# The case for sleep states in servers

Anshul Gandhi, Mor Harchol-Balter*
Carnegie Mellon University

Michael A. Kozuch
Intel Labs Pittsburgh

## ABSTRACT

While sleep states have existed for mobile devices and workstations for some time, these sleep states have largely not been incorporated into the servers in today's data centers.

Chip designers have been unmotivated to design sleep states because data center administrators haven't expressed any desire to have them. High setup times make administrators fearful of any form of dynamic power management, whereby servers are suspended or shut down when load drops. This general reluctance has stalled research into whether there might be *some* feasible sleep state (with sufficiently low setup overhead and/or sufficiently low power) that would actually be beneficial in data centers.

This paper uses both experimentation and theory to investigate the regime of sleep states that should be advantageous in data centers. Implementation experiments involve a 24-server multi-tier testbed, serving a web site of the type seen in Facebook or Amazon with key-value workload and a range of hypothetical sleep states. Analytical modeling is used to understand the effect of scaling up to larger data centers. The goal of this research is to encourage data center administrators to consider dynamic power management and to spur chip designers to develop *useful* sleep states for servers.

## 1. INTRODUCTION

While energy costs of data centers continue to double every 5 years [3], unfortunately, most of this energy is wasted. Servers are only busy 10-30% of the time on average [4], but they are often left on, while idle, utilizing 60% or more of peak power.
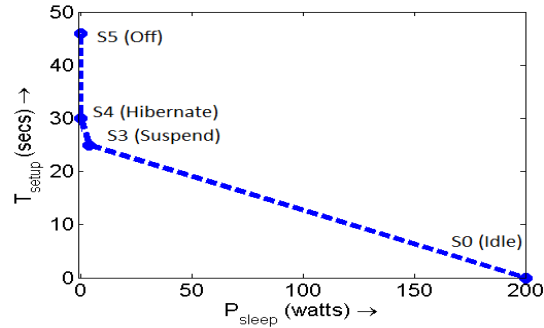
To save power, it has been proposed that servers

**Figure 1: Empirical results for a Dell desktop.**

should be put into some sleep state (or turned off) when they are not in use [7]. However, given that sleep states do not yet exist for most servers, it is hard to know how effective they will be in actually saving power. It is also hard to judge their negative effect on response time.

Prior work in dynamic power management using sleep states primarily deals with designing algorithms that figure out *when* to transition into an existing sleep state ([5, 6, 11]). Since most systems have only a handful of existing sleep states, and since sleep states on different systems may look very different, it is difficult to come to a consistent conclusion on the effectiveness of sleep states in general. While there has been some work considering the effectiveness of hypothetical sleep states ([7, 8]), the hypothetical states considered here are limited to transition times (setup times) on the order of 1 millisecond, and thus do not span the space of what is realistically possible today in servers.

The purpose of this paper is to gauge the effectiveness of a realistic range of sleep states. To determine this effectiveness fairly, we compare two algorithms. The first, `AlwaysOn`, is the status quo in power management, where the servers are left on all the time, regardless of load changes. We implement an *optimistic* version of `AlwaysOn` that actually knows the trace load ahead of time, and provisions the number of servers to exactly handle the peak load during the trace. The second algorithm, `Reactive`, is a popular algorithm in the literature [6] that reacts to the changes in load by turning on servers when load increases or putting servers into a (fixed) sleep state
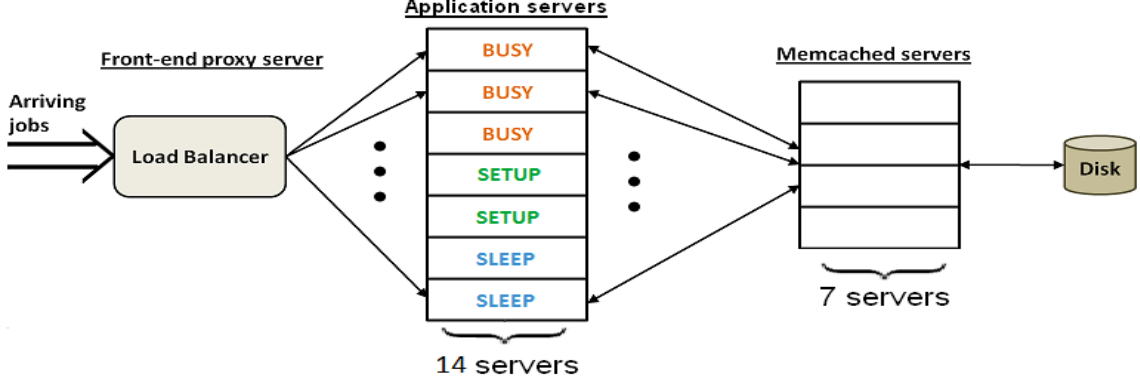
**Figure 2: Our experimental setup.**

when load drops. The `Reactive` algorithm does not know load in advance.

For each algorithm we measure the average response time, $T_{avg}$, and the average power consumption, $P_{avg}$. These yield the Performance-per-Watt, $PPW$, for each policy, defined as:

$$PPW = \frac{1}{T_{avg} \cdot P_{avg}}$$

Observe that higher $PPW$ is better. To compare the algorithms, we look at the Normalized Performance-per-Watt, $NPPW$, defined as the $PPW$ for `Reactive` normalized by that for `AlwaysOn`:

$$NPPW = \frac{PPW^{Reactive}}{PPW^{AlwaysOn}}$$

When $NPPW$ exceeds 1, we say that `Reactive` is superior to `AlwaysOn`.

A sleep state is defined by the pair, $(P_{sleep}, T_{setup})$. Here $P_{sleep}$ denotes the power consumed while sleeping (typically $P_{sleep} \ll P_{idle}$, where $P_{idle}$ is the idle power), and $T_{setup}$ denotes the time delay required to move a server from the sleep state to the on state. Furthermore, the whole time that the server is in setup mode, power is consumed at peak rate, $P_{max}$. Figure 1 shows our measurements for $(P_{sleep}, T_{setup})$ values for sleep states in a Dell desktop.

We evaluate the benefits of sleep states via implementation on a 24-server multi-tier data center, serving a web site of the type seen in Facebook or Amazon, with a key-value store workload. We use real-world arrival traces to generate load for our experiments.

Our results are surprising: For some traces, certain sleep states provide significant benefit in terms of $NPPW$, while for other traces (particularly bursty ones), even the best possible sleep state does not improve $NPPW$. We find that when the arrival rate varies slowly over time, then the $P_{sleep}$ value dictates the benefits of a given sleep state, while if the arrival rate varies quickly, then the $T_{setup}$ value

dictates the benefits of a given sleep state. Finally, we find that there is increased benefit to using sleep states as the size of the data center scales up.

## 2. EXPERIMENTAL SETUP

### Our experimental testbed

Figure 2 illustrates our data center testbed, consisting of 24 Intel Xeon E5520 servers, each equipped with two quad-core 2.26 GHz processors. We employ one of these servers as the front-end load generator running httperf [9] and another server as the front-end load-balancer running Apache, which distributes requests from the load generator to the application servers. We modify Apache on the load-balancer to also act as the capacity manager, which is responsible for suspending servers and waking them up. Another server is used to store the entire data set, a billion key-value pairs (∼500GB) on a BerkeleyDB [10] database.

Seven servers are used as memcached servers, each with 4GB of memory for caching, which serve all requests. The remaining 14 servers are employed as application servers, running Apache, which parse the incoming php requests and collect the required data from the back-end memcached servers.

We employ power management on the "front-end" application servers only, as they maintain no non-volatile state. We monitor the power consumption of servers by reading the power values off of the power distribution unit. The idle power consumption for our servers is $P_{idle} = 140W$ (with C-states enabled) and the peak power is $P_{max} = 200W$.

In our experiments, we replicate the effect of using a sleep state, $(P_{sleep}, T_{setup})$, by not sending requests to a server if it is marked for sleep, and by replacing its power consumption values by $P_{sleep}$ watts. When the server is marked for setup, we wait for $T_{setup}$ seconds before starting to send requests to the server.

**Workload**

Each key is a 9-digit number, and each value is a concatenation of random 9-digit numbers. Each generated request (or job) is a php script that runs on the application server. A job begins when the application server requests a value for a key from the memcached servers. The memcached servers provide the value, which itself is a collection of new keys. The application server then again requests values for these new keys from the memcached servers. This process can continue iteratively. In our experiments, we set the number of iterations to correspond to an average of roughly 3500 key requests per job, which translates to a mean job size of approximately 123 ms, assuming no resource contention.

**Trace-based arrivals**

Table 1 describes the traces we use. In our experiments, the seven memcached servers can together handle at most 800 job requests per second, which corresponds to roughly 400,000 key requests per second at each memcached server. Thus, we scale the arrival traces such that the maximum request rate into the system is 800 req/sec. We scale the duration of the traces to 2 hours.
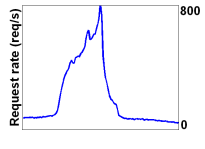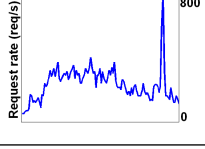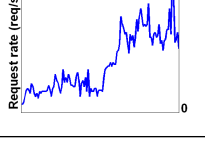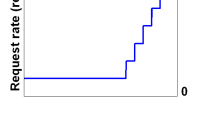
| Trace | Plot |
|---|---|
| ITA [2] <br><br> **(Slowly Varying)** |  |
| NLANR 1 [1] <br> **(Slowly Varying with Big Spike)** |  |
| NLANR 2 [1] <br><br> **(Dual Phase)** |  |
| NLANR 3 [1] <br> **(Dual Phase with Huge Variations)** |  |
| Synthetic <br><br> **(Multi-phase)** |  |

**Table 1: Description of the traces we use for experiments.**

**The AlwaysOn and Reactive algorithms**

The `AlwaysOn` algorithm maintains a fixed number of front-end servers on at all times. In our experiments, a single front-end server can handle roughly 60 req/sec while maintaining a 95%tile response time of 400ms. Based on this, `AlwaysOn` maintains $\lceil \frac{800}{60} \rceil = 14$ servers on at all times. The average data center utilization is between 30-40% for each trace under `AlwaysOn`.

By contrast, the `Reactive` algorithm tries to maintain $k_{req}(t) = \lceil \frac{\lambda(t)}{60} \rceil$ front-end servers at time $t$, where $\lambda(t)$ is the observed request rate at time $t$. If the actual number of servers at time $t$, $k(t)$, is lower than $k_{req}(t)$, then we turn on $(k_{req}(t) - k(t))$ servers (which will come online after $T_{setup}$ seconds), else we put $(k(t) - k_{req}(t))$ servers to sleep.

## 3. EVALUATION

This section evaluates the effect of different sleep states, denoted by $(P_{sleep}, T_{setup})$ in our testbed.

Figure 3 shows our experimental results for `Reactive` under different sleep states, all for the "Slowly Varying" trace (see Table 1). As expected, the average response time, $T_{avg}^{Reactive}$, increases as $T_{setup}$ is increased. The average power, $P_{avg}^{Reactive}$, increases slightly with increased $T_{setup}$ and increases significantly with increased $P_{sleep}$. Thus, the inverse of the product, $PPW^{Reactive}$, *decreases* with both $T_{setup}$ and $P_{sleep}$. By contrast, $PPW^{AlwaysOn}$ is unaffected by the sleep states and sits at a constant value of $PPW^{AlwaysOn} = 3 \cdot 10^{-6}$.

Figure 4 shows our experimental results for the Normalized Performance-per-Watt, $NPPW$, for four different arrival traces, including the Slowly Varying trace. Regions that are lightly shaded indicate the superiority of `Reactive` over `AlwaysOn`, and vice-versa. In general, $NPPW$ increases as $T_{setup} \rightarrow 0$ or $P_{sleep} \rightarrow 0$. We find that using sleep states can provide a huge benefit in terms of $NPPW$ for most of the arrival traces we consider. Interestingly, the effect of $T_{setup}$ and $P_{sleep}$ on $NPPW$ depends on the variability in arrival rates. For example, for the Dual Phase trace in Figure 4(c) (where the arrival rate varies quickly), the $T_{setup}$ value greatly effects $NPPW$ (varying by almost a factor of 2 between $T_{setup} = 20s$ and $T_{setup} = 200s$). This is because variations in arrival rate induce multiple setups. However, for the Slowly Varying trace in Figure 4(a), the $P_{sleep}$ value dictates the $NPPW$. Further, the superiority of `Reactive` over `AlwaysOn` also depends on the arrival trace. For example, for the Slowly Varying trace in Figure 4(a), our best sleep state ($T_{setup} = 20s$, $P_{sleep} = 0W$) provides a factor 2.2 improvement in $NPPW$ when com-
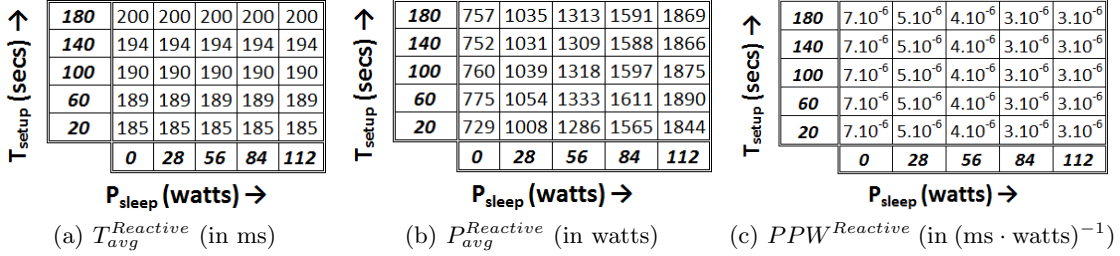
(a) $T_{avg}^{Reactive}$ (in ms)

(b) $P_{avg}^{Reactive}$ (in watts)

(c) $PPW^{Reactive}$ (in $(ms \cdot watts)^{-1}$)

**Figure 3: Results for `Reactive` under the Slowly Varying trace for a range of sleep states. For all sleep states, $PPW^{AlwaysOn} = 3 \cdot 10^{-6}$ (ms · watts)$^{-1}$.**



(a) Slowly Varying

(b) Slowly Varying with Big Spike

(c) Dual Phase

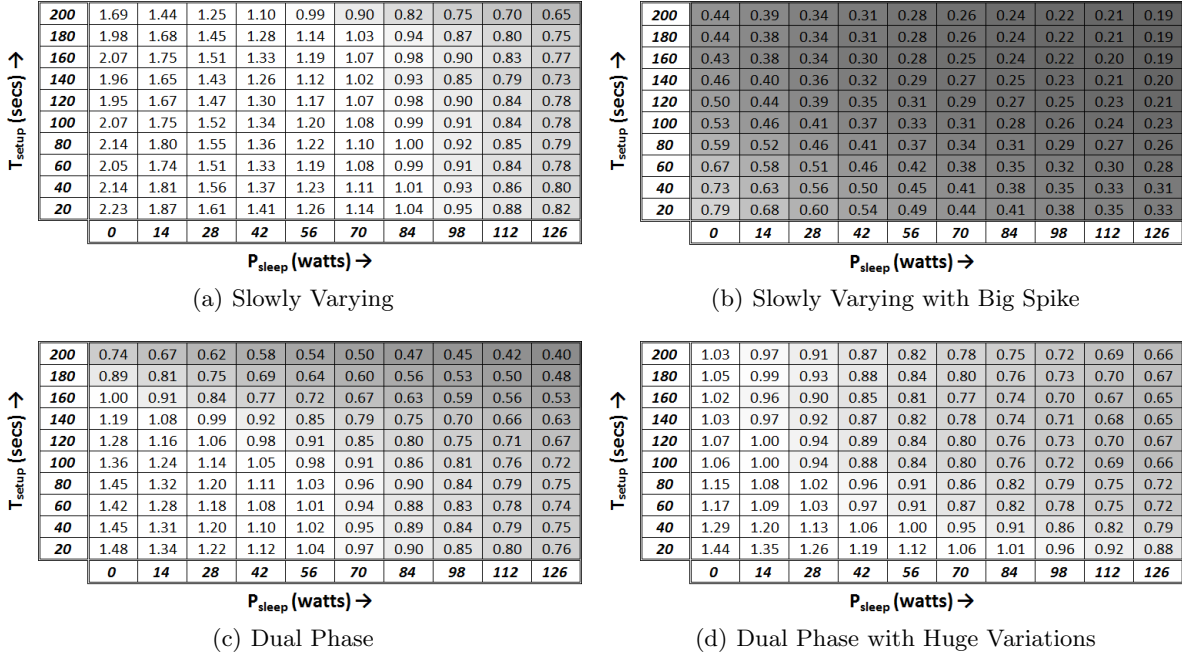(d) Dual Phase with Huge Variations

**Figure 4: Normalized Performance-per-Watt ($NPPW$) for different traces.**

pared to the idle state. However, for the Slowly Varying with Big Spike trace in Figure 4(b), even our best sleep state results in an $NPPW$ of only 0.79. We also ran experiments with $T_{setup} = 0s$ and found that $NPPW$ ranged from about 2.5 under $P_{sleep} = 0W$ to 1.1 under $P_{sleep} = 126W$ for most traces. We also evaluated the $NPPW$ metric using 95%tile response time values instead of $T_{avg}$ and found the results to be within 10% of those in Figure 4 for the different traces.

## 4. ANALYTICAL MODEL

To understand the effect of increasing the number of front-end servers, which is beyond the scope of our implementation, we develop a novel queueing-theoretic analysis. We model the data center via an $M_t/M/k$ queueing system with setup times, which is then solved via matrix-analytic methods. The $M_t/M/k$ queueing system with setup times allows us to evaluate the transient effects of changes in arrival rate as well as the effect of setup times, both of which cannot be captured by simply analyzing a sequence of $M/M/k$ queueing systems with different arrival rates.

Our queueing system is characterized by a single central queue from which the servers take requests. The maximum number of servers which are on, $k$, is set to $k = 14 \cdot 8$, mimicking the 14 8-core servers used in implementation. The $M_t$ notation indicates that the arrival process is time-varying Poisson, namely, the arrival rate varies from that suitable for 2 8-core servers (120 requests/sec) to an arrival rate requiring 14 8-core servers ($14 \cdot 60$ requests/sec), as shown for the multi-phase Synthetic trace in Table 1. The average utilization for the multi-phase trace is 30%, matching the implementation experiments. Job sizes are exponentially-distributed, with mean 120 ms, matching the implementation.

| $T_{setup}$ (secs) | $P_{sleep}$ (watts) → | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 200 | 1.78 | 1.52 | 1.33 | 1.18 | 1.06 | 0.96 | 0.88 | 0.81 | 0.75 | 0.70 |
| 180 | 1.83 | 1.56 | 1.36 | 1.21 | 1.09 | 0.99 | 0.90 | 0.83 | 0.77 | 0.72 |
| 160 | 1.87 | 1.60 | 1.40 | 1.24 | 1.11 | 1.01 | 0.93 | 0.85 | 0.79 | 0.74 |
| 140 | 1.92 | 1.64 | 1.43 | 1.27 | 1.14 | 1.04 | 0.95 | 0.88 | 0.81 | 0.76 |
| 120 | 1.97 | 1.68 | 1.47 | 1.30 | 1.17 | 1.06 | 0.98 | 0.90 | 0.83 | 0.78 |
| 100 | 2.02 | 1.73 | 1.51 | 1.34 | 1.20 | 1.09 | 1.00 | 0.92 | 0.86 | 0.80 |
| 80 | 2.08 | 1.78 | 1.55 | 1.38 | 1.24 | 1.12 | 1.03 | 0.95 | 0.88 | 0.82 |
| 60 | 2.14 | 1.83 | 1.60 | 1.42 | 1.27 | 1.16 | 1.06 | 0.98 | 0.91 | 0.85 |
| 40 | 2.20 | 1.88 | 1.64 | 1.46 | 1.31 | 1.19 | 1.09 | 1.01 | 0.94 | 0.87 |
| 20 | 2.27 | 1.94 | 1.69 | 1.50 | 1.35 | 1.23 | 1.13 | 1.04 | 0.96 | 0.90 |
|  | 0 | 14 | 28 | 42 | 56 | 70 | 84 | 98 | 112 | 126 |

(a) Theory

| $T_{setup}$ (secs) | $P_{sleep}$ (watts) → | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 200 | 2.08 | 1.71 | 1.45 | 1.26 | 1.12 | 1.00 | 0.91 | 0.83 | 0.77 | 0.71 |
| 180 | 2.11 | 1.72 | 1.45 | 1.26 | 1.11 | 0.99 | 0.90 | 0.82 | 0.75 | 0.70 |
| 160 | 2.12 | 1.75 | 1.48 | 1.29 | 1.14 | 1.02 | 0.93 | 0.85 | 0.78 | 0.72 |
| 140 | 2.15 | 1.75 | 1.47 | 1.27 | 1.12 | 1.00 | 0.90 | 0.82 | 0.76 | 0.70 |
| 120 | 2.14 | 1.75 | 1.47 | 1.27 | 1.12 | 1.00 | 0.91 | 0.83 | 0.76 | 0.70 |
| 100 | 2.05 | 1.68 | 1.42 | 1.23 | 1.09 | 0.97 | 0.88 | 0.80 | 0.74 | 0.68 |
| 80 | 2.06 | 1.69 | 1.43 | 1.24 | 1.09 | 0.98 | 0.89 | 0.81 | 0.74 | 0.69 |
| 60 | 2.16 | 1.78 | 1.51 | 1.31 | 1.16 | 1.04 | 0.94 | 0.86 | 0.79 | 0.74 |
| 40 | 2.14 | 1.74 | 1.47 | 1.27 | 1.12 | 1.00 | 0.91 | 0.83 | 0.76 | 0.71 |
| 20 | 2.14 | 1.75 | 1.47 | 1.27 | 1.12 | 1.00 | 0.91 | 0.83 | 0.76 | 0.70 |
|  | 0 | 14 | 28 | 42 | 56 | 70 | 84 | 98 | 112 | 126 |

(b) Implementation

**Figure 5:** $NPPW$ **for the multi-phase trace obtained via (a) theory and (b) implementation.**

Within our theoretical model, we analyze the `AlwaysOn` and `Reactive` algorithms. While we don't expect theory to match implementation perfectly, we believe that our model can identify regimes where `Reactive` is superior to `AlwaysOn`, that is, $NPPW > 1$. Figure 5(a) shows our analytical results for $NPPW$ as a function of the sleep state used, under the synthetic multi-phase trace. Our model suggests that the $NPPW$ is not very sensitive to $T_{setup}$ for this trace, and that `Reactive` is superior to `AlwaysOn` when $P_{sleep} \leq 70W$. Figure 5(b) shows our implementation results for the same multi-phase trace. Our implementation results agree with the insensitivity of $NPPW$ to $T_{setup}$, and match the regime where $NPPW > 1$. The above observations indicate that theory can be a useful tool for predicting the regimes where sleep states are effective.

Theory is particularly useful in understanding the effect of scaling up the data center size. In Figure 6 we concentrate on just one sleep state, ($P_{sleep} = 0W, T_{setup} = 20s$), and scale up the maximum number of servers from 7 to 70 for the multi-phase trace, while scaling the arrival rate proportionately such that the utilization is fixed at 30%. Scaling up the number of servers increases $NPPW$, making the `Reactive` algorithm more desirable as compared with `AlwaysOn`. Implementation results, shown in red crosses, seem to agree with the trends of our theoretical results.
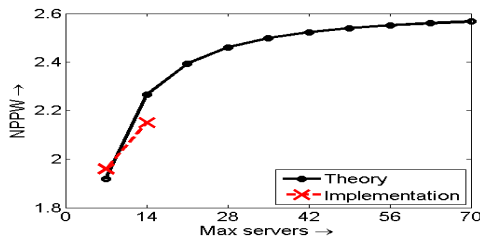


**Figure 6: Effect of scaling on** $NPPW$**.**

## 5. CONCLUSION AND FUTURE WORK

We examine the effectiveness of sleep states which are realistically feasible to implement today (with setup times ranging from 20s to 200s). Evaluation is done on a 24-server multi-tier testbed using real traces. We find that for most traces, if the power used in sleep ($P_{sleep}$) is less than half the idle power ($P_{idle}$), then a simple algorithm that reacts to current load by putting servers to sleep (`Reactive`) can increase $PPW$ by 10-100% over an optimistic static provisioning policy (`AlwaysOn`). Interestingly, for very bursty traces, even the best sleep state we consider is ineffective. We also develop a new queueing-theoretic model which allows us to predict the range of sleep states which are effective.

## 6. REFERENCES

[1] National Laboratory for Applied Network Research. Anonymized access logs. Available at ftp://ftp.ircache.net/Traces/.

[2] The internet traffic archives: WorldCup98. Available at http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[3] U.S. Environmental Protection Agency. Epa report on server and data center energy efficiency. 2007.

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

[5] Tibor Horvath and Kevin Skadron. Multi-mode energy management for multi-tier server clusters. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008.

[6] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. Napsac: Design and implementation of a power-proportional web cluster. In *First ACM SIGCOMM Workshop on Green Networking*, August 2010.

[7] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: eliminating server idle power. In *ASPLOS '09*, pages 205–216, New York, NY, USA.

[8] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th ACM International Symposium on Computer Architecture*, 2011.

[9] David Mosberger and Tai Jin. httperf—A Tool for Measuring Web Server Performance. *ACM Sigmetrics: Performance Evaluation Review*, 26:31–37, 1998.

[10] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA, 1999.

[11] Etienne Le Sueur and Gernot Heiser. Slow down or sleep, that is the question. In *USENIX Annual Technical Conference*, Portland, Oregon, USA, April 2011.