly, then there may be few opportunities for optimization because redundancy may not exist. The cost of serializing independent requests that are computationally-unique would outweigh the benefits the small amount of work sharing provides [71].

Vector Propagation: For maximum advantage, vectors of similar work should propagate together through the system. In FAWN-KV, the key-value lookups in a multiget from a client to the backend remain together throughout the lifetime of the operation. A mixed put/get workload, in contrast, would diverge once the request arrives at the backend; the backend would need to inspect the elements in the vector to separate the puts and gets into two separate vector operations. The system may then need to re-combine these two vectors before sending a response, adding further coordination and serialization overhead. Other systems have successfully used the "stages with queues" model to implement reconvergence for graphics pipelines [125]; PacketShader, for example, demonstrated that packet processing does not significantly diverge in execution to erase the benefits that GPU-accelerated vector processing can provide [61].

7.5.2 Using Vector Interfaces at Large Scale

Vector RPCs can be easily used when communicating with a single backend storage server. When the storage system consists of tens to hundreds of machines, each storing a portion of the entire dataset, a client's key requests might not map to the same backend storage server, requiring that the client issue several RPCs each consisting of requests for fewer keys. The ability to issue large multiget RPCs depends on three factors: key naming, access pattern, and the replication strategy.

Key naming: Backend nodes are typically responsible for a subset of the key-value pairs in a distributed storage system; a single storage node serves keys for multiple continuous partitions of the key space, and an index (often a distributed B-tree or a simple map) is queried to map keys to nodes. Most systems structure accesses to key-value storage based

on how keys are named. If the key is a hash of an application-specific name, then a multiget from a single client will contain keys that are randomly distributed throughout the key space, reducing the likelihood that a multiget request can be served by a single backend. If the keys are not hashed, then the application access pattern and key naming policy determines how a set of key requests in a multiget map to backends.

Access pattern: Some applications, like webmail services, will exhibit a high degree of request locality: users access and search only over their own mail. If keys corresponding to a particular user are prefixed with a unique user ID or some other identifier that specifies locality in the key space, then requests for multiple keys might be served by a small number of backend nodes, allowing the multiget RPC to maintain the original vector width. For other applications, the access pattern might not exhibit locality: Social network graphs can be difficult to partition well [60], so requests from a single client may need to access a large fraction nodes in the backend storage system.

Replication: Data in large-scale key-value storage systems are often replicated for fault-tolerance, and also for higher performance or load balancing. Because replication provides multiple choices from which to obtain the same data, it is possible to use replication to reduce the total number of nodes one needs to contact when using vector RPCs: a multiget for two keys that map to different physical nodes can be satisfied by a single node if it contains a replica for both keys.

However, the type of replication used determines whether replication can maintain the benefit of using vector RPCs for higher performance. For example, if a system uses a hot-spare replication scheme (a replica node contains the exact same data as its master), then two keys that map to different masters will not map to the same physical node. On the other hand, a mapping system like we used in FAWN-KV (overlapping chains using virtual nodes) slightly increases the probability that two keys can be served by the same node and allows a client to issue a single multiget RPC to retrieve both key-value pairs.

Application-specific replication schemes can do far better in ensuring that a client's requests need only hit one server. For example, SPAR is a middleware layer that partitions and replicates data for online social networks to ensure that a user's data (including one-hop neighbors) exists on a single server, while simultaneously minimizing the overhead of replication [105].

Simulation of worst-case pattern: To understand how a random access workload interacts with vector RPCs and replication, we use a Monte Carlo simulation to find the expected average width of vector RPCs as a key-value storage cluster scales. We assume that keys map uniformly at random to nodes (including replicas of keys). We then fix the *desired* vector width at 32, vary the number of servers N that data is distributed evenly across, and calcu-

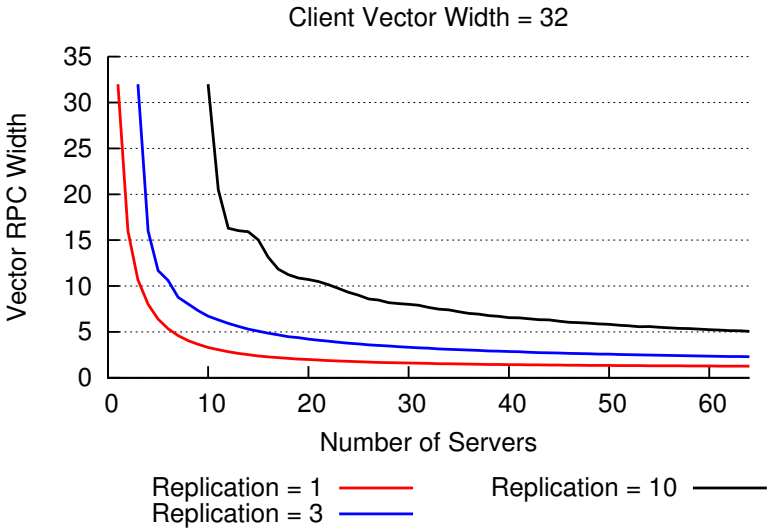


Figure 7.14: Simulation results showing the expected vector width of multiget RPCs, with the client making requests for 32 random keys, as a function of cluster size and replication factor. Data is assumed to be distributed evenly across all nodes, and replica nodes are chosen randomly (without replacement).

late the minimum number of nodes a client must contact to retrieve 32 random keys (out of a large set of keys)[2]. Each experiment is run 1000 times and we record the average minimum node count. We then divide the desired vector width (32) by this number to calculate the average expected vector RPC width.

Figure 7.14 shows how increasing the number of servers affects the average vector RPC width. For a replication factor of 1, the vector width starts at 32 for 1 server, reduces to 16 for 2 servers, etc. This line follows the formula $f(x) = \frac{32}{x-(x\times(1-\frac{1}{x})^{32})}$, which can be derived from a balls and bins analysis assuming 32 balls and x bins.

Increasing replication allows a client to maintain a higher vector width as the number of backend servers increases, because the additional choice provided by replication allows the client to pick a smaller set of nodes to cover all keys. For example, with a replication factor of 10 and 128 servers, a client can expect to contact minimum of $\sim$9 servers to retrieve

[2]This requires calculating a minimum set cover where the universe is the 32 keys requested and the sets are the N mappings of server number to keys.

the 32 random keys in the vector request, resulting in an average vector width of ∼4 key requests sent to each server.

While random replication can improve the ability to maintain higher vector RPC widths for random access patterns, these simulation results highlight the need to cluster keys intelligently (e.g., application-specific locality hints or clustered replication techniques [105]) to ensure that vectors can be propagated with maximum efficiency.

7.5.3 How to Expose Vector Interfaces

Vector interfaces to *local* key-value storage systems and devices can be implemented without requiring global modifications, whereas vector interfaces to RPC requires coordinated changes. This necessitates the question: Where and how should these vector interfaces be exposed to other components in a distributed system?

One option, as we have chosen, is that Vector RPC interfaces are exposed directly to clients interacting with key-value storage systems. For example, a social networking application may need to gather profile information given a large list of friends, and it may do so by issuing a multiget corresponding to key-value requests for each friend, instead of requesting data serially one friend after another. Multiget support in systems like `memcached` and Redis suggest that applications today already have opportunities to use these vector interfaces.

Alternatively, a client library can implicitly batch synchronous RPC requests originating from different threads into a single queue, issuing the vector request once a timeout or a threshold has been reached. Unfortunately, this creates the same opaque latency versus throughput tradeoffs found in TCP's Nagle option. In contrast, the advantage of an explicit synchronous vector interface is that the program cannot advance until the entire vector is complete, giving full control to the application developer to decide how to trade latency for throughput.

Some cluster services separate clients from backend infrastructure using load balancers or caching devices. Load balancers can coalesce requests arriving from different clients destined to the same backend, batching the requests in vector RPCs to the backend. Unfortunately, doing so can create the same poor low-load latency behavior we observe when the multiread factor is much greater than the multiget factor: the degree of coalescing should depend on load to avoid this behavior. In addition, some clients may require very low latency, whereas others may desire high throughput; vectorizing requests from different clients unfairly penalizes the former to satisfy the latter if coalesced at an intermediary.

7.5.4 Vector Network Interfaces

We have discussed vectorizing storage and RPC interfaces, but we did not need to vectorize the network socket layer interfaces because our TCP socket access patterns already work well with existing Ethernet interrupt coalescing. Our structuring of threads and queues ensures that we write in bursts to a single stream at a time. As we showed in Section 7.4.2, when several packets arrive over the network closely in time, Ethernet interrupt coalescing will deliver multiple packets to the kernel with a single interrupt (which, in turn, often triggers only one outgoing ACK packet from the kernel for the combined sequence range). The application `recv()` will process this larger stream of bytes in one system call rather than one for each packet. On the sending side, Nagle's algorithm can coalesce outbound packets, though the application must still incur a mode switch for each system call.[3]

However, vector networking interfaces (such as `multi_send()`, `multi_recv()`, or `multi_accept()`) can be useful in other environments when code repeatedly invokes the same type of network function call with different arguments close in time: for example, an event loop handler sending data might frequently call `send()` for multiple sockets, and amortizing the cost of this system call across multiple connections may be useful. Many event-based systems fall under this pattern, including systems build on `libevent` [104].

Vector network interfaces therefore can be useful depending on the structure of network event processing in a key-value storage system. For a highly-concurrent server for which event-based systems use significantly lower memory, we believe that implementing vector interfaces to the network might be necessary.

7.5.5 Vector Interfaces for Vector Hardware

Programming to vector interfaces often introduces many for loops in code. For example, preparing the multiread I/O command to the underlying storage device requires creating and populating a structure describing the offset and size of each individual I/O. This creates a unique opportunity to better use vector hardware available on emerging server platforms.

As an example, the multiget/multiread code path in the backend server contains 10 for loops that iterate over vectors whose width matches the multiget factor. A significant component of the increased latency at high vector widths comes from having to sequentially iterate through these vectors. SSE hardware today is capable of operating on 256-bit registers and GPU hardware is capable of much wider widths. Exporting the computations within

[3]The RPC package we use (Apache Thrift) explicitly disables Nagle's algorithm by default to reduce the RPC latency added by batching outgoing packets in the kernel.

these simple loops to specialized vector hardware instead of general-purpose cores could dramatically reduce the latency at larger batch sizes. This is not merely wishful thinking: developers today have tools to harness vector hardware, such as CUDA and Intel's SPMD compiler (`http://ispc.github.com/`).

7.6 Related Work

Vector interfaces: Prior systems have demonstrated the benefits of using an organization similar to vector interfaces. For example, "event batches" in SEDA [134] amortize the cost of invoking an event handler, allowing for improved code and data cache locality. Vector interfaces also share many common benefits with Cohort Scheduling [80] such as lower response time under certain conditions and improved CPU efficiency. However, Cohort Scheduling benefits only from the implicit batching that scheduling similar work in time provides, whereas vector interfaces can completely eliminate work that need not be executed once per request.

Batched execution is a well-known systems optimization that has been applied in a variety of contexts, including recent software router designs [48, 61, 86] and batched system call execution [106, 108, 119]. Each system differs in the degree to which the work in a batch is similar. The multi-call abstraction simply batches together system calls regardless of type [108], whereas FlexSC argues for (but does not evaluate) specializing cores to handle a batch of specific system calls for better cache locality. Vector interfaces target the far end of the spectrum where the work in a vector is nearly identical, providing opportunities to amortize and eliminate the redundant computation that would be performed if each request in the vector were handled independently [131]. The Stout [88] system demonstrates the throughput and latency benefits of batching key-value queries at high load and uses an adaptation algorithm to control the RPC vector width based on current latency. Using this approach can allow a system to operate at the optimal convex-hull of latency vs. throughput. Orthogonally, we show that propagating work in vectors is crucial to improving throughput and latency for a high-performance storage server.

In High Performance Computing, interfaces similar to our vector interfaces have been developed. Examples include Multicollective I/O [92], POSIX listio and `readx()`/`writex()` extensions. These extensions provide batches of I/O to an intermediate I/O director, which can near-optimally schedule requests to a distributed storage system by reordering or coalescing requests. The existence of these interfaces suggests that application designers are willing and able to use explicit vector interfaces to achieve higher performance.

Key-value stores: In recent years, several key-value storage systems optimized for flash storage have emerged to take advantage of flash storage and non-volatile memory improvements. BufferHash [14], SkimpyStash [45], FlashStore [44], and SILT [83] are examples of recent key-value systems optimized for low-memory footprint deduplication or Content Addressable Memory systems. These systems evaluate performance on a prior generation of SSDs capable of a maximum of 100,000 IOPS, whereas our work looks ahead to future SSD generations capable of much higher performance. Their evaluations typically use synthetic benchmarks or traces run on a single machine. In contrast, our work demonstrates that achieving high performance for a *networked* key-value storage system is considerably more difficult, and that achieving the performance of local microbenchmarks may require redesigning parts of local key-value systems.

Non-volatile memory uses: Several studies have explored both the construction and use of high-throughput, low-latency storage devices and clusters [11, 34, 101, 118, 132]. Most closely related is Moneta [34], which both identified and eliminated many of the existing overheads in the software I/O stack, including those from I/O schedulers, shared locks in the interrupt handler, and the context switch overhead of interrupts themselves. Our work is orthogonal in several ways, as vector interfaces to storage can be used on top of their software optimizations to yield similar benefits. In fact, our userspace interface to the NVM device begins where they left off, allowing us to explore opportunities for further improvement by using vector interfaces. Finally, we demonstrate that having the capability to saturate a NVM device using local I/O does not imply that achieving that performance over the network is straightforward.

Programming model: Our current implementation of vector interfaces and functions uses simple structures such as queues and vectors but requires programmers explicitly to use the vector interfaces (both in type and in width). In some cases, converting an operation from using a single interface to a multi-interface can be difficult. Libraries like Tame [76] provide novice programmers with basic non-vector interfaces and event-based abstractions, but can execute operations using vectorized versions of those interfaces (assuming that the function being vectorized is side-effect free).

Vector interfaces bear similarity to the "SIMT" programming model used in the graphics community, where computations are highly-parallelizable, independent, but similar in operation. GPU fixed function and programmable shader hardware matches well to these workloads where each functional core performs work in lockstep with potentially hundreds of other threads in a warp [56, 125]. The popularity of CUDA programming suggests that exposing non-vector interfaces to programmers and using vector-style execution for performance-critical sections can provide the best of both worlds, provided that vector interfaces are exposed where needed.

Chapter 8

Vector Interfaces for OS-intensive Workloads

For a distributed key-value storage system, we have demonstrated that pervasive use of vector interfaces are a scalable and effective technique to improve system efficiency. The advantages they provide can be generalized beyond networked key-value systems to OS-intensive systems such as webservers and databases. In this section, we make the case for The Vector Operating System, describing the opportunities for eliminating redundant execution found in many of these OS-intensive services and discussing the challenges in exposing these vector interfaces to more than just storage and RPC systems.

8.1 Background

Over the last decade, computing hardware has promised and delivered performance improvements by providing increasingly parallel hardware. Systems with multi-core CPUs, and GPUs with hundreds of cores have become the norm. The gauntlet has been firmly thrown down at the software community, who have been tasked to take advantage of this increased hardware parallelism.

The operating systems community has largely risen to the challenge, presenting new OS architectures and modifying existing operating systems that enable parallelism for the many-core era [25, 30, 31, 135]. Independently, application writers have rewritten or modified their applications to use novel parallel programming libraries and techniques. Both communities have improved software without changing how applications request OS resources.

In this chapter, we argue that OS-intensive parallel applications, including many of today's driving datacenter applications, must be able to express requests to the operating system using a vector interface in order to give the OS the information it needs to execute these demands efficiently. To not do so will make "embarrassingly parallel" workloads embarrassingly wasteful. We outline a new design, the Vector OS (VOS), that is able to let OS-intensive applications be not just parallel, but *efficiently* parallel.

Consider a modern webserver that receives a new connection using the `accept()` system call. `accept()` returns exactly one connection from a list of pending connections. The application then performs a series of sequential operations to handle the connection—setup, application-specific processing, and teardown. A high-load webserver may be serving many requests in parallel, but each request follows similar execution paths, wasting resources executing redundant work that could be shared across concurrent requests.

In VOS, work is executed using vector operations that are specified through vector interfaces like `vec_accept()` (which returns multiple connections rather than just one), `vec_open()`, `vec_read()`, `vec_send()`, etc. Exposing vector interfaces between applications and the operating system improves efficiency by eliminating redundant work; moreover, vector interfaces open the door to new efficiency opportunities by allowing VOS to more effectively harness vector and parallel hardware capabilities already present in many machines, such as SSE vector instructions and GPUs.

This part of the thesis argues that vector interfaces are critical to achieving efficient parallelism, thus requiring changes to the OS interface that applications program to. But this departure from a decades-old interface raises several questions and challenges that are as exciting as they are difficult: Should developers explicitly specify work in terms of vectors of resources? If not, how should vector execution be hidden from the programmer without introducing excessive complexity into the OS? What are the semantics of vector system call completion? Should they be synchronous or asynchronous, and how should they handle exceptions? We address many of these questions in this paper, but do not yet have answers to all of them. Nonetheless, we believe that the Vector OS can enable OS-intensive applications to make the maximum use of today's increasingly vector and parallel hardware.

8.2 The Parallel Landscape

Hardware parallelism is growing. Processors continue to gain speed by adding cores; graphics engines improve rendering performance by adding shaders; solid state drives improve throughput by adding more flash chips; and CPUs increase throughput by making

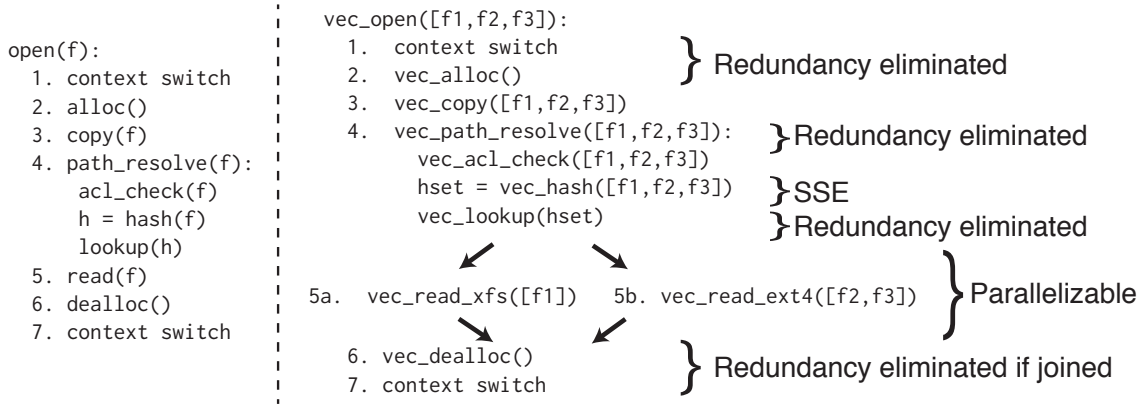


Figure 8.1: Pseudocode for open() and proposed vec_open(). vec_open() provides opportunities for eliminating redundant code execution, vector execution when possible, and parallel execution otherwise.

their vector instructions, such as SSE, even wider.[1] These trends are expected to continue into the foreseeable future, barring a revolution in device physics.

OS-intensive, parallel applications are adapting to parallel hardware. `memcached` moved from single-threaded to multi-threaded; event-based webservers such as `node.js` and Python Twisted are improving support for massive concurrency; languages and threading libraries such as OpenMP, Cilk, Intel Thread Building Blocks, and Apple's Grand Central Dispatch library all encourage applications to break work into smaller independent tasks, which the libraries can then assign to cores.

Parallel operating systems are making parallel execution possible. Recent work on improving operating system support for parallelism has made great strides in *enabling* parallel execution of code by removing locks, improving cache locality and utilization for data, carefully allocating work to processors to minimize overhead, and so on [31, 61, 119, 135].

Parallelism is necessary, but not necessarily efficient. While parallel execution on parallel hardware may sound ideal, it completely ignores *efficiency*, a metric that measures how much of a system's resources are necessary to complete some work [18]. We therefore turn our attention to "efficient parallelism," or making the best overall use of resources on a parallel system.

[1]Intel's Advanced Vector Extensions support 256-bit wide register operations.

8.3 Vectors will be Victors

Providing efficient parallelism requires that operating systems do the following:

1. *Eliminate redundancy*: Identify what work is common or redundant across a set of tasks and execute that work exactly once.

2. *Vectorize* when possible: Work should be expressed as operations over vectors of objects so it can be both executed efficiently using hardware techniques such as SSE and better optimized by compilers.

3. *Parallelize* otherwise: If code paths diverge, continue to execute work in parallel.

Consider the example of the `open()` system call in Figure 8.1 (left) simplified from the Linux 2.6.37 source code. Opening a file requires context switches, memory operations (e.g., copying the filename into a local kernel buffer in `alloc()`), performing access control checks, hashing file names, directory entry lookup, and reads from a filesystem. Today, simultaneous `open()` calls can mostly be executed in parallel, but doing so would not be as efficient as it could be.

Figure 8.1 (right) shows the set of operations required for `vec_open()`, which provides a vector of filenames to the operating system. In this example, file f1 exists in an XFS filesystem while the other two reside in an ext4 filesystem. The code path is similar to `open()`, with the exception that the interfaces are capable of handling vectors of objects, e.g, `vec_hash()` takes in several filenames and returns a vector of hashes corresponding to those filenames. These vector interfaces package together similar work and can improve efficiency by using the techniques described above.

Eliminating Redundancy: When provided vectors of objects, a vector system call can share the common work found across calls. Examples of common work include context switching, argument-independent memory allocations, and data structure lookups. `alloc()` is an argument-independent memory allocation that returns a page of kernel memory to hold a filename string; the `vec_alloc()` version need only fetch one page if the length of all filename arguments fits within that page, eliminating several additional page allocations necessary for each file if processed one by one. `vec_path_resolve()` requires traversing the directory tree, performing ACL checks and lookups along the way. If files share common parent directories, resolution of the common path prefix need only occur once for all files instead of once per file.

As yet another example, `lookup(hash)` must search a directory entry hash list to find the one entry that matches the input hash. In `vec_lookup(hset)`, the search algorithm

need only traverse the list once to find all entries corresponding to the vector of hashes. While this search can be performed in parallel, doing so would needlessly parse the hash list once for each input hash, wasting resources that could be dedicated to other work.

Vector interfaces provide general redundancy-eliminating benefits (e.g., reducing context switches), some of which can be provided by other batching mechanisms [119]. But vector interfaces also can enable specialized algorithmic optimizations (e.g., hash lookup) because all operations in a batch are the same, even though the operands may differ.

HW Vectorizing: Certain operations can be performed more efficiently when they map well to vector hardware already available but underutilized by existing operating systems. An implementation of `vec_hash()` might use SSE instructions to apply the same transformations to several different filenames, which may not be possible when dealing with precisely one filename at a time.

Parallelizing: Not all work will perfectly vectorize throughout the entire execution. In our `vec_open()` example, three files are stored on two different filesystems. While most of the code can be vectorized because they share common code paths, performing a `read()` from two different filesystems would diverge in code execution. Both calls can occur in parallel, however, which would use no more resources than three parallel `read()` calls; in this case it would probably use fewer resources because there will exist common work between files f2 and f3 within `vec_read_ext4()`.

When code paths diverge, the system should automatically determine whether to join the paths together. In Figure 8.1, there is an optional barrier before `vec_dealloc()`—should both forked paths complete together, joining the paths can save resources in executing deallocations and context switches. But should they diverge in time significantly, it may be better to let the paths complete independently.

8.3.1 Quantifying Redundant Execution

How much redundancy exists for OS-intensive parallel workloads running on highly parallel hardware? We explore this question by looking at the system call redundancy of Apache 2.2.17 server web requests between four different files in two different directories. We use `strace` to record the system calls executed when serving a single HTTP GET request for each file. Each request was traced in isolation and we show the trace for the request with the median response time out of five consecutive requests for each file.

Figure 8.2 shows when each system call was executed for each request for these four different static files. Each request invoked the same set of system calls regardless of which file was served, and the timings between system calls were similar, with variances attributable

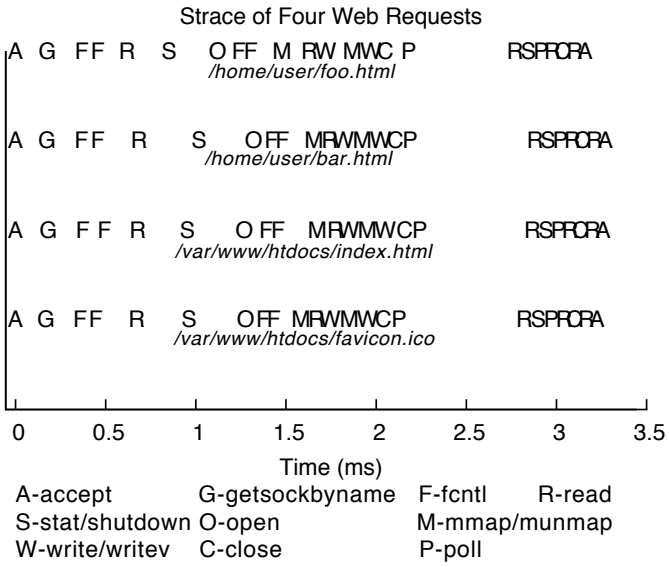


Figure 8.2: Trace of four different web requests serving static files shows the same system calls are always executed, and their execution paths in time are similar.

to OS scheduling. This simple experiment demonstrates that real applications provide opportunities to take advantage of the benefits of vector interfaces.

8.3.2 Scaling Redundancy With Load

Efficiency matters most when parallel systems are operating at high load, so we must understand how redundancy scales as a function of incoming load. We argue that at high load, redundancy is abundantly available, making the Vector OS approach an appealing solution.

To illustrate this concept, Figure 8.3 shows how offered load affects the redundancy available in a parallel system. Here, we define redundancy loosely as the amount of work that is identical to some unique work currently being done in the system. In the best case, each additional request is identical to existing requests in the system, and requests arrive simultaneously; redundancy therefore increases linearly with load.

In the worst case, requests are uncoordinated: they differ in both type and/or arrival time. At low load, redundancy is thus hard to find. As load increases, redundancy increases for several fundamental reasons. First, a system has a finite number of different types of

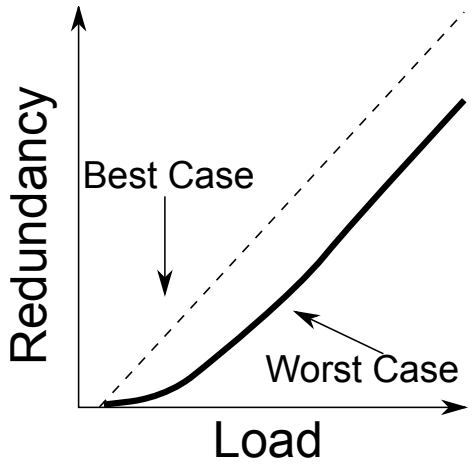


Figure 8.3: At high load, redundancy linearly grows with incrementally offered load.

tasks it is likely to execute; a similar argument has been made for applications running on GPUs [56]. By the pigeonhole principle, redundancy must increase once a request of each type already exists in the system. In addition, high load systems are increasingly becoming specialized for one type of workload (databases and filesystems, caching, application-logic, etc.), further shrinking the number of distinct types of work. Distributed systems often partition work to improve data locality or further increase specialization within a pool of servers, again increasing the chance that a high-load system will see mostly homogeneous requests.

These homogeneous requests may also arrive closely in time: Many systems already batch work together at the interrupt level to cope with interrupt and context switching overheads, creating an under-appreciated opportunity. Today's operating systems make limited use of this interrupt coalescing, but this mechanism plays a more fundamental role in the Vector OS: It provides the basis for nearly automatic batching of I/O completions, which are delivered to the application and processed in lockstep using convenient vector programming abstractions. These "waves" of requests incur additional latency only once during entry into the system.

8.4 Towards the Vector OS

The Vector OS must address several difficult interface and implementation challenges. The design must expose an appropriate set of vector interfaces to allow application writers to

both easily write and understand software running on parallel hardware. VOS must also organize and schedule computation and I/O to retain vectorization when possible and parallel execution otherwise.

Batching imposes a fundamental tradeoff between efficiency gains by having a larger batch, and adding latency by waiting for more requests to batch. However, at higher loads, this tradeoff becomes a strict win in favor of vectorization, as the efficiency gains enable *all* requests to execute faster because they spend less time waiting for earlier requests to complete. VOS must therefore make it easy for applications to dynamically adjust their batching times, much as has been done in the past for specific mechanisms such as network interrupt coalescing. Fortunately, even at lower load, Figure 8.2 shows that calls that arrive together (e.g., subsequent requests for embedded objects after an initial Web page load) present a promising avenue for vectorization because their system call timings are already nearly identical.

8.4.1 Interface Options

Today's system call interface destroys opportunities for easy vectorization inside the kernel—system calls are synchronous and typically specify one resource at a time. VOS must be able to package similar work (system calls, internal vector function calls) together to be efficiently parallel. We enumerate several interface possibilities below and describe their impact on the efficiency versus latency tradeoff.

1. Application-agnostic changes: One way to provide opportunities for making vector calls without application support is to introduce system call queues to coalesce similar requests. An application issues a system call through libc, which inserts the call into a syscall queue while the application waits for its return. Upon a particular trigger (a timeout, or a number threshold), VOS would collect the set of requests in a particular queue and execute the set using a vector interface call—this implementation can build upon the asynchronous shared page interface provided by FlexSC [119]. Another approach is to rewrite program binaries to submit multiple independent system calls as one multi-call by using compiler assistance [106, 108]. Both approaches transparently provide VOS with collections of system calls which it could vectorize, but these approaches have several drawbacks: First, applications do not decide when to issue a vector call, so they cannot override the timing logic built into the OS or compiler, leading to a frustrating tension between application writers and OS implementers. Second, a single thread that wishes to issue a set of similar synchronous system calls (e.g., performing a bunch of `read()` calls in a for loop), will still execute all reads serially even if there exists no dependence between them.

2. Explicit vector interface: Applications can help VOS decide how to coalesce vector calls by explicitly preparing batches of similar work using the vector interface to system calls, such as `vec_open()`, `vec_read()`, etc. VOS can use this knowledge when scheduling work because it knows the application thread will be blocked until all submitted work completes. This assumes these vector calls are synchronous, though a non-synchronous completion interface (e.g., return partial results) may be useful for some applications.

As an example of how an application might use explicit vector interfaces, the core event-loop for an echo server may look as follows (in simplified pseudocode):

```
fds = vec_accept(listenSocket);
vec_recv(fds, buffers);
vec_send(fds, buffers);
```

As the application processing between `vec_recv()` and `vec_send()` becomes more complicated, raw vector interfaces may prove difficult to use. The benefit is that the OS is relieved of deciding how and when to vectorize, leaving the application as the arbiter for the efficiency versus latency tradeoff and eliminating that complexity from the OS.

3. Libraries and Languages: Although our focus is on the underlying system primitives, many applications may be better served by library and language support built atop those primitives. Several current event-based language frameworks appear suitable for near-automatic use of vector interfaces, including `node.js` and Python Twisted. Programs in these frameworks specify actions that are triggered by particular events (e.g., a new connection arrival). If the actions specified are side-effect free, they could be automatically executed in parallel or even vectorized as is done with some GPU programming frameworks such as CUDA. System plumbing frameworks such as SEDA [134] that use explicit queues between thread pools also present a logical match to underlying vector interfaces, with, of course, non-trivial adaptation.

Finally, the progress in general-purpose GPU programming is one of the most promising signs that programming for vector abstractions is possible and rewarding. Both CUDA and OpenCL provide a "Single Instruction Multiple Thread" (SIMT) abstraction on top of multi-processor vector hardware that simplifies some aspects of programming these systems. Although programmers must still group objects into vectors, the abstraction allows them to write code as if the program were a stream of instructions to a single scalar processor. We believe that the amazing success of GPGPUs in high-performance computing is a telling sign that programmers who want performance are willing and able to "think vector" in order to get it.

8.5 Discussion

We believe that restructuring for a vector OS is needed to improve the efficiency of OS-intensive parallel software, but to do so brings challenges and accompanying opportunities for OS research:

Heterogeneity: As the multiple-filesystem example of `vec_open()` showed, VOS will need to move efficiently between single-threaded execution of shared operations, vector execution of SIMD-style operations, and parallel execution when code paths diverge. We believe it is clear that forcing the OS programmer to manually handle these many options is infeasible; VOS will require new library, language, or compiler techniques to render the system understandable. In addition to code divergence, VOS's code should also be appropriately specialized for the available hardware, be it a multi-core CPU, an SSE-style CPU vector instruction set, an integrated GPU or a high-performance discrete GPU.

Application to databases: StagedDB [62] showed the benefits of a specific type of work-sharing made available through staged construction of query processing in a database. Examples of shared work included re-using the results of a database scan across many concurrent queries. Follow up work showed that aggressive work sharing could end up serializing query execution beyond the benefit that work sharing could provide for in-memory workloads [71]. In their system, the work being shared often involved the shared table scan, with each individual query performing a unique operation on that shared data that could not be vectorized across queries. We believe that for database workloads that exhibit more computational similarity, such query serialization can be avoided, but for computationally-unique workloads, vector interfaces are less useful.

Passing down vector abstractions through distributed storage: Many high-performance computing systems use I/O directors as a middleware layer between compute and storage systems. Multi-collective I/O is an interface presented to application designers that allows these I/O directors to more effectively use the underlying storage system. Given that these applications already use these "vector"-like interfaces in applications, we believe that these abstractions can and should be pushed down to the storage layer as we showed in Chapter 7 for similar efficiency and throughput benefits.

Chapter 9

Future Work and Conclusion

The FAWN approach has the potential to provide significant energy and cost savings by reducing the peak power consumed by datacenters. This dissertation demonstrates that achieving this potential involves software-hardware co-design, and that existing software system stacks require a new set of techniques, designs, and abstractions to harness the full capabilities of the FAWN approach. In this section, we discuss the landscape for prior and future FAWN systems, the future work left by the dissertation, and conclude with a brief summary of the contributions of the work.

9.1 Evolving Server and Mobile Platforms

The FAWN project began in 2007 because of the enormous opportunity to improve energy efficiency by combining wimpy nodes and, at the time, relatively high-end flash. Over the last several years, our comparisons have demonstrated a still significant but smaller improvement over evolving brawny platforms.

One reason for this difference is that we have often chosen off-the-shelf systems that leave something to be desired: The fixed power overheads of non-CPU components tended to dominate power consumption, or the storage or networking I/O capability were not balanced well with processing capability. A sign that further efficiency improvements are possible is that custom architectures designed by SeaMicro, Tilera, and others have demonstrated tight integration of wimpy nodes with balanced communication architectures (be-

tween cores) that provide the same performance as our wimpy node prototypes, while being more than four times as energy efficient.[1]

Simultaneously, designers of "brawny" platforms have needed to increase throughput while avoiding the power wall. Importantly, they have done so by taking advantage of some of the same properties by which FAWN improves energy efficiency: by using slower, simpler cores. For example, our winning entry for the 2010 10GB JouleSort competition used an Intel Xeon L3426, which operated 4 cores (8 HyperThreads) at a clock rate of just 1.86GHz and contained 8MB of L2 cache, about the same clock rate as the Atom-based FAWN systems we compare against.

An analysis of mobile processor evolution shows a similar trend: the high-end 15" Apple Macbook Pro line has moved from using a 2.8GHz dual-core Core2 Duo with 6MB of L2 cache in 2009 to using a 2.2 GHz quad-core with 6MB of L3 cache in 2011, a significant decrease in both frequency and per-core cache size. Because manufacturers have applied energy-efficient designs targeted originally for wimpy processors to mobile and server platforms, we expect that the efficiency differences between wimpy systems and "efficient server" platforms will shrink.

9.2 Implications and Outlook

In Section 2.1, we outlined several power scaling trends for modern computer systems. Our workload evaluation in Chapter 5 suggested that these trends hold for CPU in real systems—and that, as a result, using slower, simpler processors represents an opportunity to reduce the total power needed to solve a problem if that problem can be solved at a higher degree of parallelism.

In this section, we draw upon the memory scaling trends we discussed to present a vision for a future FAWN system: Individual "nodes" consisting of a single CPU chip with a modest number of relatively low-frequency cores, with a small amount of DRAM stacked on top of it, connected to a shared interconnect. This architecture is depicted in Figure 9.1. The reasons for such a choice are several:

Many, many cores: The first consequence of the scaling trends is clear: A future energy-efficient system for data-intensive workloads will have many, many cores, operating at quite

[1]They have so far avoided integrating storage into their platforms. We believe that this integration will be the next important step in ensuring these platforms are used for rich, stateful services like key-value storage as opposed to stateless, simple webservers.

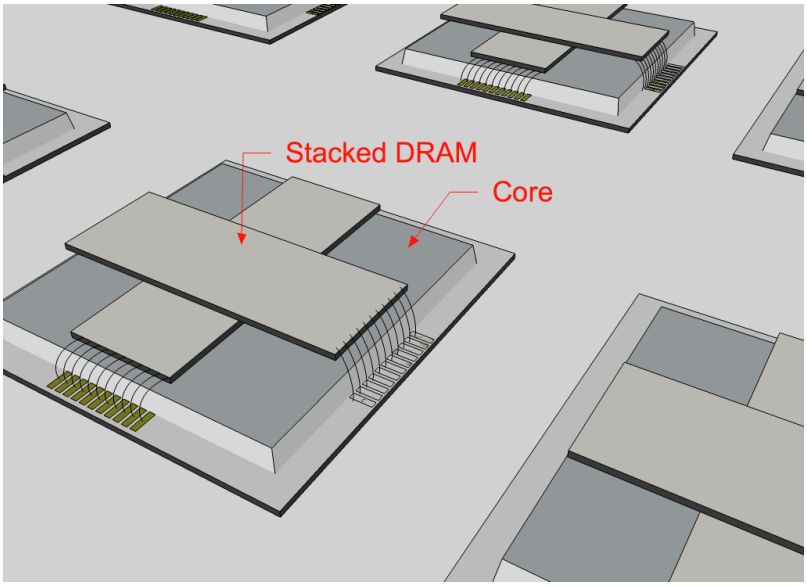


Figure 9.1: Future FAWN vision: Many-core, low-frequency chip with stacked DRAM per core.

modest frequencies. The limits of this architecture will be the degree to which algorithms can be parallelized (and/or load-balanced), and the static power draw imposed by CPU leakage currents and any hardware whose power draw does not decrease as the size and frequency of the cores decrease.

However, the move to many-core does not imply that individual chips must have modest capability. Indeed, both Intel and Tilera have produced system with 48–100 cores on a single chip. Such a design has the advantage of being able to cheaply interconnect cores on the same chip, but suffers from limited off-chip I/O and memory capacity/bandwidth compared to the amount of CPU on chip.

Less memory, stacked: We chose a stacked DRAM approach because it provides three key advantages: Higher DRAM bandwidth, lower DRAM latency (perhaps half the latency of a traditional DIMM bus architecture) and lower DRAM power draw. The disadvantage is the limited amount of memory available per chip. Using the leading edge of today's DRAM technologies, an 8Gbit DRAM chip could be stacked on top of a small processor; 1GB of DRAM for a single or dual-core Atom is at the low end of an acceptable amount of memory for many workloads. From the matrix multiplication workload in the previous section, we

expect that this decision will result in a similar efficiency "flip-flop": Workloads that fit in memory on a single FAWN node with 1GB of DRAM would run much more efficiently than they would on a comparable large node, but the FAWN node would be less efficient for the range of problems that exceed 1GB but are small enough to fit in DRAM on a more conventional server.

Implications: This vision for future FAWN system abuts roadmaps for traditional server systems. The similarities are in the need for increasing software parallelism and motivations for treating individual nodes more like distributed systems [25, 135], but they differ in the tighter coupling of compute and memory and the emphasis on ensuring software reduces per-core memory consumption.

Innovations in hardware should continue, but harnessing these improvements will likely require revisiting software-hardware abstractions, programming models, and applications together. In this light, this dissertation suggests some techniques, interfaces, and perspectives that should apply to these future systems as well.

9.3 Future Work

Broadly categorized, there are several directions of future work that this dissertation leaves open.

More In-depth Workload Studies This dissertation studied the application of the FAWN approach primarily to distributed key-value storage systems supplemented with a broad categorization and analysis of other data-intensive workloads. Performing a deep study of these other data-intensive workloads will always remain an area of future work.

Changing Markets As these workload studies continue to show the potential of the FAWN approach, the market is likely to respond in several different ways, each of which will open up further research studies. First, several different startup companies are producing FAWN-like wimpy node systems each with different hardware architectures. For example, SeaMicro machines use a tightly integrated compute and memory fabric, but move all other peripheral services such as storage to a separate hardware I/O virtualized shared infrastructure. While the system can work as a "plug-and-play" replacement, many of the software techniques to access local SSDs described in Section 7 may not apply to a shared virtualized environment, requiring the use of a different set of software-hardware co-design strategies.

Second, hardware vendors are likely to produce many more systems that span the range between wimpy nodes and high-speed server designs. For many workloads, mobile processors and low-power servers may provide a better tradeoff between computational features and energy efficiency than either of the other extreme design points, assuming these systems remain balanced for a given workload.

Last, many studies, including ours, have used current prices of hardware to understand total cost of ownership numbers. This will of course remain a moving target as the market responds to the demand for high-speed and low-speed systems alike.

Non-volatile memories This thesis has been influenced by the increasing throughput and reduced latencies of non-volatile memories such as flash, PCM, and others, but we have treated these non-volatile memories as storage devices instead of "persistent memory." When technologies like PCM and memristor can deliver nanosecond access times, their use as a DRAM replacement or substitute becomes more attractive. Abstractions such as NVHeaps [38] and other data structures and algorithms [132] have begun to shed light on better ways to use these systems, and discovering how higher-level applications can use these abstractions effectively is an exciting avenue of future work.

FAWN-KV opted for using log-structured writing both to avoid random writes on flash as well as to provide efficient failover operations. Modern flash devices now perform greater degrees of log-structuring in their flash translation layer and still expose the block-based abstraction to the operating system and applications. But if applications are already log-structuring writes, should the underlying device duplicate this effort? We believe that opening up the interface between the application and the device can help both entities, perhaps through the use of "I/O personalities" that specify the types of access (e.g., random/sequential, small/large) the application is likely to need [16].

Lastly, we have only begun to understand the potential impact of fast, *predictable* non-volatile memories on operating system design. Whereas performance from hard disks is unpredictable due to mechanical delays, solid state devices potentially can have more predictable access latencies. Today, block erasures and TRIM commands are not well hidden, but smarter scheduling or improved application-device communication might make devices more predictable. This predictability changes many aspects of operating systems. For example, we used the relative predictability of I/O performance from SSDs when tuning the constants for the SATA interrupt mitigation driver to some success. Similarly, instead of using heavyweight storage hardware interrupts to abruptly notify the operating system of completed I/O, the OS can schedule the completion of I/O at a convenient time to provide higher performance and better quality-of-service, eliminating the need for hardware interrupts except as a backup mechanism.

Heterogeneous systems Our work, along with others, has demonstrated that no one system design will work well for all workloads. As mentioned in Chapter 3, several hetereogenous node designs that combine wimpy cores with brawny cores can bridge this gap. Prior work has investigated the use of these asymmetric core designs for microbenchmarks and CPU scheduler design [51], but understanding how higher-level software components in a distributed system should be exposed to this asymmetry, if at all, remains an open and intriguing question. As heterogeneity increases, e.g., by supplementing systems with GPUs and other specialized hardware, so do we need to revisit abstractions, interfaces, and use-cases further. Recent work has begun exploring this research area in particular for GPUs [114].

Quality of Service per Joule While we have demonstrated that the FAWN approach can significantly improve energy efficiency as measured by "work done per Joule", Quality of Service (QoS) is becoming an increasingly important measurable statistic. Many datacenter services need to provide service level agreements such as 99%-ile latency below some threshold, a target throughput level, or a specific availability such as "five nines" of uptime. FAWN demonstrates that a latency or reliability-agnostic metric such as work done per Joule can be improved but comes at the cost of higher latency and worse behavior under high load [68]. Some questions remain, such as: Is worst-case performance fundamentally worse for the FAWN approach, or are they by-products of system engineering? Do FAWN systems fundamentally produce lower availability because of increased node count, or does FAWN maintain or increase availability by reducing the per-node recovery work unit? Do the strong scaling requirements of FAWN systems make them harder to manage? Ultimately, these questions are a subset of the general metric "Quality of Service per Joule", which measures how much compute energy is required to provide a particular Quality of Service level.

Improving QoS/J could involve, for example, deliberately over-provisioning a cluster to handle peak loads more gracefully, as suggested in [68]. Here, energy proportionality becomes more important because systems are driven explicitly below peak capability to provide more headroom for workload variation. But doing so increases the peak power consumed by the datacenter, increasing infrastructure costs and energy use. Whether this approach improves QoS/J at the datacenter level is therefore an open question.

9.4 Conclusion

This dissertation proposed FAWN, a Fast Array of Wimpy nodes, and analyzed the trends, tradeoffs, and software techniques needed to improve datacenter energy efficiency using FAWN. Our evaluation of the FAWN approach on several different datacenter workloads demonstrated that balanced system designs optimized for per-node efficiency can improve aggregate cluster energy efficiency but requires revisiting several layers of the software stack to account for the different constraints and properties of FAWN systems. At the application layer, we developed a distributed key-value storage system called FAWN-KV that used energy-efficient processors paired with flash, identifying a combination of techniques to effectively use flash, conserve memory, and efficiently handle failover in a large FAWN cluster. Our study of several other datacenter workloads identified key differences between FAWN nodes and traditional nodes that can make deploying existing software less efficient on FAWN nodes.

We then explored the use of modern balanced systems with fast SSDs, identifying and eliminating bottlenecks in the existing I/O stack, motivating our proposal for new vector interfaces to fast I/O devices. We then demonstrated subsequent changes to networked application server design to take advantage of the lower-level vector interfaces provided by the operating system, storage devices, and RPC systems. Last, we generalized the design behind vector interfaces to all system interfaces, identifying mechanisms to eliminate redundant work found in many datacenter workloads and described the associated interfaces and tradeoffs such a proposal could provide.

In summary, this dissertation provided multiple sources of evidence to suggest that novel, efficient software techniques are needed to fully take advantage of the FAWN approach to improving datacenter energy efficiency. Finally, I believe the lessons, abstractions, and insights provided by this work will apply generally to all future balanced systems.

Bibliography

[1] Greenplum: the petabyte-scale database for data warehousing and business intelligence. http://www.greenplum.com/, 2009. URL retrieved August 2009. [Cited on page 24.]

[2] Riak, an open source scalable data store. http://www.basho.com/products_riak_overview.php, 2009. [Cited on page 25.]

[3] Teradata. http://www.teradata.com/, 2009. URL retrieved August 2009. [Cited on page 24.]

[4] Aster data. http://www.asterdata.com, 2011. [Cited on page 24.]

[5] Calxeda. http://www.calxeda.com, 2011. [Cited on page 23.]

[6] Apache Cassandra. http://cassandra.apache.org, 2011. [Cited on page 25.]

[7] Flexible I/O Tester. http://freshmeat.net/projects/fio/, 2011. [Cited on pages 37 and 61.]

[8] Hadoop. http://hadoop.apache.org/, 2011. [Cited on page 23.]

[9] Paraccel analytic platform. http://www.paraccel.com, 2011. [Cited on page 24.]

[10] Apache Thrift. https://thrift.apache.org/, 2011. [Cited on page 77.]

[11] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Onyx: A prototype phase change memory storage array. In *Proc. HotStorage*, Portland, OR, June 2011. [Cited on pages 74 and 102.]

[12] Gene Amdahl. Validity of the single processor approach to large-scale computing capabilities. In *Proc. AFIPS (30)*, pages 483–485, 1967. [Cited on page 7.]

[13] Hrishikesh Amur, James Cipar, Varun Gupta, Gregory R. Ganger, Michael A. Kozuch, and Karsten Schwan. Robust and flexible power-proportional storage. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, Indianapolis, IN, June 2010. [Cited on page 8.]

[14] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proc. 7th USENIX NSDI*, San Jose, CA, April 2010. [Cited on pages 24, 74 and 102.]

[15] Anandtech. The sandy bridge preview. `http://www.anandtech.com/show/3871/the-sandy-bridge-preview-three-wins-in-a-row`. [Cited on page 59.]

[16] David G. Andersen and Steven Swanson. Rethinking flash in the data center. *IEEE Micro*, 2010. (Invited commentary article). [Cited on page 117.]

[17] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009. [Cited on pages 6, 15, 25, 46, 54, 58 and 74.]

[18] Eric Anderson and Joseph Tucek. Efficiency matters! In *Proc. HotStorage*, Big Sky, MT, October 2009. [Cited on pages 58 and 105.]

[19] Jonathan Appavoo, Volkmar Uhlig, and Amos Waterland. Project kittyhawk: Building a global-scale computer. In *Operating Systems Review*, volume 42, pages 77–84, January 2008. [Cited on page 25.]

[20] Mohit Aron and Peter Druschel. Soft timers: Efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems*, 18(3): 197–228, 2000. [Cited on page 69.]

[21] Jens Axboe. [PATCH 1/3] block: add blk-iopoll, a NAPI like approach for block devices. `http://lwn.net/Articles/346256/`. [Cited on page 69.]

[22] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid SSD/RAM memory allocation made easy. In *Proc. 8th USENIX NSDI*, Boston, MA, April 2011. [Cited on page 24.]

[23] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007. [Cited on pages 7, 8, 15 and 25.]

[24] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009. [Cited on pages 1, 5, 6 and 8.]

[25] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009. [Cited on pages 103 and 116.]

[26] Andreas Beckmann, Ulrich Meyer, Peter Sanders, and Johannes Singler. Energy-efficient sorting using solid state disks. http://sortbenchmark.org/ecosort_2010_Jan_01.pdf, 2010. [Cited on page 53.]

[27] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-core key-value store. http://gigaom2.files.wordpress.com/2011/07/facebook-tilera-whitepaper.pdf. [Cited on page 23.]

[28] BerkeleyDB Reference Guide. Memory-only or Flash configurations. http://www.oracle.com/technology/documentation/berkeley-db/db/ref/program/ram.html. [Cited on page 39.]

[29] W. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang. Evaluation of existing architectures in IRAM systems. In *Workshop on Mixing Logic and DRAM, 24th International Symposium on Computer Architecture*, June 1997. [Cited on page 23.]

[30] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proc. 8th USENIX OSDI*, San Diego, CA, December 2008. [Cited on page 103.]

[31] Silas Boyd-Wickizer, Austin Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of linux scalability to many cores. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, October 2010. [Cited on pages 103 and 105.]

[32] Randal E. Bryant. Data-Intensive Supercomputing: The case for DISC. Technical Report CMU-CS-07-128, School of Computer Science, Carnegie Mellon University, May 2007. [Cited on page 1.]

[33] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, March 2009. [Cited on page 22.]

[34] Adrian M. Caulfield, Arup De, Joel Coburn, Todor Mollov, Rajesh Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *IEEE Micro*, December 2010. [Cited on pages 60, 74, 76, 78 and 102.]

[35] Sang Kil Cha, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G. Andersen. SplitScreen: Enabling efficient, distributed malware detection. In *Proc. 7th USENIX NSDI*, San Jose, CA, April 2010. [Cited on page 56.]

[36] Geoffrey Challen and Mark Hempstead. The case for power-agile computing. In *Proc. HotOS XIII*, Napa, CA, May 2011. [Cited on page 26.]

[37] Byung-Gon Chun, Gianluca Iannaccone, Giuseppe Iannaccone, Randy Katz, Gunho Lee, and Luca Niccolini. An energy case for hybrid datacenters. In *Proc. HotPower*, Big Sky, MT, October 2009. [Cited on page 26.]

[38] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. In *Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, 2011. [Cited on page 117.]

[39] Adrian Cockcroft. Millicomputing. In *Proc. 12th Intl. Workshop on High Performance Transaction Systems*, Pacific Grove, CA, October 2007. [Cited on page 22.]

[40] Hui Dai, Michael Neufeld, and Richard Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Baltimore, MD, November 2004. [Cited on page 24.]

[41] John D. Davis and Suzanne Rivoire. Building energy-efficient systems for sequential workloads. Technical Report MSR-TR-2010-30, Microsoft Research, March 2010. [Cited on page 53.]

[42] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th USENIX OSDI*, San Francisco, CA, December 2004. [Cited on pages 9, 23 and 24.]

[43] Biplob Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proc. USENIX ATC 2010*, Boston, MA, June 2010. [Cited on page 24.]

[44] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3:1414–1425, September 2010. [Cited on pages 34, 74 and 102.]

[45] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: RAM space skimpy key-value store on flash. In *Proc. ACM SIGMOD*, Athens, Greece, June 2011. [Cited on pages 74 and 102.]

[46] Guiseppe DeCandia, Deinz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, WA, October 2007. [Cited on pages 25, 29 and 42.]

[47] Dell XS11-VX8. Dell fortuna. `http://www1.euro.dell.com/content/topics/topic.aspx/emea/corporate/pressoffice/2009/uk/en/2009_05_20_brk_000`, 2009. [Cited on page 22.]

[48] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009. [Cited on page 101.]

[49] Fred Douglis, Frans Kaashoek, Brian Marsh, Ramon Caceres, Kai Li, and Joshua Tauber. Storage alternatives for mobile computers. In *Proc. 1st USENIX OSDI*, pages 25–37, Monterey, CA, November 1994. [Cited on page 24.]

[50] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz André Barroso. Power provisioning for a warehouse-sized computer. In *International Symposium on Computer Architecture (ISCA)*, San Diego, CA, June 2007. [Cited on page 8.]

[51] Alexandra Fedorova, Juan Carlos Saez, Daniel Shelepov, and Manuel Prieto. Maxmimizing performance per watt with asymmetric multicore systems. volume 52, pages 48–57, December 2009. [Cited on pages 25 and 118.]

[52] Richard Freitas, Joseph Slember, Wayne Sawdon, and Lawrence Chiu. GPFS scans 10 billion files in 43 minutes. IBM Whitepaper, http://www.almaden.ibm.com/storagesystems/resources/GPFS-Violin-white-paper.pdf, 2011. [Cited on pages xvi and 60.]

[53] fusion-io. Fusion-IO. http://www.fusionio.com. [Cited on pages 73 and 74.]

[54] Lakshmi Ganesh, Hakim Weatherspoon, Mahesh Balakrishnan, and Ken Birman. Optimizing power consumption in large scale storage systems. In *Proc. HotOS XI*, San Diego, CA, May 2007. [Cited on page 26.]

[55] A. Gara, M. A. Blumrich, D. Chen, G L-T Chiu, et al. Overview of the Blue Gene/L system architecture. *IBM J. Res and Dev.*, 49(2/3), May 2005. [Cited on page 25.]

[56] Michael Garland and David B. Kirk. Understanding throughput-oriented architectures. *Communications of the ACM*, 53(11):58–66, November 2010. [Cited on pages 102 and 109.]

[57] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, October 2003. [Cited on pages 24 and 32.]

[58] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, distributed data structures for Internet service construction. In *Proc. 4th USENIX OSDI*, San Diego, CA, November 2000. [Cited on page 24.]

[59] James Hamilton. Cooperative expendable micro-slice servers (CEMS): Low cost, low power servers for Internet scale services. http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_CEMS.pdf, 2009. [Cited on pages 22 and 23.]

[60] James Hamilton. Scaling at MySpace. http://perspectives.mvdirona.com/2010/02/15/ScalingAtMySpace.aspx, 2010. [Cited on page 97.]

[61] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*, New Delhi, India, August 2010. [Cited on pages 96, 101 and 105.]

[62] Stavros Harizopoulos and Anatassia Ailamaki. StagedDB: Designing database servers for modern hardware. *IEEE Data Engineering Bulletin*, 28(2):11–16, June 2005. [Cited on pages 96 and 112.]

[63] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. QPipe: A simultaneously pipelined relational query engine. In *Proc. ACM SIGMOD*, Baltimore, MD, June 2005. [Cited on page 96.]

[64] Urs Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 2010. [Cited on pages 6, 7, 10 and 11.]

[65] Intel. Penryn Press Release. `http://www.intel.com/pressroom/archive/releases/20070328fact.htm`, 2007. [Cited on page 11.]

[66] Iozone. Filesystem Benchmark. `http://www.iozone.org`. [Cited on page 37.]

[67] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proc. EuroSys*, Lisboa, Portugal, March 2007. [Cited on page 9.]

[68] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 314–325, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. [Cited on pages 26, 27, 49, 50 and 118.]

[69] JFFS2. The Journaling Flash File System. `http://sources.redhat.com/jffs2/`. [Cited on pages 32 and 40.]

[70] Bobby Johnson. Facebook, personal communication, November 2008. [Cited on page 40.]

[71] Ryan Johnson, Stavros Harizopoulos, Nikos Hardavellas, Kivanc Sabirli, Ippokratis Pandis, Anastasia Ailamaki, Naju G. Mancheril, and Babak Falsafi. To share or not to share? In *Proc. VLDB*, Vienna, Austria, September 2007. [Cited on pages 96 and 112.]

[72] Randy H. Katz. Tech titans building boom. IEEE Spectrum, February 2009. [Cited on pages 1, 5 and 6.]

[73] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *Proc. USENIX Annual Technical Conference*, New Orleans, LA, January 1995. [Cited on page 24.]

[74] Peter M. Kogge, Toshio Sunaga, Hisatada Miyataka, Koji Kitamura, and Eric Retter. Combined DRAM and logic chip for massively parallel systems. In *Proceedings of the 16th Conference for Advanced Research in VLSI*, pages 4–16. IEEE Press, 1995. [Cited on page 23.]

[75] Jonathan Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of historical trends in the electrical efficiency of computing. In *Proceedings of IEEE Annals of the History of Computing*, pages 46–54, 2011. [Cited on page 16.]

[76] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *Proc. USENIX Annual Technical Conference*, Santa Clara, CA, June 2007. [Cited on page 102.]

[77] Vivek Kundra. State of public sector cloud computing. `http://www.cio.gov/documents/StateOfCloudComputingReport-FINALv3_508.pdf`, 2010. [Cited on page 7.]

[78] T. Kushner, A. Y. Wu, and A. Rosenfeld. Image Processing on ZMOB. *IEEE Trans. Comupters*, 31(10), 1982. [Cited on page 23.]

[79] Willis Lang, Jignesh M. Patel, and Srinath Shankar. Wimpy node clusters: What about non-wimpy workloads? In *Sixth International Workshop on Data Management on New Hardware*, Indianapolis, IN, June 2010. [Cited on pages 26 and 50.]

[80] James R. Larus and Michael Parkes. Using cohort scheduling to enhance server performance. In *Proc. USENIX Annual Technical Conference*, Berkeley, CA, June 2002. [Cited on page 101.]

[81] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory SSD in enterprise database applications. In *Proc. ACM SIGMOD*, Vancouver, BC, Canada, June 2008. [Cited on page 24.]

[82] Yinan Li, Bingsheng He, Qiong Luo, and Ke Yi. Tree indexing on flash disks. In *Proceedings of 25th International Conference on Data Engineering*, March 2009. [Cited on page 24.]

[83] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011. [Cited on pages 34 and 102.]

[84] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *International Symposium on Computer Architecture (ISCA)*, Beijing, China, June 2008. [Cited on pages 22 and 23.]

[85] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *Proc. 6th USENIX OSDI*, San Francisco, CA, December 2004. [Cited on page 25.]

[86] Tudor Marian. *Operating Systems Abstractions for Software Packet Processing in Datacenters*. PhD thesis, Cornell University, January 2011. [Cited on page 101.]

[87] Gaurav Mathur, Peter Desnoyers, Deepak Ganesan, and Prashant Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Boulder, CO, October 2006. [Cited on page 24.]

[88] John C. McCullough, John Dunagan, Alec Wolman, and Alex C. Snoeren. Stout: An adaptive interface to scalable cloud storage. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010. [Cited on page 101.]

[89] David Meisner, Brian T. Gold, and Thomas F. Wenisch. PowerNap: Eliminating server idle power. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, March 2009. [Cited on page 26.]

[90] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *International Symposium on Computer Architecture (ISCA)*, San Jose, CA, June 2011. [Cited on pages 8 and 26.]

[91] Memcached. A distributed memory object caching system. `http://memcached.org/`, 2011. [Cited on pages 24 and 29.]

[92] Gokhan Memik, Mahmut T. Kandemir, WeiKeng Liao, and Alok Choudhary. Multicollective i/o: A technique for exploiting inter-file access patterns. volume 2, August 2006. [Cited on page 101.]

[93] Microsoft Marlowe. Peering into future of cloud computing. `http://research.microsoft.com/en-us/news/features/ccf-022409.aspx`, 2009. [Cited on pages 22 and 23.]

[94] Jeffrey C. Mogul, Jayaram Mudigonda, Nathan Binkert, Parthasarathy Ranganathan, and Vanish Talwar. Using asymmetric single-ISA CMPs to save energy on operating systems. In *"IEEE Micro"*, pages 26–41, 2008. [Cited on page 25.]

[95] Iulian Moraru and David G. Andersen. Exact pattern matching with feed-forward bloom filters. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX11)*, ALENEX 2011. Society for Industrial and Applied Mathematics, January 2011. [Cited on pages 56 and 58.]

[96] Daniel Myers. On the use of NAND flash memory in high-performance relational databases. M.S. Thesis, MIT, February 2008. [Cited on pages 24, 32 and 37.]

[97] Suman Nath and Phillip B. Gibbons. Online maintenance of very large random samples on flash storage. In *Proc. VLDB*, Auckland, New Zealand, August 2008. [Cited on pages 32 and 35.]

[98] Suman Nath and Aman Kansal. FlashDB: Dynamic self-tuning database for NAND flash. In *Proceedings of ACM/IEEE International Conference on Information Processing in Sensor Networks*, Cambridge, MA, April 2007. [Cited on pages 24 and 32.]

[99] Netezza. Business intelligence data warehouse appliance. `http://www.netezza.com/`, 2006. [Cited on page 24.]

[100] nilfs. Continuous snapshotting filesystem for Linux. `http://www.nilfs.org`. [Cited on page 40.]

[101] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. In *Operating Systems Review*, volume 43, pages 92–105, January 2010. [Cited on page 102.]

[102] Milo Polte, Jiri Simsa, and Garth Gibson. Enabling enterprise solid state disks performance. In *Proc. Workshop on Integrating Solid-state Memory into the Storage Hierarchy*, Washington, DC, March 2009. [Cited on pages 18, 24, 32, 43, 61 and 82.]

[103] Project Voldemort. A distributed key-value storage system. `http://project-voldemort.com`. [Cited on pages 25 and 29.]

[104] Niels Provos. libevent. `http://monkey.org/~provos/libevent/`. [Cited on page 100.]

[105] Josep M. Pujol, Vijay Erramilli, Georgos Siganos, Xiaoyuan Yang, Nikos Laoutaris, Parminder Chhabra, and Pablo Rodriguez. The little engine(s) that could: Scaling online social networks. In *Proc. ACM SIGCOMM*, New Delhi, India, August 2010. [Cited on pages 97 and 99.]

[106] Amit Purohit, Charles P. Wright, Joseph Spadavecchia, and Erez Zadok. Cosy: Develop in user-land, run in kernel-mode. In *Proc. HotOS IX*, Lihue, Hawaii, May 2003. [Cited on pages 101 and 110.]

[107] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, CA, January 2002. [Cited on page 32.]

[108] Mohan Rajagopalan, Saumya K. Debray, Matti A. Hiltunen, and Richard D. Schlichting. Cassyopia: Compiler assisted system optimization. In *Proc. HotOS IX*, Lihue, Hawaii, May 2003. [Cited on pages 101 and 110.]

[109] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey Chase. Ensemble-level power management for dense blade servers. In *International Symposium on Computer Architecture (ISCA)*, Boston, MA, June 2006. [Cited on page 1.]

[110] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *IEEE Computer*, 34(6):68–74, June 2001. [Cited on page 23.]

[111] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. JouleSort: A balanced energy-efficient benchmark. In *Proc. ACM SIGMOD*, Beijing, China, June 2007. [Cited on pages 16, 21 and 46.]

[112] Luigi Rizzo. Polling versus interrupts in network device drivers. *BSDConEurope*, November 2001. [Cited on page 69.]

[113] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992. [Cited on page 24.]

[114] Chris Rossbach, Jon Currey, B. Ray, Mark Silberstein, and Emmitt Witchel. PTask: Operating system abstractions to manage gpus as compute devices. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011. [Cited on page 118.]

[115] Steven W. Schlosser, John Linwood Griffin, David F. Nagle, and Gregory R. Ganger. Filling the memory access gap: A case for on-chip magnetic storage. Technical Report CMU-CS-99-174, Carnegie Mellon University, November 1999. [Cited on page 23.]

[116] Seagate. New tools for the digital age. http://www.seagate.com/docs/pdf/whitepaper/DigitalAge_AVpro_Tools.PDF. [Cited on page 43.]

[117] SeaMicro. Seamicro. http://www.seamicro.com, 2010. [Cited on page 23.]

[118] Eric Seppanen, Matthew T. O'Keefe, and David J. Lilja. High performance solid state storage under linux. In *26th IEEE Symposium on Massive Storage Systems and Technologies*, May 2010. [Cited on page 102.]

[119] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, October 2010. [Cited on pages 101, 105, 107 and 110.]

[120] David C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malô, France, October 1997. [Cited on page 85.]

[121] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM*, San Diego, CA, August 2001. [Cited on page 31.]

[122] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST 2008)*, San Jose, CA, February 2008. [Cited on page 26.]

[123] Ginger Strand. Keyword: Evil, Google's addiction to cheap electricity. *Harper's Magazine*, page 65, March 2008. [Cited on page 1.]

[124] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proc. HotPower*, Vancouver, Canada, October 2010. [Cited on page 11.]

[125] Jeremy Sugerman, Kayvon Fatahalian, Soloman Boulos, Kurt Akeley, and Pat Hanrahan. GRAMPS: A programming model for graphics pipelines. In *ACM Transactions on Graphics*, January 2009. [Cited on pages 96 and 102.]

[126] Alexandra Sullivan. Energy star for data centers. EPAGreenGrid:2009. [Cited on page 7.]

[127] Alex Szalay, Gordon Bell, Andreas Terzis, Alainna White, and Jan Vandenberg. Low power Amdahl blades for data intensive computing, 2009. [Cited on pages 22 and 23.]

[128] Niraj Tolia, Zhikui Wang, Manish Marwah, Cullen Bash, Parthasarathy Ranganathan, and Xiaoyun Zhu. Delivering energy proportionality with non energy-proportional systems – optimizing the ensemble. In *Proc. HotPower*, San Diego, CA, December 2008. [Cited on pages 8 and 15.]

[129] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *Proc. ACM SIGMOD*, Providence, RI, June 2009. [Cited on page 24.]

[130] Vijay Vasudevan, David G. Andersen, Michael Kaminsky, Lawrence Tan, Jason Franklin, and Iulian Moraru. Energy-efficient cluster computing with FAWN: Workloads and implications. In *Proc. e-Energy 2010*, Passau, Germany, April 2010. (invited paper). [Cited on page 74.]

[131] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky. The case for VOS: The vector operating system. In *Proc. HotOS XIII*, Napa, CA, May 2011. [Cited on page 101.]

[132] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy Campbell. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*, San Jose, CA, February 2011. [Cited on pages 102 and 117.]

[133] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Proc. 1st USENIX OSDI*, pages 13–23, Monterey, CA, November 1994. [Cited on page 11.]

[134] Matt Welsh, David Culler, and Eric Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001. [Cited on pages 77, 101 and 111.]

[135] David Wentzlaff, Charles Gruenwald, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. An operating system for multicore and clouds: Mechanisms and implementation. In *Proc. 1st ACM*

Symposium on Cloud Computing (SOCC), Indianapolis, IN, June 2010. [Cited on pages 103, 105 and 116.]

[136] Colin Whitby-Strevens. The transputer. In *International Symposium on Computer Architecture (ISCA)*, Los Alamitos, CA, June 1985. [Cited on page 23.]

[137] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proc. 6th International Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, October 1994. [Cited on page 24.]

[138] Demetrios Zeinalipour-Yazti, Song Lin, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar. MicroHash: An efficient index structure for flash-based sensor devices. In *Proc. 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, December 2005. [Cited on page 24.]

[139] Qingbo Zhu, Zhifeng Chen, Lin Tan, Yuanyuan Zhou, Kimberly Keeton, and Jon Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)*, Brighton, UK, October 2005. [Cited on page 26.]