

Energy Conservation in Datacenters through Cluster Memory Management and Barely-Alive Memory Servers

Vlasia Anagnostopoulou*, Susmit Biswas*, Alan Savage*,
Ricardo Bianchini†, Tao Yang*, Frederic T. Chong*

*Department of Computer Science, University of California, Santa Barbara

†Department of Computer Science, Rutgers University

ABSTRACT

As a result of current resource provisioning schemes in Internet services, servers end up less than 50% utilized almost all the time. At this level of utilization, the servers' energy efficiency is less than half their efficiency at peak utilization. A solution to this problem could be consolidating workloads into fewer servers and turning others off. However, services typically resist doing so. A major reason is the fear of slow response times during re-activation in handling traffic spikes. Another reason is that services want to maximize the amount of main memory available for data caching across the server cluster.

In this paper, we propose an approach that does not completely shutdown idle servers and allows free memory space to be used for cooperative data caching. Specifically, we make two key contributions. First, we propose to send servers to a new "barely-alive" power state, instead of turning them off after consolidation. Our barely-alive servers allow remote accesses to their main memories even when all processing cores have been turned off. Second, we design a distributed middleware that accommodates barely-alive servers and is capable of dynamically re-sizing the amount of cache space across the cluster to the minimum required to respect the service's service-level agreement (SLA). Any memory that is not in use by the middleware can be used by applications. Our trace-driven simulations of a server cluster using our middleware and barely-alive servers show very encouraging results.

1. INTRODUCTION

Energy represents a large fraction of the operational cost of Internet services. As a result, previous works have proposed approaches for conserving energy in these services, such as consolidating workloads into a subset of servers and turning others off [7, 8, 11, 16, 17] and leveraging dynamic voltage and frequency scaling [8, 9, 10].

Consolidation is particularly attractive for two reasons. First, current resource provisioning schemes leave server utilizations under 50% almost all the time [10]. At these utilizations, server energy efficiency is very low [4]. Second, current servers consume a significant amount of energy even when they are completely idle [4]. Despite its benefits, services typically do not use this technique. A major reason is the fear of slow response times during re-activation in handling traffic spikes. Another reason is that services want to maximize the amount of main memory available for data caching

across the server cluster, thereby avoiding disk accesses or content re-generation.

In this paper, we propose an approach that does not completely shutdown idle servers, keeps in-memory application code/data untouched, and allows free memory space to be used for cooperative application data caching. Specifically, we propose to send servers to a new "barely-alive" power state, instead of turning them completely off after consolidation. In barely-alive state, a server's memory can still be accessed, even if many of its other components are turned off. In fact, we consider one design in which all cores are turned off and remote accesses to memory are effected by a very low-power embedded processor at the network interface. The new state does not affect the consolidation algorithm, which can stay the same as before [7, 16].

To manage the memory itself, we also design a distributed middleware that transforms the servers' main memories into a large cooperative cache. The main contributions of the middleware are (1) its ability to accommodate barely-alive servers; and (2) its ability to dynamically re-size the amount of cache space across the cluster to the minimum required to respect the SLA (or equivalently to achieve a desired average hit ratio). The sizing of the cache is based on a distributed implementation of a Stack algorithm [13] for variable-sized objects. Any memory that is not in use by the middleware can be used by applications. The middleware also directs the servers to change power states based on their amount of resource activity.

Our preliminary evaluation uses trace-driven simulation of a server cluster running a search engine application. We model the application using data from AOL and Ask.com. Our results show that we can conserve 24% energy and 49% power for this application.

The remainder of the paper is organized as follows. Next, we discuss the background and related work. In Section 3, we detail the barely-alive power state. Section 4 describes our design for the distributed middleware. In Sections 5 and 6, we describe our preliminary simulation infrastructure and results. We discuss our conclusions in Section 7.

2. BACKGROUND AND RELATED WORK

2.1 Data-set and Request Distribution

Each datacenter of an Internet service may consist of thousands of servers running many applications. The servers are connected to each other in a hierarchical fashion. Servers located on a rack communicate with high bandwidth through

fairly inexpensive commodity switches. In contrast, communication across racks occurs through an expensive large-port-count switch. Despite its large numbers of ports, this switch provides lower bandwidth per server, since the servers in each rack must share a relatively small number of uplinks to the higher-level switch.

Because of this hierarchical architecture, applications and data-sets are mapped to servers to reduce communication through the high-level switch. For example, the data-set of a large application may be partitioned across racks so that servers do not have to communicate intensively across the high-level switch. Our study in this paper considers one of these partitions, calling it simply a cluster.

Within the cluster, load (i.e., client requests) may be assigned to each server via a front-end device or using some other distribution approach (e.g., Round-Robin DNS). In this paper, we focus on the Locality-Aware Request Distribution (LARD) algorithm [15] and, in particular, its distributed implementation (PRESS) [6]. The main idea of LARD and PRESS is to create a cooperative cache out of the servers’ main memories. The request distribution sends each request to the server caching the requested content locally, as long as the server is not overloaded. Otherwise, a less-utilized server is selected and the content is replicated. In the LARD implementation, a front-end device maintains the caching information. In PRESS, there is no front-end and each server informs all others about any changes to its local fraction of the cache.

2.2 Stack Algorithm

There has been extensive research on approaches to predict the hit ratio of a cache hierarchy, given its capacity, the replacement policy, and a sequence of memory accesses. An efficient approach is the Stack algorithm, which can compute the hit ratio that would be achieved by all cache sizes using a single pass over the stream of memory accesses. In its simplest implementation, the algorithm uses a linked list to keep track of the memory references [13]. More efficient implementations typically keep track of the reuse distances of the references instead of the references themselves, and use AVL tree structures [5]. In our system, we consider a single-level memory hierarchy, i.e. main memory, and use LRU as the replacement policy.

2.3 Related Work

Barely-alive servers. Many papers have studied dynamic workload consolidation and server turn off [7, 8, 11, 17], an approach that was originally called Load Concentration in [16]. The idea is to adjust the number of active servers dynamically, based on the load offered to the service. During periods of less-than-peak load, the workload can be concentrated (either through state migration or request distribution) on a subset of the servers and others can be turned off. In this paper, we propose how to make this approach to energy conservation more practical through the creation of a new server power state.

Interestingly, the Advanced Configuration and Power Interface (ACPI) [1] defines a sleep state, called S2, wherein the CPU is powered off, but the memory controller and main memory remain powered on. Unfortunately, S2 is not implemented in current motherboard and CPU architectures. Instead, they implement the next lower power state (S3), because there has been no use for S2 until now and S3 provides

extra energy savings by turning off the memory controller. We advocate that, using the S2 state, energy savings can be achieved without losing cache space or disk throughput.

Recently, Meisner et al. proposed PowerNap, an approach to energy conservation that rapidly transitions servers between active and “nap” state [14]. In nap state, a server is not operational. PowerNap requires server hardware to provide fast and reliable transitions between active and deep low-power states, and server software to avoid unwanted transitions to active state (e.g., due to clock interrupts). The combination of consolidation and our barely-alive state imposes fewer requirements on server hardware and software.

Even more recently, Agarwal et al. proposed a similar idea to our barely-alive state with Somniloquy [3]. Somniloquy augments the network interface of a PC to be able to turn most components off during periods of idleness, while retaining network connectivity. Our barely-alive state targets servers and requires the network interface to perform memory accesses as well. Thus, the barely-alive state requires a more powerful embedded processor.

Remote memory access without intervention from the host CPU is also the idea behind Remote Direct Memory Access (RDMA). A few high-end network interface cards, such as InfiniBand [12], provide RDMA capabilities. Although we intentionally abstract the mechanisms required by RDMA (e.g., address registration and memory pinning) in this paper, we do rely on the same functionality. Our contribution is in combining RDMA with the ability to potentially turn the entire CPU off, except for the memory controller.

Middleware. Our middleware borrows cooperative-caching ideas from LARD and PRESS. In fact, PRESS was designed to explore user-level communication and RDMA-based network interfaces. However, our middleware extends these systems by accommodating the barely-alive state and using a Stack algorithm to carefully size the cache. The Stack algorithm has been studied for stand-alone systems with fixed-sized cache blocks, e.g. [13, 5]. In this paper, we propose a distributed implementation of the stack algorithm for server clusters with variable-sized objects.

3. BARELY-ALIVE SERVERS

Services would like to conserve server energy during periods of less-than-peak load using consolidation and server turn off. However, servers cannot typically be turned off, since their re-activation can produce high response times (e.g., due to operating system reboot) during traffic spikes and their main memories are required for data caching. Furthermore, the servers may need to stay on for the service to achieve a higher aggregate disk throughput.

Given these constraints, we propose the barely-alive power state. Servers can be sent to barely-alive state, instead of off state, after the workload is consolidated on another set of servers. A barely-alive server allows remote access to its main memory, even though most of its other components are turned off to conserve energy. Because the memory contents (including the operating system) are not affected, transitioning to and from barely-alive state can be very fast.

With respect to the processing of memory accesses in barely-alive state, we envision two possible approaches. The first leaves one of the cores active and responsible for receiving and performing accesses to the server’s main memory. *Because all accesses are performed by the host processor,*

all memory addressing can be done using virtual addresses. This approach does not require changes to current multi-core hardware, besides the ability to turn off cores independently.

Unfortunately, as the number of cores per CPU increases with each processor generation, it is less likely that we will be able to manage the cores' power states independently. As a result, we will not be able to leave a single core active, increasing the energy consumption of the barely-alive state. Thus, our second approach, the one we actually evaluate in this paper, turns off all the cores but keeps the memory controller on (even if the controller is on chip). Remote memory accesses occur through a very low-power embedded processor built into the network interface. This processor accesses memory by driving the memory controller directly, just as in regular DMA operations involving the network interface. *Because the embedded processor does not understand virtual addresses, the remote memory accesses have to specify physical addresses or be translated to physical addresses in software. (Our middleware takes the latter approach.)*

Obviously, the embedded processor has to implement some sort of (RDMA) communication protocol to be able to receive memory access requests coming from active servers and reply to them. As our target system is a server cluster, this communication protocol can be lean and simple. Because the barely-alive state is essentially independent of this protocol, we do not discuss it further.

Regardless of the exact implementation of the barely-active state, the consolidation algorithm can be the same as before. In other words, servers can be transitioned from active to barely-alive state and back exactly at the same points as a standard consolidation algorithm would turn them off and on, respectively.

4. MIDDLEWARE

In this section, we propose a distributed middleware for cooperative data caching that accommodates barely-alive servers. The middleware keeps using a server's memory for caching even after the server has been sent to barely-alive state.

4.1 Cache Accesses

In our initial design, applications interact with the middleware mainly by calling runtime routines for storing and fetching objects from the cooperative cache. A store call places the object in the local memory cache of the caller. When the local cache is full, enough LRU objects are ejected to create space for the new object. Objects are immutable, so each store call returns a new object id. A fetch call checks the cooperative cache directory. If the object is cached, it is retrieved from one of the servers caching it (this node can be the local server) and returned to the caller process. On a cache miss, the fetch call returns a special flag. At that point, the application is responsible for fetching the object from disk or re-generating it, and then possibly storing it in the cache. The middleware allows fetch and store requests to originate at any server and does not seek to provide strong data consistency; weak consistency is typically enough for Internet services and many of their applications.

The middleware also provides three invalidate calls: invalidate-object, invalidate-local-cache, and invalidate-full-cache. An application can use these calls to invalidate data from the cache at different granularities. For example, applications that update objects (each update effectively creates a new

object) may use the invalidate-object call to invalidate an old version of an object. In addition, the crawler of a search engine may want to invalidate the entire cache periodically, because Web data changes over time. Invalidating an object cached by a barely-alive server works fine, because when the barely-alive server is activated, it realizes that the object should be invalidated (as described below). To prevent the loss of cache space at the barely-alive servers in invalidate-intensive scenarios, they can be periodically activated, while some of the active servers can be sent to barely-alive state.

In more detail, cache accesses are performed by a daemon on each active server of a cluster (e.g., a rack of servers). Objects are always named by their ids. Thus, each daemon maintains a table relating the ids of the objects stored locally to their local virtual and physical addresses. This table is accessible by the network interface, so that objects can be accessed when the server is in barely-alive state.

The set of daemons implement the PRESS algorithm, as described in Section 2.1. Each daemon broadcasts any changes it makes to its local cache, as a result of store requests, to the rest of the cluster. The broadcast message specifies the server number within the cluster and the object id. With the broadcast information, each daemon will have a full directory of the entire contents of the cooperative cache stored in a hash table. Assuming that each directory entry requires 10 bytes (4 bytes for the id and 6 bytes for a bitmap listing the servers caching the object), and the average object size is 4 KBytes, the directory at each server would require at most 0.25% of the amount of memory used for caching. Each daemon also broadcasts the CPU utilization of its local server, whenever the utilization becomes more than a threshold percentage different than the last broadcast utilization.

We had to change the PRESS algorithm to deal with barely-alive servers. Specifically, barely-alive servers do not run the daemon while in that state. Essentially, this means that a barely-alive server does not broadcast its caching or utilization information (or even listen to the information broadcast by other servers). This is not a problem for two reasons. First, since barely-alive servers do not receive middleware calls themselves, they never need to process a store operation (which would cause a change to its local cache). Second, the consolidation algorithm would not decide to turn off a server that caches popular data, so we can expect a barely-alive server to generally exhibit a much lower utilization than the active servers. Moreover, in our variation of PRESS, an active server that receives a fetch request for an object cached by a barely-alive server has to serve the request itself by accessing the cached data remotely.

When a barely-alive server is activated (e.g., by the consolidation algorithm), the daemon is re-started. The daemon then contacts any active server to know what part of its cache contents are still valid; it can reuse the space taken by invalid objects.

4.2 Sizing the Cache

A key aspect of our design is the ability of the middleware to re-size the cooperative cache. The goal is to use just enough memory for the cache as required by the service's SLA. In fact, we assume that the SLA requirement can be translated into a hit-ratio requirement. Shrinking the cache to this minimum size can be critical in the context of consolidation, since more memory can be used by other

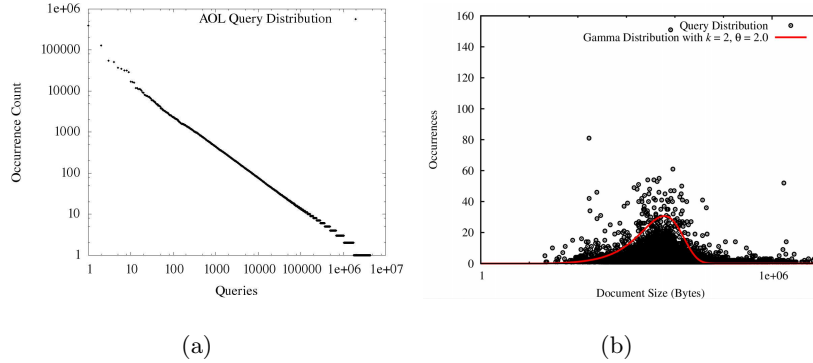


Figure 1: (a) AOL query frequency (Zipf), and (b) object size distribution (Gamma).

applications. A second goal is to size all local caches uniformly, so that we are eventually able to manage the caching of data from multiple applications with a simple scheme.

As aforementioned, our approach to sizing the cooperative cache relies on a distributed implementation of the stack algorithm. Each application using the middleware has its own set of stacks. The server daemon maintains the local stack information of each application separately. Object stores update the local stack in the obvious way. Object stores are handled by inserting new entries into the stack, whereas object invalidations are handled by removing entries from the stack. Servers also communicate at fixed time intervals (time windows), e.g. every hour. During each round of communication, each server broadcasts the local cache size that it would require to achieve the desired hit ratio. With information from all nodes, each server can compute the average of the required local cache sizes and reconfigure its local cache to that size. The local stack information is then reset.

Barely-alive servers also keep their local stack information. (The stack structure is accessible by the network interface processor when the CPU cores are off.) However, they do not broadcast that information or resize their local caches in that state. Instead, our middleware activates these servers at the end of each time window, so that they can participate in the re-sizing protocol. These periodic activations also enable the refreshing of the cache of the barely-alive servers. Transitions to and from barely-alive state are fast enough that these activations should not be a problem.

Our initial middleware design stores each cached object in contiguous physical addresses. This design simplifies the processing of cache accesses at servers in barely-alive state. To accommodate objects of variable sizes with reduced external fragmentation, we will also consider breaking each local cache into three partitions: one for small objects (e.g., objects < 4KBytes), one for medium objects (e.g., objects > 4 KBytes and < 20 KBytes), and one for large objects (e.g., objects > 20 KBytes). There will then be three stacks at each server, one for each partition. Periodically, the partitions can be re-sized based on the stack information and the desired hit ratio.

5. SIMULATION ENVIRONMENT

5.1 Simulator Overview

We simulate a cluster of servers executing a snippet generator for a search engine. We assume each search query

generates a list of the top 10 URLs for that query and passes them to the snippet generator. The snippet generator then scans the pages associated with those URLs to generate a snippet for each one. Within our snippet generation system we assume that the full dataset is available at all times, regardless of the number of active servers. That is, the snippet generator will never be unable to answer a request. For simplicity, we assume that the full dataset is replicated across all the disks of the cluster in such a way to ensure this availability. The following subsections detail our assumptions and methods for the generated workload, processing times, latency, and per server power consumption.

5.2 Workload

We obtained a 7-day trace representing a fraction of the traffic directed to a popular search engine (Ask.com). Due to privacy and commercial concerns, the trace only includes information about the number of queries per second. We used curve fitting techniques on Ask.com trace and model the query rate as a sinusoid.

In order to generate a complete workload, we also analyze publicly available traces that contain all submitted queries (36.39 Million) to AOL over a duration of 3 months in 2006. The number of occurrences of individual queries is shown in Figure 1(a). The figure shows that the distribution of object requests follows Zipf's law [2]. We generate queries for our synthetic trace following this distribution.

The application cache we have tested is a page cache for dynamic snippet generation. Given a search query, a search engine returns 10 top-ranked URLs and a snippet-generation service produces a dynamic description for each URL by scanning through the original Web pages and selecting sentences/phrases relevant to the query. As popular URLs appear in return results of different queries, there is potential for a high cache hit ratio.

In determining the page size for the cache, we took the following approach. We ran a sample of AOL queries against Ask.com, downloaded the content pointed to by the URLs listed in the returned results, and computed the content size. We found a median size of 6Kbytes following a Gamma distribution. The size distribution can be seen in Figure 1(b). Based on these results, we use a cache page size of 4 KBytes in our experiments, but a smaller size could be chosen to reduce fragmentation.

We used these parameters for object access and size distribution to develop a synthetic trace generator. In our exper-

Component	Active state	Barely-alive state
Intel i7 CPU	115W	0
Memory Controller	15W	15W
1 Gbit/sec NI	5W	5W
2xActive Hitachi Deskstar Deskstar 7K1000	24W	0
4GB DRAM	16W	16W
Small embedded CPU	-	1W
Total	175W	37W
Potential savings		138W

Table 1: Server power consumption by component.

iments, we use a trace of an entire day that was generated using our generator.

5.3 Simulator Parameters

We have built an analytical simulator to evaluate our barely-alive power state. We have not yet implemented the complete functionality of the middleware in the simulator, including the distributed stack algorithm.

Table 1 lists the simulated power consumptions in active and barely-alive state. Both sets of parameter values are representative of real systems. The table shows that the power consumption of a barely-alive server is only 37 Watts. The assumption is that the CPU and the disk are turned off, while the processor embedded in the network interface consumes 1W. A fully utilized active server consumes 175 Watts (it keeps the embedded processor off). We assume that the power consumption varies as $(0.5 + 0.5 \times utilization) \times 175$ [4]. Thus, going to barely-alive state reduces power consumption by 58% $(1 - 37/88)$ with respect to staying idle in active state.

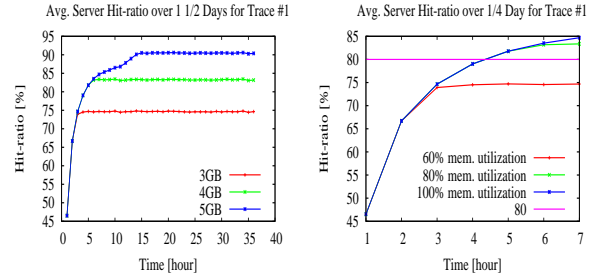
6. EXPERIMENTS AND RESULTS

6.1 Barely-Alive Servers

Figure 2(a) depicts the workload we used to evaluate our barely-alive power state. It represents an entire day of accesses to a service. We simulated 50 servers. When all servers are always active, we assume that the peak utilization during the day is 80%, as seen in Figure 2(b). For these average utilizations, the resulting average power consumptions are shown in Figure 2(c).

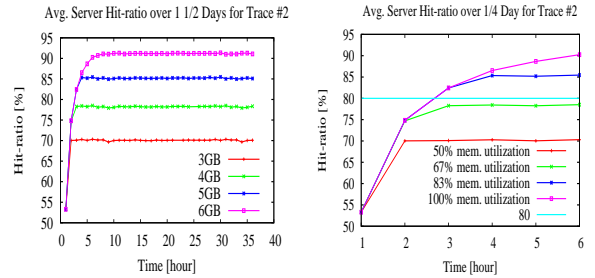
We seek to conserve energy by consolidating workloads onto a subset of servers during periods of light/moderate load and sending others to barely-alive state. Instead of utilizing all servers at relatively low utilization levels (under 60%) during part of the day, we utilize a percentage of servers x at 60% (at most) and the other $100 - x$ percent are sent to barely-alive state. Figure 2(d) depicts the values of x required to maintain the average utilizations from Figure 2(b). The figure shows that the number of active servers decreases during a large part of the day. Given the number of servers in active and barely-alive state, Figure 2(e) depicts the average power consumptions over time.

Finally, Figure 2(f) depicts the energy savings for a range of server provisioning schemes. Our results so far have assumed that the provisioning is such that the peak utilization during the day is 80%. However, this provisioning may leave too little slack to handle increases in load and seasonal effects over longer periods of time. For this reason, we also



(a) Avg. hit ratio for Trace #1

(b) Zoom into 1st 7 hours of Trace #1



(c) Avg. hit ratio for Trace #2

(d) Zoom into 1st 6 hours of Trace #2

Figure 3: Hit ratios over time for two traces.

plot results for peak utilizations of 50% and 65%. When the peak utilization is 80%, our approach is able to save 13% of the daily energy and up to 44% power. When the peak utilization over the day is 50%, i.e. the system is overprovisioned by a factor of 2, we observe a daily energy savings of 24% and a power savings of up to 49%.

6.2 Dynamic Memory Resizing

We have not yet implemented the distributed stack algorithm in our simulator. Here, we limit ourselves to discussing the cache behavior of two traces generated as described in Section 5.2. Figures 3(a) and 3(c) show the average hit ratio for the traces, for various cache configurations, and for assumed 80% SLA hit-ratio. Based on this behavior, we propose a simple heuristic that we plan to incorporate into our final stack implementation.

Interestingly, from figures 3(b) and 3(d), we can see that the warm-up phase of the traces actually takes several hours. This is particularly problematic for search services like the one we simulate, because these services may have to invalidate their cached content periodically (e.g., every two days), since Web content changes over time. This set of figures is the inspiration for our heuristic. It suggests that larger caches may allow the system to reach its target hit rate substantially faster. Thus, during the warm-up phase, the heuristic should disregard the predictions of the stack algorithm and use as large a cache as it can. After the miss rate has reached the target, the heuristic should respect the algorithm's predictions and shrink the size of the cache to the smallest size that still achieves the target hit rate.

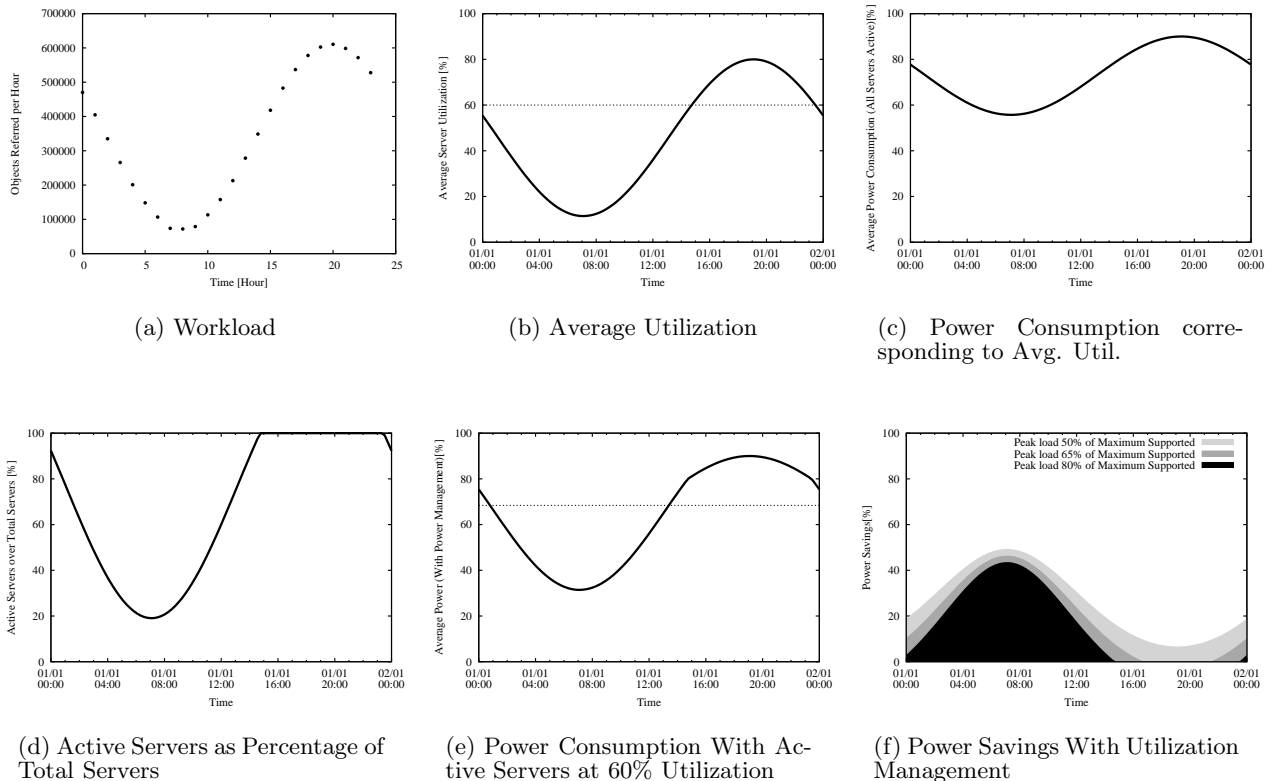


Figure 2: Workload during a day (a). Average server utilization when all servers are always active (b). We assume that the peak utilization during the day is 80%. Resulting average power consumption (c). Fraction of active servers, when the barely-alive state is used (d). Power consumption as servers are sent to barely-alive state under lighter load (e). Energy savings for peak utilizations varying from 50% to 80% (f).

7. CONCLUSION

In this paper, we sought to make consolidation and server turn off more practical. With that in mind, we made two main contributions. First, we proposed a new power state, called barely-alive. When a server is in this state, its memory can still be accessed, even though the entire CPU is turned off. Second, we proposed a middleware for cooperative caching that accommodates barely-alive servers. The middleware also dynamically re-sizes the cache, according to a desired hit ratio, using a distributed stack algorithm. Our preliminary results showed that our proposed caching middleware and barely-alive power state offer promising energy savings and the mechanisms to facilitate cluster-wide memory management for application-level caching.

8. REFERENCES

- [1] ACPI Consortium. Advanced Configuration and Power Interface. <http://www.acpi.info/>.
- [2] L. Adamic. Zipf, Power-laws, and Pareto - A Ranking Tutorial. Technical report, HP Labs, October 2000.
- [3] Y. Agarwal et al. Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage. In *Proceedings of NSDI*, April 2009.
- [4] L. A. Barroso and U. Holzle. The Case for Energy-Proportional Computing. *IEEE Computer*, 40(12), 2007.
- [5] B. T. Bennett and V. J. Kruskal. LRU Stack Processing. *IBM Research Journal*, 19(4), 1975.
- [6] E. V. Carrera and R. Bianchini. PRESS: A Clustered Server Based on User-Level Communication. *IEEE TPDS*, 16(5), 2004.
- [7] J. Chase et al. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of SOSR*, October 2001.
- [8] Y. Chen et al. Managing Server Energy and Operational Costs in Hosting Centers. In *Proceedings of SIGMETRICS*, June 2005.
- [9] M. Elnozahy, M. Kistler, and R. Rajamony. Energy-Efficient Server Clusters. In *Proceedings of PACS*, 2002.
- [10] X. Fan, W.-D. Weber, and L. A. Barroso. Power Provisioning for a Warehouse-sized Computer. In *Proceedings of ISCA*, June 2007.
- [11] T. Heath et al. Energy Conservation in Heterogeneous Server Clusters. In *Proceedings of PPOPP*, June 2005.
- [12] J. Liu et al. High Performance RDMA-Based MPI Implementation over InfiniBand. In *Proceedings of ICS*, 2003.
- [13] R. L. Mattson et al. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2), 1970.
- [14] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. In *Proceedings of ASPLOS*, March 2009.
- [15] V. Pai et al. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Proceedings of ASPLOS*, 1998.
- [16] E. Pinheiro et al. Load Balancing and Unbalancing for Power and Performance in Cluster-Based Systems. In *Proceedings of COLP*, September 2001.
- [17] K. Rajamani and C. Lefurgy. On Evaluating Request-Distribution Schemes for Saving Energy in Server Clusters. In *Proceedings of ISPASS*, March 2003.