# TimeThief: Leveraging Network Variability to Save Datacenter Energy in On-line Data-Intensive Applications

Balajee Vamanan
Purdue University
bvamanan@ecn.purdue.edu

Hamza Bin Sohail
Purdue University
hsohail@ecn.purdue.edu

Jahangir Hasan
Google Inc.
jahangir@google.com

T. N. Vijaykumar
Purdue University
vijay@ecn.purdue.edu

## Abstract

Datacenters running on-line, data-intensive applications (OLDIs) consume significant amounts of energy. However, reducing their energy is challenging due to their tight response time requirements. A key aspect of OLDIs is that each user query goes to all or many of the nodes in the cluster, so that the overall time budget is dictated by the tail of the replies' latency distribution. Previous work proposes to achieve load-proportional energy by slowing down the computation at lower datacenter loads based directly on response times (i.e., at lower loads, the proposal exploits the average slack in the time budget provisioned for the peak load). In contrast, we propose TimeThief to reduce energy by exploiting the latency slack in the sub-critical replies which arrive before the deadline (e.g., 80% of replies are 3-4x faster than the tail). This slack is present at all loads. While the previous work shifts the leaves' response time distribution to consume the slack at lower loads, TimeThief reshapes the distribution at all loads by slowing down individual sub-critical nodes without increasing missed deadlines. Specifically, TimeThief exploits the slack in the network budget. Further, TimeThief leverages Earliest Deadline First scheduling to largely decouple critical requests from the queuing delays of sub-critical requests which can then be slowed down without hurting critical requests. Using at-scale simulations, we show that without adding to missed deadlines, TimeThief saves 12% and 20% energy at 90% and 30% loading, respectively, in a datacenter with 512 nodes.

## 1 Introduction

Datacenters host many of modern Internet services today such as Web Search, social networking, e-commerce, and cloud computing. Datacenters consume tens of megawatts of electric power [8], which accounts for millions of dollars in annual operating costs [30]. Of their total power, modern datacenters spend about 10% on cooling and power distribution overheads (their Power Usage Effectiveness is 1.12 [15]) and about 5% on networking equipment, leaving about 85% for servers of which memory and disk take up 45% and processors consume 55% (i.e., 47% of total) [8, 15, 23]. TimeThief focuses on the substantial processor power.

Many of Internet services are provided by on-line, data-intensive applications (OLDIs) which often process vast amounts of Internet data (e.g., Web Search and Key-Value stores) [25]. Such services typically operate under tight response time budgets set by service-level agreements (SLAs) (e.g., 200 ms for a Web Search query) [16]. Processing of a query often involves hundreds or thousands of servers working in parallel on memory-resident data [7, 11]. OLDIs have two distinguishing characteristics. (1) They employ a multi-level tree-like software architecture where each query goes to *all or many* leaves. Consequently, though only a few leaves' replies are slow, the overall SLA budget is dictated by the tail of the leaves' reply latency distribution [11] (e.g., the 99.9$^{th}$ percentile leaf latency in a 1000-leaf tree). Replies arriving after the deadline are dropped for responsiveness. (2) The network contributes to significant variability in the latency of the leaves' replies, as we explain in Section 2 (e.g., a request or reply takes 2-30 ms in the network [5, 37, 38]). Network variations occur at all datacenter loads though the spread is greater at higher loads.

Using low-power or sleep modes is a common approach to saving energy. Unfortunately, OLDIs' time budgets and inter-arrival times are too short for the transition latencies of low-power modes [24, 25]. As such, the low-power modes would incur many deadline violations [23]. Alternately, an insightful recent work, called Pegasus [23], achieves load-proportional energy by slowing down the leaf computation at lower datacenter loads while carefully ensuring that SLAs are not violated (e.g., at night times [25]). Pegasus exploits the mean slack at lower loads in the time budget provisioned for the peak load.

In contrast, we propose *TimeThief* to reduce energy by exploiting sub-critical leaves' latency slack (e.g., 80% of leaves in *every* query complete within a 3$^{rd}$-4$^{th}$ of the budget.). This slack is present at all loads (modern datacenters operate at high loads during the day [25]). Pegasus exploits the mean load-related slack, common to all leaves at lower loads, to *shift* the response time distribution. Instead, TimeThief *reshapes* the response time distribution at all loads by slowing down individual sub-critical leaves so that they are closer to, but within, the deadline than the default distribution. Specifically, TimeThief exploits the slack in the *network budget*. TimeThief achieves significant savings even at the peak load, which occurs often and where Pegasus has no opportunity. Thus, TimeThief converts the performance disadvantage of latency tails [11] into an energy advantage.

TimeThief employs two ideas. First, TimeThief trades time across system layers, borrowing from the network layer and lending to the compute layer. Each query results in a request-compute-reply-aggregate sequence where the requests from parents to the leaves and replies from the leaves to their parents see variability in the network. OLDIs break up the total time budget into a component each for request, compute,

reply, and aggregate. We make the *key* observation that because request comes before compute, the slack in faster requests can be transferred to their corresponding compute without any prediction or risk of missing the deadline. Unlike request, unfortunately, reply comes after compute and reply latency is unpredictable due to the highly-timing-dependent nature of network latencies (Section 2). Therefore, the slack in faster replies cannot be transferred easily to their compute. As such, TimeThief exploits the request but not the reply slack.

Second, despite the slack, such slowing down is challenging in the presence of long tails and SLA guarantees. Even though a sub-critical request has slack, slowing it down may hurt another, critical request that is queued behind the sub-critical request. To address this issue, we leverage the well-known idea of Earliest Deadline First (EDF) scheduling [22] to decouple critical requests from the queuing delays of sub-critical requests by placing the former ahead of the latter in the leaf servers' queues. Conventional implementations and Pegasus cannot exploit EDF because they do not distinguish between critical and sub-critical requests. Due to its decoupling, EDF pulls in the tail and reshapes the leaves' response time distribution (without improving the mean), enabling TimeThief to use the *per-leaf* slack to shift further the distribution closer to the deadline than with network slack alone. Though this shift lengthens the mean service time, such an increase does not worsen throughput. Because OLDIs' response times are sensitive to tail latencies, compute-queuing delays are kept low even at high loads via high throughput-parallelism (i.e., there is compute-throughput slack even at high loads). As such, TimeThief's longer service times tap into this throughput slack without causing loss of throughput.

Finally, TimeThief employs two key mechanisms to realize the above ideas. Transferring the request slack from the network to the compute is challenging due to lack of fine-grained (sub-ms) synchronization between a parent and the leaves. To address this issue, we leverage the well-known Explicit Congestion Notification (ECN) in IP [32] and TCP timeouts to inform the leaves whether a request encountered timeout or congestion in the network and hence does not have slack. Further, because the slack lengths are tens of milliseconds, we use power management schemes with response times of 1 ms, similar to Pegasus (e.g., Running Average Power Limit (RAPL) [1]).

In summary, the paper's contributions are:

- TimeThief reshapes the response time distribution at all loads by slowing down individual sub-critical leaves without increasing SLA violations;
- TimeThief exploits the request (network) slack on a per-leaf, per-query basis;
- TimeThief leverages EDF to largely decouple critical requests from the slowing down of sub-critical requests; and
- TimeThief leverages (a) network signals such as TCP timeouts and ECN to circumvent the lack of fine-grained synchronization between parent and leaves and (b) modern, low-latency power management to fit within OLDI timescales.

Using at-scale simulations, we show that without adding to missed deadlines TimeThief saves about 12% and 20% energy at 90% and 30% loading, respectively, in a datacenter with 512 nodes.

## 2 Background

As discussed, OLDIs typically employ a tree-based software architecture where the data to be queried resides in the leaf nodes' memory for fast access [7, 11] (see Figure 1). For instance, in Web Search and Key-Value store, the search index and the key-value pairs are partitioned across the leaves in a well load-balanced manner (e.g., using good hashing). In Web Search, every query is broadcast to all the leaves whose results are aggregated based on some ranking scheme (e.g., Google's PageRank). Typical use of key-value stores involve looking up several keys, so that each top-level request generates lookups in several hundreds of leaves, as noted in [23] (e.g., a user's Facebook page typically comprises of several hundreds of objects).

Each query involves a request-compute-reply-aggregate sequence where the query generates requests to the leaves going through multiple levels in the tree (see Figure 1); each leaf looks up its memory to compute its result and sends a reply to its parent which often aggregates the replies from all the children and sends the aggregated result up the tree potentially involving aggregations on the way to the root which sends the overall response. The key point here is that each query needs to wait for the replies from either all the leaves (Web Search) or several hundreds of leaves (Key-value stores). Consequently, the overall response time of a query is affected by the slowest leaf so that the mean overall response time, and therefore the SLA budget, includes the $99^{th}$ - $99.9^{th}$ percentile leaf latency in a 1000-node cluster, known as the latency tail problem [11]. To maintain interactive user experience, the parents wait for replies only until the deadline and drop the replies that miss the deadline. Because the dropped replies affect response quality and revenue, OLDIs keep the fraction of missed deadlines low (e.g., 1%).

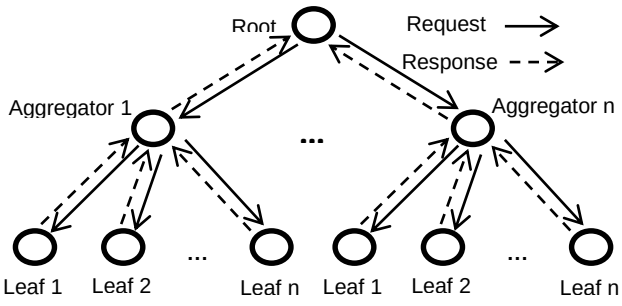


**Figure 1: OLDI software architecture**

There is a wide variation in the leaves' reply latency due to variations in network and compute; as noted before, this variation is among the sub-queries within a query, not across queries. Requests from parents to leaves (and responses) may take varying time due to collisions at the packet buffers with the leaves' replies for multiple queries. Due to the tree-like software architecture and mostly balanced workload among the leaves, the leaves send their replies to the parent at about the same time; this phenomenon is called in-cast [5, 37, 38]. Because all the replies are destined for the same input port of the same node (parent), the replies are queued in the same packer buffer at the relevant datacenter network switch. Because in-casts are inevitable, the switches are provisioned with enough buffering to handle a few in-casts. However, the buffers are kept shallow for cost and latency reasons [5]. Therefore, multiple queries' in-casts occurring at about the same time and colliding at the buffers result in delays and buffer overflows; multiple queries are processed in parallel for high throughput. Further, there are also background flows from other applications on the cluster due to consolidation or to updating the OLDI data (e.g., Web index). Such collisions cause TCP time-outs and re-transmits resulting in the replies falling in the tail or exceeding the time budget. While such collisions are uncommon in general, they are common enough to affect the $90^{th}$-$99.9^{th}$ percentile latencies (e.g., in *every* query, 80% of replies incur 5 ms latency whereas the last 1% incur 20 ms). Further, such collisions are highly timing-dependent and therefore are highly unpredictable; the TCP-flow propagation delay for a leaf to realize that a collision has occurred is too long for the leaf to delay or slow down its sending rates (hence reactive schemes are unlikely to work).

# 3 TimeThief

Recall from Section 1 that TimeThief exploits the network slack in requests. TimeThief slows down the individual, sub-critical leaves, to save energy without increasing SLA violations. To ensure that slowing down sub-critical requests does not hurt the critical requests that are queued behind the sub-critical requests, TimeThief employs Earliest Deadline First (EDF) scheduling [22] that prioritizes the critical requests ahead of the sub-critical requests.

## 3.1 Request slack

Requests that arrive before their budgeted deadlines have slack which TimeThief transfers to compute. Fortunately, because request comes before compute, this slack can be identified without prediction or the risk of missing the deadlines (recall from Section 2 that predicting network latencies is hard). However, requests originate at the parent node and compute occurs at a leaf, making it hard to accurately estimate the slack. Unfortunately, clock skew of several milliseconds between the parent and the leaf nearly rules out estimating slacks of similar magnitudes. Inter-node synchronization at such fine time granularity is hard [26, 28].

Instead of attempting to precisely determine the request slack, we use signals from the network about the presence or absence of packet drop and of imminent network congestion (typically due to an in-cast collision, as described in Section 2). Presence of these signals could mean no slack due to delays in the network whereas absence confirms some slack. While there may still be some slack even in the former case, we conservatively assume there is none. Because congestion is uncommon in datacenters that host OLDIs, our conservative assumption does not degrade our savings.

Determining the exact slack amount involves two cases: packet drop and imminent congestion. The former case results in retransmission which is marked by the sender (parent) with a packet header bit. The latter case of imminent congestion is signaled by Explicit Congestion Notification (ECN) [32]. Network switches detect imminent congestion when packet buffers are occupied above certain watermarks signifying queuing delays, and use ECN bits in packet headers to pass this information. Thus, the leaf can determine if there was packet drop and/or imminent congestion by looking at the packet header. If the entire request did not encounter packet drop or imminent congestion, we set the *request slack* to be *request budget – median network latency.* However, in the presence of either packet drop or imminent congestion, we conservatively assume zero slack.

However, this *request slack* has to be attenuated (i.e., scaled) before being applied as a slowdown to account for the fact that slower computation affects all the queued requests and not just the current request. One other subtle issue is that going to a lower power setting in CPUs requires choosing a slowdown factor. While we know the total slack amount, we do not know how long the current request will take and therefore, we cannot compute a slowdown factor. Fortunately, both these issues – attenuation and unknown service time – can be addressed by observing that the budget accounts for worst-case queuing delays and worst-case service times. Further, some slack is spent in RAPL latency. Therefore, we set

$$slowdown = (request\ slack - RAPL_{latency}) * scale / budget$$

where *scale* is a factor to further moderate the slowdown. Scale depends on both load and applications (i.e., service time distributions and budgets). Higher load implies lower value for *scale* to reduce the slowdown factor and impact on throughput. Instead of using statically configured *scale* values for each application, we employ a simple control algorithm that dynamically determines *scale.* . The algorithm monitors the difference between request+compute times of completed queries and the request+compute budget at each leaf server every 5 seconds. While the compute time is known for completed queries, the request time is not and therefore, we conservatively assume the full request budget or median network latency depending on ECN or timeout marks. If the difference is more than 5% of the budget, we increase *scale* by 0.05. Else, we reduce *scale* by 0.05 until there is room or the *scale* is 0. Thus, there is a guard band of 5% to avoid SLA violations. Even at the peak load, there is room to exploit. However, Pegasus cannot exploit this room because it does not distinguish critical requests from sub-critical requests, at the *same* leaf server. TimeThief saves energy even at the peak

load by slowing down sub-critical requests using a non-zero *scale* value without directly affecting critical requests that have 0 *total slack* (*scale* does not matter). *Further,* EDF shields critical requests from the queuing effects that arise from the slowing down of sub-critical requests. *Thus,* by using per-request slack and EDF, TimeThief saves energy at all load. We use scale values of 0.7, 0.4, and 0.2 for 30%, 60%, and 90% utilization respectively.

To set the core's speed as per the slowdown factor, we employ RAPL [1], which requires less than 1 ms, making it suitable for OLDI timescales. RAPL allows per-core power control (e.g., Intel's Enhanced Speed Step). One issue is that modern processors employ Simultaneous Multithreading (SMT) [36] where the slack for each SMT context may be different. We conservatively use the worst of the contexts' individual slowdown factors to avoid violating deadlines. Because the number of SMT contexts per core is only a few (e.g., 2-8), this conservative assumption does not diminish our opportunity. More SMT contexts may improve throughput but worsen single-thread latency which is key for OLDIs.

When we explored slowing down main memory in addition to the CPU, the fact that memory is shared among all the cores of a server severely limits the memory slowdown factor in the presence of such a conservative assumption. For instance, for a 32-core server, the memory slowdown factor would have to be the worst among all the 32 cores' factors, which would likely be zero. Therefore, we slow down only the cores and not memory. Nonetheless, because CPUs contributes about 60% of server power [8], our opportunity remains significant.

### 3.2 Deadline-based compute-queuing

Recall from Section 1 that the presence of slack is not sufficient to guarantee avoiding missing of the deadlines. Slowing down a sub-critical request which has slack may hurt another critical request that is queued behind the sub-critical request. To address this issue, we exploit Earliest Deadline First (EDF) scheduling that decouples critical requests from the queuing delays of sub-critical requests by placing the former ahead of the latter in the leaf server's queues.

The decoupling is not perfect due to the fact that arriving critical requests may still see elongated, residual service times of sub-critical requests in the absence of pre-emption (whose delays would not be suitable in our context of tight deadlines). Nevertheless, the decoupling enables EDF to pull in the tail and to reshape the leaves' response time distribution; the mean response time does not improve because as critical requests' response times get shorter the sub-critical requests' times get longer. However, EDF enables TimeThief to use per-leaf slack to slow down sub-critical requests, thereby *further* shifting the distribution closer to the deadline. Though such slow down lengthens the mean service time, such an increase taps into the throughput slack, and hence does not worsen throughput. Still, the throughput slack may not be enough to exploit the full total slack in which case we give up some energy savings to avoid throughput loss.

In our implementation, we timestamp the requests as they arrive at the leaf server and compute their deadlines before queuing them in a task queue implemented as a priority queue. Worker threads process the requests in the priority order.

## 4 Methodology

TimeThief involves two aspects: network latency and compute power. We use real-system measurements for compute power, and at-scale simulations for network latency. The compute aspects involve only one server because over long periods of time all servers are statistically identical in response times and power consumption and hence real-system measurements are feasible. Further, because tail effects are more pronounced in large clusters (e.g., 1000 node) to which we do not have access, we rely on simulations to study the network aspect.

**Benchmarks:** We simulate an OLDI benchmark, *Web Search (Search)*, from CloudSuite 2.0 We generate *Search*'s index from Wikipedia. In our runs, *Search* supports peak queries-per-second rates of 3000 using 100 threads per leaf server at 90% utilization (corresponding to a modern server with 4 sockets, 12 cores per-socket, and 2 SMT contexts per core). We use a parent-to-leaf fan-out of 32 (a standard value). For each query, we randomly choose a node to be the parent (Section 2). We set the budgets as: total 200 ms, request 25 ms, reply 25 ms, leaf compute 75 ms (*Web Search*), and aggregate and remaining network (aggregate-root communication) 75 ms. The network and compute budgets are the 99$^{th}$ percentile latencies achieved by, respectively, our network using D$^2$TCP and compute nodes at the peak load. We target less than 1% missed deadlines (i.e., these deadlines are tight and do not offer any "easy" opportunity for TimeThief). The network and compute budgets are in line with [5, 37, 38] and [34], respectively. TimeThief focuses on request, compute and reply for a total of 125 ms (*Web Search*) which is the deadline in our experiments. We use request sizes of 2 KB and reply sizes of 16-64 KB chosen uniformly randomly, and background flow sizes of 1 and 10 MB chosen uniformly randomly (Section 2); the total traffic is split evenly between OLDI and background flows. These message characteristics match publicly-available distributions from production OLDIs [9]. In all our experiments, the network utilization is 20% which is realistic for datacenters [5] (i.e., the network is over-provisioned and yet incurs in-cast collisions).

**Network latency:** Using *ns-3* [3], a widely-used simulator, we simulate a fat-tree topology which is typical of datacenter networks [4]. There are 64 racks with each rack having up to 16 servers (i.e., a 1000-server cluster). Each server connects to the top-of-rack (ToR) switch via a 10 Gbps link. Going up from the ToR level, there is a bandwidth over-subscription of 2x at each level, as is typical [4]. We sized the packet buffers in the ToR switches to match typical buffer sizes of shallow-buffered switches in real data centers (4MB) [5]. We set the

link latencies to 20 µs, achieving an average of round-trip time (RTT) of 200 µs, which is representative of datacenter network RTTs. To reduce the effects of in-cast collisions, we add a 1-ms jitter to each leaf's reply [14].

To simulate a deadline-aware TCP implementation that exploits the separate request-reply budgets (Section 2.3), we use D$^2$TCP [37] on top of ns-3's TCP New Reno protocol [2]. (Code obtained from D$^2$TCP's authors). All D$^2$TCP parameters (e.g., deadline imminence factor) match those in [37] and are available with the code. We set $RTO_{min}$ for all the protocols to be 20 ms. We use the same separate request-reply budgets and D$^2$TCP in both the baseline (no power management) as well as TimeThief. The latencies we observe closely match those reported in other papers, including production runs [37].

**All together:** In *ns-3*, we simulate TimeThief's EDF scheduling (Section 3.2) and compute the total slack as a function of the request. We compute the per-query slowdown factor based on the total slack. Using these slowdown factors and our power-latency measurements, we compute TimeThief's energy savings.

# 5 At-scale simulation results

Now we show our at-scale results. We start with comparing the energy savings of TimeThief over the baseline, the main result of the paper. We then show a binning of requests based on their CPU core's power state for TimeThief.

## 5.1 Energy savings

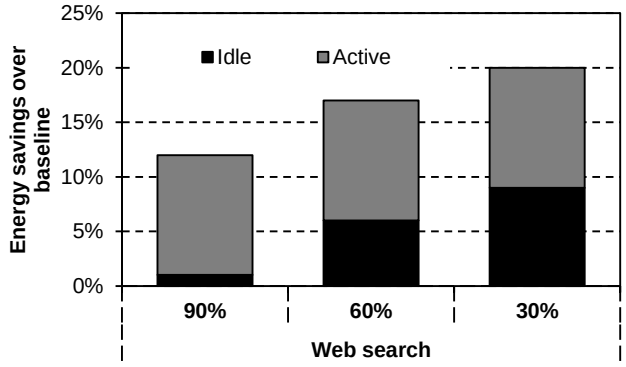


**Figure 2: At-scale CPU energy savings**

Figure 2 compares the energy savings of TimeThief over a baseline cluster without power management. The Y axis shows the total energy savings (including idle) and the X axis shows the benchmarks running at 90% (peak), 60%, and 30% load. In all the three systems, less than 1% of queries exceed the 125-ms (*search*) request-compute-reply budgets (i.e., they all meet our target of less than 1% missed deadlines).

TimeThief achieves significant savings both at low as well as high loads. For instance, at 90% and 30% loads, TimeThief achieves about 12% and 20% energy savings over the baselines, respectively. By slowing down, TimeThief saves both active and idle energy. As the load decreases, idle power savings increase, as expected. Further, TimeThief saves about 12% energy at the peak load during which the power consumption is more than twice than that during 30% load (it is misleading to compare the savings percentages at different loads which correspond to different amounts of power consumption). Because datacenter loads are moderate to high during half the day (diurnal pattern), TimeThief's savings are significantly higher than Pegasus's which saves 0% energy at peak load (not shown).

## 5.2 Power states

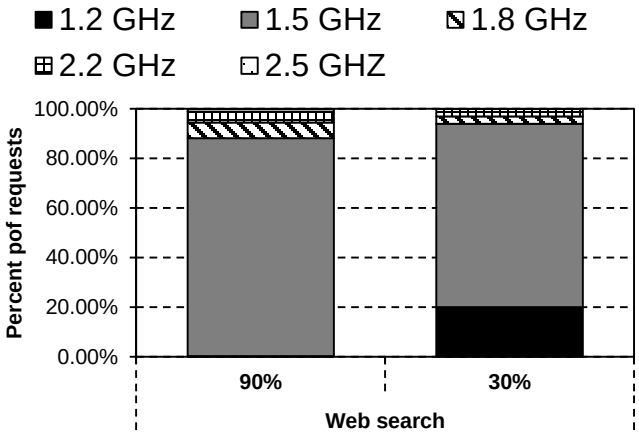


**Figure 3: Power-state distribution**

To understand TimeThief's energy savings, we bin the requests based on the CPU core's power state for each request. Each power state corresponds to a core clock speed which is scaled based on the request's slowdown factor. Figure 3 shows the fraction of requests in each bin for at 90% (peak) and 30% loads. The bins span 1.2 GHz to 2.5 GHz.

We see from Figure 3 that TimeThief even at 90% load slows down 85% of the requests by 20% or more which corresponds to the second-slowest state (1.5 GHz). As the load decreases to 30% and the slack increases, TimeThief uses the slowest state for many requests (20%) and saves more energy.

# 6 Conclusion

We proposed *TimeThief* to reduce energy by exploiting sub-critical replies' latency slack. While previous work *shifts* the leaves' response time distribution to consume the slack at lower loads, TimeThief *reshapes* the distribution at all loads by slowing down individual sub-critical nodes without increasing missed deadlines. TimeThief exploits slack in the network budget. Further, TimeThief leverages Earliest Deadline First scheduling to decouple critical requests from the queuing delays of sub-critical requests which can then be slowed down without hurting critical requests. Using at-scale simulations, we showed that without adding to missed deadlines, TimeThief saves 12% and 20% energy at 90% and 30% loading, respectively, in a datacenter with 512 nodes.

# References


1. Intel® 64 and IA-32 Architectures Software Developer Manuals *Systems Programming Guide, part 2*, 2013.
2. Iperf - The TCP/UDP Bandwidth Measurement Tool *https://iperf.fr/*.
3. The ns-3 discrete-event network simulator, http://www.nsnam.org/.
4. Al-Fares, M., Loukissas, A. and Vahdat, A. A scalable, commodity data center network architecture *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, ACM, Seattle, WA, USA, 2008.
5. Alizadeh, M., Greenberg, A., Maltz, D.A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S. and Sridharan, M. Data center TCP (DCTCP) *Proceedings of the ACM SIGCOMM 2010 conference*, ACM, New Delhi, India, 2010.
6. Aydin, H., Melhem, R., Moss, D., Mej, P. and a, A. Power-Aware Scheduling for Periodic Real-Time Tasks. *IEEE Trans. Comput.*, *53* (5). 584-600.
7. Barroso, L.A., Dean, J. and Holzle, U. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, *23* (2). 22-28.
8. Barroso, L.A. and Hölzle, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool, 2009.
9. Benson, T., Akella, A. and Maltz, D.A. Network traffic characteristics of data centers in the wild *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, ACM, Melbourne, Australia, 2010.
10. Chen, Y., Alspaugh, S., Borthakur, D. and Katz, R. Energy efficiency for large-scale MapReduce workloads with significant interactive analysis *Proceedings of the 7th ACM european conference on Computer Systems*, ACM, Bern, Switzerland, 2012.
11. Dean, J. and Barroso, L.A. The tail at scale. *Commun. ACM*, *56* (2). 74-80.
12. Delimitrou, C. and Kozyrakis, C. Paragon: QoS-aware scheduling for heterogeneous datacenters *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ACM, Houston, Texas, USA, 2013.
13. Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A.D., Ailamaki, A. and Falsafi, B. Clearing the clouds: a study of emerging scale-out workloads on modern hardware *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ACM, London, England, UK, 2012.
14. Floyd, S. and Jacobson, V. The synchronization of periodic routing messages *Conference proceedings on Communications architectures, protocols and applications*, ACM, San Francisco, California, USA, 1993.
15. Google. Efficiency: How we do it *http://www.google.com/about/datacenters/efficiency/internal/*.
16. Hoff, T. Latency is Everywhere and it Costs You Sales - How to Crush it *http://highscalability.com/blog/2009/7/25/latency-iseverywhere-and-it-costs-you-sales-how-to-crush-it.html.*, 2009.
17. Isci, C., Buyuktosunoglu, A., Cher, C.-Y., Bose, P. and Martonosi, M. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society, 2006.
18. Kleinrock, L. *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.
19. Lang, W. and Patel, J.M. Energy management for MapReduce clusters. *Proc. VLDB Endow.*, *3* (1-2). 129-139.
20. Lee, J. and Kim, N.S. Optimizing throughput of power- and thermal-constrained multicore processors using DVFS and per-core power-gating *Proceedings of the 46th Annual Design Automation Conference*, ACM, San Francisco, California, 2009.
21. Lin, C. and Brandt, S.A. Improving Soft Real-Time Performance through Better Slack Reclaiming *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, IEEE Computer Society, 2005.
22. Liu, C.L. and Layland, J.W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, *20* (1). 46-61.
23. Lo, D., Cheng, L., Govindaraju , R., Barroso, L.A. and Kozyrakis, C. Towards Energy Proportionality for Large-Scale Latency-Critical Workloads *The 41th Annual International Symposium on Computer Architecture*, Minnesota, MN, 2014, 301-312.
24. Meisner, D., Gold, B.T. and Wenisch, T.F. PowerNap: eliminating server idle power *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ACM, Washington, DC, USA, 2009.
25. Meisner, D., Sadler, C.M., Andr, L., Barroso, Weber, W.-D. and Wenisch, T.F. Power management of online data-intensive services *Proceedings of the 38th annual international symposium on Computer architecture*, ACM, San Jose, California, USA, 2011.
26. Moon, S.B., Skelly, P. and Towsley, D., Estimation and removal of clock skew from network delay measurements. in *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, (1999), 227-234.
27. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T. and Venkataramani, V. Scaling Memcache at Facebook *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, USENIX Association, Lombard, IL, 2013.
28. Paxson, V. On calibrating measurements of packet transit times *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, ACM, Madison, Wisconsin, USA, 1998.
29. Pillai, P. and Shin, K.G. Real-time dynamic voltage scaling for low-power embedded operating systems *Proceedings of the eighteenth ACM symposium on Operating systems principles*, ACM, Banff, Alberta, Canada, 2001.
30. Raghavendra, R., Ranganathan, P., Talwar, V., Wang, Z. and Zhu, X. No "power" struggles: coordinated multi-level power management for the data center *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ACM, Seattle, WA, USA, 2008.
31. Rajamani, K., Rawson, F., Ware, M., Hanson, H., Carter, J., Rosedahl, T., Geissler, A., Silva, G. and Hua, H. Power-performance management on an IBM POWER7 server *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, ACM, Austin, Texas, USA, 2010.
32. Ramakrishnan, K., Floyd, S. and Black, D. *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC Editor, 2001.
33. Ranganathan, P., Leech, P., Irwin, D. and Chase, J. Ensemble-level Power Management for Dense Blade Servers *Proceedings of the 33rd annual international symposium on Computer Architecture*, IEEE Computer Society, 2006.
34. Ren, S., He, Y. and McKinley, K. A Theoretical Foundation for Scheduling and Designing Heterogeneous Processors for Interactive Applications *the 11th International Conference on Autonomic Computing (ICAC 14)*, USENIX Association, Philadelphia, PA, 2014.
35. Sharma, N., Barker, S., Irwin, D. and Shenoy, P. Blink: managing server clusters on intermittent power *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ACM, Newport Beach, California, USA, 2011.
36. Tullsen, D.M., Eggers, S.J. and Levy, H.M. Simultaneous multithreading: maximizing on-chip parallelism *Proceedings of the 22nd annual international symposium on Computer architecture*, ACM, S. Margherita Ligure, Italy, 1995.
37. Vamanan, B., Hasan, J. and Vijaykumar, T.N. Deadline-aware datacenter tcp (D2TCP) *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, ACM, Helsinki, Finland, 2012.
38. Wilson, C., Ballani, H., Karagiannis, T. and Rowtron, A. Better never than late: meeting deadlines in datacenter networks *Proceedings of the ACM SIGCOMM 2011 conference*, ACM, Toronto, Ontario, Canada, 2011.
39. Yang, H., Breslow, A., Mars, J. and Tang, L. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ACM, Tel-Aviv, Israel, 2013.