CARB: A C-State Power Management Arbiter For Latency-Critical Workloads

Xin Zhan[†], Reza Azimi[†], Svilen Kanev[§], David Brooks[§], Sherief Reda[†]
[†]Brown University, Providence, RI [§]Harvard University, Cambridge, MA

Abstract—Latency-critical workloads in datacenters have tight response time requirements to meet service-level agreements (SLAs). Sleep states (c-states) enable servers to reduce their power consumption during idle times; however entering and exiting c-states is not instantaneous, leading to increased transaction latency. In this paper we propose a c-state arbitration technique, CARB, that minimizes response time, while simultaneously realizing the power savings that could be achieved from enabling c-states. CARB adapts to incoming request rates and processing times and activates the smallest number of cores for processing the current load. CARB reshapes the distribution of c-states and minimizes the latency cost of sleep by avoiding going into deep sleeps too often. We quantify the improvements from CARB with `memcached` running on an 8-core Haswell-based server.

Index Terms—Latency-critical workloads, energy-efficient, c-state, feedback controller, workload consolidation

1 Introduction

Minimizing tail request latency is a dominant optimization target in datacenters because whole groups of requests are often held behind by the slowest one. In an application service tree, the negative effects of a single slow request can easily get amplified severalfold when moving closer to the root. For example, Dean and Barroso show a latency degradation of 10×, when measured at the root of the tree, as opposed to at an individual node [1]. Such performance irregularities lead to violations of service-level agreements (SLAs) and low levels of utilization in datacenters [2].

Processor idleness, especially at mid- and low-utilization points, interferes with request tail latencies [3], [4]. The *latency cost of sleep* is the result of a request arriving while a processor core is in a sleep state (c-state), and having to pay additional latency for the transition to active mode (C0) before being processed. Deeper sleep states lead to larger latency transitions, which further exacerbates the problem at low server utilizations. Given that servers spend most of their time at low utilizations [2], c-states lead to a dilemma as enabling them saves power but increases response time.

In this paper, we observe that there is an optimal number of active cores that minimizes tail latencies, and that any larger number of active cores beyond the optimal simultaneously worsens performance and power. This optimal number is a function of the request rate and the application. Based on this observation, we propose a c-state arbitration technique, CARB, which unobtrusively monitors request latencies for the target workload and optimally adjusts the number of active cores to minimize response time and power. CARB reshapes the distribution of c-states and minimizes the latency cost of sleep by avoiding going into deep sleeps too often. We demonstrate CARB on an 8-core Haswell-based server using `memcached` [5]. Compared to a traditional scheduler that spreads the load across as many cores as possible, CARB effectively reduces tail latencies by up to 51% and reduces system power by 6%.

The rest of this paper is organized as follows. We quantify the effects of idleness on request latencies for latency-sensitive datacenter applications in Section 2. We then describe how CARB consolidates request processing on as few cores as necessary in Section 3. Finally, we demonstrate its effectiveness on an Intel Haswell-based server in Section 4.

2 Motivation & Overview

Power savings states, i.e., c-states, enable processors to save power consumption during idle periods where no instructions are available to execute. New processors offer deeper c-states for more power savings during idle periods. For example, Intel's Haswell architecture offers the following 5 c-states (e.g., C1, C1E, C3, C6 and C8) [6]. While c-states enable processors to achieve power savings, entry to and exit from a c-state by a core incurs a latency overhead during which the core cannot be utilized. For example, it is estimated that the C3 and C6 states require, respectively, 80 μs and 104 μs [3]. These entry-exit latencies can have significant performance effects on workloads whose request processing latencies are of similar magnitude.

We illustrate the negative performance effects of deep sleep on our 8-core Haswell-based Xeon server in Figure 1. We report the 95th percentile response time and average power consumption for the `memcached` application as a function of the number of requests per second (RPS). The plots show that enabling c-states introduces a latency overhead that is a function of RPS, but it reduces power consumption. For instance, at low RPS values (e.g., 10K), the increase in the 95th response time is up to 2×, but the power savings are about 20%. As RPS increases, there are naturally fewer opportunities for cores to go idle, and as a result the overhead of c-states diminishes. Figure 2 provides

Manuscript submitted: 26-Oct-2015. Manuscript accepted: 12-Jan-2016. Final manuscript received: 22-Jan-2016.

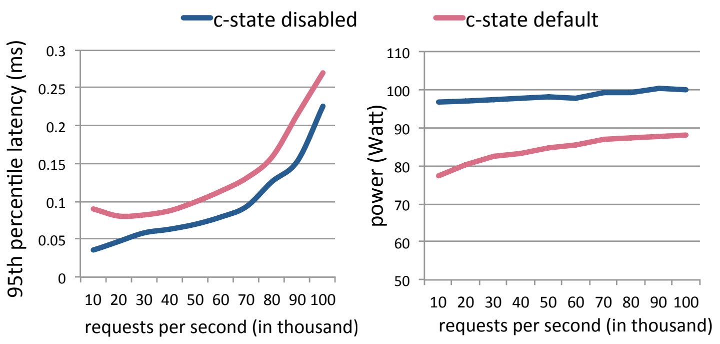

Fig. 1. Impact of enabling versus disabling c-states on 95-th percentile latency and power consumption for various RPS.

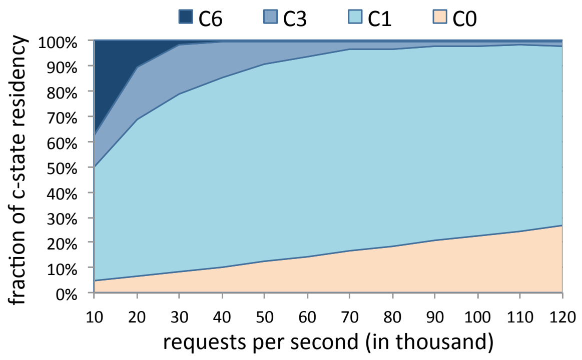

Fig. 2. Fraction of time spent by the entire processor at various c-states.

the fraction of time spent by the entire processor (averaged over 8 cores) in various c-states. The plot shows that at low RPS values, idleness periods are long enough to induce deep sleep states with larger delay penalties.

One way to mitigate this increase in latency is to use fewer cores. We observe the relationship between the number of active cores and latencies for `memcached` in Figure 3, where we plot the measured 95th response time as a function of the number of active cores at RPS=25K, 50K, and 75K. The plot for each RPS value has a clear minimum where performance is optimal. To the left of the minimum, the number of active cores is not sufficient to handle the load, and latency dramatically increases due to queueing. More interestingly, to the right of the minimum, performance is also worse due to the c-state latency effect identified earlier. At the optimal point, the entry-exit overheads are minimal because the busy cores have the minimum amount of idle time that allows them to handle the incoming load.

Based on these observations, we propose a *c-state arbiter* which arbitrates the number of active cores in search for this optimum. Such behavior is in contrast with traditional OS fairness policies, which aim to spread load across all cores, and closer to the goals of packing schedulers [7]. Packing cores, or limiting an application's core allocation, has been well-studied, most frequently in a multi-application scenario with the goal of workload isolation [8], i.e. not falling off the "performance cliff" shown to the left on Figure 3. On the contrary, this paper demonstrates that too many cores can also be detrimental to performance even in the single-application case. While previously such effects have been attributed to cache sharing [9] or I/O interrupt scheduling [4], we add deep sleep as a reason to prefer packing. C-state management is highly relevant to applications that are latency-sensitive and that lead to frequent sleeps, where the sleep overhead is comparable to the request latency [3].

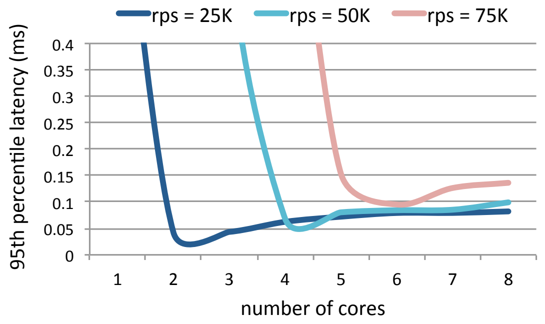

Fig. 3. 95th percentile response time as a function of number of arbitrated active cores for RPS=25K, 50K and 75K.

3 CARB DESIGN

CARB is a feedback-based controller that arbitrates the minimum number of sufficient cores for a given request rate. CARB collects the real time request rate $r(k)$ and response time $y(k)$ (time is discrete and denoted by k) as control inputs and arbitrates the number of active cores.

At each control epoch, CARB adjusts the number of active cores $c(k) \in [c_{\min}, c_{\max}]$ towards the optimal. CARB has three working states: 1) *idle state* **S0**, where it measures the request rate $r(k)$ and the response time $y(k)$ and determines the next state $s(k+1)$; 2) *scaling up state* **S1**, where it increases the number of active cores by a step size $\Delta(k)$ until the response time cannot be further improved, then switches back to **S0**; and 3) *scaling down state* **S2**, where it decreases the number of active cores by $\Delta(k)$ until the response time cannot be further improved, then switches back to **S0**. In more detail, when the controller resides in **S0**, the state transitions and control logic are given in Algorithm 1.

Algorithm 1 Control logic at **S0**

```
1: if $r(k) > r(k-1) + \delta_r$ then
2:     $c(k) \leftarrow c(k-1) + \Delta(k)$; $s(k+1) \leftarrow$ S1
3: else if $r(k) < r(k-1) - \delta_r$ then
4:     $c(k) \leftarrow c(k-1) - \Delta(k)$; $s(k+1) \leftarrow$ S2
5: else if $y(k) > y(k-1) + \delta_y$ then
6:     $c(k) \leftarrow c(k-1) + \Delta(k)$; $s(k+1) \leftarrow$ S1
7: else
8:     $c(k) \leftarrow c(k-1)$ ; $s(k+1) \leftarrow$ S0
```

δ_r and δ_y are sensitivity thresholds to filter out the noise in request rate and response time so that unnecessary oscillation can be avoided, and are determined empirically.

At states **S1** and **S2**, CARB scales the number of active cores (up for **S1** and down for **S2**) towards the optimal as given in Algorithms 2 and 3.

Algorithm 2 Control logic at **S1**

```
1: if $y(k) < y(k-1) + \delta_y$ then
2:     $c(k) \leftarrow c(k-1) + \Delta(k)$; $s(k+1) \leftarrow$ S1
3: else
4:     $c(k) \leftarrow c(k-1) - \Delta(k)$; $s(k+1) \leftarrow$ S0
```

Algorithm 3 Control logic at **S2**

```
1: if $y(k) < y(k-1) + \delta_y$ then
2:     $c(k) \leftarrow c(k-1) - \Delta(k)$; $s(k+1) \leftarrow$ S2
3: else
4:     $c(k) \leftarrow c(k-1) + \Delta(k)$; $s(k+1) \leftarrow$ S0
```

At initialization, we set $k = 0$, $c(0) = c_{\max}$, and $s(0) = \mathbf{S2}$ while measuring $r(0)$ and $y(0)$. In all steps, CARB checks that $\Delta(k)$ leads to a $c(k) \in [c_{min}, c_{max}]$ before each change inside the loop. It is crucial to identify the most effective step size $\Delta(k)$, particularly when CARB is operating on the *left side* to the optimal on the curve in Figure 3. To ensure the SLA will not be violated, CARB should move out of the *left side* within the minimum number of steps. A constant $\Delta(k)$ can be set based on user preferences and might be chosen differently for scaling up and scaling down. To address the situation of potential bursts in request load, which requires scaling up capacity rapidly, CARB sets the number of cores to the maximum when the request rate $r(k)$ increases beyond a threshold r_{th}, then attempts to scale down cores afterwards.

We also examined other controllers based on proportional-integral-derivative (PID) controllers and gradient descent methods. PID controllers require analytical models for the output to identify their optimal parameters, which is quite challenging in our system due to the variations in the response time from queueing effects. On the other hand, CARB does not require an analytical objective function. Similarly, optimal control methods (e.g., gradient descent or Newton's method) require a differentiable objective function. We have found that noise arising from measurements and queueing effects lead to erroneous gradient calculations, which make these methods relatively unstable. As our problem is a local one-dimensional unconstrained optimization problem, our bang-bang based CARB controller gives us good results.

4 Evaluation

4.1 Evaluation Methodology

Server: We evaluate CARB on an Intel Haswell-based server using a Xeon E5-2630 V3 8-core processor with 32GB of DDR4 memory and a 10 Gbe network controller. The server runs Ubuntu 14.04. We measure power consumption by sensing the external current at the 120 V AC socket with a sampling rate of 10 Hz. Hardware control of frequency (Intel TurboBoost) is enabled on the processor.

Workloads: To evaluate the effectiveness of CARB, we choose `memcached` [5], a memory object caching workload. The `data caching` benchmark from CloudSuite [10] is used to generate request load and to collect end-to-end delay statistics.

Request load trace: Since real load traces of a data caching cluster are rarely available for access, we use a synthetic trace. This way, we can control the range and the frequency of the fluctuation of the requests. A time series trace can be generated using: $r(k+1) = \sum_{i=0}^{m-1} \omega(i) r(k-i) + \Phi\alpha(k)$, where $r(k)$ is the request load at time k, ω is a vector defining the weights on the last m samples; Φ is a parameter that describes how much the request load will fluctuate between two consecutive elements in the series; and $\alpha(k)$ is a random number drawn from a normal distribution.

Implementation: CARB is implemented using Python. The number of active cores can be changed either by setting core affinity (`cpuset`), or by taking cores away from the OS. In either case, inactive cores go the deepest sleep state and the application needs no changes. CARB only needs the ability to monitor request rate and response time of the application. The request rate can be measured from the network socket and the response time can be observed from the server by timing the request service time. Both of them can be measured without modifying the target application. Although in our case `data caching` has the interface to monitor the request rate and response time.

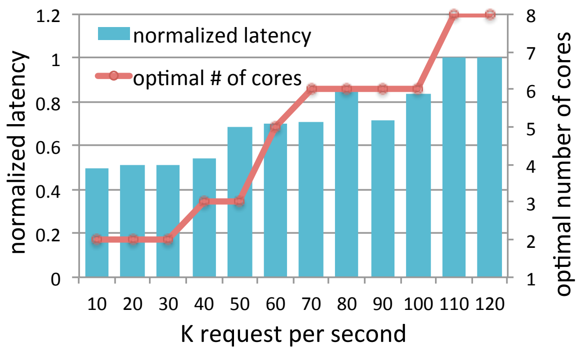

Fig. 4. The normalized 95th latency with the optimal number of cores.

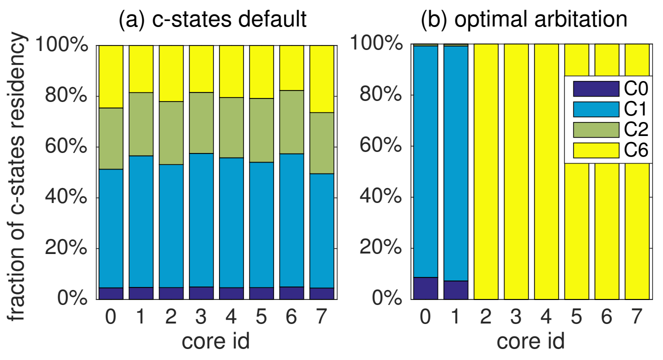

Fig. 5. Fraction of time spent by each core in various c-states under various arbitration for RPS=10K. Subfigure (a) gives the default case when all cores are active; Subfigure (b) gives the case when 2 active cores are arbitrated.

4.2 Experimental results

Static results. We first demonstrate the optimal number of cores and the response time difference between using all cores and the optimal number of cores. We vary the request rate from 10 K to 120 K with a step of 10 K and measure the 95th percentile of the response time when all 8 cores are enabled and when the optimal number of cores are enabled using CARB. Results normalized to the response time of 8 cores, together with the corresponding number of optimal cores, are given in Figure 4. By consolidating the requests onto a subset of cores, response time can be reduced by up to 51%. In order to better demonstrate how CARB works, Figure 5 plots the change in c-states distribution when the request rate is 10 K, using 8 cores and optimal number of cores (two in this case). The optimal number of active cores is usually less than eight when the request rate is less than 100 K. We observe that the optimal number of cores has to be larger than one, i.e., $c_{\min} = 2$. One explanation for this is that, with only one core available, all system and background processes are scheduled together and interfere with the `memcached` process. Thus, in our dynamic experiments, we set the lower bound of scaling down cores as two cores.

Dynamic results. In this experiment we evaluate CARB with 90-minute synthetic request traces. We set the step size $\Delta(k)$ to 1 for both scaling up and scaling down states. The threshold parameter, r_{th}, is chosen as 20 KRPS. The sensitivity parameter δ_r is set as 5 KRPS and δ_y is chosen as 10% of the average response time. Figure 6 shows results with a slow varying trace, where we plot the request rate, the 95th percentile of the response time, power, and the number of active cores over time for three cases: (1) default c-state management, (2) disabling c-states, and (3) CARB. The response time using CARB is almost half that of the default c-state management when the request load is very low and overall 26% lower, while consuming 6.1% less power. Compared to disabling c-states, CARB reduces power by 23% while offering similar response times. The results are summarized in Figure 8. Thus, CARB delivers response times close to the case with c-states disabled and consumes less power than the default c-state governor.

We repeat the same test with a fast varying request rate trace. The corresponding results are given in Figure 7 and Figure 8. The results show that the response time of CARB closely follows the case with disabled c-states while request load is low. After the load spikes at 28 mins and 57 mins, to be conservative, CARB scales up to the maximum number of cores and then searches down for the optimal. Overall, CARB reduces response time by 25% over the c-state default with 5% power savings.

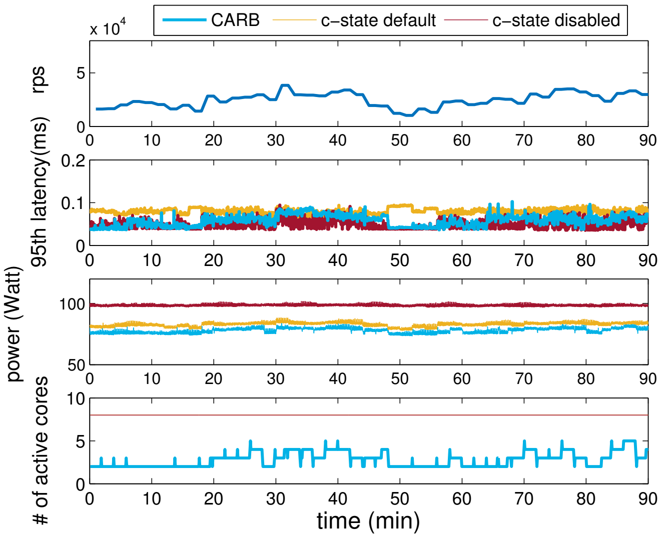

Fig. 6. Dynamic results of `memcached` with slow varying request trace.

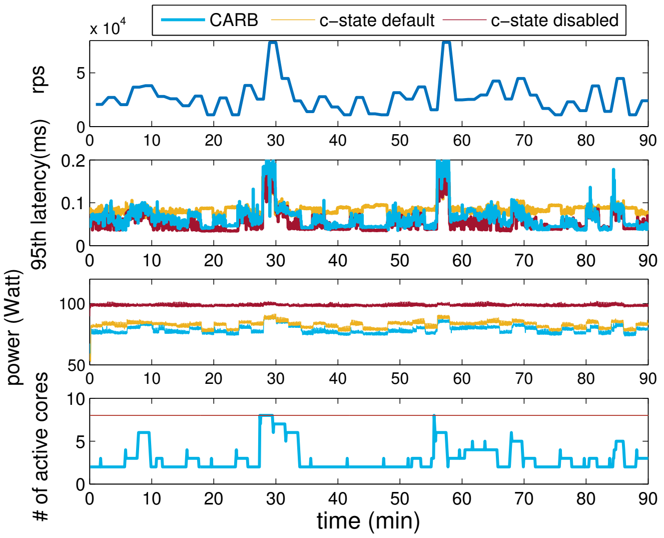

Fig. 7. Dynamic results of `memcached` with fast varying request trace.

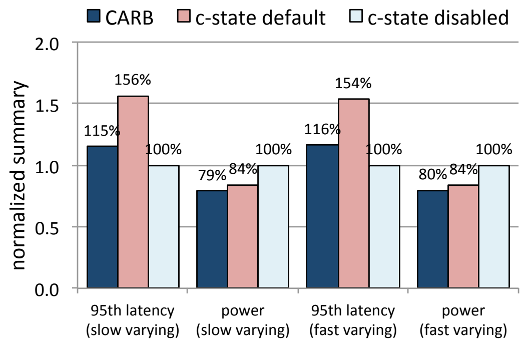

Fig. 8. Summary of dynamic experiments of `memcached`.

5 Conclusions

For latency-critical workloads with sub-millisecond response times, c-state transitions constitute a good portion of overall latency, especially when the request load is relatively low. In this case, consolidating the load on a subset of cores improves both latency and energy efficiency. We devised a controller, CARB, which arbitrates the core allocation of `memcached`, and manages to find the minimum number of cores to optimize latency and power. In addition to `memcached`, we believe that CARB is particularly attractive for latency-sensitive workloads, where the overhead of sleep state transitions is comparable to the response time.

Acknowledgments: We thank the anonymous reviewers for their comments. The research of X. Zhan, R. Azimi and S. Reda is supported by NSF grants 1305148 and 1438958.

References

[1] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, 2013.

[2] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: an introduction to the design of warehouse-scale machines," *Synthesis Lectures on Computer Architecture*, 2013.

[3] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks, "Tradeoffs between power management and tail latency in warehouse-scale applications," in *Workload Characterization (IISWC)*, 2014.

[4] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Cloud Computing (SoCC)*, 2014.

[5] "Memcached: A distributed memory cache system," 2012.

[6] N. Kurd, M. Chowdhury, E. Burton, T. Thomas, C. Mozak, B. Boswell, P. Mosalikanti, M. Neidengard, A. Deval, A. Khanna, N. Chowdhury, R. Rajwar, T. Wilson, and R. Kumar, "Haswell: A family of ia 22 nm processors," *Solid-State Circuits, IEEE Journal of*, vol. 50, pp. 49–58, Jan 2015.

[7] D. G. Feitelson, "Packing schemes for gang scheduling," in *Job Scheduling Strategies for Parallel Processing*, Springer, 1996.

[8] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pp. 450–462, 2015.

[9] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors," in *ACM SIGOPS Operating Systems Review*, 2007.

[10] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'12, pp. 37–48, 2012.