Improving Datacenter Energy Efficiency Using a Fast Array of Wimpy Nodes

Vijay Vasudevan
vrv+@cs.cmu.edu

October 12, 2010

THESIS PROPOSAL

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David G. Andersen (Chair)
Luiz A. Barroso[1], Gregory R. Ganger, Garth A. Gibson, Michael E. Kaminsky[2]

Carnegie Mellon University, [1]Google, Inc., [2]Intel Labs Pittsburgh

Abstract

Energy has become an increasingly large financial and scaling burden for computing. With the increasing demand for and scale of Data-Intensive Scalable Computing (DISC), the costs of running large data centers are becoming dominated by power and cooling. In this thesis we propose to help reduce the energy consumed by large-scale computing by using a FAWN: A Fast Array of Wimpy Nodes. FAWN is an approach to building datacenters using low-cost, low-power hardware devices that are individually optimized for energy efficiency (performance/watt) rather than raw performance alone. FAWN nodes are individually resource-constrained, motivating the development of distributed systems software with efficient processing, low memory consumption, and careful use of flash storage.

In this proposal, we investigate the applicability of FAWN to data-intensive workloads. First, we present FAWN-KV: a deep study into building a distributed key-value storage system on a FAWN prototype. We then present a broader classification and workload analysis showing when FAWN can be more energy-efficient, and under what conditions that wimpy nodes perform poorly. Based on our experiences building software for FAWN, we finish by presenting Storage Click: a software architecture for providing efficient processing of remote, small storage objects.

Keywords: Energy efficiency, cluster computing

1 Thesis Overview

Energy has become an increasingly large financial and scaling burden for computing. With the increasing demand for and scale of Data-Intensive Scalable Computing (DISC) [18], the costs of running large data centers are becoming dominated by power and cooling: studies have projected that by 2012, 3-year datacenter energy-related costs will be double that of server equipment expenditures [47]. On a smaller scale, power and cooling are serious impediments to the achievable density in data centers [48]: 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 server-class machines built from "commodity components." The peak power provisioned for each datacenter can exceed hundreds of megawatts, a level of consumption where choosing datacenter locations based on abundant access to cheap energy has now become popular and common [55, 35].

Given the degree to which today's largest datacenters are affected by energy, in this thesis we propose to help reduce the energy consumed by large-scale computing by using a FAWN: A Fast Array of Wimpy Nodes. FAWN is an approach to building datacenters through the use of low-cost, low-power hardware devices that are individually more optimized for energy efficiency (performance/watt) rather than raw performance alone. The abundant parallelism found in 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.

FAWN focuses primarily on data-intensive workloads. Whereas traditional HPC and transaction-processing systems perform complex computations and synchronization on small amounts of data, DISC systems often require computations (both simple and complex) across petabytes of data that tend to be more I/O-bound than CPU-bound on traditional systems. The FAWN approach balances the I/O gap between processing and storage while choosing a specific balance that optimizes for energy efficiency (in terms of work done per Joule). We focus primarily on two particular FAWN instantiations using off-the-shell hardware consisting of embedded/low-power processors paired with consumer-class Flash storage.

The challenges of using FAWN are more than simply a matter of choosing a different hardware platform. This proposal focuses on answering three research questions: First, how does a FAWN architecture change the way distributed systems are built? Second, when is the FAWN architecture appropriate, and when do traditional architectures win out? Third, how does the need for high-performance, low-latency small object retrieval on wimpy platforms inform the design of individual operating systems for DISC systems?

We propose to answer these questions through three specific in-depth explorations based on the FAWN approach to building DISC systems:

1. FAWN-KV: a deep study into building a distributed key-value storage system on FAWN.

2. Workload exploration: a broader classification and workload analysis showing when FAWN can be more energy-efficient.

3. Storage Click: a software architecture for providing efficient processing of remote, small storage objects.

The rest of this proposal is structured as follows: Section 2 describes background on the problem and the principles in which the FAWN approach is rooted. Section 3 discusses FAWN-KV – the design, implementation, and evaluation of a key-value storage system on a FAWN prototype. Section 4 provides more insight into the applicability of FAWN to other workloads. Section 5 describes Storage Click: the motivation, the proposed solution to be completed for the thesis, and related work in this area. Finally, Section 6 outlines the proposed timeline for the research.

2 Background

Datacenter energy efficiency is important given the tremendous growth of cloud services. Cluster distributed systems consisting of hundreds of thousands of machines are now prevalent around the world, and the energy-related financial burden imposed by datacenter power and cooling requirements is beginning to dominate the total cost of ownership for datacenters. At today's energy prices, the cost to power a datacenter server is only a fraction (perhaps 10%) of the total cost of ownership (TCO) of the server [15], but the proportion of a server's TCO begins to be dominated by energy when considering all energy-related costs, such as cooling and infrastructure costs.

There are a number of energy-related costs needed to power hundreds of thousands of machines in a single warehouse. The density of the datacenters that house the machines is limited by the ability to supply and cool 10–20 kW of power per rack and up to 10–20 MW per datacenter [35]. Future datacenters are being designed with a maximum power draw of 200 MW [35], or the equivalent of nearly 200,000 residential homes, requiring dedicated electrical substations to feed them.

A new datacenter can be expected to be operational for at least fifteen years to amortize the cost of construction, whereas the average server's lifetime is on the order of three to four years [15]. As a result, the upfront costs of building a datacenter to support fifteen years of growth is high. The main challenge is that the infrastructure required to support a datacenter must necessarily plan for *peak capacity*. The designed peak power draw of the datacenter informs the design and subsequent cost of building the datacenter.

As an example: datacenters builders have been focused on reducing a datacenter's *Power Usage Effectiveness*, or PUE. Simply put, the PUE is the ratio of total power draw to aggregate server power draw. The average PUE in 2009 was estimated to be 3–for every watt of power delivered to a server, the datacenter infrastructure required another two watts to deliver the power and remove the heat generated by the server. State-of-the-art datacenters have reduced the PUE to about 1.1, so that only an additional 10% of power is used to deliver power to servers (there are also additional losses when distributing the power to the individual components on the server that increase the PUE by another 10 or 20%). But providing this low of a PUE has required innovation in battery backup systems, efficient power supplies, voltage regulators, and state-of-the-art cooling infrastructures, which all require significant capital investments. While many of these can be amortized over the lifetime of a datacenter rather than the lifetime of a server, the main takeaway is that supporting the peak power draw of a datacenter comprises a major cost in a datacenter today.

The peak power of a datacenter is determined by the aggregate required 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 measured as 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.

The question we seek to answer in this thesis is: How can we improve the energy efficiency of datacenter servers?

2.1 What is FAWN?

A significant fraction of the proposed thesis revolves around **FAWN: A Fast Array of Wimpy Nodes**, an approach to building clusters using low-power, low-speed nodes at scale [8]. The central observation of this work 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 many (perhaps 10x) 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 is significantly slower per-node than a multi-GHz multicore server system balanced with multiple disks.

To perform a fixed amount of work in the same amount of time as a traditional cluster, we require using more components in parallel. This implicitly relies on the ability of a workload to parallelize well, also known as an "embarrassingly parallel" workload. The FAWN approach may not improve energy efficiency for workloads that cannot be parallelized or whose computation requires a serialized component in the computation (because of Amdahl's Law [6]). Fortunately, many (but not all [14, 31]) DISC workloads are embarrassingly parallel because of the data-oriented nature of the workloads. It is for these types of workloads that we believe the FAWN approach will work well.

2.2 Metric: Work done per Joule

Evaluating large systems using only performance metrics such as throughput or latency is slowly falling out of favor as energy and programming ease inform the design of modern large scale systems. There are several metrics for energy efficiency, but the one we focus on is "work done per Joule" of energy, or equivalently, "performance per Watt."

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: it relies on being able to parallelize workloads, which is often explicitly provided by data-intensive computing models such as MapReduce [22] and Dryad [33] 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—for example, 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.

One metric we do not study in detail is the cost of software development. As we will show in this thesis, 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, though it is likely that software development costs will necessarily work in favor of "brawnier" platforms [31].

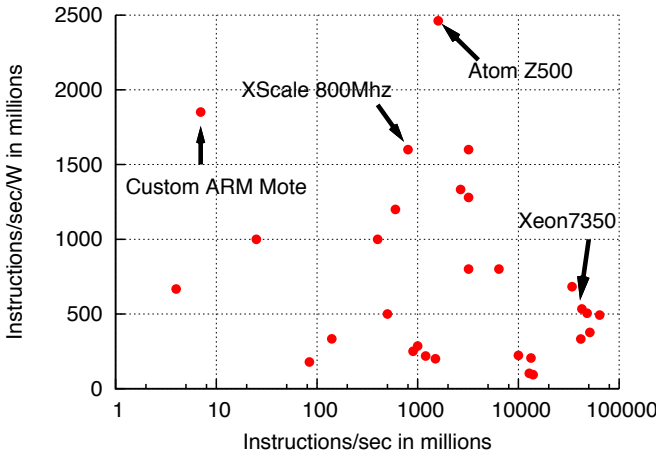

Figure 1: Max speed (MIPS) vs. Instruction efficiency (MIPS/W) in log-log scale. Numbers gathered from publicly-available spec sheets and manufacturer product websites.

2.3 Principles

The FAWN approach to building *balanced* cluster systems has the potential to achieve high performance and be fundamentally more energy-efficient than conventional architectures for serving massive-scale I/O and data-intensive workloads.

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

2.3.1 CPU Trends

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. The 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 come at the cost of energy efficiency. Branch prediction, 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 L2/3 caches [32]. These techniques do not increase the speed of basic computations, but do increase power consumption, making faster CPUs less energy efficient.

A primary energy-saving benefit of Dynamic Voltage and Frequency Scaling (DVFS) for CPUs was its ability to reduce voltage as it reduced frequency [61], but 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 the efficiency benefit of DVFS today [56].

FAWN Approach: A FAWN cluster's slower CPUs dedicate more transistors to basic operations. These CPUs execute significantly more *instructions per Joule* than their faster counterparts (Figure 1): multi-GHz superscalar quad-core processors can execute approximately 100 million instructions per

Joule, assuming all cores are active and avoid stalls or mispredictions. Lower-frequency in-order CPUs, in contrast, can provide over 1 billion instructions per Joule—an order of magnitude more efficient while still running at 1/3rd 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 [31], and as we show throughout this work, unavoidable fixed costs can eliminate the benefits of extremely slow but energy-efficient processors.

2.3.2 Memory trends

The previous section examined the trends that cause CPU power to increase drastically with an increase in designed 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 power-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 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. The reasons for this are several: 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, and 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. 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 out of the box.

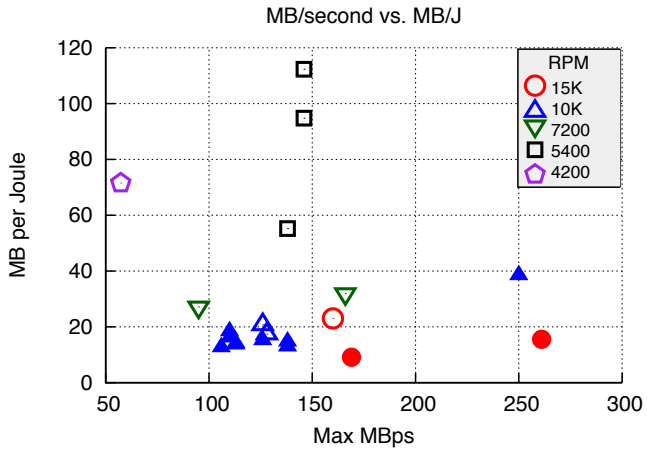

Figure 2: Power increases with rotational speed and platter size. Solid shapes are 3.5" disks and outlines are 2.5" disks. Speed and power numbers acquired from product specification sheets.

2.3.3 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 demonstrates this tradeoff by plotting the efficiency versus speed for several modern hard drives, including enterprise, mobile, desktop, and "Green" products.[1]

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.

Implications: Energy-efficient clusters constrained by storage capacity requirements will continue to use 2.5" disk drives because they provide the lowest energy per bit, but Flash storage will continue to make in-roads in the datacenter, particularly for the remote small object retrieval systems that large clusters 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 appropriate [8].

[1] The figure uses MB/s data from vendor spec sheets, 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.

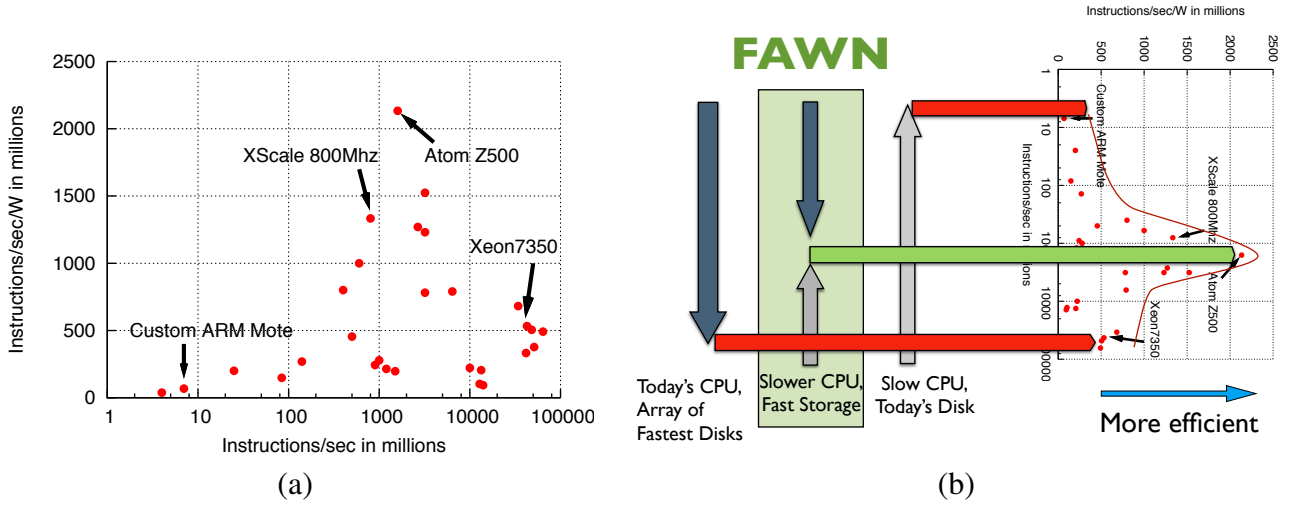

Figure 3: (a) Processor efficiency when adding fixed 0.1W system overhead. (b) A FAWN system chooses the point in the curve where each individual node is balanced and efficient.

2.4 Fixed power costs

Non-CPU components such as memory, motherboards, and power supplies have begun to dominate energy consumption [14], 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 [58]. Despite improved power scaling technology, 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 3 plots processor efficiency when adding a fixed 0.1W cost for system components such as Ethernet. Because powering 10Mbps Ethernet dwarfs the power consumption of the tiny sensor-type processors that consume only micro-Watts of power, their 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.

2.5 System balance

Balanced systems are a principle required for energy-efficient clusters [50], but balance alone does not maximize energy efficiency. Figure 3b takes the speed vs. efficiency graph for processors in Figure 3a and illustrates where several different "balanced" systems operate in the curve. The FAWN approach chooses a particular balance where each individual node is optimized for energy efficiency.

This highlights the importance of combining both balance and efficiency together. While balance is necessary to avoid wasting resources (and energy), even a balanced system can be inefficient if the amount of energy to operate at a higher speed is disproportionately high. The FAWN approach focuses on finding this balanced and efficient point in the curve. While the specific point on the curve that optimizes for energy efficiency will change over time, the general shape of the curve should hold for the foreseeable future.

Figure 3a above shows the speed vs. efficiency curve for processors as a proxy for entire system efficiency. Individual components that exhibit the superlinearity in speed vs. power will further shape the entire system curve much like in Figure 3a, whereas components that are perfectly linear in speed vs.

power (a constant efficiency) can be factored out. Fixed power costs, on the other hand, will further push the optimal point towards brawnier systems.

2.6 Proportionality vs. Efficiency

Another major challenge for datacenters is that, despite being provisioned for peak power, the *average* utilization of the datacenter can be quite low—anywhere from 5%[37] to 20% [14]. Ideally, the 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 peak power draw), a feature termed "energy proportionality" [14]. Unfortunately, individual servers are not energy proportional because of their high fixed power draw even when idle: servers can consume 30-50% of their peak power at 0% utilization. Worse yet, 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 [15].

Achieving energy proportionality in a datacenter thus may require "ensemble-level techniques" [58], such as turning portions of a datacenter off completely [7]. 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 [14], 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.

Although providing energy proportionality is a complementary approach to saving energy in datacenters, this work focuses only on energy efficiency.

3 FAWN-KV

The FAWN-KV distributed key-value storage system is a system we designed and implemented to help answer the question: How does a FAWN architecture change the way distributed systems are built? In this section, we briefly articulate the reasons for targeting this workload, the unique challenges that we had to address to answer the question, and the relevant portions of this joint work that I plan to include in my thesis.

Large-scale data-intensive applications, such as high-performance key-value storage systems, are growing in both size and importance; they now are critical parts of major Internet services such as Amazon (Dynamo [23]), LinkedIn (Voldemort [45]), and Facebook (`memcached` [39]).

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 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 system 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 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.

FAWN-KV is designed specifically with the FAWN hardware in mind, and is able to exploit the advantages and avoid the limitations of wimpy nodes with flash memory for storage. Specifically, the FAWN hardware poses several challenges:

1. FAWN nodes have a lower memory capacity per core,
2. Flash is relatively poor for small random writes,
3. More nodes in a FAWN system leads to more frequent failures than a traditional system with fewer nodes.

FAWN-KV is a system designed to deal with these challenges that both uses most of the available I/O capability of each individual wimpy node, and together harnesses the aggregate performance of each node while being robust to individual node failures.

The thesis work will include the relevant details about the design and implementation of our memory-efficient in-memory hash index, our log-structured key-value module, FAWN-DS, to support fast sequential writes and fast random reads on flash, and our mechanisms for restoring replication of data efficiently on failures and arrivals into the system. Each of these components will be supported with experimental data verifying the efficacy of the design and implementation. Some of these details will be culled from our FAWN-KV paper, to which we refer the reader [8].

4 Workload Analysis

Our evaluation of FAWN-KV demonstrated that significant energy efficiency benefits were attainable using the FAWN architecture for an I/O-bound workload. But these benefits could only be fully reaped following a redesign of the distributed system that runs atop a cluster of FAWN nodes.

Of course, in a datacenter, not all workloads will be completely I/O-bound to the same degree, and some may not be I/O-bound at all. The goal of the next part of this thesis is to understand when the FAWN architecture applies to a wider variety of workloads, and what features critically determine this applicability.

Motivation for this research comes not only from our own interests in exploring the workload space for FAWN, but interest within the broader research community as well. Some researchers have discussed the potential for using wimpy nodes for other types of DISC and HPC workloads [57, 29, 19, 9, 28, 49] to varying degrees of success in improving energy efficiency, while others note that wimpy nodes may not be the most efficient platform for many out-of-the-box applications, instead arguing for a hybrid approach [20, 38]. Some workloads are clearly not good targets for FAWN (they do not exhibit the workload parallelism required for this approach to apply well [38]). Others are seemingly I/O-bound based on intuition, but data shows that the efficiency of those workloads on FAWN is lower than traditional systems [20].

Based upon the principles in Section 2, wimpy nodes are expected to be fundamentally more efficient because of their lower complexity and lower speed, up until the point where fixed costs dominate. Understanding the disparity between expectation and result is the major motivation for this part of the thesis.

4.1 Approach

Many have attempted to apply the FAWN approach to existing software systems, measuring the energy efficiency benefits to be small or non-existent in some cases [20, 38], or bring with them other caveats, such as increased response time variability [49]. Existing software systems can be complex to analyze, so our approach in this work is to create and/or perform microbenchmarks on two types of systems, FAWN nodes and traditional nodes, comparing energy efficiency and trying to understand the fundamental reasons for the results. Microbenchmarks allow us to isolate the individual features of a particular complex workload that may influence the energy efficiency comparison of real applications.

4.2 Taxonomy

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

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

The first of these 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, and represent the low-hanging fruit for the FAWN approach, as described in our earlier work [8]. The second category includes CPU and memory-bound workloads, where the running time is limited by the speed of the CPU or memory system.

The last two categories represent workloads where the FAWN approach may be less useful. Latency-sensitive workloads require fast responses times to provide, for example, an acceptable user-experience; anything too slow (e.g., more than 50ms) impairs the quality of service unacceptably. 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).

The thesis will analyze and present results from this categorization and series of microbenchmarks. In addition, we analyze the impact of running below peak utilization on wimpy hardware, showing that fixed power costs play an even larger role in these circumstances. An example of an insight from this work is depicted in Figure 4, which shows the efficiency of a memory-intensive floating point matrix-transpose multiply microbenchmark on both a brawny and wimpy platform. The measured energy efficiency is affected by the size of the matrix being multiplied because of cache size, and the most efficient platform for a particular matrix size flips back and forth because of the differences of the cache sizes between the two architectures. Such discontinuities can greatly affect the measured energy efficiency comparison between brawny and wimpy platforms.

This work represents mostly prior work that is summarized in our e-Energy 2010 paper [59] as well as our winning 10GB 2010 Joulesort submission [60]. A summary of findings from the paper is as follows:

- The FAWN approach works well for embarrassingly-parallel workloads but not for those with strict latency targets that FAWN systems are unable to meet.

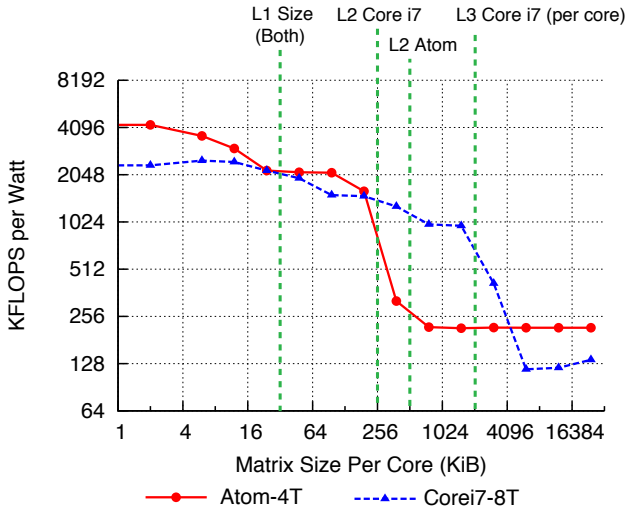

Figure 4: Efficiency vs. Matrix Size. Green vertical lines show cache sizes of each processor.

- Differences in cache and memory size play a large role when comparing the energy efficiency of wimpy and brawny platforms.
- Code optimized for a particular platform can skew the energy efficiency comparisons, but may be unavoidable in practice.
- Fixed power costs largely dominate energy efficiency metrics, particularly when systems are not used at 100% utilization.

5 Storage Click

In the course of upgrading our FAWN infrastructure to state-of-the-art wimpy platforms, we have discovered that there are a few reasons why today's operating systems[2] inefficiently handle small I/O requests, and this inefficiency plays a large role in the applicability of the FAWN approach. First, small block I/O handling in the kernel has been tuned to the characteristics and performance of rotating disks, and modern Flash devices break many of these assumptions. Second, specialized distributed small object stores like FAWN-KV stress the interface between software layers and hardware, resulting in frequent context switches, data copies, and high network and storage interrupt loads that significantly reduce performance, particularly on wimpy platforms.

The primary question that we try to answer is: "how should operating systems be modified to efficiently support distributed small object stores?" To answer this, we propose **Storage Click**, a software architecture for high-performance, efficient processing of remote, small storage objects. Storage Click consists of three major components:

1. Improved interrupt mitigation and polling algorithms for flash storage.
2. Optimization of the block I/O codepath for flash storage using the `multiread/multiwrite` interface.

[2]We focus on Linux because it is both popular and open source. We believe that other popular operating systems such as Windows experience similar issues given the magnitude of improvement in IOPS rate of flash devices in the last three years.

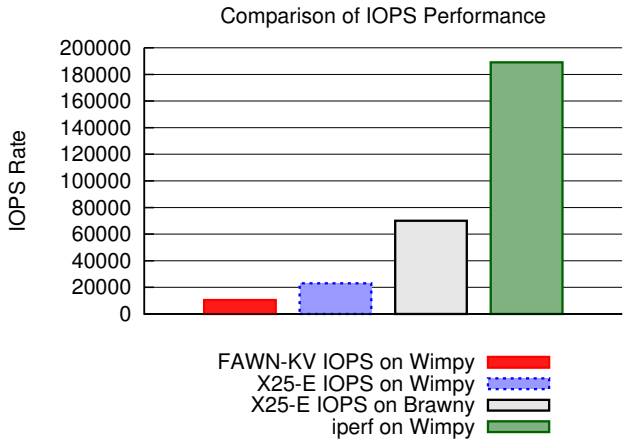

Figure 5: IOPS rate of various benchmarks and configurations. The right two bars show the peak capabilities of the X25-E and network packet reception, whereas the left two bars show the FAWN-KV application performance and X25-E IOPS rate obtained on a wimpy platform. The wimpy platform is unable to saturate the raw capabilities of both the network and the storage in these experiments.

3. Streamlined in-kernel processing for simple fast-path operations and deferral to userspace for complex slow-path operations.

We begin our discussion by briefly detailing the experiments that inspired Storage Click, and then describe each component in more detail, in order of decreasing maturity of exploration in the area.

5.1 Motivation

Benchmarking FAWN-KV on modern low-power nodes: FAWN-KV implements a distributed key-value store optimized for flash storage. A client issues key-value requests either directly to backend FAWN nodes or through a front-end intermediary. We implemented a Java-based client library that uses Thrift [1] to communicate directly to the backend using the FAWN-KV protocol (put/get), and also implemented a FAWN-KV YCSB module in order to use the Yahoo! Cloud Serving Benchmark [21]. We used a Core i7-based client load generator and one FAWN node consisting of a 1.6GHz single-core Atom (N450), 2GB of DDR2 DRAM, and one Intel 32GB X25-E Flash solid state drive.

The left-most bar in Figure 5 shows the number of key-value requests served for this benchmark. Depicted in the same graph is the raw 512-byte IOPS rate of the X25-E device measured using a microbenchmark tool (fio [2]) on the same wimpy Intel Atom N450 platform, and the IOPS rate of the device as measured on a brawny Core i7 platform.[3] The farthest right bar shows the number of 1KB packets processed by the same wimpy platform using the `iperf` benchmarking tool, demonstrating that the wimpy platform is capable of receiving (but not processing) nearly 200,000 packets per second.

The two major takeaways of this graph are as follows: First, the raw IOPS rate of the X25-E obtained using the wimpy platform is a factor of three worse than the theoretical capabilities of the device (as shown by the IOPS rate obtained on a multi-core brawny platform). Second, the wimpy platform is

[3]Note that the application performance does not match the performance of raw I/O for several reasons: First, the microbenchmark issues requests directly to the device in 512B sectors rather than through the filesystem layer in 4KB pages; second, the application must process and interpret the data; third, the application interacts with the networking layer in addition to local storage, whereas the microbenchmark uses local storage only.

capable of receiving high packet loads, but the the resulting application-level IOPS rate is more than an order of magnitude worse because of the combination of network packet processing and storage I/O. These experiments demonstrate the difficulty of achieving efficient remote small object storage, particularly on wimpy platforms. Storage Click is the umbrella project that we propose to address this problem. We discuss the major components of Storage Click independently in the next several sections.

5.2 Interrupt Mitigation for Flash Storage

While flash hardware has improved tremendously in the past three years, software support from applications and operating systems has lagged behind. The CPU-I/O gap has narrowed so quickly that many OS assumptions about the cost of an I/O have been shattered completely: a random seek on a magnetic disk has plateaued at about five milliseconds on average, whereas a random read from flash takes only 100 microseconds (fifty times lower) and is dropping by a factor of two every year [16]. Operating systems software stacks are only now being revisited/rehauled to accommodate solid state devices [52, 11, 13].

Modern flash devices are capable of providing tens of thousands of IOPS from a single device: the X25-M drive can perform 70,000 512-byte random reads per second (when reading directly from the device instead of the filesystem, which restricts access to 4KB pages), and the Fusion-IO ioDrive can perform nearly 160,000 512-byte IOPS—three orders of magnitude higher than a single magnetic disk. Without techniques to reduce interrupt load, balanced wimpy and brawny systems will spend a significant portion of time handling interrupts instead of performing application work.

5.2.1 Background

There are two ways to reduce the load of interrupts for high IOPS devices: interrupt coalescing and interrupt mitigation. **Interrupt coalescing** is a technique performed by an I/O device that combines the interrupt generated by a single event with subsequent events in time, effectively interrupting a host system only once for multiple events. **Interrupt mitigation** is a technique performed by the host operating system that switches from interrupt-driven operation to polling-based operation, which is particularly useful during high load situations. These techniques are complementary and help reduce interrupt load on the host system.

Interrupt coalescing support exists for high-speed network cards, but does not yet exist for flash devices. Therefore, we must currently rely on interrupt mitigation support in the operating system to reduce interrupt load for flash devices.

Existing interrupt mitigation for block devices: 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 general framework of NAPI and has shown improvements to IOPS performance and CPU utilization for traditional systems [12]. 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 (a feature also required to benefit from the internal parallelism of modern flash devices). If the number of IOs retrieved on an

initial interrupt is high, the blk-iopoll system remains in polling mode to check if more commands will be completed soon.

However, we have had mixed success in straightforwardly applying these changes to wimpy systems. 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, single-core machine such as the Intel Atom N450, 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.

Figure 6 illustrates how the NAPI mitigation approach scales with load. At low load, the cost of performing an interrupt is low because the system is already underutilized. At high load, switching to polling improves overhead because staying polling is more efficient when there is always more work to do. However, at medium load, the cost of switching to polling to do very little additional work is a performance penalty. We note that similar challenges affect network processing using NAPI when a fast machine operates at below 100% utilization [12].

5.2.2 Research Ideas

Our initial attempts to improve upon the interrupt mitigation algorithm used by NAPI have been fruitful, fundamentally relying on trading off slightly higher latency for higher throughput. Whereas the NAPI interrupt mitigation approach uses a spin loop to perform polling, an alternative approach can be taken by using event- and timer-based logic [10] to decide when to actually service requests from a network device [51], 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. The spinloop method used by NAPI scales more gracefully as load increases as we depict in Figure 6a, whereas our deferral-based polling avoids the suboptimal operation load region (Figure 6b) from which NAPI suffers. Specifically, deferral introduces a higher cost at cost at low load, 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.

5.2.3 Preliminary Results

We have used the flexible I/O tester [2] to measure the rate of retrieving 512-byte random IOPS directly through the block layer. With our Intel single-core Atom N450 paired with an X25-M flash device on the default 2.6.32 Linux kernel, we measured the system as capable of performing around 23,000 IOPS. We then patched the kernel to enable blk-iopoll support for the AHCI driver, and discovered that performance dropped to 20,000 IOPS. We measured the distribution of completions and found that most interrupts completed only 1 command before re-enabling interrupts. Therefore, the performance drops due to the cost of disabling interrupts, polling to complete one command, then re-enabling interrupts. This confirms that the interrupt from the device occurs before enough commands can be completed, so that the internal parallelism of the X25-M cannot be saturated, and the OS does extra work on top of an already heavyweight interrupt event.

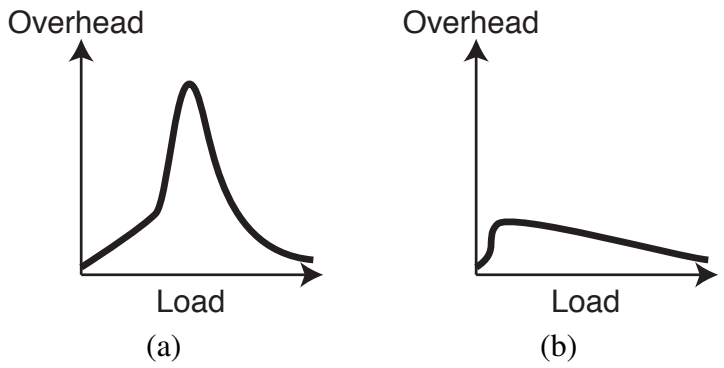

Figure 6: Illustration of load vs. overhead. (a) Spin-loop polling can incur a high overhead during periods of medium load. (b) Deferral-based polling avoids the high cost of switching from interrupt to polling when polling isn't effective.

Other inefficiencies: After profiling the run of this benchmark, we discovered a few issues that limit the performance of random block I/O on the wimpy platform, some stemming from the assumption that block I/Os are expensive, as they would be for magnetic disks:

1. The benchmark program calls `gettimeofday` several times per request to precisely measure the latency of each request for statistics. Our system uses the HPET chip (High-Performance Event Timer) to perform precise timing, which carries a high-overhead.

2. On each block operation, the kernel calls a function to add entropy to the kernel's entropy pool for /dev/random.

To address these issues, we modified the benchmark utility to read the timestamp counter directly from the processor, avoiding a system call and an expensive HPET interaction. We also disabled the entropy pool generation on each block request. Finally, we compiled a kernel to specifically support the Intel Atom using the Intel ICC compiler. These changes were simple workarounds to issues in the block I/O codepath and motivates the proposed multiread/multiwrite interface described in Section 5.3.

Results: With all of these changes, we were able to improve the baseline performance from 20,000K IOPS (with blk-iopoll enabled), to 35,500K IOPS, nearly doubling baseline performance. These changes were not simply optimized for one platform: we measured the same changes on an Intel Nehalem system paired with three Intel X25-M drives, and performance improved from 110K IOPS by default to 180K IOPS with our changes.

These modifications relied on some magic constants for optimal performance. One challenge is to determine *when* to switch from interrupt-driven mode to deferred-polling mode to optimize for the best IOPS rate within a latency bound, and for how long to defer before performing the polling work. Currently, our changes come at the cost of high latency, because the deferring strategy introduces a worst case of several hundreds of microseconds of delay. For today's flash devices that have a OS-perceived access time of 200 microseconds or more, this is acceptable, but we are investigating ways to reduce this delay as flash access times continue to drop.

5.3 Optimizing Block I/O for Flash: Multi-read and Multi-write

The difficulty of providing high performance I/O on wimpy platforms is due to a confluence of several challenges:

- **High interrupt load:** As discussed previously, a modern Flash device can deliver hundreds of thousands of I/Os per second.
- **Long, inefficient block I/O codepath:** The code executed for each individual block request is not optimized for the low latency of flash (e.g., the entropy calculation on each block I/O mentioned above).
- **Internal flash parallelism:** Modern flash devices require many parallel requests to saturate their IOPS capabilities [44] because they internally place data on multiple independently-accessible flash planes.

While interrupt mitigation techniques can help avoid interrupting a system tens to hundreds of thousands of times per second, achieving high IOPS on wimpy platforms may still struggle to issue enough commands to flash (in parallel) before the device interrupts the host system.

5.3.1 Research Idea: Multi-read and Multi-write

A solution to this problem is to amortize the cost of executing the code path by issuing *multiple* block I/Os to the device at once. This provides several benefits: First, it can help reduce interrupt load by ensuring that the flash device receives (and thus completes) commands in bursts so that an interrupt will allow the OS to process multiple, related commands at once. Second, batching the request sent through the OS reduces the number of times the block I/O codepath must be executed. Third, issuing multiple requests at the same time inherently can take advantage of the internal parallelism of modern flash devices.

Thus, we propose the use of the **multi-read** and **multi-write** interface. The existing `read()` and `write()` calls primarily deal with one request at a time; we propose to extend the OS interface to include an analogous set of `multiread()` and `multiwrite()` calls to read and write from multiple locations off of a device in one request.

Note that the `multiread()` and `multiwrite()` interface is very different from the existing scatter-gather I/O `readv` and `writev` interface: scatter-gather read I/O is used to reduce the number of system calls and data copies involved in reading a sequential portion of data from a file descriptor into several different userspace buffers, whereas our goal is, logically, to read from multiple different locations of a file descriptor into at least one buffer. Similarly, `multiwrite()` should write from several different buffers to several different offsets to a file descriptor, whereas the existing `writev()` call writes sequentially to the existing offset of the file descriptor.

Implementation plan: One important question we hope to understand is: how far do we need to push the multiread abstraction in the OS to reap its benefits? For example, a simple multiread implementation could simply turn a multiread system call into an iterative set of calls to the read() call within the OS. This would not reduce the number of traversals within the block I/O code path, but it would avoid system call overhead.

Another approach is to modify the existing interface to the block I/O system to support multiple buffers and offsets in many of the existing structures (e.g., struct request). This would reduce the number of times the block I/O codepath is executed, issuing and constructing the final requests only when

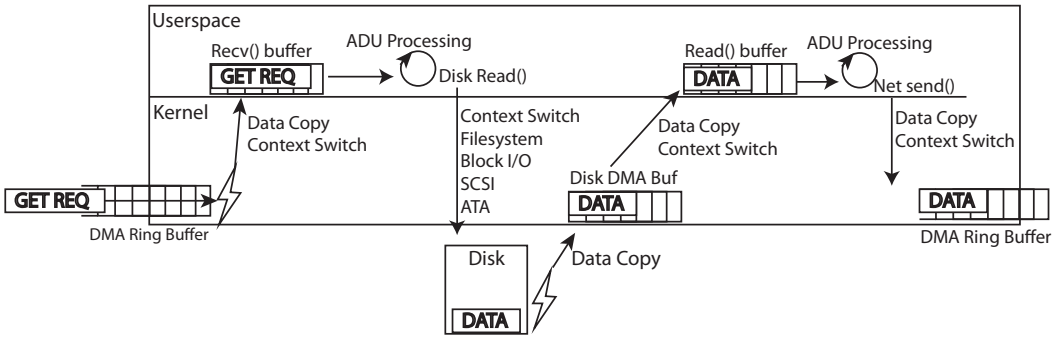

Figure 7: Lifecycle of an incoming key-value get request. Each request incurs several data copies, interrupts (depicted by lightning bolts), and context switches.

interacting directly with the device, which understands only existing single-op commands. This could be done at the block layer by coalescing multiple independent read() calls in time, or exposed as a multiread() system call, which would avoid the need to add logic to determine when to perform batching of I/O in the kernel.

The extreme, stretch goal would be to explore the possibility (and benefit) of pushing a multi-read interface directly to the device, but such an exploration requires device support that does not exist in current off-the-shelf flash devices.

5.4 In-kernel Fast Path ADU Processing

Userspace processing of small remote storage requests (referred to here as Application Data Units or ADUs) in today's operating systems must incur several context switches and data copies that can reduce performance, particularly on wimpy platforms. Figure 7 depicts the lifecycle of a key-value request over the network being processed by a userspace application.

The overhead of processing a single key-value request is high, particularly when the amount of Application Data Unit (ADU) processing is small. There are four data copies, two interrupts, and four context switches for a single request.

Context switches can be amortized by batching together several ADUs and processing them at the same time, increasing the average latency of responses depending on the degree of batching performed, but data copies are unavoidable in typical userspace processing.

There are two contrasting approaches to eliminating context switches and data copies: performing all work in userspace and performing all work in the kernel. When ADU processing involves no kernel transitions, userspace networking driver support can eliminate context switches, while RDMA network support can help eliminate data copies by reading data from the network directly into the application address space. A major benefit of this approach is that remaining in userspace retains application programming ease, while the major drawbacks are that accessing storage through system calls incurs context switches and datacopies, and userspace operation requires implementing networking driver support and protocols such as TCP in userspace instead of using the mature codebase in the kernel. The opposite approach is to perform *all* processing in the kernel, as in-kernel web servers and load balancers do [34, 5, 4]. This can avoid all context switches and data copies but restricts the programmer's flexibility to write complex code.

5.4.1 Research Issues

Storage Click attempts to find a middle-ground between the two extremes: it must avoid the data copies and context switches that reduce performance for userspace ADU processing, and must enable applications that require complex ADU processing that may be untenable to implement in the kernel. We therefore need to separate the work to perform complex/slow-path processing in userspace, while performing common fast-path operations (such as caching, storage, and routing) in the kernel.

There are several questions we need to answer to properly implement this split functionality:

- How do we balance the complexity of in-kernel functionality with the performance benefits doing so might provide?
- What interface do we present to application developers to allow them to build ADU-specific functionality in the kernel?
- How do we communicate slow-path operations to the userspace program?
- How do we ensure ordering and safety of shared data between the userspace and kernel implementations?

We are building upon preliminary work on Storage Click that provides the substrate for efficient networking and ADU processing in the kernel. Specifically, we have an in-kernel network event library that allows us to process ADUs in the kernel with minimal data copying. We plan to extend this with protocol-specific parsing libraries (e.g., libraries to read Thrift/Protocol Buffer data formats). The system would then link with a user-written policy that receives "batches of ADUs", where it decides on a per-ADU basis whether to process it using fast-path or slow-path.

An example policy would be to take put/get key-value requests in FAWN-KV and send them to the fast-path handler, which interacts with an in-kernel memory cache and a direct interface to storage and networking; all other requests (such as those used for maintenance operations) would be directed through the slow-path code where it forwards the request to the userland application.

5.5 Plan of Action

Implementing a full, flexible Storage Click system is a significant undertaking. The goal of this part of the thesis work will be to:

1. Measure existing systems' ability to perform small I/O,
2. Identify bottlenecks and design deficiencies in current implementations and propose solutions to these deficiencies,
3. Implement the storage/flash-interface side of the Storage Click infrastructure (multiread/multiwrite),
4. Implement a rudimentary fast-path vs. slow-path mechanism.

We also note that the three major research components proposed above (interrupt mitigation for flash, multiread/multiwrite, and in-kernel fastpath ADU processing) will have a dependent interaction with each other: for example, the multiread interface may change the way that interrupts are returned from the device: the interrupt mitigation techniques used may depend on the semantics of this behavior. This work will include measuring the impact of various parameters of each component and its interaction.

5.6 Related Work

Storage Click follows a long line of work in improving the efficiency (and hence performance) of networked systems.

Efficient Networked Servers: With the growth of the Internet in the late 1990s, there were several research projects focused on web serving dynamic content and high-speed packet processing. Lazy Receive Processing [25] and IO-lite [43] are examples of research aimed at improving efficiency by eliminating data copies for I/O and ensuring fair and stable performance during periods of overload. We similarly take approaches to reduce the number of redundant data copies, but do not focus explicitly on handling graceful degradation on overload. Facebook's implementation of memcached has been reported to support nearly 1 million memcache operations per second, but the average load on any server is only about 100,000 requests/sec, far enough from overload that degradation is relatively less important [54].

Flash [42] and SEDA [62] are both webservers designed for high-performance and high-concurrency using event-driven architectures. Modern high performance systems like memcached use the libevent framework [46] for event-driven scalable polling of thousands of network sockets from userland. Storage Click is aimed at providing similar functionality of event-driven operation in the kernel (klibevent) to reduce the overhead of context switches.

AFPA [34] is a modular in-kernel processing architecture for high-performance networked servers. This work focuses mostly on webserving, and they demonstrate the significant (three-fold) benefits of avoiding data copies and context switches, performing application processing in the same software interrupt context as the TCP/IP stack, and slimming down the code path for requests. Their system did not provide a fast-path/slow-path mechanism, nor did they focus heavily on performance of small random I/O or interfacing with modern flash devices.

RAMCloud [41] proposes the use of networked memory servers as the basis for data-intensive computing clusters. One goal is to provide end-to-end RPC latencies of just 10 microseconds. Even a fast 10Gbps network switch has a switch-to-switch latency of 1 microsecond, so meeting this target requires processing an RPC at the server in a few microseconds. To achieve this goal, RAMCloud must similarly optimize the operating system stack to minimize any extra work. One difference between our approaches is that in RAMCloud, all processing can logically remain in userspace: the in-memory database can be run in a userspace program with the incoming network data copied directly into userspace memory using RDMA or a userspace network driver. Today's Ethernet currently does not support RDMA (though support has been announced [3]), and implementing a different userspace network driver for each card may be difficult. In contrast, Storage Click currently requires interfacing with devices that only the kernel can access. The separation of simple fast-path operations from complex slow-path operations provides a middle-ground between implementing all logic in the kernel and supporting all I/O in userspace.

FlexSC [53] proposes the use of exception-less system calls, noting that the impact of system calls on high-performance servers implemented in userspace is high due to cache pollution and pipeline flushes. Their implementation uses shared memory pages between userspace and kernel to coordinate asynchronous system call execution without requiring a mode switch. This technique may help improve the performance of the slow-path by reducing the cost of system calls, but it does not eliminate the data copies and interrupt overheads for I/O-intensive work.

Packet Processing: The speed of networking continues to increase every year, with 10Gbps Ethernet in datacenters today, 40Gbps Ethernet/Infiniband either on its way or in HPC clusters, and 100Gbps Ethernet/Infiniband proposals. Unfortunately, the speed of individual cores has hit a plateau, and today's systems improve single machine throughput by moving to many-core processing architectures. Worse yet, the interrupt load imposed by the network has traditionally scaled with the throughput of the network.

Techniques like Receive-Side-Scaling have been designed to distribute the work of packet processing among many cores, ensuring that similar flows or packets are assigned the same core to avoid cache thrashing. To avoid dedicating proportionally more cores to processing network packets, our research is focused on exploring the tradeoff of latency versus throughput in the context of datacenter workloads.

Storage Click inherits its name from the Click Modular Router [36], which provides a flexible packet processing framework in the kernel for high performance. While Click deals primarily with packets, we must deal with "application data units" that may involve application-specific logic: retrieving data from local storage such as RAM (for caching) or flash/disks (for persistent storage), sending such processed ADUs over the network. Storage Click's implementation will likely not focus as much on modularity as Click, however.

Recent work on using GPUs for packet forwarding has identified that the existing Linux kernel stack spends over 50% of its processing cycles with memory allocations and dealing with high per-packet overheads [30]. Unfortunately the code has not been made available, so we will likely have to re-implement some of their optimizations and modifications to the stack. But this effort is in support of the research contribution that is more focused on the in-kernel interfaces required to allow flexible programmability and high performance on wimpy nodes. PacketShader builds upon ideas presented in the RouteBricks [24] project, an architecture and implementation of scalable software routers. Key to high performance in RouteBricks is the pinning of cores to queues to prevent each core's cache from getting thrashed. We imagine that similar techniques will be useful in Storage Click, e.g., pinning a core (or cores) to polling network I/O, interfacing with a disk, etc.

Interrupt coalescing/mitigation in network devices became popular in the late 1990s as a way to deal with the receiver livelock problem [25] and to improve the efficiency of high-speed packet reception in FreeBSD [51]. Linux soon adopted the same technique and called it NAPI ("New API"), allowing for easy configuration of when to switch between polling and interrupt modes. Recently, the same infrastructure has been created in the Linux I/O stack for block devices, known as blk-iopoll [11]. Based on some initial experiments on FAWN systems, blk-iopoll can actually *reduce* performance when not properly tuned; Storage Click tries to improve upon existing interrupt mitigation techniques, particularly for Flash SSDs that do not support interrupt coalescing on the device itself.

Extensible Operating Systems: Implementing logic in the kernel necessarily makes it more difficult to integrate userspace applications for the average developer. Many of our design decisions are informed by the work on extensible operating systems, such as SPIN [17], the Exokernel project [26, 27], and Scout OS [40]. The Exokernel proposed eliminating all abstractions from applications to let them manage the underlying hardware as they wished, with the exception of ensuring security/isolation of the underlying hardware. SPIN took an alternative approach of "downloading" code into the kernel and allowing the user applications overwrite kernel functions where applicable, providing safety using language constructs and capabilities. Storage Click conceptually follows more of the Exokernel approach, implementing many of the lower-level actions (network handling, storage I/O, memcached) itself and allowing users to use these exported interfaces. In contrast, Storage Click looks at a relatively restricted set of application functionality in the kernel, allowing most of the existing kernel implementations to remain as-is.

6 Timeline

The chart in Figure 8 describes the timeline for the thesis research. The timeline factors into account job interview season between February and May. The three month period dedicated to writing the dissertation also will allow for some lag time to complete portions of the research needed to complete the thesis.

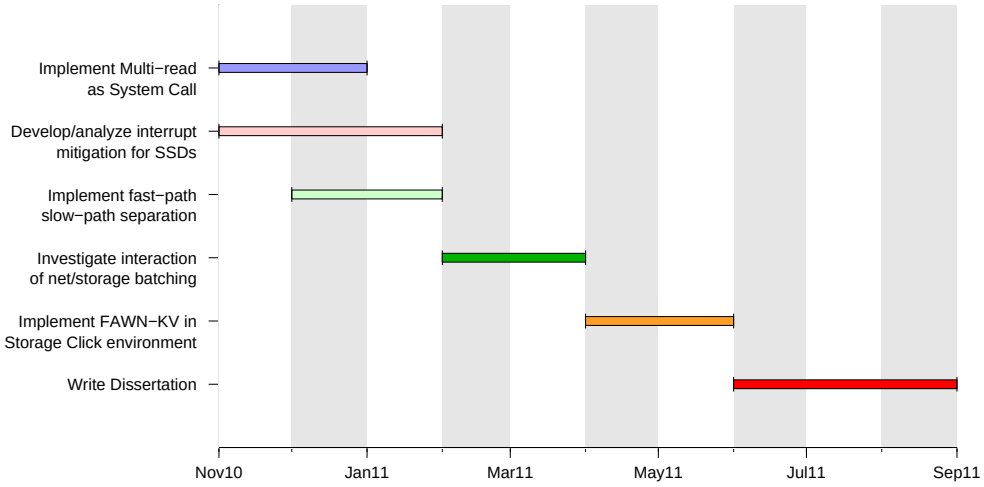

Figure 8: Timeline for thesis work

References

[1] Apache Thrift. `http://incubator.apache.org/thrift/`.

[2] Flexible I/O Tester. `http://freshmeat.net/projects/fio/`.

[3] Infiniband trade association announces RDMA over Converged Ethernet (RoCE); new specification to bolster low latency ethernet adoption in the enterprise data center. `http://www.infinibandta.org/content/pages.php?pg=press_room_item&rec_id=663`.

[4] The Linux Virtual Server Project. `http://www.linuxvirtualserver.org/`.

[5] The TUX kernel webserver. `http://www.redhat.com/docs/manuals/tux/`.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] Jens Axboe. blk-iopoll, a polled completion API for block devices. `http://lwn.net/Articles/346187`.

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

[13] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid RAM/SSD memory allocation made easy. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010. (Work in Progress Report).

[14] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[15] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[16] Andreas Bechtolsheim. Memory technologies for data intensive computing. In *Proc. 13th Intl. Workshop on High Performance Transaction Systems*, Pacific Grove, CA, October 2009.

[17] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 267–284, Copper Mountain, CO, December 1995.

[18] 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.

[19] 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.

[20] 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.

[21] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, Indianapolis, IN, June 2010.

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

[23] 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.

[24] 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.

[25] Peter Druschel and Gaurav Banga. Lazy receiver processing (lrp): A network subsystem architecture for server systems.

[26] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, CO, December 1995.

[27] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Hector M. Briceno, Russell Hunt, and Thomas Pinckney. Fast and flexible application-level networking on exokernel systems. volume 20, pages 49–83, February 2002.

[28] 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.

[29] 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.

[30] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. In *Proc. ACM SIGCOMM*, New Delhi, India, August 2010.

[31] Urs Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 2010.

[32] Penryn Press Release. `http://www.intel.com/pressroom/archive/releases/20070328fact.htm`.

[33] 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.

[34] Philipe Joubert, Robert B. King, Rich Neves, Mark Russinovich, and John M. Tracey. High-performance memory-based web severs: Kernel and user-space performance. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2001.

[35] Randy H. Katz. Tech titans building boom. IEEE Spectrum, February 2009.

[36] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[37] Vivek Kundra. State of public sector cloud computing. `http://www.cio.gov/documents/StateOfCloudComputingReport-FINALv3_508.pdf`.

[38] 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.

[39] A distributed memory object caching system. `http://www.danga.com/memcached/`.

[40] David Mosberger and Larry Peterson. Making paths explicit in the scout operating system. In *Proc. 2nd USENIX OSDI*, Seattle, WA, October 1996.

[41] 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.

[42] Vivek Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proc. USENIX Annual Technical Conference*, Monterey, CA, June 1999.

[43] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proc. 3rd USENIX OSDI*, New Orleans, LA, February 1999.

[44] 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.

[45] A distributed key-value storage system. `http://project-voldemort.com`.

[46] Niels Provos. libevent. `http://monkey.org/~provos/libevent/`.

[47] Asfandyar Qureshi, Rick Weber, Hari Balakrishnan, John Guttag, and Bruce Maggs. Cutting the electric bill for internet-scale systems. In *Proc. ACM SIGCOMM*, Barcelona, Spain, August 2009.

[48] 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.

[49] Vijay Janapa Reddi, Benjamin Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using small cores: Quantifying the price of efficiency. Technical Report MSR-TR-2009-105, Microsoft Research, August 2009.

[50] Suzanne Rivoire, Mehul A. Shah, Parthasarathy Ranganathan, and Christos Kozyrakis. JouleSort: A balanced energy-efficient benchmark. In *Proc. ACM SIGMOD*, Beijing, China, June 2007.

[51] Luigi Rizzo. Polling versus interrupts in network device drivers. *BSDConEurope*, November 2001.

[52] Mohit Saxena and Michael M. Swift. FlashVM: Virtual memory management on flash. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2010.

[53] Livio Soares and Michael Stumm. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proc. 9th USENIX OSDI*, Vancouver, Canada, October 2010.

[54] Jason Sobel. Building Facebook: Performance at massive scale, June 2010. Keynote talk at ACM Symposium on Cloud Computing 2010.

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

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

[57] Alex Szalay, Gordon Bell, Andreas Terzis, Alainna White, and Jan Vandenberg. Low power Amdahl blades for data intensive computing, 2009.

[58] 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.

[59] 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).

[60] Vijay Vasudevan, Lawrence Tan, David Andersen, Michael Kaminsky, Michael A. Kozuch, and Padmanabhan Pillai. FAWNSort: Energy-efficient sorting of 10GB, July 2010. 2010 JouleSort Competition `http://sortbenchmark.org`.

[61] 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.

[62] 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.