Are sleep states effective in data centers?

Anshul Gandhi, Mor Harchol-Balter
Carnegie Mellon University

Michael A. Kozuch
Intel Labs

***Abstract*—While sleep states have existed for mobile devices and workstations for some time, these sleep states have not been incorporated into most of the servers in today's data centers. High setup times make data center 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 investigates the regime of sleep states that would be advantageous in data centers. We consider the benefits of sleep states across three orthogonal dimensions: (i) the variability in the workload trace, (ii) the type of dynamic power management policy employed, and (iii) the size of the data center.

Our implementation results on a 24-server multi-tier testbed indicate that under many traces, sleep states greatly enhance dynamic power management. In fact, given the right sleep states, even a naïve policy that simply tries to match capacity with demand, can be very effective. By contrast, we characterize certain types of traces for which even the "best" sleep state under consideration is ineffective. Our simulation results suggest that sleep states are even more beneficial for larger data centers.

I. Introduction

Energy costs of data centers continue to double every 5 years [4], but what is most disappointing is that much of this power is *wasted*. Servers are only busy 10-30% of the time on average [5, 27] (despite virtualization), but they are often left on, while idle, utilizing 60% or more of peak power. The total reported cost (in terms of power, space, etc) of idle servers exceeds $19 billion per year and results in over 11 million tons of unnecessary CO_2 emissions each year [12].

The de facto power management policy used in data centers today is *AlwaysOn*, which leaves as many servers on as needed to handle the estimated peak demand. This is clearly wasteful of power during periods of lower loads. To reduce this "waste", it has been proposed [3, 19, 24, 25, 26] that idle servers should be put into some sleep state (or turned off). This is commonly referred to as *dynamic power management*. A big challenge with dynamic power management is that sleeping servers incur a high *setup time* to get them back on again. Given this high setup time, it is not at all obvious whether sleep states are useful or not.

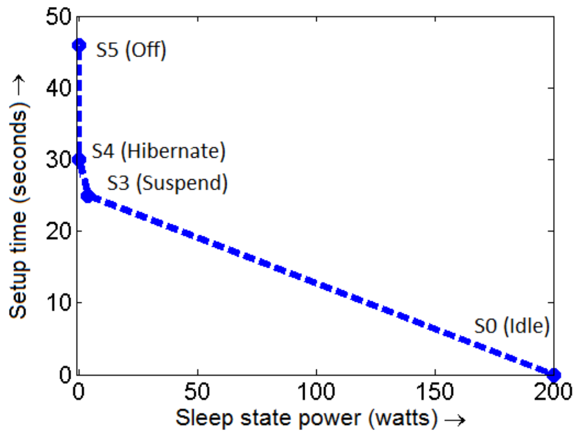


Figure 1. Existing sleep states in a Dell desktop.

Another challenge with dynamic power management is the scarcity of sleep states in today's computers. Servers in today's data centers do not yet support sleep states. Desktops and laptops support sleep states, but are only equipped with a handful of sleep states. Figure 1 shows the sleep states (and off and idle states) in a Dell desktop. We see that, apart from the off state, there are only two sleep states.

Given that sleep states do not yet exist for most servers, it is hard to estimate how much power they can actually save. Further, given that there is a *setup time* needed to bring sleeping servers back online, it is hard to judge the negative effect of sleep states on performance.

Prior work [9, 10, 13, 16, 18, 28, 29, 31] evaluated dynamic power management using only *existing* sleep states. That is, they only consider dynamic power management using the S3, S4 and S5 states, as shown in Figure 1. Given the limited range of existing sleep states, and because sleep states on different systems may look very different, it is difficult to assess the full potential of dynamic power management. While there has been some work [19, 20] considering the effectiveness of hypothetical sleep states, the hypothetical states considered in [19, 20] 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. There is also a long list of papers that look at dynamic power management assuming zero setup times (see, for example, [6, 17]). However, our focus in this paper is on dynamic power management using sleep states with realistic (non-zero) setup times.

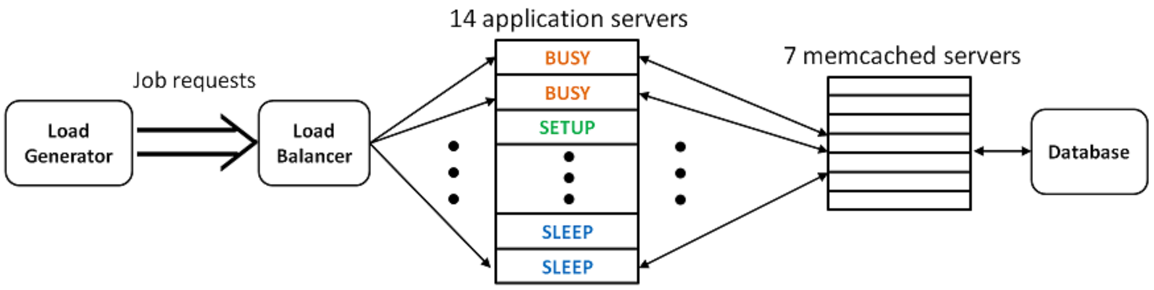


Figure 2. Our experimental setup.

The purpose of this paper is to gauge the effectiveness of a realistic *range* of sleep states. 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. In this paper, we consider a hundred different hypothetical sleep states, ranging from $(P_{sleep} = 0W, T_{setup} = 20s)$ to $(P_{sleep} = 126W, T_{setup} = 200s)$.

We evaluate the benefits of sleep states for use in dynamic power management 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. Since sleep states don't yet exist in servers, we "fake" their effect in implementation by marking a server as off when we would be putting it to sleep and then delaying resumption of service at the server for T_{setup} seconds when it would be in setup. To understand the effect of sleep states on larger data centers, which is beyond the scope of our implementation, we resort to a discrete-event simulator. We use real-world arrival traces to generate load for our experiments and simulations.

Our first experiments involve the simple dynamic power management policy called *Reactive*. Reactive responds to changes in load by putting servers to sleep when load drops and turning servers back on when the load increases (see Section IV-A for more details). We evaluate Reactive on a wide range of sleep states and compare it with AlwaysOn. For each policy (Reactive and AlwaysOn), we measure the 95th percentile of response times, T_{95}, since it captures the response time of 95% of the customers, and not just the average customer. We also measure the average power consumption, P_{avg}. These yield the Performance-per-Watt, PPW, for each policy, defined as:

$$PPW = \frac{1}{T_{95} \cdot P_{avg}}$$

Observe that higher PPW is better. To compare the policies, 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.

We find that for sufficiently "good" sleep states (low enough P_{sleep} and T_{setup}), under most traces, Reactive beats AlwaysOn with respect to $NPPW$. However, this is not always true. There are some traces (sufficiently bursty ones) where Reactive is inferior to AlwaysOn for *all* sleep states that we consider.

One might think that the fact that Reactive does not always dominate AlwaysOn is less a matter of the sleep state than it is the fact that Reactive is a very simple policy. To test out this theory, we therefore introduce a more sophisticated dynamic power management policy which we call *SoftReactive*. SoftReactive avoids needless state transitions in servers by being conservative about scaling down capacity (see Section V-A for more details). Under SoftReactive, we find that for sufficiently "good" sleep states, SoftReactive beats AlwaysOn with respect to $NPPW$ for *all* traces we consider (when using SoftReactive, we define $NPPW$ as the ratio between $PPW^{SoftReactive}$ and $PPW^{AlwaysOn}$). However, for bursty traces, the sleep state needed is on the order of the best sleep state (S3) currently available for today's Desktop machines (but not yet available for servers).

Finally, we consider how our results might change as the size (number of servers) of the data center increases. Using simulation, we consider data center sizes ranging from 14 servers to 1,400 servers, and scale the load proportionately so that the average load remains the same. We find that as the data center size grows, the setup time has less and less effect on performance. Thus, we can get away with larger values of T_{setup}. This finding holds true for both Reactive and SoftReactive.

This paper is based on a previous workshop paper, [11], wherein we consider a much narrower version of the same problem. In particular, in that paper, we only consider the Reactive policy, but do not introduce SoftReactive. We also

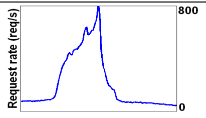
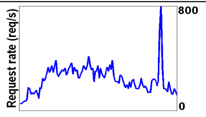
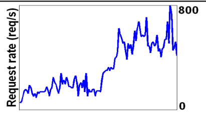
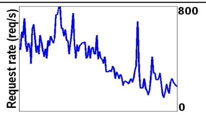
Trace	Plot
ITA [2] **(Slowly Varying)**	
NLANR 1 [1] **(Slowly Varying with Big Spike)**	
NLANR 2 [1] **(Dual Phase)**	
NLANR 3 [1] **(Dual Phase with Huge Variations)**	

Table I
DESCRIPTION OF THE TRACES WE USE FOR EXPERIMENTS.

do not include any simulation results in [11] and do not examine the effects of scaling to 1,400 servers. The work in this paper was motivated by the response we got from the workshop.

The rest of the paper is organized as follows. We describe our multi-tier implementation testbed in Section II. We then describe the AlwaysOn policy in Section III, which we will use as a yardstick to compare against. In Section IV, we describe the Reactive policy and examine the effectiveness of sleep states for Reactive. In Section V, we describe our SoftReactive policy and examine the effectiveness of sleep states for SoftReactive. We consider the effect of data center size on the effectiveness of sleep states in Section VI, and conclude in Section VII.

II. EXPERIMENTAL SETUP

A. *Our experimental testbed*

Figure 2 illustrates our data center testbed, consisting of 24 Intel Xeon 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 [21]. Another server is used as the front-end load-balancer running Apache, which distributes requests from the load generator to the 14 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. Each application server communicates with 7 memcached servers, each with 4GB of memory for caching, to retrieve data required to service the requests. Another server is used to store the entire data set, a billion key-value pairs ($\sim$500GB) on a BerkeleyDB [23] database.

We employ power management on the "front-end" application servers only, as they maintain no persistent state. We monitor the power consumption of the application servers, P_{avg}, by reading the power values from the power distribution unit. We do not consider the power consumed by the memcached servers. The idle power consumption for our servers is $P_{idle} = 140W$ (with C-states enabled) and the peak power observed for our experiments 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 sending requests to the server, and replace its power consumption values during the T_{setup} seconds with $P_{max} = 200W$.

B. *Workload*

We design a key-value workload to model realistic multi-tier applications such as the social networking site, Facebook, or e-commerce sites like Amazon [8]. Each generated request (or job) is a php script that runs on the application server. A request begins with the application server requesting 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 3,500 key-value requests per job, which translates to a mean service time of approximately 123 ms, assuming no resource contention. The request size distribution is highly variable, with the largest request being roughly 20 times the size of the smallest request.

In this paper, we use the Zipf [22] distribution to model the popularity of requests. To minimize the unpredictable effects of misses in the memcached layer, we tune the parameters of the Zipf distribution so that only a negligible fraction of requests miss in the memcached layer.

C. *Traces*

Table I shows the traces we use. We choose these particular traces because of the huge variability in request rates that they exhibit, which makes them challenging for dynamic power management. 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/s. We scale the duration of the traces to 2 hours. The 4 traces include a Slowly Varying trace [2], a Big Spike trace [1], a Dual Phase trace [1], and a Dual Phase with Huge Variations trace [1].

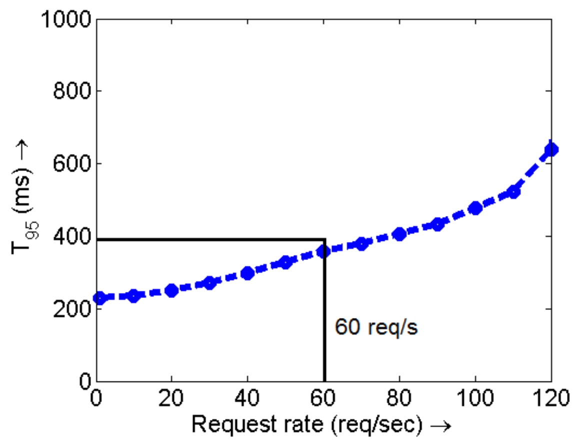


Figure 3. Each front-end server can handle 60 req/s.

III. The AlwaysOn policy

AlwaysOn [7, 13, 30] is a static power management policy that is currently used by most of the industry. The policy maintains a fixed number of front-end servers on at all times. In order to determine the number of front-end servers AlwaysOn should provision, we look at the request rate that each front-end server can handle. Typically, data center operators require that the 95th percentile of response times, T_{95}, stays below a certain threshold. Throughout the paper, we use 400 ms as the threshold. In our experiments, a single front-end server can handle roughly 60 req/s while maintaining a T_{95} of 400ms, as shown in Figure 3. Based on this, AlwaysOn maintains $\lceil \frac{800}{60} \rceil = 14$ servers on at all times, where 800 req/s is the peak request rate for each of our traces. Note that realistically, one doesn't know the peak request rate ahead of time. However, we empower AlwaysOn by assuming that the peak request rate is known in advance.

Since AlwaysOn provisions for the peak request rate, it results in a good T_{95}. However, the P_{avg} under AlwaysOn is usually high. Note that since AlwaysOn is a static power management policy, **the T_{95} and P_{avg} for a given trace under AlwaysOn are unaffected by the choice of sleep states**. This makes AlwaysOn a good policy to compare against when examining the effectiveness of sleep states for Reactive and SoftReactive. In particular, we consider a sleep state to be useful, if the PPW for a policy (Reactive or SoftReactive) under the given sleep state is better than the PPW for AlwaysOn. Thus, *a sleep state is considered useful if $NPPW > 1$.*

IV. Effect of sleep states on reactive

A. The Reactive policy

The Reactive policy tries to maintain $\lceil \frac{\lambda(t)}{60} \rceil$ front-end servers at time t, where $\lambda(t)$ is the observed request rate at time t, and 60 req/s is the request rate that can be handled by a single front-end server. If the actual number of servers at time t is lower than $\lceil \frac{\lambda(t)}{60} \rceil$, then we turn on the required additional servers (which will come online after T_{setup} seconds), else, we put the extra servers to sleep. Note that the servers can only go into a *single* fixed sleep state.

B. Evaluation

This section evaluates the effect of different sleep states, denoted by (P_{sleep}, T_{setup}), on Reactive.

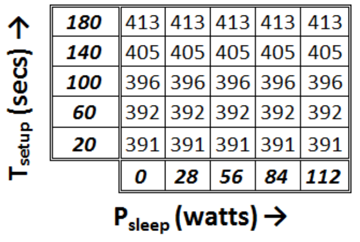
Figure 4 shows our experimental results for Reactive under different sleep states, all for the "Slowly Varying" trace (see Table I). As expected, the 95th percentile of response times, $T_{95}^{Reactive}$, increases as T_{setup} is increased. The average power, $P_{avg}^{Reactive}$, increases slightly 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 $PPW^{AlwaysOn} = 1.7 \cdot 10^{-6}$ (ms · watts)$^{-1}$.

Figure 5 shows our experimental results for the Normalized Performance-per-Watt, $NPPW$, for four different arrival traces, including the Slowly Varying trace. Lighter regions indicate higher $NPPW$, where $NPPW > 1$ indicates that Reactive is superior to AlwaysOn. In general, $NPPW$ increases as $T_{setup} \to 0$ or $P_{sleep} \to 0$.

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 request rates. For example, for the Dual Phase trace in Figure 5(c) (where the request rate varies quickly), the T_{setup} value greatly affects $NPPW$ (varying by more than a factor of 4 between $T_{setup} = 20s$ and $T_{setup} = 200s$). This is because variations in request rate induce multiple setups. However, for the Slowly Varying trace in Figure 5(a), the P_{sleep} value dictates the $NPPW$.

We also find that **the superiority of Reactive over AlwaysOn depends on the arrival trace**. For example, for the Slowly Varying trace in Figure 5(a), our best sleep state, $(P_{sleep} = 0W, T_{setup} = 20s)$, provides a factor 2.1 improvement in $NPPW$ when compared to AlwaysOn. However, **for the Slowly Varying with Big Spike trace in Figure 5(b), even our best sleep state results in an $NPPW$ of only 0.74**. Likewise, for the Dual Phase with Huge Variations trace in Figure 5(d), our best sleep state results in an $NPPW$ of 0.94. 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.

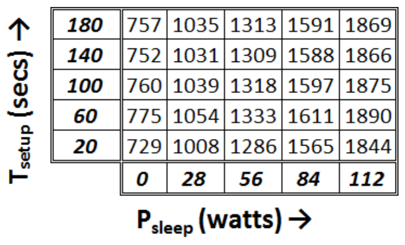
Figure 6 further illustrates the effect of arrival traces on the usefulness of sleep states. In general, as T_{setup} drops,



T_{setup} (secs) →
P_{sleep} (watts) →

	0	28	56	84	112
180	413	413	413	413	413
140	405	405	405	405	405
100	396	396	396	396	396
60	392	392	392	392	392
20	391	391	391	391	391

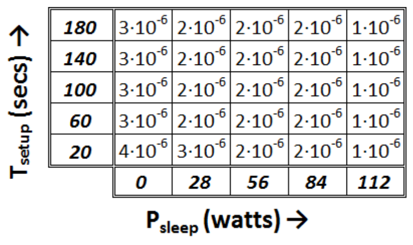
(a) $T_{95}^{Reactive}$ (in ms)



T_{setup} (secs) →
P_{sleep} (watts) →

	0	28	56	84	112
180	757	1035	1313	1591	1869
140	752	1031	1309	1588	1866
100	760	1039	1318	1597	1875
60	775	1054	1333	1611	1890
20	729	1008	1286	1565	1844

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

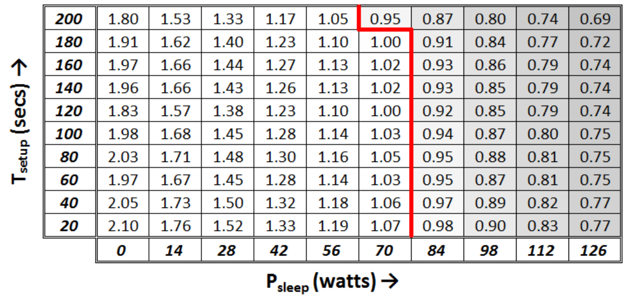


T_{setup} (secs) →
P_{sleep} (watts) →

	0	28	56	84	112
180	$3\cdot10^{-6}$	$2\cdot10^{-6}$	$2\cdot10^{-6}$	$2\cdot10^{-6}$	$1\cdot10^{-6}$
140	$3\cdot10^{-6}$	$2\cdot10^{-6}$	$2\cdot10^{-6}$	$2\cdot10^{-6}$	$1\cdot10^{-6}$
100	$3\cdot10^{-6}$	$2\cdot10^{-6}$	$2\cdot10^{-6}$	$2\cdot10^{-6}$	$1\cdot10^{-6}$
60	$3\cdot10^{-6}$	$2\cdot10^{-6}$	$2\cdot10^{-6}$	$2\cdot10^{-6}$	$1\cdot10^{-6}$
20	$4\cdot10^{-6}$	$3\cdot10^{-6}$	$2\cdot10^{-6}$	$2\cdot10^{-6}$	$1\cdot10^{-6}$

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

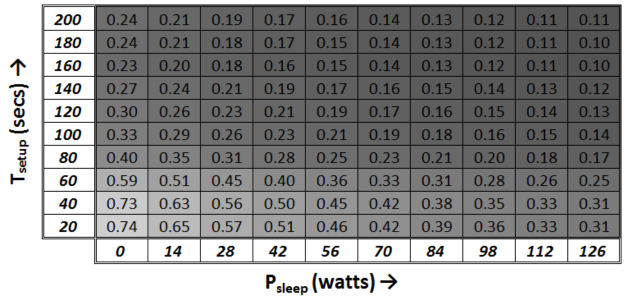
Figure 4. Results for Reactive with respect to (a) T_{95}, (b) P_{avg}, and (c) PPW, under the Slowly Varying trace for a range of sleep states. For all sleep states, $PPW^{AlwaysOn} = 1.7 \cdot 10^{-6}$ (ms · watts)$^{-1}$.



T_{setup} (secs) →
P_{sleep} (watts) →

	0	14	28	42	56	70	84	98	112	126
200	1.80	1.53	1.33	1.17	1.05	0.95	0.87	0.80	0.74	0.69
180	1.91	1.62	1.40	1.23	1.10	1.00	0.91	0.84	0.77	0.72
160	1.97	1.66	1.44	1.27	1.13	1.02	0.93	0.86	0.79	0.74
140	1.96	1.66	1.43	1.26	1.13	1.02	0.93	0.85	0.79	0.74
120	1.83	1.57	1.38	1.23	1.10	1.00	0.92	0.85	0.79	0.74
100	1.98	1.68	1.45	1.28	1.14	1.03	0.94	0.87	0.80	0.75
80	2.03	1.71	1.48	1.30	1.16	1.05	0.95	0.88	0.81	0.75
60	1.97	1.67	1.45	1.28	1.14	1.03	0.95	0.87	0.81	0.75
40	2.05	1.73	1.50	1.32	1.18	1.06	0.97	0.89	0.82	0.77
20	2.10	1.76	1.52	1.33	1.19	1.07	0.98	0.90	0.83	0.77

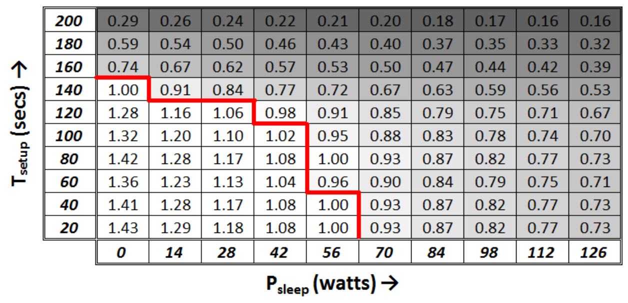
(a) Slowly Varying trace



T_{setup} (secs) →
P_{sleep} (watts) →

	0	14	28	42	56	70	84	98	112	126
200	0.24	0.21	0.19	0.17	0.16	0.14	0.13	0.12	0.11	0.11
180	0.24	0.21	0.18	0.17	0.15	0.14	0.13	0.12	0.11	0.10
160	0.23	0.20	0.18	0.16	0.15	0.14	0.13	0.12	0.11	0.10
140	0.27	0.24	0.21	0.19	0.17	0.16	0.15	0.14	0.13	0.12
120	0.30	0.26	0.23	0.21	0.19	0.17	0.16	0.15	0.14	0.13
100	0.33	0.29	0.26	0.23	0.21	0.19	0.18	0.16	0.15	0.14
80	0.40	0.35	0.31	0.28	0.25	0.23	0.21	0.20	0.18	0.17
60	0.59	0.51	0.45	0.40	0.36	0.33	0.31	0.28	0.26	0.25
40	0.73	0.63	0.56	0.50	0.45	0.42	0.38	0.35	0.33	0.31
20	0.74	0.65	0.57	0.51	0.46	0.42	0.39	0.36	0.33	0.31

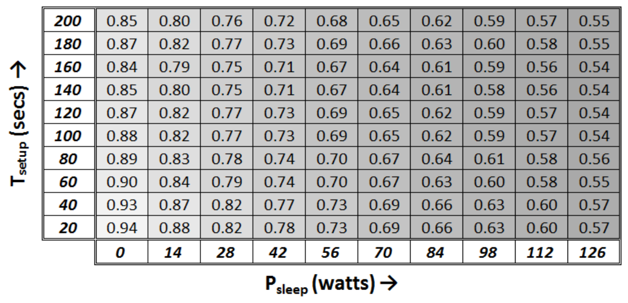
(b) Slowly Varying with Big Spike trace



T_{setup} (secs) →
P_{sleep} (watts) →

	0	14	28	42	56	70	84	98	112	126
200	0.29	0.26	0.24	0.22	0.21	0.20	0.18	0.17	0.16	0.16
180	0.59	0.54	0.50	0.46	0.43	0.40	0.37	0.35	0.33	0.32
160	0.74	0.67	0.62	0.57	0.53	0.50	0.47	0.44	0.42	0.39
140	1.00	0.91	0.84	0.77	0.72	0.67	0.63	0.59	0.56	0.53
120	1.28	1.16	1.06	0.98	0.91	0.85	0.79	0.75	0.71	0.67
100	1.32	1.20	1.10	1.02	0.95	0.88	0.83	0.78	0.74	0.70
80	1.42	1.28	1.17	1.08	1.00	0.93	0.87	0.82	0.77	0.73
60	1.36	1.23	1.13	1.04	0.96	0.90	0.84	0.79	0.75	0.71
40	1.41	1.28	1.17	1.08	1.00	0.93	0.87	0.82	0.77	0.73
20	1.43	1.29	1.18	1.08	1.00	0.93	0.87	0.82	0.77	0.73

(c) Dual Phase trace



T_{setup} (secs) →
P_{sleep} (watts) →

	0	14	28	42	56	70	84	98	112	126
200	0.85	0.80	0.76	0.72	0.68	0.65	0.62	0.59	0.57	0.55
180	0.87	0.82	0.77	0.73	0.69	0.66	0.63	0.60	0.58	0.55
160	0.84	0.79	0.75	0.71	0.67	0.64	0.61	0.59	0.56	0.54
140	0.85	0.80	0.75	0.71	0.67	0.64	0.61	0.58	0.56	0.54
120	0.87	0.82	0.77	0.73	0.69	0.65	0.62	0.59	0.57	0.54
100	0.88	0.82	0.77	0.73	0.69	0.65	0.62	0.59	0.57	0.54
80	0.89	0.83	0.78	0.74	0.70	0.67	0.64	0.61	0.58	0.56
60	0.90	0.84	0.79	0.74	0.70	0.67	0.63	0.60	0.58	0.55
40	0.93	0.87	0.82	0.77	0.73	0.69	0.66	0.63	0.60	0.57
20	0.94	0.88	0.82	0.78	0.73	0.69	0.66	0.63	0.60	0.57

(d) Dual Phase with Huge Variations trace

Figure 5. Normalized Performance-per-Watt ($NPPW$) for different traces under Reactive.

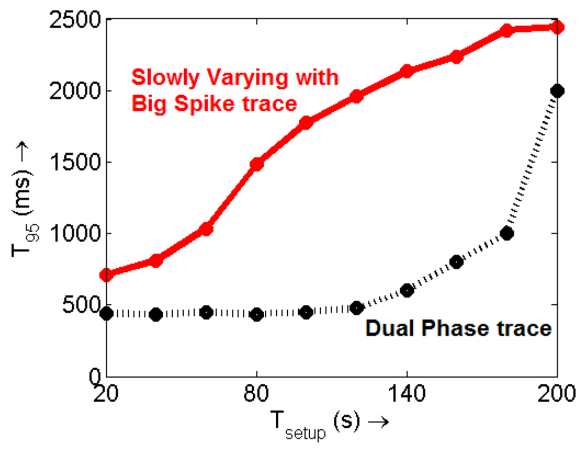


Figure 6. *Effect of arrival traces on usefulness of sleep states for Reactive. For the sleep states in the above plot,* $P_{sleep} = 0W$.

we see that T_{95} drops as well. Recall that ideally our system should achieve a T_{95} of 400 ms. For the Slowly Varying with Big Spike trace (solid line), we see that T_{95} drops almost linearly with T_{setup}, but is always well above 400 ms. This is because of the big spike in the trace, which cannot be handled even by $T_{setup} = 20s$. By contrast, for the Dual Phase trace (dotted line), we see that T_{95} drops quickly to about 400 ms as T_{setup} drops from 200s to 140s, and is relatively constant thereafter. Thus, even a high setup time ($T_{setup} = 140s$) is fine for the Dual Phase trace while the lowest setup time ($T_{setup} = 20s$) is still insufficient for the Big Spike trace.

We also investigated Reactive with C-states instead of sleep states. Recall that our idle state already has C-states enabled. In Reactive with C-states, the "unneeded" servers, as computed via the Reactive policy, transition to the best C-state ($P_{sleep} = 140W$, $T_{setup} = 0s$). Our results suggest that Reactive with C-states is always worse than AlwaysOn,

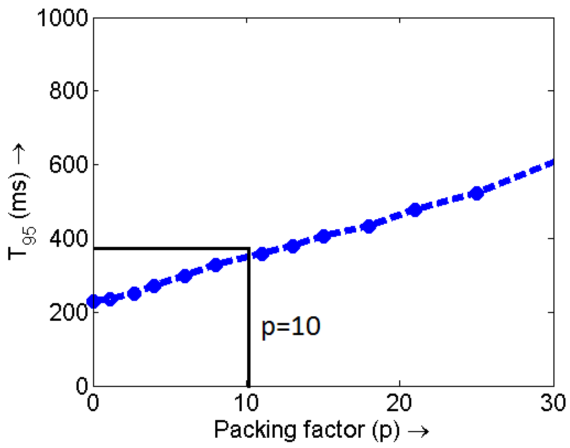


Figure 7. *T_{95} vs. packing factor for a single server under SoftReactive.*

with $NPPW$ ranging from 0.68 – 0.97 for the traces we use. This is because C-states do not provide any significant power savings over the idle state. Further, the T_{95} under AlwaysOn is usually significantly lower than Reactive with C-states since AlwaysOn load balances requests among all 14 application servers at all times.

V. The SoftReactive policy

One might think that the less-than-ideal performance of Reactive stems from the fact that it is too slow to turn servers on when needed. However, an equally big concern is the fact that Reactive is quick to turn servers *off* when not needed, and hence does not have those servers available when load subsequently rises. This rashness is particularly problematic in the case of bursty workloads, such as those in Table I. To remedy this problem, we now introduce the *SoftReactive* policy.

A. Description of SoftReactive

SoftReactive addresses the problem of scaling down capacity by being *very conservative in turning servers off* while doing nothing new with respect to turning servers on (the turning on algorithm is the same as in Reactive). We will show that *by simply taking more care in turning servers off*, SoftReactive is able to outperform Reactive with respect to T_{95}, while simultaneously keeping P_{avg} low.

t_{wait} → / ↓ (P_{sleep}, T_{setup})	$\frac{t^*_{wait}}{4}$	$\frac{t^*_{wait}}{2}$	t^*_{wait}	$2t^*_{wait}$	$4t^*_{wait}$
(0W, 20s)	1.63	1.64	**1.62**	1.57	1.49
(56W, 100s)	1.15	1.15	**1.14**	1.12	1.08
(126W, 200s)	0.84	0.84	**0.85**	0.86	0.86

Table II
Effect of t_{wait} on $NPPW$ for the Slowly Varying trace. We see that $t_{wait} = t^*_{wait} \approx T_{setup} \cdot \frac{P_{max}}{P_{idle}}$ works well for a range of sleep states.

When to turn a server off?
Under SoftReactive, each server decides *autonomously* when to turn off. When a server goes idle, rather than turning off immediately, it sets a timer of duration t_{wait} and sits in the idle state for t_{wait} seconds. If a request arrives at the server during these t_{wait} seconds, then the server goes back to the busy state (with zero setup cost); otherwise the server is turned off.

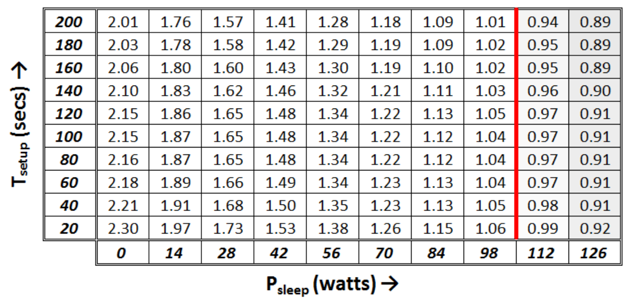
A natural question that arises is how to choose the optimal value of t_{wait}, which we denote as t^*_{wait}. A good choice for t^*_{wait} is $t^*_{wait} \approx T_{setup} \cdot \frac{P_{max}}{P_{idle}}$, where $P_{max} = 200W$ and $P_{idle} = 140W$ are the power consumption of the server when in setup and when in the idle state respectively. The intuition behind this idea is to equate the energy consumed waiting in the idle state, $(t^*_{wait} \cdot P_{idle})$, to the energy consumed when turning a server on, $(T_{setup} \cdot P_{max})$. Table II verifies our choice of t^*_{wait} for the Slowly Varying trace under SoftReactive. We see that our choice of t^*_{wait} works well for various sleep states.

The idea of setting a timer before turning off an idle server has been proposed before (see, for example, [14, 15, 18]), however, *only for a single server*. For a multi-server system, *independently* setting timers for each server can be inefficient, since we can end up with too many idle servers. Thus, we need a more coordinated approach for using timers in our multi-server system which takes routing into account, as explained below.

How to route jobs to servers?
Timers prevent the mistake of turning off a server just as a new arrival comes in. However, they can also waste power by leaving too many servers in the idle state. We want to keep only a small number of servers (just the *right* number) in this idle state. To do this, we introduce a routing scheme that tends to concentrate jobs onto a small number of servers, so that the remaining (unneeded) servers will naturally "timeout." Our routing scheme uses an *index-packing* idea, whereby all on servers are indexed from 1 to n. Then we send each request to the lowest-numbered on server that currently has fewer than p requests, where p stands for *packing factor* and denotes the optimal number of requests that a server can serve concurrently. For example, in Figure 7, we see that to meet a T_{95} of 400 ms, the packing factor for our servers is $p = 10$. When all on servers are already packed with p requests each, additional arrivals are routed to servers via the join-the-shortest-queue routing.

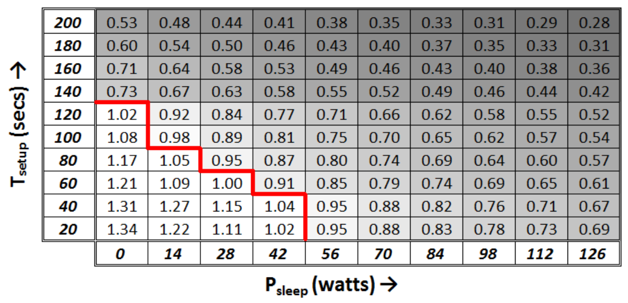
B. Evaluation

We now examine the effectiveness of sleep states under SoftReactive. Figure 8 shows our experimental results for $NPPW$ for all four arrival traces. Lighter regions indicate higher $NPPW$, where $NPPW > 1$ indicates that SoftReactive is superior to AlwaysOn.



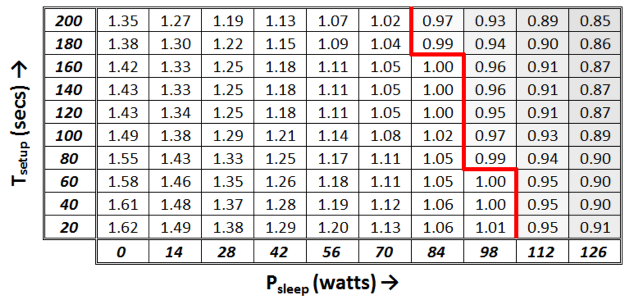
T_{setup} (secs) →										
200	2.01	1.76	1.57	1.41	1.28	1.18	1.09	1.01	0.94	0.89
180	2.03	1.78	1.58	1.42	1.29	1.19	1.09	1.02	0.95	0.89
160	2.06	1.80	1.60	1.43	1.30	1.19	1.10	1.02	0.95	0.89
140	2.10	1.83	1.62	1.46	1.32	1.21	1.11	1.03	0.96	0.90
120	2.15	1.86	1.65	1.48	1.34	1.22	1.13	1.05	0.97	0.91
100	2.15	1.87	1.65	1.48	1.34	1.22	1.12	1.04	0.97	0.91
80	2.16	1.87	1.65	1.48	1.34	1.22	1.12	1.04	0.97	0.91
60	2.18	1.89	1.66	1.49	1.34	1.23	1.13	1.04	0.97	0.91
40	2.21	1.91	1.68	1.50	1.35	1.23	1.13	1.05	0.98	0.91
20	2.30	1.97	1.73	1.53	1.38	1.26	1.15	1.06	0.99	0.92
	0	14	28	42	56	70	84	98	112	126
	P_{sleep} (watts) →									

(a) Slowly Varying trace



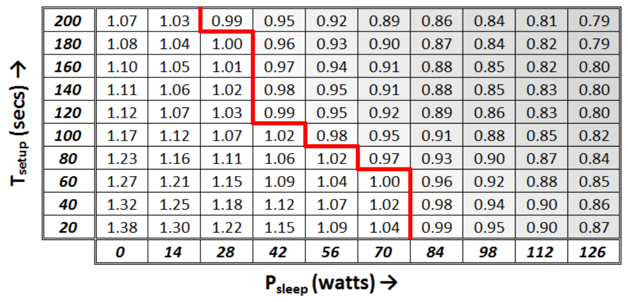
T_{setup} (secs) →										
200	0.53	0.48	0.44	0.41	0.38	0.35	0.33	0.31	0.29	0.28
180	0.60	0.54	0.50	0.46	0.43	0.40	0.37	0.35	0.33	0.31
160	0.71	0.64	0.58	0.53	0.49	0.46	0.43	0.40	0.38	0.36
140	0.73	0.67	0.63	0.58	0.55	0.52	0.49	0.46	0.44	0.42
120	1.02	0.92	0.84	0.77	0.71	0.66	0.62	0.58	0.55	0.52
100	1.08	0.98	0.89	0.81	0.75	0.70	0.65	0.62	0.57	0.54
80	1.17	1.05	0.95	0.87	0.80	0.74	0.69	0.64	0.60	0.57
60	1.21	1.09	1.00	0.91	0.85	0.79	0.74	0.69	0.65	0.61
40	1.31	1.27	1.15	1.04	0.95	0.88	0.82	0.76	0.71	0.67
20	1.34	1.22	1.11	1.02	0.95	0.88	0.83	0.78	0.73	0.69
	0	14	28	42	56	70	84	98	112	126
	P_{sleep} (watts) →									

(b) Slowly Varying with Big Spike trace



T_{setup} (secs) →										
200	1.35	1.27	1.19	1.13	1.07	1.02	0.97	0.93	0.89	0.85
180	1.38	1.30	1.22	1.15	1.09	1.04	0.99	0.94	0.90	0.86
160	1.42	1.33	1.25	1.18	1.11	1.05	1.00	0.96	0.91	0.87
140	1.43	1.33	1.25	1.18	1.11	1.05	1.00	0.96	0.91	0.87
120	1.43	1.34	1.25	1.18	1.11	1.05	1.00	0.95	0.91	0.87
100	1.49	1.38	1.29	1.21	1.14	1.08	1.02	0.97	0.93	0.89
80	1.55	1.43	1.33	1.25	1.17	1.11	1.05	0.99	0.94	0.90
60	1.58	1.46	1.35	1.26	1.18	1.11	1.05	1.00	0.95	0.90
40	1.61	1.48	1.37	1.28	1.19	1.12	1.06	1.00	0.95	0.90
20	1.62	1.49	1.38	1.29	1.20	1.13	1.06	1.01	0.95	0.91
	0	14	28	42	56	70	84	98	112	126
	P_{sleep} (watts) →									

(c) Dual Phase trace



T_{setup} (secs) →										
200	1.07	1.03	0.99	0.95	0.92	0.89	0.86	0.84	0.81	0.79
180	1.08	1.04	1.00	0.96	0.93	0.90	0.87	0.84	0.82	0.79
160	1.10	1.05	1.01	0.97	0.94	0.91	0.88	0.85	0.82	0.80
140	1.11	1.06	1.02	0.98	0.95	0.91	0.88	0.85	0.83	0.80
120	1.12	1.07	1.03	0.99	0.95	0.92	0.89	0.86	0.83	0.80
100	1.17	1.12	1.07	1.02	0.98	0.95	0.91	0.88	0.85	0.82
80	1.23	1.16	1.11	1.06	1.02	0.97	0.93	0.90	0.87	0.84
60	1.27	1.21	1.15	1.09	1.04	1.00	0.96	0.92	0.88	0.85
40	1.32	1.25	1.18	1.12	1.07	1.02	0.98	0.94	0.90	0.86
20	1.38	1.30	1.22	1.15	1.09	1.04	0.99	0.95	0.90	0.87
	0	14	28	42	56	70	84	98	112	126
	P_{sleep} (watts) →									

(d) Dual Phase with Huge Variations trace

Figure 8. Normalized Performance-per-Watt ($NPPW$) for different traces under SoftReactive.

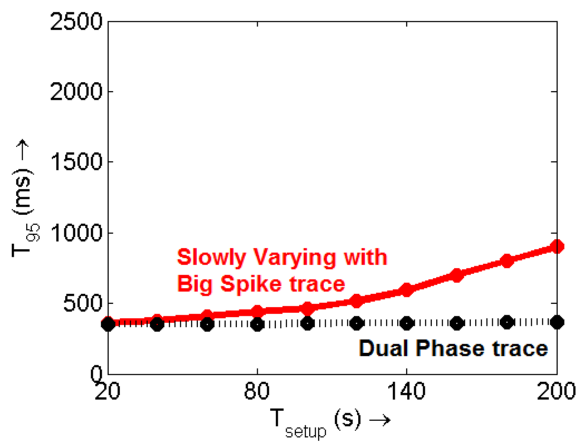


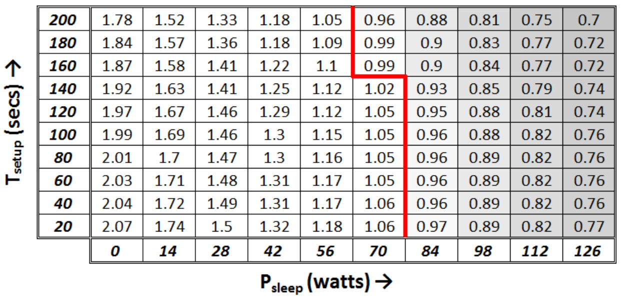
Figure 9. *Effect of arrival traces on usefulness of sleep states for SoftReactive. For the sleep states in the above plot,* $P_{sleep} = 0W$.

We find that **SoftReactive enhances the effectiveness of sleep states for all arrival traces we consider**. This can be seen by comparing each of the traces in Figure 5 with the corresponding ones in Figure 8. Interestingly, **under SoftReactive, the T_{setup} value does not significantly affect $NPPW$ for most of the traces we consider**. The $NPPW$ value is largely invariant for a given P_{sleep} and a given trace in Figure 8. For example, if we consider the Dual Phase trace with $P_{sleep} = 0W$, we see that under SoftReactive, $NPPW$ decreases by only 17% as we move from $T_{setup} = 20s$ to $T_{setup} = 200s$. By contrast, under the same Dual Phase trace with $P_{sleep} = 0W$, the $NPPW$ decreases by 80% under Reactive as we move from $T_{setup} = 20s$ to $T_{setup} = 200s$. The insensitivity of SoftReactive to T_{setup} is to be expected since SoftReactive is designed to avoid setup times.

While SoftReactive is fairly insensitive to the choice of sleep states, it is sensitive to the arrival trace. For example, for the Slowly Varying trace in Figure 8(a), our best sleep state, namely ($P_{sleep} = 0W$, $T_{setup} = 20s$), provides a factor 2.3 improvement in $NPPW$ (huge win) when compared to AlwaysOn. By contrast, for the Slowly Varying with Big Spike trace in Figure 8(b), our best sleep state results in an $NPPW$ of 1.34 (smaller win over AlwaysOn). The sensitivity of SoftReactive to arrival traces is further illustrated in Figure 9. For the Slowly Varying with Big Spike trace (solid line), we see that T_{95} drops almost linearly from about 900 ms at $T_{setup} = 200s$ to about 350 ms at $T_{setup} = 20s$. By contrast, for the Dual Phase trace (dotted line), we see that T_{95} is largely insensitive to T_{setup}, and lies between 350 ms and 400 ms.

In Section IV-B, we saw that there were no sleep states for which Reactive was effective on some traces (see Figures 5(b) and 5(d)). With SoftReactive, in contrast, there is a set of sleep states, namely $T_{setup} \leq 60s$ and $P_{sleep} \leq 28W$, for which $NPPW > 1$ under all traces (see Figures 8(a) - 8(d)). It is sleep states in this range that we would recommend hardware designers to focus on. The superiority of SoftReactive over Reactive is because of the huge improvement with respect to T_{95}, which is evident from Figures 6 and 9.

We also investigated SoftReactive with C-states. Our results suggest that SoftReactive with C-states, just like

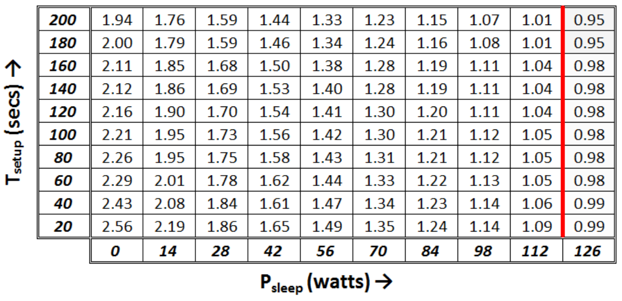


T_{setup} (secs) ↑

200	1.78	1.52	1.33	1.18	1.05	0.96	0.88	0.81	0.75	0.7
180	1.84	1.57	1.36	1.18	1.09	0.99	0.9	0.83	0.77	0.72
160	1.87	1.58	1.41	1.22	1.1	0.99	0.9	0.84	0.77	0.72
140	1.92	1.63	1.41	1.25	1.12	1.02	0.93	0.85	0.79	0.74
120	1.97	1.67	1.46	1.29	1.12	1.05	0.95	0.88	0.81	0.74
100	1.99	1.69	1.46	1.3	1.15	1.05	0.96	0.88	0.82	0.76
80	2.01	1.7	1.47	1.3	1.16	1.05	0.96	0.89	0.82	0.76
60	2.03	1.71	1.48	1.31	1.17	1.05	0.96	0.89	0.82	0.76
40	2.04	1.72	1.49	1.31	1.17	1.06	0.96	0.89	0.82	0.76
20	2.07	1.74	1.5	1.32	1.18	1.06	0.97	0.89	0.82	0.77
	0	**14**	**28**	**42**	**56**	**70**	**84**	**98**	**112**	**126**

P_{sleep} (watts) →

(a) Simulation: Reactive

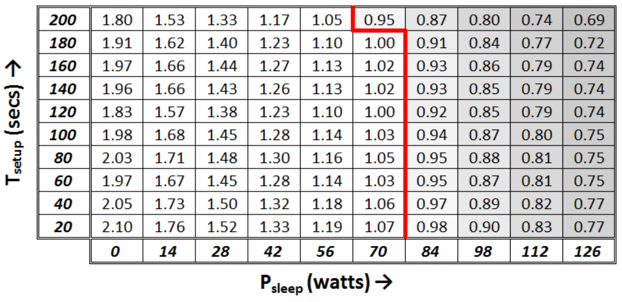


T_{setup} (secs) ↑

200	1.94	1.76	1.59	1.44	1.33	1.23	1.15	1.07	1.01	0.95
180	2.00	1.79	1.59	1.46	1.34	1.24	1.16	1.08	1.01	0.95
160	2.11	1.85	1.68	1.50	1.38	1.28	1.19	1.11	1.04	0.98
140	2.12	1.86	1.69	1.53	1.40	1.28	1.19	1.11	1.04	0.98
120	2.16	1.90	1.70	1.54	1.41	1.30	1.20	1.11	1.04	0.98
100	2.21	1.95	1.73	1.56	1.42	1.30	1.21	1.12	1.05	0.98
80	2.26	1.95	1.75	1.58	1.43	1.31	1.21	1.12	1.05	0.98
60	2.29	2.01	1.78	1.62	1.44	1.33	1.22	1.13	1.05	0.98
40	2.43	2.08	1.84	1.61	1.47	1.34	1.23	1.14	1.06	0.99
20	2.56	2.19	1.86	1.65	1.49	1.35	1.24	1.14	1.09	0.99
	0	**14**	**28**	**42**	**56**	**70**	**84**	**98**	**112**	**126**

P_{sleep} (watts) →

(b) Simulation: SoftReactive

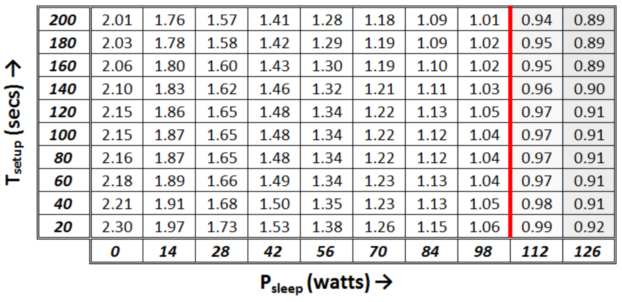


T_{setup} (secs) ↑

200	1.80	1.53	1.33	1.17	1.05	0.95	0.87	0.80	0.74	0.69
180	1.91	1.62	1.40	1.23	1.10	1.00	0.91	0.84	0.77	0.72
160	1.97	1.66	1.44	1.27	1.13	1.02	0.93	0.86	0.79	0.74
140	1.96	1.66	1.43	1.26	1.13	1.02	0.93	0.85	0.79	0.74
120	1.83	1.57	1.38	1.23	1.10	1.00	0.92	0.85	0.79	0.74
100	1.98	1.68	1.45	1.28	1.14	1.03	0.94	0.87	0.80	0.75
80	2.03	1.71	1.48	1.30	1.16	1.05	0.95	0.88	0.81	0.75
60	1.97	1.67	1.45	1.28	1.14	1.03	0.95	0.87	0.81	0.75
40	2.05	1.73	1.50	1.32	1.18	1.06	0.97	0.89	0.82	0.77
20	2.10	1.76	1.52	1.33	1.19	1.07	0.98	0.90	0.83	0.77
	0	**14**	**28**	**42**	**56**	**70**	**84**	**98**	**112**	**126**

P_{sleep} (watts) →

(c) Implementation: Reactive



T_{setup} (secs) ↑

200	2.01	1.76	1.57	1.41	1.28	1.18	1.09	1.01	0.94	0.89
180	2.03	1.78	1.58	1.42	1.29	1.19	1.09	1.02	0.95	0.89
160	2.06	1.80	1.60	1.43	1.30	1.19	1.10	1.02	0.95	0.89
140	2.10	1.83	1.62	1.46	1.32	1.21	1.11	1.03	0.96	0.90
120	2.15	1.86	1.65	1.48	1.34	1.22	1.13	1.05	0.97	0.91
100	2.15	1.87	1.65	1.48	1.34	1.22	1.12	1.04	0.97	0.91
80	2.16	1.87	1.65	1.48	1.34	1.22	1.12	1.04	0.97	0.91
60	2.18	1.89	1.66	1.49	1.34	1.23	1.13	1.04	0.97	0.91
40	2.21	1.91	1.68	1.50	1.35	1.23	1.13	1.05	0.98	0.91
20	2.30	1.97	1.73	1.53	1.38	1.26	1.15	1.06	0.99	0.92
	0	**14**	**28**	**42**	**56**	**70**	**84**	**98**	**112**	**126**

P_{sleep} (watts) →

(d) Implementation: SoftReactive

Figure 10. $NPPW$ for the Slowly Varying trace obtained via simulation for (a) Reactive and (b) SoftReactive, and via implementation for (c) Reactive and (d) SoftReactive. We see that simulation and implementation match well for almost all sleep states.

Reactive with C-states, is always worse than AlwaysOn, with $NPPW$ ranging from 0.69 – 0.94 for the traces we use. Note that $t^*_{wait} = 0s$ for SoftReactive with C-states since $T_{setup} = 0s$.

VI. Effect of Scale

Thus far, we have only looked at the effectiveness of sleep states on an implementation testbed with 14 front-end servers. In this section, we examine the effectiveness of sleep states for larger data centers. Since this is beyond the scope our implementation testbed, we resort to simulations.

A. *Simulation details*

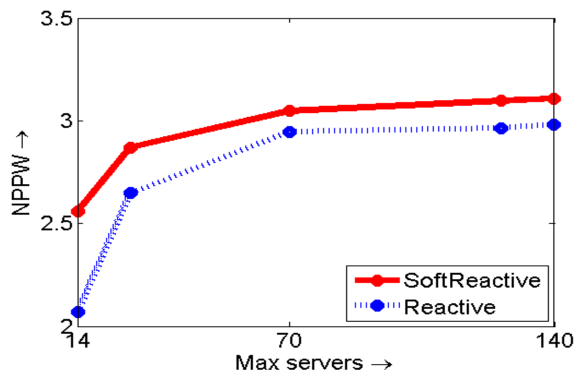
Our discrete event simulator, written in about 3000 lines of C++, models a data center as described in Section II-A. In particular, our simulator comprises: (i) a load generator, which can replay arrival traces, (ii) a load balancer, which dispatches arrivals to front-end servers, and is also responsible for suspending servers and waking them up, and (iii) 8-core front-end servers, that can either be busy, idle, sleeping, or in setup mode. We model each request as a task that requires a certain number of CPU cycles. To model the variability in requests, we use exponentially-distributed task sizes, with a mean of 123ms, matching the implementation. The simulator carefully tracks every arriving request and calculates the response time for each request based on its task size and the CPU contention at the servers.

Using the simulator, we replay the exact same traces we use for implementation (see Table I); we also use the same range of P_{sleep} (0W to 126W) and T_{setup} (20s to 200s). During the setup time, the servers consume $P_{max} = 200W$, as in our implementation. We vary the data center size from 14 servers to 1,400 servers. Since our memcached servers are always on, and are thus not a bottleneck, we ignore these servers in our simulation. We have implemented the AlwaysOn, Reactive and SoftReactive policies in our simulator based on their descriptions in the previous sections.

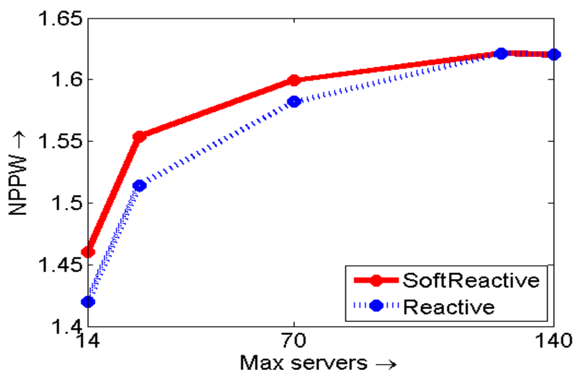
While we don't expect simulations to match implementation perfectly, we believe that our simulator can broadly identify regimes where Reactive or SoftReactive is superior to AlwaysOn, that is, regimes where $NPPW > 1$. Figures 10(a) and 10(b) show our simulation results for the Slowly Varying trace under Reactive and SoftReactive respectively. We see that our simulation results agree with our implementation results with respect to $NPPW$ in Figures 10(c) and 10(d). The above observations indicate that our simulator can be a useful tool for predicting the regimes where sleep states are effective.

B. *Evaluation*

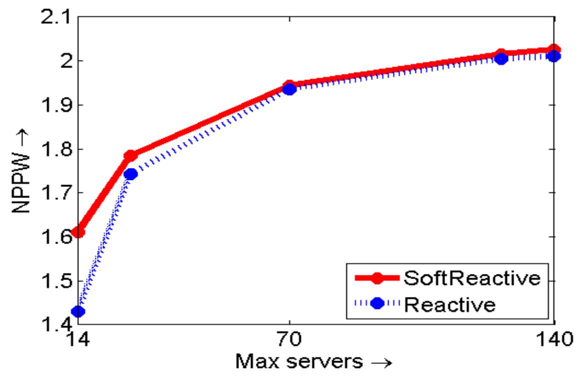
We now use our simulator to examine the effect of data center size on the effectiveness of sleep states for both Reactive and SoftReactive. Figure 11 shows our simulation results for all four traces under our best sleep state, $(P_{sleep} = 0W, T_{setup} = 20s)$. In our simulations, we vary the data center size from 14 servers to 140 servers, and scale the request rate for each trace proportionately, so as to



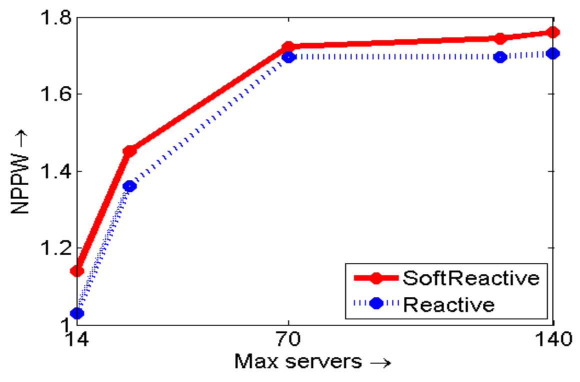
(a) Slowly Varying trace



(b) Slowly Varying with Big Spike trace



(c) Dual Phase trace



(d) Dual Phase with Huge Variations trace

Figure 11. Effect of scaling on $NPPW$ under our best sleep state: $(P_{sleep} = 0W, T_{setup} = 20s)$. Note that the range of y-values ($NPPW$) varies across the traces.

maintain the same load per server.

We see that **scaling up the number of servers increases $NPPW$, making the Reactive and SoftReactive policies more desirable as compared to AlwaysOn**. In particular, $NPPW$ increases by about 50%, on average (across all traces), as the data center size increases from 14 servers to 140 servers, for both Reactive and SoftReactive. The reason for this is that as the size of the data center goes up, the probability that all cores are simultaneously busy goes down. Thus, an incoming request has higher chances of finding an idle core, thereby lowering T_{95} and increasing PPW for Reactive and SoftReactive. By contrast, for the over-provisioned AlwaysOn, T_{95} is always good whereas P_{avg} is always high, regardless of the data center size. Thus, the PPW under AlwaysOn does not change much with the data center size. The net effect is an increase in $NPPW$ for Reactive and SoftReactive.

The improvement in $NPPW$ under Reactive and SoftReactive as the data center size increases cannot go on forever. The reason is that there is a natural lower bound on T_{95}, namely the T_{95} provided by AlwaysOn. Once we reach this lower bound on T_{95}, $NPPW$ cannot improve further. Our simulations indicate that for our traces, by the time we reach 140 servers, we have almost achieved this $NPPW$. We also ran simulations for 1,400 servers and found that $NPPW$ does not increase by more than 10% when compared to the $NPPW$ under 140 servers.

We also find that **the $NPPW$ for Reactive converges to that of SoftReactive as the size of the data center increases**. This is again because of the fact that there is a natural upper bound on $NPPW$.

VII. Conclusion

In this paper we examine the effectiveness of sleep states which are realistically feasible to implement in today's servers (with setup times ranging from 20s to 200s). Evaluation is done on a 24-server multi-tier testbed using real traces and a hundred different hypothetical sleep states. We find that for a large range of sleep states, a simple policy, Reactive, that reacts to current load by putting servers to sleep, can significantly increase Performance-per-Watt (PPW) over an optimistic static provisioning policy, AlwaysOn. In particular, for most traces, if the power used in sleep (P_{sleep}) is less than half the idle power (P_{idle}), Reactive can increase PPW by 10-100% over AlwaysOn. However, for very bursty traces, even the best sleep state with $P_{sleep} = 0W$ and $T_{setup} = 20s$ is ineffective.

We next introduce a sophisticated dynamic power management policy, SoftReactive, that builds on Reactive by being more conservative when scaling down capacity. We find that by simply taking more care in putting servers to sleep, we can harness the full potential of sleep states. In particular, under SoftReactive, there is a set of sleep states for which SoftReactive is superior to AlwaysOn ($NPPW > 1$) for all traces. This set of sleep states includes at least all sleep states with $T_{setup} \leq 60s$ and $P_{sleep} \leq 28W$.

Finally, we examine the effect of data center size on the effectiveness of sleep states via simulation. We find that the effectiveness of sleep states for large data centers is even more pronounced. In particular, we find that $NPPW$ under Reactive and SoftReactive increases by about 50% as the data center size grows from 14 servers to 140 servers. Further, we find that the $NPPW$ under Reactive converges to that under SoftReactive. Thus, larger data centers can exploit the benefits of sleep states even with a simple dynamic power management policy.

REFERENCES

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

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

[3] Yuvraj Agarwal, Stefan Savage, and Rajesh Gupta. Sleepserver: A software-only approach for reducing the energy consumption of pcs within enterprise environments. In *Proceedings of USENIX ATC 2010*, pages 285–299.

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

[5] L. A. Barroso and U. Holzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[6] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, and Amin M. Vahdat. Managing energy and server resources in hosting centers. In *Proceedings of SOSP 2001*, pages 103–116.

[7] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *Proceedings of NSDI 2008*, pages 337–350.

[8] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of SOSP 2007*, pages 205–220.

[9] V. Devadas and H. Aydin. On the Interplay of Voltage/Frequency Scaling and Device Power Management for Frame-based Real-Time Embedded Applications. *IEEE Transactions on Computers*, 61(1):31–44, January 2012.

[10] A. Gandhi, Y. Chen, D. Gmach, M. Arlitt, and M. Marwah. Minimizing data center sla violations and power consumption via hybrid resource provisioning. In *Proceedings of IEEE IGCC 2011*, pages 49–56.

[11] A. Gandhi, M. Harchol-Balter, and M. Kozuch. The case for sleep states in servers. In *Proceedings of HotPower 2011*.

[12] Green Grid. Unused Servers Survey Results Analysis. http://www.thegreengrid.org/en/Global/Content/white-papers/UnusedServersSurveyResultsAnalysis, 2010.

[13] Tibor Horvath and Kevin Skadron. Multi-mode energy management for multi-tier server clusters. In *Proceedings of PACT 2008*, pages 270–279.

[14] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of SOSP 2001*, pages 117–130.

[15] J. Kim and T. S. Rosing. Power-aware resource management techniques for low-power embedded systems. In S. H. Son, I. Lee, and J. Y-T Leung, editors, *Handbook of Real-Time and Embedded Systems*. Taylor-Francis Group LLC, 2006.

[16] Andrew Krioukov, Prashanth Mohan, Sara Alspaugh, Laura Keys, David Culler, and Randy Katz. Napsac: Design and implementation of a power-proportional web cluster. In *Proceedings of Workshop on Green Networking 2010*, pages 15–22.

[17] Seung-Hwan Lim, Bikash Sharma, Byung Chul Tak, and Chita R. Das. A dynamic energy management in multi-tier data centers. In *Proceedings of IEEE ISPASS 2011*, pages 257–266.

[18] Yung-Hsiang Lu, Eui-Young Chung, T. Simunic, T. Benini, and G. de Micheli. Quantitative comparison of power management algorithms. In *Proceedings of DATE 2000*, pages 20–26.

[19] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: eliminating server idle power. In *Proceedings of ASPLOS 2009*, pages 205–216.

[20] 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 ISCA 2011*, pages 319–330.

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

[22] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, 46:323–351, December 2005.

[23] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley db. In *Proceedings of USENIX ATC 1999*, pages 183–192.

[24] Parthasarathy Ranganathan. Recipe for efficiency: principles of power-aware computing. *Commun. ACM*, 53:60–67, April 2010.

[25] Joshua Reich, Aman Kansal, Michel Gorackzo, and Jitendra Padhye. Sleepless in seattle no longer. In *Proceedings of USENIX ATC 2010*, pages 315–328.

[26] Peng Rong and M. Pedram. Hierarchical power management with application to scheduling. In *Proceedings of ISLPED 2005*, pages 269–274.

[27] Bill Snyder. Server virtualization has stalled, despite the hype. http://www.infoworld.com/print/146901, December 2010.

[28] Etienne Le Sueur and Gernot Heiser. Slow down or sleep, that is the question. In *Proceedings of USENIX ATC 2011*, pages 217–222.

[29] Bhuvan Urgaonkar and Abhishek Chandra. Dynamic provisioning of multi-tier internet applications. In *Proceedings of ICAC 2005*, pages 217–228.

[30] Akshat Verma, Gargi Dasgupta, Tapan Kumar Nayak, Pradipta De, and Ravi Kothari. Server workload analysis for power minimization using consolidation. In *Proceedings of USENIX ATC 2009*, pages 355–368.

[31] M. Ware, K. Rajamani, M. Floyd, B. Brock, J.C. Rubio, F. Rawson, and J.B. Carter. Architecting for power management: The IBM POWER7 approach. In *Proceedings of HPCA 2010*, pages 1–11.