Reducing Response Time with Preheated Caches

Mathias Gottschlag and Frank Bellosa

Karlsruhe Institute of Technology
gottschlag@ira.uka.de bellosa@kit.edu

Abstract. CPU performance is increasingly limited by thermal dissipation, and soon aggressive power management will be beneficial for performance. Especially, temporarily idle parts of the chip (including the caches) should be power-gated in order to reduce leakage power. Current CPUs already lose their cache state whenever the CPU is idle for extended periods of time, which causes a performance loss when execution is resumed, due to the high number of cache misses when the working set is fetched from external memory. In a server system, the first network request during this period suffers from increased response time. We present a technique to reduce this overhead by preheating the caches in advance before the network request arrives at the server: Our design predicts the working set of the server application by analyzing the cache contents after similar requests have been processed. As soon as an estimate of the working set is available, a predictable network architecture starts to announce future incoming network packets to the server, which then loads the predicted working set into the cache. Our experiments show that, if this preheating step is complete when the network packet arrives, the response time overhead is reduced by an average of 80%.

Keywords: Leakage Power, Caches, Preheating, Response Time, Working Set Estimation

1 Introduction

CPU performance is increasingly limited by thermal dissipation. While smaller feature sizes provide us with additional transistors which could be used to implement more and more cores on one chip, the increased power density will soon create a situation where a significant portion of the available chip area has to be powered off (*dark silicon* [5]). Several techniques have been developed to make use of additional chip area even if continuous usage of the whole chip would violate thermal limits. One such technique is *Computational Sprinting*, which lets a CPU temporarily utilize all of the chip area to reduce the response time of the system. At other points in time, the chip is operating with significantly reduced power dissipation to keep the average power below the thermal limit [12].

We envision computational sprinting to be useful in a data center environment, because many web services have critical response time requirements. In such a setting, a server system would process several requests at full performance

and therefore with low response times, and then would enter a low-power state in order to reduce the chip temperature. As leakage power is responsible for a significant part of the overall power consumption in modern CPUs [8], deep CPU sleep states (e.g., the ACPI PC7 state of modern Intel CPUs) shut down as much of the chip area as possible, including the caches [13]. As a result, the caches lose their state during idle periods. When the CPU resumes execution, the server application therefore incurs a performance loss because the working set has to be fetched from external memory. For the nginx web server serving a static web site, we have measured the response time overhead caused by cold caches to be as high as 35%.

The high response time from the initially cold caches would affect the tail latency of the system, even if, as described above, we expect servers to process requests in bursts between low-power periods. In modern large-scale web services, however, the tail latency of single systems is critical: Because operations are often parallelized on hundreds of machines (e.g., database shards), the final result will be delayed even if a single sub-operation experiences increased latency [2]. It is therefore important to reduce the impact of power management on the tail latency of server systems.

Zhu et al. propose anticipatory wakeups as a technique to hide the exit latency of CPU sleep states by waking the CPU up in advance before an event occurs [17]. Our experiments, however, have shown that the exit latency is low compared to the latency overhead caused by cold caches. Additionally, anticipatory wakeups can only function when the wakeup source is well predictable, whereas incoming network packets are hardly predictable. In this paper, we build upon the theory of anticipatory wakeups and extend it to solve these two problems. The key contributions of this paper, which are elaborated in Section 3, are as follows:

- We describe a technique to construct a fine-grained estimate of the working set of a server application. Such an estimate is required for efficient cache preheating. We generate the estimate by analyzing the tag bits of the last-level cache.
- We present a technique to predict future incoming network packets, in order to enable the system to wake up early and preheat the cache in anticipation of the network packets. Our design uses a network architecture which implements congestion control in a central arbiter. Having an overview over all packets sent in the near future, this arbiter is able to announce future packets to the receiver system.
- We discuss a mechanism to preheat the cache in anticipation of an event, so that the response time to that event is reduced significantly because the working set is already present in the caches.

These contributions are evaluated with measurements of a prototype implementation (see Sections 4 and 5). We outline further improvements and future work in Section 6.

2 Background and Related Work

Leakage-Power Reduction: Leakage power dominates the power consumption of modern CPUs with small feature sizes [8], and as caches constitute a significant portion of the chip area, many approaches have been developed to reduce their leakage power. These approaches can be categorized into destructive and nondestructive techniques, depending on whether the cache contents are destroyed by the power management technique. Nondestructive techniques usually have lower impact on performance, but also provide lower power-saving advantages. For example, drowsy caches [3] can temporarily reduce the supply voltage of the cache to a point where the memory cells retain their content, but no access is possible. Significantly better leakage power reduction can be achieved by completely disconnecting memory cells from the power supply. Gated-V_{dd} [11] is a technique which performs such destructive power gating and combines it with a dynamically resizable cache. The decision to shrink or grow the cache is based on the number of cache misses during a time interval, and the inactive parts of the cache are disabled to conserve power.

This policy is completely reactive and uses very simple heuristics. Some dynamic situations however require a more flexible predictive approach for maximum performance: For example, the cache should be completely disabled during idle periods, but the content should be restored before the system is reactivated again. Such predictive tasks require information which is commonly only known to the operating system.

In a similar scenario, Zhu et al. therefore propose strong software engagement of the OS with the power management mechanisms of the hardware [17]. Especially, they show that the operating system can hide the wakeup latency of current CPUs by predicting future events and waking the CPU up in advance, so that it is fully awake by the arrival time of the events [17]. We extend these anticipatory wakeups with a technique to predict future incoming network packets and with a mechanism to preheat the caches when they have been flushed by the low-power state.

Centrally Arbitrated Networks: To be able to preheat the cache contents for an incoming network packet, our system needs to know the arrival time of that packet in advance. We use a network architecture with a central arbiter to predict future packets. Such networks have been studied in-depth [4,10,14], albeit with a different goal: Central arbiters are frequently used to implement connection switching, which has been proposed for low-latency traffic in data center networks.

Packet switching requires queues in all switches in order to deal with temporary congestion in some network segments, and these queues can significantly delay the queued packets. Connection switching, however, can provide superior network latencies compared to networks with traditional packet switching and congestion control [10]. In networks based on connection switching, a network arbiter temporarily allocates a connection with a fixed guaranteed bandwidth to

a pair of systems. As the arbiter has a global view of the network, it can prevent any congestion by exclusively allocating network links to a single connection [4]. Despite the differences between such a network architecture and current network stacks, connection switching can be implemented on top of off-the-shelf ethernet hardware [14] and can even coexist with traditional packet switching by assigning different types of packets to different priority classes [10].

Adaptive Pre-Paging: Whenever a network packet has been predicted, our design loads the predicted working set of the server application into the cache. Similar techniques have been developed to reduce the overhead caused by the migration of virtual machines [6]:

Post-copy migration of virtual machines achieves low downtimes by immediately resuming the virtual machine at the target system and then using on-demand paging to move the working set from the source system to the target. The problem of this technique is that initially, right after execution has been resumed on the target system, the whole working set is still placed on the source system. Therefore, many expensive page faults are generated. One approach to reduce the number of page faults is to already move the predicted working set of the virtual machine to the target system before execution is resumed (*adaptive pre-paging*) [6]. We use a similar approach to improve performance right after a system has resumed from a deep sleep state. However, instead of preventing page faults, we try to prevent cache misses by loading the estimated working set into the cache before execution is resumed. Our design therefore predicts the working set with cache line granularity instead of page granularity.

Adaptive pre-paging is further extended by Zhang et al. in their Picocenter virtualization system [16], which uses adaptive pre-paging to quickly restore virtual machines from checkpoints. The Picocenter system differentiates between different types of events which can reactivate a virtual machine (e.g., network packets which target different server applications) and creates a separate working set prediction for each type, by logging which pages have been accessed in the past after similar events. We employ a similar technique to maintain separate predicted working sets, and we select one of them to be loaded into the cache depending on the target port of the incoming network packet. In contrast to the Picocenter virtualization system, though, our design can already predict the type of future incoming network packets before they arrive.

3 Design

We present a system which loads the working set into the cache right before a network packet arrives. As shown in Figure 1, our design consists of two phases: Initially, in the working set estimation phase, the system estimates the working set of the active server application. Once an estimate is available, the system enters the cache preheating phase. It resumes regular operation, with one exception: Whenever the system is woken up from a deep sleep state, the predicted working set is fetched into the CPU caches in order to reduce the response time of the

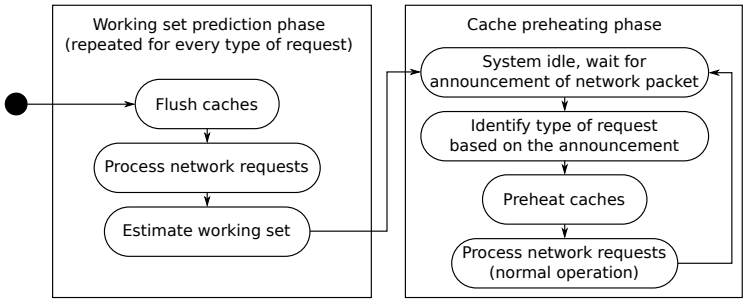


Fig. 1: The two phases of our cache preheating solution: First, the working set of the server application is estimated, then the estimate is used to preheat the caches for all following network requests. Incoming network packets are announced by an external component (described in Section 3.2).

system. For most server applications, the principle of locality is valid even over long timeframes, so the working set does not significantly change over time. Therefore, working set estimation is only performed once, but the predicted working set is reused to preheat the caches many times.

Working set estimation and cache preheating are implemented as part of the OS and are designed to work with arbitrary unmodified server applications. Similarly, the extensions to the network architecture as described in Section 3.2 are completely transparent to both the server application and its clients.

3.1 Working Set Estimation

In current systems, working set estimation is usually performed with page granularity, for example to provide efficient virtual memory. Often, the application only requires parts of a page, though. A cache preheating system should not load more data into the cache than necessary, so the working set must be predicted with cache line granularity. On current hardware, we have identified two hardware mechanisms which can be used to provide fine-grained information about the current working set.

First, some CPU architectures are able to trace and record all cache misses. The list of the cache misses and the accessed memory locations can be analyzed to create an estimate of the application's working set. For example, current Intel processors provide processor event-based sampling (PEBS) as a tracing facility for various types of events [7]. PEBS monitors an event counter and stores a copy of the most important CPU registers (along with the accessed memory address in case of memory events) to a buffer whenever the counter reaches a user-defined value. In theory, this facility can be used to trace all cache misses. In practice, however, whenever an event is logged, PEBS frequently misses other events which occur at approximately the same time [9]. It is therefore neither an effective working set estimation mechanism, nor is it efficient, as it also causes significant overhead when every event is recorded.

Alternatively, the working set can be estimated by analyzing cache contents. The tag bits in the cache can be translated into a list of physical addresses which have been accessed by the application since the last cache flush. Among others, the ARM Cortex-A15 and Cortex-A57 cores provide the RAMINDEX register [1] which can be used to read and write arbitrary portions of cache memory, including tag bits. In the absence of any conflict or capacity cache misses, the resulting list of addresses is complete and, unlike any simple PEBS-based tracing mechanism, does not miss some addresses which have been accessed. Our experiments show that directly after a cache flush there are rarely any conflict or capacity misses.

As cache tag bit analysis is a viable technique for working set estimation, we use a ARM Cortex-A15 system as the basis of our design: First, the caches are flushed to remove any unwanted data from the cache, and hardware prefetching is temporarily disabled to ensure that only accessed data is loaded into the cache. Afterwards, the server resumes normal operation and processes incoming network requests. After one or more requests have been processed, the last-level cache tag memory is read and analyzed. The result is a list of all memory locations which have been accessed since the cache flush. Future invocations of the server application might access slightly different memory locations, though. For example, network buffers are likely placed at different locations. To remove such dynamic regions from the working set, all these steps are repeated several times (8 times in our prototype). The final working set estimate then only contains those memory locations which have been repeatedly accessed.

3.2 Network Packet Prediction

Once a good working set estimate is available, the system switches back to regular operation, but activates cache preheating. The cache preheating mechanism however not only requires a prediction of the working set, but the system also needs to know when to preheat the caches. Because the arrival time of network packets is usually highly nondeterministic, anticipatory cache preheating is not possible with traditional network architectures. Our solution makes use of a centrally arbitrated network architecture such as Fastpass [10] to predict future incoming network packets.

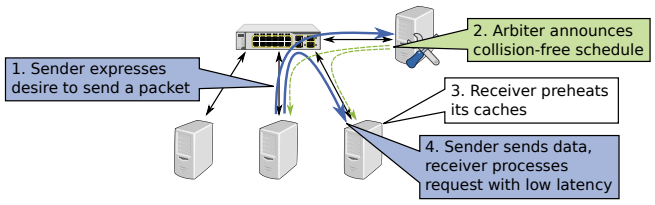


Fig. 2: Future network packets are announced by the network arbiter in advance, so that the receiver can preheat the caches in anticipation of the packets.

Figure 2 shows how a central arbiter can announce future packets: First, the sender requests a time slot to send the packet. The network arbiter receives all such requests from all systems in the network, and creates a schedule for the packets. Normally, this schedule only needs to be sent to the sender systems so that they know when to send their packets. In our system, however, the schedule is also sent to all affected receiver systems as an announcement of future network packets. When a system receives such an announcement while it is in a low-power state with flushed caches, it wakes up a CPU core which then starts to preheat the caches so that the network packet can be efficiently processed.

Ideally, the receiver system not only knows in advance when packets arrive, but also which type of request they carry. When different types of requests are processed, the server applications can have significantly different working sets. While the network is mostly oblivious to the type of request carried by a network packet, some indicators are transferred along with the data (e.g., the target TCP port). We modify the network arbitration scheme so that the sender system not only announces the target address to the arbiter, but also includes the target port. The arbiter forwards this information to the receiver system, which can, depending on the port, preheat the predicted working set of the corresponding server application.

3.3 Preheating

When an incoming packet has been announced by the network arbiter, the receiving system wakes the CPU and loads the estimated working set into the last-level cache. The main problem here is the short time span between the announcement and the arrival of the packet. For example, Fastpass calculates schedules only 65 microseconds in advance [10]. Waking up the CPU requires half of that time already [13], so only approximately 30 microseconds are left to preheat the cache. Preheating is therefore highly time-critical. However, the nginx web server only requires 235.8 KiB to serve a static website from RAM, and even a TPC-C-like MariaDB workload only requires 621.8 KiB to serve most requests. The required memory bandwidth to load these working sets into RAM in the available time (7.5 GiB/s and 19.8 GiB/s respectively) is well within the capabilities of current server hardware.

To preheat the last-level cache, the preheating code loops over all memory locations in the working set and loads them into the cache. To effectively utilize all the available memory bandwidth, the memory locations are sorted by increasing physical address. A linear access order minimizes the number of DRAM row activations and therefore improves memory throughput. As sorting is costly, the data is sorted as a preprocessing step during the working set estimation phase.

Additionally, the working set description is run-length encoded. Compression of the working set description increases the preheating memory throughput, as less additional data needs to be fetched from RAM. Run-length encoding provides significantly lower computational complexity compared to more complex cache state compression methods found in literature (e.g., dictionary-based compression [15]). The decoding overhead is low enough that it can be mostly hidden behind

memory operations. Also, the regular structure of the addresses enables sufficient compression factors: Applications frequently access long consecutive memory regions, so many consecutive cache lines can be described in one "run". As an example, the working set description of the nginx web server can be reduced by 68%, from 14528 bytes (one 32-bit address per 64-byte cache line in the working set) down to 4596 bytes, thereby increasing preheating performance by 5.7%.

These optimizations produce a fairly optimized memory access pattern which utilizes most of the available memory bandwidth. However, on modern systems, a single core often cannot saturate the memory bandwidth anymore. On our prototype platform, parallelizing the preheating code on two cores results in a 10% performance gain.

4 Evaluation

We have conducted a prototypical evaluation of our design, in order to answer the following questions: Can cache preheating be used to reduce the response time to network requests when the caches have been flushed? Is such preheating efficient enough so that it is a viable technique when combined with existing network architectures?

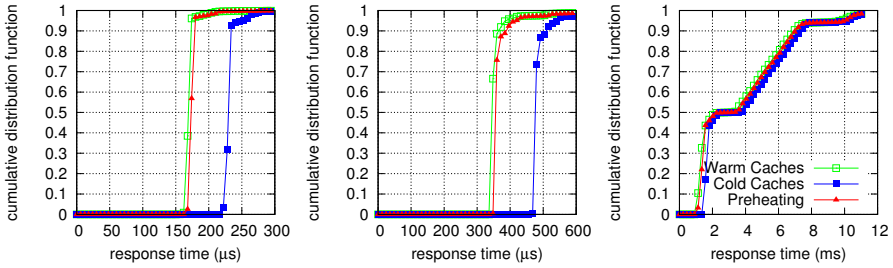
In this paper, we present a proof of concept based on a limited prototype which, while not being functionally complete, is able to show that cache preheating is a viable technique. Our prototype is designed to run on a system with ARM Cortex-A15 cores, and all benchmarks are executed on a Hardkernel Odroid-XU3 single board computer. This system provides a Samsung Exynos 5422 SoC with four Cortex-A15 cores and four Cortex-A7 cores.

The system's network support is limited to an USB ethernet adapter, which prevents any meaningful network latency benchmarks. Therefore, our prototype is not yet integrated with a real predictable network architecture, but instead simulates the network architecture in the benchmark client. The benchmarked server application is executed on a Cortex-A15 along with the cache preheating software, whereas the benchmark client is executed on a Cortex-A7 core on the same system, connected by a local TCP connection. As the SoC provides separate last-level caches for the different types of cores, this setup mostly isolates the cache footprints of the two processes. For the response time comparisons below, the benchmark client optionally flushes the caches of the Cortex-A15 cores to simulate CPU sleep states and optionally triggers cache preheating before issuing any request. The Cortex-A15 cores have private 64 KiB L1 caches as well as a shared 2 MiB L2 cache. The latter has shown to be large enough to accomodate the working sets of our benchmarks. Applications with a larger working set require a more complex approach to working set estimation.

4.1 Response Time Reduction

We measure the response time of several benchmark applications to show that preheating effectively mitigates the performance penalty of cold caches. We

compare the response time with warm caches, flushed caches, and after caches have been first flushed and then preheated. Our three main benchmark applications are the nginx web server and the memcached key-value store, both serving static data, and a more complex dynamic TPC-C-like workload (DBT-2) executed on the MariaDB database.



(a) memcached (b) nginx (c) MariaDB (DBT-2)

Fig. 3: Cumulative histogram of the response time with and without cache preheating.

	response time (µs)			CPI			working set	preheating
	warm	cold	preheated	warm	cold	preheated		
nginx	367.6	498.0	388.5	3.63	5.78	4.12	235.8 KiB	77.4 µs
memcached	178.8	238.8	188.5	3.81	5.66	4.30	142.3 KiB	53.0 µs
MariaDB	3970	4320	4069	2.20	2.44	2.29	621.8 KiB	146.4 µs

Table 1: Averaged benchmark results with cold, warm and preheated caches as well as the corresponding cycles per instruction, predicted working set size and preheating costs.

Figure 3 shows the cumulative histogram of the response time of 100000 requests to the three applications. In all three cases, the average response time of requests is significantly reduced by cache preheating compared to when the requests hit cold caches as shown in Table 1. On average, the response time overhead (difference between response times for cold and warm caches) is reduced by 79.8%. To show that this improvement can be attributed to cache preheating, we also measure the average number of cycles per instruction (CPI). The CPI are a good indicator for the effectiveness of cache preheating, because a reduced number of cache misses is only beneficial for performance if it in turn reduces the number of CPU stall cycles. Our experiments show that the response time improvement is accompanied by 66% less cache misses (on average) as well as significantly improved CPI.

4.2 Preheating Cost

Packets are only announced several dozens of microseconds in advance, thereby limiting the time available for loading the working set into the caches. Along with the response time, we have also measured the time required for preheating (in the "preheating" column of Table 1). The cost of preheating is approximately linear to the size of the predicted working set.

We have stated in Section 3.3 that preheating must not take more than 30 microseconds. Our prototype clearly violates this limit for all selected benchmarks, with preheating taking between 53.0 and 146.4 microseconds. The achieved average memory throughput is 3.17 GiB/s, which is close to the maximum throughput which can be achieved with well optimized code.

5 Discussion

Our evaluation shows that cache preheating improves response time significantly over cold caches. The potential response time reduction is large enough that the resulting energy savings should compensate the energy cost of preheating. In our prototype, however, cache preheating requires up to four times more time than is available between the CPU waking up and the request packet arriving. As a result, the cache would not be completely preheated by the time the network request arrives. In this section, we make the case that our preheating design provides a benefit even in these scenarios. Further, we argue that server hardware should be able to preheat cache working sets before the network request arrives.

In our prototype, network requests would arrive with preheating still in progress. At that time, our system could naively complete preheating and process the request afterwards. From our experiments, we can deduct that our approach still improves response times over cold caches: The delay from preheating's tardiness is less than the reduction of request processing time it achieves, causing a net improvement of response time.

With memcached for example, we found preheating to overshoot the 30 µs available (see Section 3.3) by 23 µs. However, preheating reduced the request processing time by 50µs (see Table 1), thereby lowering the response time by 27 µs (11%) overall. Similarly, even for the MariaDB benchmark which overshoots the preheating deadline by almost 120 µs, our preheating approach would still provide a 130 µs response time reduction.

In practice, server hardware will require significantly less time, though, and can preheat the caches in time for the arriving network packets. Current server systems provide significantly higher memory bandwidth than the hardware platform of our prototype. The memory throughput of the Hardkernel Odroid-XU3 system is merely 3.17 GiB/s in our benchmarks, and even the *pmbw* parallel memory bandwidth benchmark only achieves slightly better results for a completely sequential access pattern. According to the pmbw benchmark, a recent Intel Skylake system with dual-channel memory in contrast provides almost 8 times more bandwidth. This performance increase should allow preheating to

be completed in time before the corresponding network packet arrives, even for complex workloads such as the presented MariaDB benchmark.

6 Conclusion and Outlook

Deep CPU sleep states have negative effects on server response times, yet such power management methods are required in order to improve overall performance in a world with significant amounts of dark silicon. We have identified frequent cache flushes as the most problematic side effect of aggressive power management. Previous work usually suggested different power management methods which keep the cache state intact, but waste significant amounts of energy instead or which reduce performance. We argue that a more efficient system can be built if the operating system is in charge of cache content management.

We propose a system which preheats the caches in anticipation of events which cause the system to resume from a deep sleep state, in order to mitigate the effect of cold caches. When the arrival time of the next wakeup event is known, the estimated working set can be loaded into the cache in order to reduce the cache miss rate shortly after the event. We also describe a method to predict future incoming network packets with the help of a centrally arbitrated network architecture. Benchmarks show that such cache preheating can mitigate most of the overhead caused by cold caches. The time required to preheat the caches is too long in our current prototype though, due to the low memory bandwidth of our prototype platform. We show that cache preheating still results in a net response time reduction, and we argue that current server hardware provides enough memory bandwidth that cache preheating is completed quickly enough.

Our working set estimation code is currently limited to certain ARM CPU cores. On Intel CPUs, we are therefore evaluating whether PEBS—despite its limitations—can be used to trace all cache misses and to derive the working set from them. Additionally, we are currently integrating our cache preheating system with the Fastpass [10] network architecture, in order to be able to use cache preheating in a representative server system and to evaluate its effect on power usage in such a system.

References

1. Cortex-A15 Technical Reference Manual: 4.3.57. RAM Index Register. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0438c/BABEJEAJ.html, accessed: 2015-05-05
2. Dean, J., Barroso, L.A.: The Tail at Scale. Communications of the ACM 56(2), 74–80 (Feb 2013)
3. Flautner, K., Kim, N.S., Martin, S., Blaauw, D., Mudge, T.: Drowsy Caches: Simple Techniques for Reducing Leakage Power. In: Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02). pp. 148–157. IEEE (2002)

4. Grosvenor, M.P., Schwarzkopf, M., Moore, A.W.: R2D2: Bufferless, Switchless Data Center Networks Using Commodity Ethernet Hardware. In: ACM SIGCOMM Computer Communication Review. vol. 43, pp. 507–508. ACM (2013)
5. Hardavellas, N., Ferdman, M., Falsafi, B., Ailamaki, A.: Toward dark silicon in servers. IEEE Micro 31, 6–15 (2011)
6. Hines, M.R., Gopalan, K.: Post-Copy Based Live Virtual Machine Migration Using Adaptive Pre-Paging and Dynamic Self-Ballooning. In: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'09). pp. 51–60. ACM (2009)
7. Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 3: System Programming Guide. No. 325384-058US (Apr 2016)
8. Kim, N.S., Austin, T., Baauw, D., Mudge, T., Flautner, K., Hu, J.S., Irwin, M.J., Kandemir, M., Narayanan, V.: Leakage Current: Moore's Law Meets Static Power. Computer 36(12), 68–75. IEEE (2003)
9. Larysch, F.: Fine-Grained Estimation of Memory Bandwidth Utilization. Master Thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany (Mar 2016)
10. Perry, J., Ousterhout, A., Balakrishnan, H., Shah, D., Fugal, H.: Fastpass: A Centralized "Zero-Queue" Datacenter Network. ACM SIGCOMM Computer Communication Review 44(4), 307–318 (2015)
11. Powell, M., Yang, S.H., Falsafi, B., Roy, K., Vijaykumar, T.: Gated-V_{dd}: A Circuit Technique to Reduce Leakage in Deep-Submicron Cache Memories. In: Proceedings of the 2000 International Symposium on Low Power Electronics and Design (ISPLED). pp. 90–95. ACM (2000)
12. Raghavan, A., Luo, Y., Chandawalla, A., Papaefthymiou, M., Pipe, K.P., Wenisch, T.F., Martin, M.M.K.: Computational Sprinting. In: Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA). pp. 1–12. IEEE (2012)
13. Schöne, R., Molka, D., Werner, M.: Wake-up latencies for processor idle states on current x86 processors. Computer Science - Research and Development 30(2), 219–227. Springer (2015)
14. Vattikonda, B.C., Porter, G., Vahdat, A., Snoeren, A.C.: Practical TDMA for Datacenter Ethernet. In: Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12). pp. 225–238. ACM (2012)
15. Vishnoi, A., Panda, P.R., Balakrishnan, M.: Cache Aware Compression for Processor Debug Support. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE'09). pp. 208–213. European Design and Automation Association (2009)
16. Zhang, L., Litton, J., Cangialosi, F., Benson, T., Levin, D., Mislove, A.: Picocenter: Supporting Long-lived, Mostly-idle Applications in Cloud Environments. In: Proceedings of the 11th European Conference on Computer Systems (EuroSys'16). p. 37. ACM (2016)
17. Zhu, Q., Zhu, M., Wu, B., Shen, X., Shen, K., Wang, Z.: Software Engagement with Sleeping CPUs. In: 15th Workshop on Hot Topics in Operating Systems (HotOS XV). USENIX Association (May 2015)