

Power Proportional Cluster

Sara Alspaugh, Laura Keys, Andrew Krioukov
Computer Science Division
University of California, Berkeley
{alspaugh,laurak,krioukov}@cs.berkeley.edu

December 14, 2009

Abstract

Energy consumption is a major and costly problem in modern data centers. Estimates suggest that 20 megawatts is a common power requirement costing over \$38M per year. A large fraction of this energy goes to powering idle machines that are doing no useful work. We identify two causes of this inefficiency: low server utilization and a lack of power proportionality in server machines.

We propose a novel cluster manager that schedules load and turns on and off machines to create a power proportional cluster out of non-power proportional nodes. This approach both increases server utilization and improves power proportionality.

We implement a cluster manager for HTTP workloads and show how to construct a scheduling algorithm to predictively turn on and off machines while meeting performance requirements. Through evaluations using both synthetic and real workload traces we show a 50% reduction in energy consumption with no significant performance penalty.

1 Introduction

The rise of web services and cloud computing is driving the growth of large data centers created by companies such as Google, Microsoft, and Amazon. These data centers consume astonishing amounts of energy; 20 megawatts is a common power requirement [14]. Worst of all, more than half of the time the servers that make up these data centers are idle

but still using nearly peak power [2]. This is caused by two compounding factors: a lack of power proportionality and low server utilization.

A server is power proportional if its energy consumption is proportional to its load. For instance, a lightly loaded server should use only a small fraction of total power. However, modern servers are demonstrably bad at “doing nothing.” Even energy-efficient servers consume nearly half of their full power when doing virtually no work [2]. Less efficient servers are typically far worse, often consuming more than 75% of full power when idle [7]. Due to this lack of power proportionality in servers, keeping them underutilized wastes large amounts of energy.

Unfortunately, many servers in data centers have low levels of utilization. Figure 1 shows the average CPU utilization of servers at Google over a six-month period. The average utilization is around 30%, with most servers operating between 10% and 50% of total capacity. We can compute the percent of wasted energy as follows:

$$\frac{(\text{idle pwr})(\% \text{ idle})}{(\text{idle pwr})(\% \text{ idle}) + (\text{active pwr})(\% \text{ active})}$$

Using the numbers reported by Google, this implies that over 53% of the energy consumed by their servers is wasted doing nothing. It is estimated that Google spends over \$38M per year powering its data centers [17]. Thus, over \$20M is being wasted.

This paper develops and analyzes a method for eliminating this inefficiency by constructing a power proportional cluster out of non-power proportional

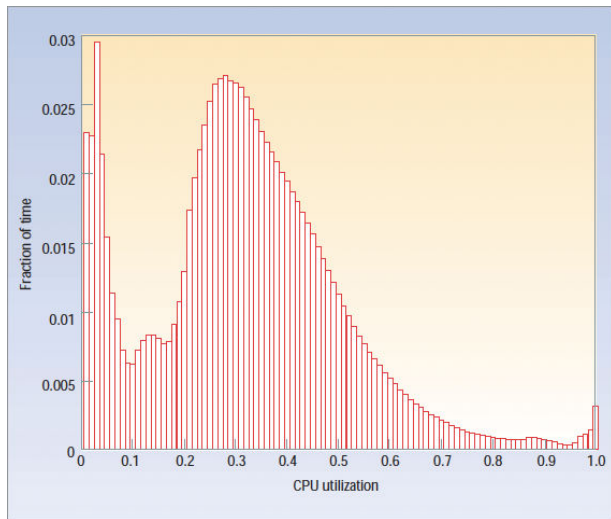


Figure 1: Average CPU utilization of more than 5,000 servers at Google during a six-month period [2].

nodes. We address both of the issues that cause poor energy efficiency: low server utilization and a lack of power proportionality.

Central to our design is a novel cluster manager that determines the minimal set of servers required to service a given workload and saves energy by packing the work onto fewer machines, and putting the remaining servers into sleep states. We implement a cluster manager for HTTP workloads and show a significant power reduction with no significant increase in server response time or decrease in throughput.

2 Prior Work

Previous research has touched on various aspects of power proportional cluster design, from studies searching for the most efficient cluster nodes to techniques that allow nodes to go to sleep without losing network presence. Most of these projects rely on simulation or analysis rather than actual system design and implementation, so we attempt to incorporate all of these areas into our full system design.

2.1 Power Proportionality

Studies on server power proportionality motivate the need to put servers into low-power sleep states when they're not being used. Average processors currently available consume at least 40-50% of their maximum power even when they are completely idle. In terms of obtaining higher performance per Watt, these processors are more energy-efficient when they are more highly utilized. However, a survey of the utilization of Google servers reveals that most servers spend the majority of their time underutilized [2]. These underutilized servers waste a great deal of energy that could be conserved if they were more power proportional.

To say an individual server is power proportional could mean one of two things: either the hardware has near-zero power usage when idle, or the hardware can put itself into lower power states as it becomes less utilized. As individual servers are not yet very power proportional, achieving a power proportional cluster can be accomplished in one of two ways: either by using servers that have a low idle, baseline power or by putting servers into low-power sleep states when they are not in use.

2.2 Low-Power Cluster Building Blocks

A number of recent papers advocate building data center clusters out of so-called "wimpy" nodes, low power mobile-class processors instead of the traditional heavyweight servers. However, the ideal processor node has both high performance and low power, in addition to an idle power near zero, so while these wimpy nodes do not quite achieve the ideal, they are a push in the right direction. These papers tackle reducing baseline power but do not address ways of completely eliminating the energy wasted during idle periods.

FAWN advocates cluster architecture based on many low-power embedded processors instead of fewer higher power servers [1]. Part of the problem FAWN addresses is that of unbalanced systems, in particular the growing CPU-I/O gap. Individual CPUs are becoming so fast that even the fastest I/O

subsystems cannot keep the CPU fully utilized, thus allowing potential CPU cycles to go wasted. Even with these low-power, low performance nodes, cluster idle power is still 2/3 of its peak power, which could be addressed only by putting individual nodes within the cluster to sleep when idle.

Another study compares energy consumption of clusters built out of different homogeneous building blocks [16]. The authors build a number of homogeneous clusters, ranging from server-class to mobile and embedded processors, and use initial performance-to-power ratios to narrow down their choices. These initial ratios suggest that heavyweight servers and mobile nodes are both worth considering for energy-efficient cluster building blocks, so the authors compare energy-consumption for DryadLINQ-style distributed workloads across both classes of nodes. Ultimately, they find that the mobile-class computing nodes attain high performance at a lower power and result in the lowest energy consumption, making them an ideal cluster building block.

2.3 Proxying

Though computers are often left idling, they generally cannot change to low-power sleep states because they must be available to respond to potential network traffic.

The technique of proxying has been studied as an option for putting power-hungry servers into a low-power mode without disrupting their network presence. Migrating active TCP connections from multiple machines to one proxy machine that maintains these connections can allow greater power savings by allowing the proxied nodes to sleep without disrupting applications [3]. Proxying has been implemented successfully. However, its original purpose was to move connections between machines, not necessarily to allow the newly-freed machines to be put to sleep. Thus, it has not been deployed as part of a full-system solution for decreasing server power usage.

One collaborative study details a breakdown of the types of network traffic that computers in home and office environments see in order to better understand how to design a proxy for such environments [15]. The study verifies that the majority of computers in

both settings have large amounts of idle time and that network packets often have sparse interarrival times. It classifies network protocol packets into several types, including *don't ignore* and *don't wake*. The *don't ignore* packets require the proxy server to take some action, which can involve either waking the sleeping node so that it can respond to the packet, or responding on behalf of the sleeping node; an example might include DHCP traffic. *Don't wake* packets can either be ignored or require some action that the proxy can perform, such as broadcasted ARP traffic regarding other machines. Based on the results of their studies, the authors call for power-awareness in future systems, both in protocol design and application configuration.

2.4 Energy-efficient Web Servers

The topic of energy-efficient web servers in particular has been touched on previously by Guerra et al citeWebServer. The project assumes the use of a web server cluster comprised of a front-end and back-end nodes that can be turned on and off. This cluster is then used to analyze the feasibility of using varying on-and-off homogeneous back-end nodes to meet set time constraints for a modeled web server workload.

The authors simulate both the web server and the workload, modeling the incoming requests as a Pareto distribution. They run experiments with workloads of various magnitudes to show how well a varying on-and-off cluster can do at meeting QoS deadlines. They present a trade-off between request interarrival times, energy usage, and percentage of QoS deadlines met. The trade-off is not particularly surprising, with both shorter interarrival times and stricter QoS deadlines requiring higher energy consumption, as machines need to stay on longer. However, they do not physically implement their design and hence cannot present actual power measurements from their runs.

3 Building a Power Proportional Cluster

3.1 Hardware

An important part of constructing a power proportional cluster is choosing the proper building blocks. As a first step, we require that our chosen cluster hardware support certain power management functions, in particular, low-power sleep states (or S states) as described in the Advanced Configuration and Power Interface (ACPI) specification [12]. We are interested in not only the kinds of sleep states supported, but also in the power used in each of these sleep states, along with the time needed to transition to and from them. To this end, we measured on a number of machines the time it takes to move from awake (or state S1) to sleep (or state S3) and from sleep to awake again. This data is given in Table 1. In S3, alternatively referred to as *standby*, *sleep*, or *suspend to RAM*, the power to the CPU is shut off and the contents of its registers are flushed to memory, which is put in slow refresh mode, and many devices are powered off. We focus on S3 because in general we have found that other sleep states tend not to be supported. It is an open question as to whether it is useful in general to have several sleep states with varying levels of power usage and transition times or whether only one sleep state is really all that is needed.

We also measured, for two machines, the power used while awake and asleep, the results of which are given in Table 2. We ultimately chose to build our cluster using Intel Atom machines, but we are interested in the BeagleBoard because of its extremely low energy consumption. The BeagleBoard is a low cost, low power platform based on the OMAP3530 processor featuring a 600 MHz ARM Cortex-A8, along with 256 MB RAM and 256 MB NAND flash [10]. While in our current cluster, we use only one type of node, we can imagine potentially achieving greater power savings by using a heterogeneous mix of hardware types, such as one that includes both Atoms and less powerful but more BeagleBoards. However, we leave this as a topic for future consideration.

	to S3	from S3
Atom	1.0	2.4
Centrino	2.5	4.9
Nehalem	1.5	4.9

Table 1: Node transition time (s)

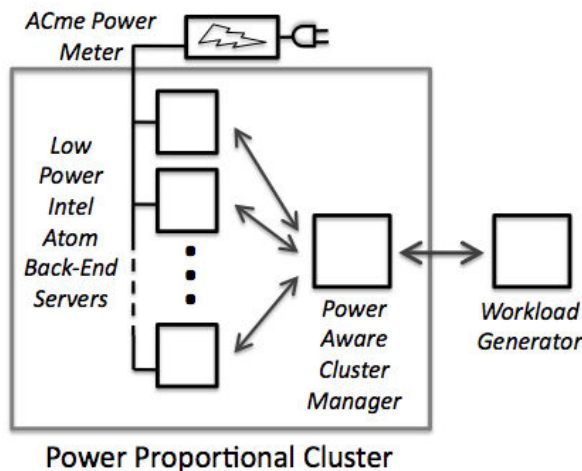


Figure 2: Our system architecture

3.2 System Architecture

Our cluster consists of a set of back-end servers along with one front-end cluster manager that runs a scheduler and receives incoming connections. The scheduler decides based on the current load how many back-end nodes to keep awake, and keeps the rest asleep, with the ultimate goal of minimizing the number of awake servers, and by extension, minimizing the total cluster power usage. In addition to running a server, each of the back-end nodes runs a small daemon that listens on a known port for the signal from the manager to go to sleep. To wake up a node, the manager uses Wake-On-LAN, which allows it to send a specific Ethernet packet to cause the node to wake up. When the manager receives a connection, it makes a new connection to one of the awake back-end servers and forwards requests associ-

	Idle		Peak		Sleep	
	<i>No USB</i>	<i>With USB</i>	<i>No USB</i>	<i>With USB</i>	<i>No USB</i>	<i>With USB</i>
BeagleBoard	1.7	3.4	–	3.4	–	–
Intel Atom	22.1		24.0		0.1	

Table 2: Node power usage (W)

```

on_new_request {
    if (active_servers < desired_servers)
        turn_on_server();
}

on_end_request {
    if (server.requests == 0 &&
        active_servers > desired_servers)
        turn_off_server_after_timeout();
}

```

Figure 4: Algorithms for bringing up and down servers. The goal is to maintain the desired number of servers as specified in Figure 3.

ated with that connection along to that sever. This setup is depicted in Figure 2.

In addition to this arrangement, we considered a possible alternative. In this alternative arrangement, a standard load balancer would sit in front of the back-end nodes, and the scheduler would reside in separate node and make decisions about which nodes to keep awake and which to put to sleep from there. It would notify the load balancer as to which nodes are available to load balance among. This would remove the scheduler from the critical path of requests and allow a standard out-of-the-box load balancer to be used.

3.3 Scheduler

The goal of the cluster scheduler is to keep awake the minimal number of machines necessary for servicing the current workload and for handling potential load spikes. The two main challenges are how to direct incoming requests and how to determine the necessary

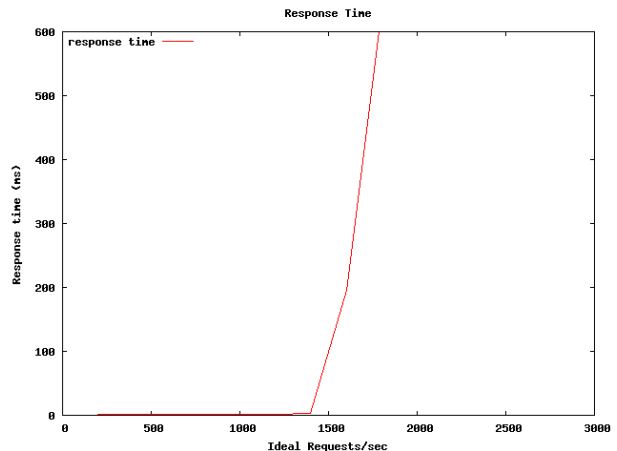


Figure 5: Response times at increasing request rates for an Apache server on an Atom node.

number of servers to keep awake.

For deciding where to direct traffic, the scheduler uses a simple algorithm in which incoming requests are sent to the server with the highest load that is still below the maximum limit. This policy allows us to maximize the utilization of those servers that are awake and increases the chances that a lightly loaded server will complete all requests and become eligible for sleeping. The main complication is that for certain types of traffic, including HTTP, a single connection can be used to send multiple requests in a pipelined manner[9]. This complicates redirecting traffic from one back-end server to another. The simplest solution is to force the client to reconnect for each request, however, this is inefficient. A more efficient approach would involve the cluster manager creating connections to multiple back-end servers for

$$\text{desired_servers} := \frac{(\text{overprovision multiplier})(\text{incoming request rate}) + (\text{overprovision constant})}{(\text{maximum request rate per server})}$$

Figure 3: Determining the desired number of servers. The overprovision factors ensure extra capacity for bursts of requests.

one incoming connection. The downside of this approach is that it requires the manager to understand and parse application layer protocols (e.g. HTTP).

The second function of the cluster scheduler is to keep awake the minimal number of servers needed. To do this it must measure the current workload, estimate the maximum load a given server can service and predictively turn on additional servers. The current workload can be quantified in several ways: rate of requests, current number of requests being processed, or required resources (e.g. CPU time, memory, etc.). For our implementation we initially focus on small, static HTTP requests, thus we assume that the resources required to service each request are roughly equal. We quantify our workload as the rate of incoming requests. There are many ways to measure this. We compute the incoming request rate at a given time as the number of requests observed in the most recent full window of time, for a given window size (e.g. one second). A more sophisticated approach could keep a exponentially weighed moving average of the request arrival frequency:

$$\text{avg} = \alpha \text{avg} + (1 - \alpha) \frac{1}{\text{interarrival time}}$$

Here α is a variable which specifies how much to weigh history versus the current measurement.

Next, the scheduler must determine the maximum rate each server can sustain. Since we are dealing with interactive workloads, we strive to minimize the response time. Figure 5 shows the response time vs load for a single backend server running Apache. The response time remains very low until the knee of the curve after which it increases steeply. The scheduler should operate at the knee to maximize server utilization while keeping a low response time. In the current implementation this point is set statically, however, in the future we will explore a more dynamic ap-

proach.

Finally, the cluster scheduler must predictively turn on additional servers. To do this we incorporate over-provisioning factors when determining the necessary number of servers, as shown in Figure 3. The over-provision multiplier ensures we can handle a multiplicative increase in load while the over-provision constant ensure extra capacity at low load. This calculation is repeated every time a new request is received or completes and is used to determine when to bring up and down servers as shown in Figure 4.

4 Evaluation

4.1 Workloads

We use a synthetic stress-test to find the performance points at which our power-proportional cluster would need to turn on more machines. Using `httperf`, we ramp our workload up from an easily manageable number of requests to much higher request rates that overload the servers.

The first workload is an `httperf` run that requests the same single 24KB file over and over. Caching of this file allows both the simple and power-aware schedulers to perform better than the following heterogeneous workload, but we focus more on the heterogeneous file workload as it is more representative of actual web server traffic.

The second workload is still a synthetic `httperf` run, but it requests files obtained from actual ISP web server traces [5]. The file requests fall into a Pareto distribution, with a few files being requested quite frequently and a large number of files requested very infrequently, as shown in Figure 6.

This workload of heterogeneous file requests cre-

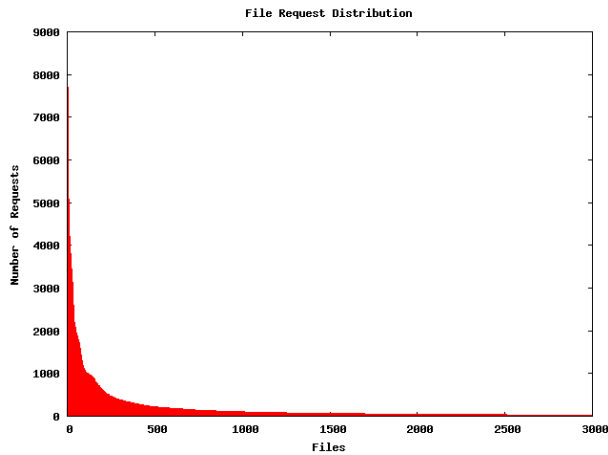


Figure 6: Frequency of ISP File Requests

ates a connection between the workload generator machine and the front-end load balancer machine, sends one HTTP get request, and then closes the connection. While this method induces a lot of overhead costs, it allows us to easily decouple requests from their associated flows and distribute them to different machines as decided by the load balancer. It sends a set amount of requests over the course of the run and sends new requests at a specified rate. This rate is incremented linearly over the course of the workload run to test the cluster at different utilizations.

A third workload we examine is based on the aforementioned heterogeneous workload. It uses `httperf` to send a steadily increasing number of requests/sec to the web server, but when it reaches some maximum rate, it mirrors the first half of the workload and decreases the request rate. This so-called “Christmas tree” workload demonstrates the cluster manager’s ability to turn machines both on and off when necessary.

We present the details for a fourth workload though do not show any results for it in this work. This last workload is a realistic one, based on Apache web server logs from a variety of web servers, each responsible for different amounts of traffic [6]. At the lower end of the spectrum is the web server for a small computer science department; the high end is a web

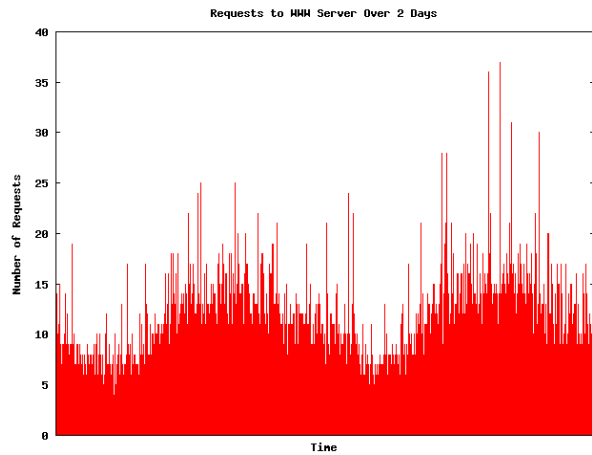


Figure 7: Number of Requests to WWW Server Over 2 Days

server for the 1998 World Cup. The ISP web server whose file distribution we show in Figure 6 falls somewhere in the middle. The access patterns to these files, shown in Figure 7 follow a clear diurnal pattern, with dips in traffic corresponding to late-night hours and large spikes during the daytime.

Knowing the profile of the workloads, such as maximum expected spikes or times of day with little expected traffic, enables us to achieve better energy savings by tuning the settings of our scheduler. Each web server’s traces correspond to different levels of traffic our web server would see. We currently do not have the infrastructure in place to implement this real workload because the larger web servers, such as the World Cup one, require large numbers of requests to be sent every second, which goes beyond the limited number of connections available to us in Linux.

4.2 Experimental Setup

To evaluate our design we ran our workloads on a small cluster. Our cluster consists of four 1.6 GHz Intel Atom dual core processors with 2 GB RAM and 160 GB hard drive. The power usage information for these nodes is given in Table 2. All nodes run Ubuntu 8.10. Three of the nodes serve as back-end

Apache servers, while the remaining front-end node serves as the scheduler. All nodes are connected via a single switch, so network latencies are on the order of sub-milliseconds. We used a separate machine to run the load generator, `httperf`. We measure the cumulative power usage of the three back-end nodes using an the ACme AC meter [13] that outputs power readings every second. Additionally, data on the back end nodes is fully replicated; any web request can be serviced by any of the back end nodes.

4.3 Results

We use the simple, power-oblivious scheduler HAProxy [11] as a baseline of comparison for our power-aware scheduler and examine differences in power, throughput, and response times between the two schedulers.

The power usage results of our experiments with the simple scheduler and heterogeneous workload are not particularly surprising. The simple scheduler had all 3 backend servers on for the duration of the test, so its power usage is practically constant regardless of the utilization of each back end machine, hovering around 80 Watts, as shown in Figure 8(a).

Power usage due to the power-aware scheduler is more dynamic: it corresponds to a step function, starting out with the power of a single machine at work and then gradually adding a second and third machine to the mix. Points at which the power jumps up a “step” shows the sustained request rate at which more machines are required to maintain good cluster throughput and response times. These points occur around 700 requests/sec to go from one to two machines and around 1300 requests/sec to go from two to three machines.

Our cluster exhibits a degree of power-proportionality; however, we want to ensure that we do not sacrifice performance in our quest for power savings. Figures 9(a) and 9(b) show the request rates our clusters sustain.

We attempt to send a linearly increasing number of requests/second during the course of the workload. The actual number of requests/second corresponds to the number of connections we successfully make to a back end machine, and the responses/second is the

number of replies we get back from these successful connections. Up until the ideal request/rate of around 1000, the ideal request rate and actual request rate are the same. Both the simple and power-aware schedulers can accommodate this load, even though the power-aware cluster has only 2 machines turned on at that point.

However, after the request rate of 1024, both systems exhibit a drop in throughput and never sustain a future request rate higher than 1024. We realized that there are some internal Linux settings that affect our ability to maintain any more connections than that, so with an unmodified system, we turn our attention to the operating range less than 1024 requests/second.

Looking at the response times for these two schedulers in Figures 10(a) and 10(b), we see that both maintain very low times, less than 10s of milliseconds, up until the point where throughput drops. Thus, we believe that our scheduler can perform just as well as accepted schedulers like HAProxy while cutting power usage required to maintain this good performance.

To exhibit the cluster’s ability to turn machines both on and off as workload changes, we run the aforementioned Christmas tree workload with a maximum request rate of 1000 requests/second to keep us within the default Linux-limited operating range. The workload ramps up the number of requests/second from 100 requests/second up to 1000 and then decreases the rate back down to 100. Figure 11(a) shows this ramping up and down of request rates, along with the matching actual request and response rates. For the most part, the actual requests and responses matches up with the ideal. However, around 1000 requests/second, throughput suddenly drops. Due to time constraints, we were not able to run this test enough to get any good statistics about this problem and its causes, so we avoid discussing it here and dismiss it as a single data point.

Figure 11(b) shows power usage of the power-aware scheduler. Once again, we can see the step function showing the points at which new machines must be turned on to handle the increasing workload, but then we can also see the times machines are turned off as the workload decreases. If we were to compare the to-

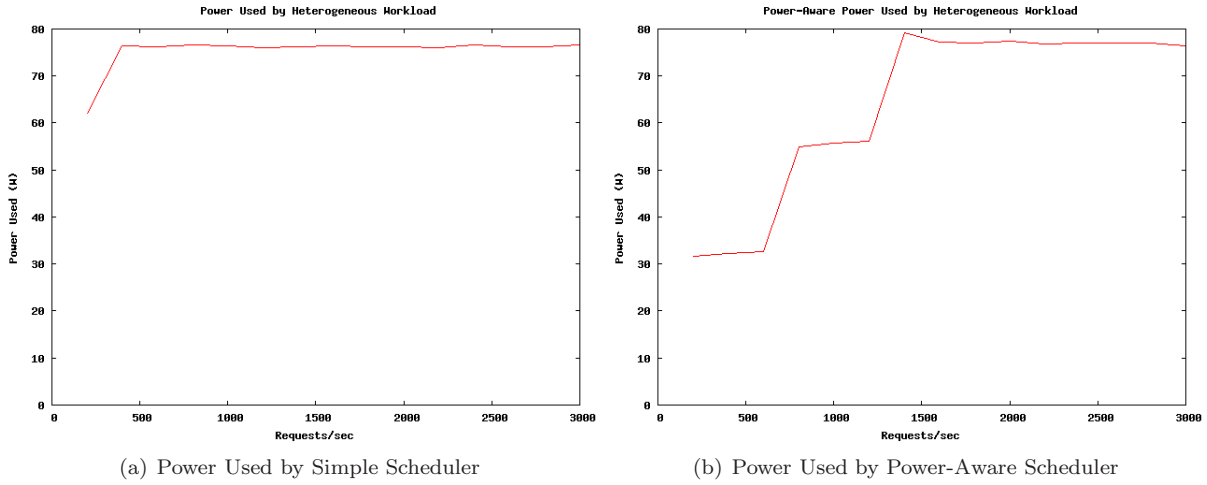


Figure 8: Power Usage in Heterogeneous Workload

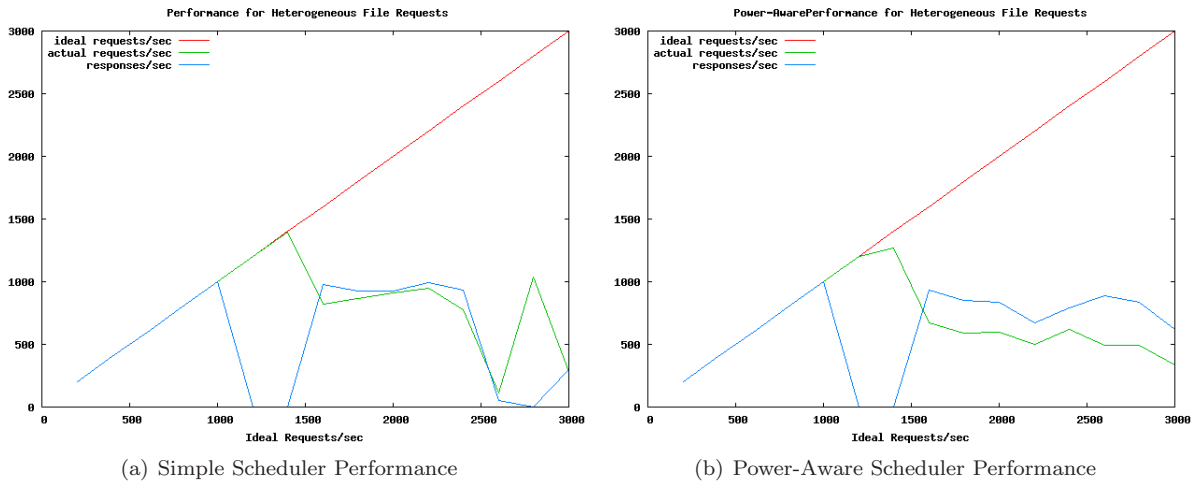


Figure 9: Performance Rates of Each Scheduler

tal energy used in the simple scheduler cluster versus in the power-aware cluster for this rising and falling workload, we would see nearly half the energy used in the power-aware cluster as in the simple one. This energy savings comes while maintaining the same level of performance as the simple cluster.

A final verification of our power-aware scheduler

is ensuring that incoming requests are actually being scheduled to the highest utilized machine. Figure 12 shows the number of outstanding requests at each end node during the Christmas Tree workload. We can see that while only one machine is turned on, all requests go to one server. When a second machine is turned on, some requests are diverted to the sec-

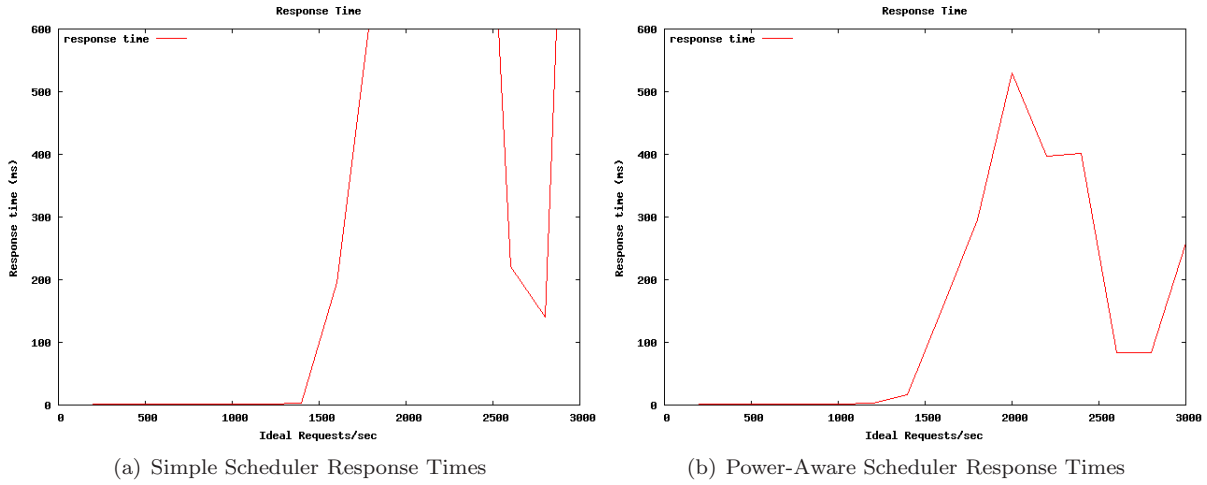


Figure 10: Response Times of Each Scheduler

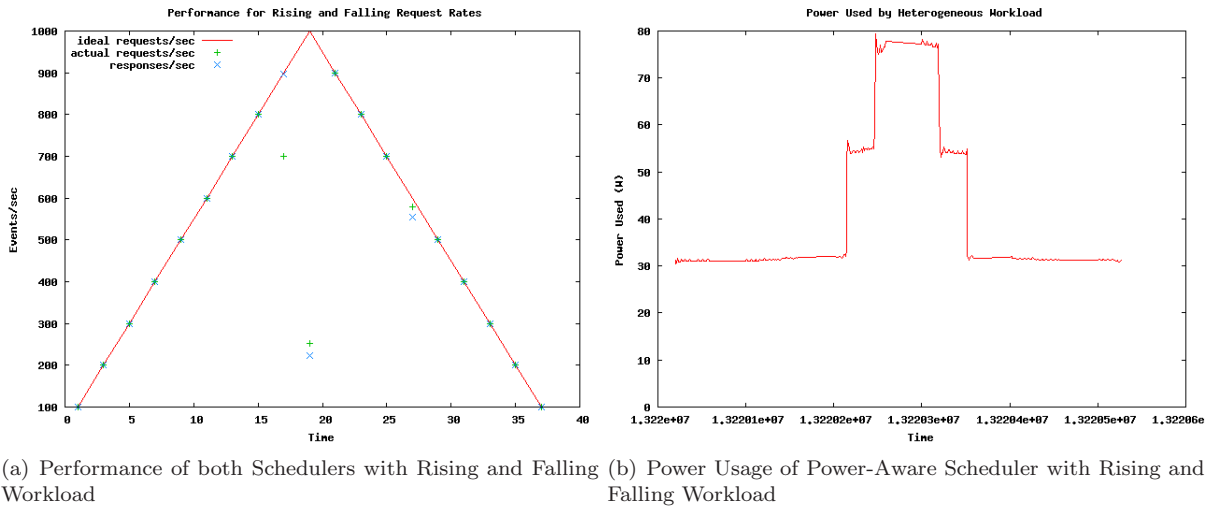


Figure 11: Rising and Falling “Christmas Tree” Workload on Power-Aware Cluster

and one from the first one because the first reached a maximum threshold and cannot take on all the new requests. When a third server is powered on and added to the mix, the first two are not yet utilized to their maximum, so the third server never actually sees traffic. In that sense, we overprovision slightly

to be able to handle a reasonable spike in traffic.

As the incoming requests decrease to 600 requests/second, the first server ends up taking all the requests, but then shortly thereafter, it starts sharing with the second server again. This change in which machines are servicing requests has to do with rate

sampling in the scheduler. The scheduling algorithm attempts to bin-pack the requests onto the most utilized machine, but only up until the machine sees a certain threshold incoming rate, at which point it turns some of the requests over to other machines. The instantaneous rate the machines see sometimes fluctuates a bit, so it's likely that the rate oscillated slightly around the first server's threshold. Further decrease in request rates diverts the second server's traffic to the original first server. The second server's queue eventually reaches zero, and it is turned off, with all requests going to the first server.

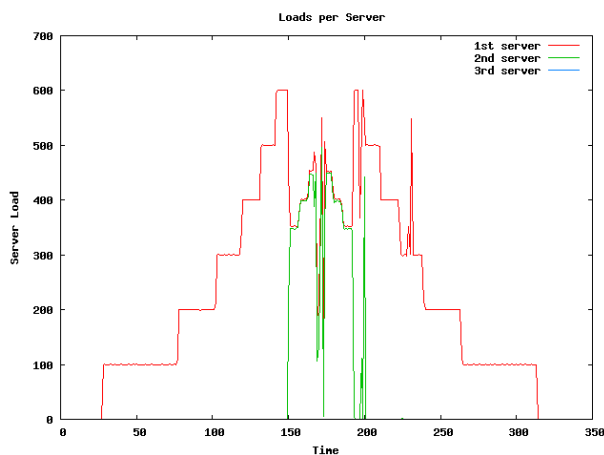


Figure 12: Request Load on Each Back End Node

5 Discussion

The initial results show the potential of building a power-proportional cluster to save energy without sacrificing performance. However, our initial approach exposes several key limitations and also opens many possible future directions.

The most significant limitation of the current implementation is the connection limit imposed by Linux. Due to limits on the number of file descriptors, sockets and ports per process and per machine, we cannot always accept connections at the cluster manager despite having capacity on the back-

end servers. There are several possible solutions to this problem. We could increase the limits by changing the `ulimit` settings and using more efficient event detection functions. Currently, we use the `select` system call to detect network events (e.g. incoming data and available buffer space for writing). `select` linearly scans all of the connections for new events, thus, its performance is $O(n)$ in the number of connections. An alternative approach is to use `epoll` which notifies the caller when an event occurs on any of the open connections. It is fully event driven and does not scan the list of connections, thus, its performance is $O(1)$.

More fundamentally, the cluster manager is currently a single bottleneck through which all incoming request traffic must flow. In future work we will explore ways to parallelize the cluster manager so that it can run on multiple machines in a distributed manner. A different approach would be to use a high-performance hardware load balancer with the cluster manager simply dictating the number of servers that should be on. This method would take the power management off of the critical path.

Other directions for future work include exploring different scheduling policies and handling more diverse workloads. Currently, the scheduler uses a static threshold for turning on and off machines. Oscillations are reduced by requiring the load to stay past the threshold for a set period of time. A more sophisticated algorithm can use two different thresholds for turning machines on and off. Also, the operating point can be determined dynamically through an algorithm such as AIMD [4].

Finally, we will explore adapting the cluster manager to handle more complex workloads and applications. This includes services that maintain state such as Map-Reduce [8]. For these applications we can either split the storage layer from the computation or explore turning off servers with replicated state [19]. We will also explore a mix of batch and interactive workloads that go beyond serving static HTTP content, such as CloudStone [18].

6 Conclusion

To tackle the problem of energy-inefficient clusters, we took a two-pronged approach: building a custom cluster out of energy-efficient hardware and making the number of powered-on machines proportional to the incoming workload. We studied a few hardware options but ultimately went with the Intel Atom mobile-class processors as our cluster building blocks.

We then implemented a power-efficient web server out of these low-power processors, complete with a front-end that acts a proxy for the back-ends. Using a simple formula that equates necessary powered-on servers with current load, we modified a simple scheduler to adaptively put servers to sleep and awaken them as needed.

Based on synthetic workloads meant to stress the servers, we find that our power-aware scheme can save power at low incoming workload rates. As a large proportion of servers in the real-world sit idle for a good amount of the day, we expect our system design to enable noticeable power savings when deployed at the large scale. Even with power savings, we do not sacrifice performance. Machines that have been added to a cluster with overprovisioning in mind, that is, to handle large spikes in traffic, remain in the cluster, but power is not wasted on them during periods of lower traffic.

In the future, we plan to make our power-aware cluster more general-purpose. In addition to handling a web server workload, we will add capacity for both interactive and batch-type jobs in order to make the best use of machines that are required to be on anyway. We are also considering the merits of a cluster built out of heterogeneous nodes as a method of further reducing power usage.

7 Acknowledgments

Thanks to Randy H. Katz and David Culler for their encouragement of this project and for the initial idea. Many thanks to Jon Kuroda and the RADLab support staff for their never-ending efforts to support our research projects. Additional thanks to Ken Lutz for the custom-designed Atom clusters, and to Fred

Jiang and Stephen Dawson-Haggarty for their assistance with the ACme power meters, and to the countless members of the UC Berkeley RADLab and LoCal communities who have engaged in lively conversation and debate with us over the semester and helped us to refine our ideas.

References

- [1] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: A fast array of wimpy nodes. In *22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, October 2009.
- [2] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.
- [3] M. Bernaschi, F. Casadei, and P. Tassotti. Sockmi: a solution for migrating tcp/ip connections.
- [4] D. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, June 1989.
- [5] ClarkNet-HTTP. <http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html>, 1998.
- [6] P. Danzig, J. Mogul, V. Paxson, and M. Schwartz. Traces available in the internet traffic archive. <http://ita.ee.lbl.gov/html/traces.html>.
- [7] S. Dawson-Haggerty, A. Krioukov, and D. Culler. Power optimization: a reality check. *EECS Department, University of California, Berkeley, Tech. Rep.*, Oct. 2009.
- [8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137–150.
- [9] R. Fielding. Hypertext transfer protocol http 1.1. <http://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html>.

- [10] R. Fielding. Beagleboard system reference manual. http://beagleboard.org/static/BBSRM_latest.pdf, October 2009.
- [11] HAProxy: The reliable, high performance TCP/HTTP load balancer. <http://haproxy.1wt.eu/>.
- [12] Intel Corporation. Advanced configuration and power interface specification, June 2009.
- [13] X. Jiang, S. Dawson-Haggerty, P. Dutta, and D. Culler. Design and implementation of a high-fidelity ac metering network. In *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks*, San Francisco, CA, April 2009.
- [14] R. H. Katz. Tech titans building boom. *IEEE Spectrum*, February 2009.
- [15] S. Nedeveschi, J. Chandrashenkar, B. Nordman, S. Ratnasamy, and N. Taft. Skilled in the art of being idle: Reducing energy waste in networked systems. In *NSDI '09: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, 2009.
- [16] A. omitted due to double-blind review policy for ISCA 2010. The search for energy-efficient data center building blocks. 2009.
- [17] A. Qureshi, R. Weber, H. Balakrishnan, J. Guttag, and B. Maggs. Cutting the Electric Bill for Internet-Scale Systems. In *ACM SIGCOMM*, August 2009.
- [18] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, A. Fox, and D. Patterson. Cloudstone: Multi-platform multi-language benchmark and measurement tools for web 2.0. *Proc. Cloud Computing and Its Applications*, 2008.
- [19] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: a powerproportional, distributed storage system. *MSR-TR-2009-153*, November 2009.