Karlsruher Institut für Technologie

Reduced Response Time with Preheated Caches

Masterarbeit
von

cand. inform. Mathias Gottschlag

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Wolfgang Karl
Betreuender Mitarbeiter:	Prof. Dr. Frank Bellosa

Bearbeitungszeit: 1. Dezember 2015 – 31. Mai 2016

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 31. Mai 2016

Abstract

CPU performance is increasingly limited by thermal dissipation, and soon aggressive power management will be beneficial for performance. Especially, large parts of the chip (including the caches) will be frequently power-gated in order to reduce leakage power. Therefore, the cache content is lost whenever the CPU is idle, 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. For common workloads, we measured as much as 34% response time overhead due to cold caches.

In this thesis, 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 last-level cache in anticipation of the packets. When the network packet arrives after this preheating step is complete, the server application can process the network request with warm caches and therefore with improved performance.

We evaluate our design with a proof-of-concept prototype based on Linux, on a system with ARM CPUs. Our experiments show that cache preheating can reduce the response time overhead caused by cold caches by an average of 80%. We demonstrate that our prototype does not cause significant runtime overhead, and we show that the time required to load the working set into the cache is low enough that it is possible to exploit the full potential of cache preheating on current server CPUs.

Acknowledgments

I would like to express my gratitude to the members of the Operating Systems Group at the KIT and especially my supervisor Prof. Bellosa for their support and guidance during my work on this thesis. I am particularly thankful for the opportunity to present my work at EuroSys 2016 as well as for the extensive feedback on this thesis, the EuroSys poster and a workshop paper submitted to ROME 2016. A special thanks goes to Marius Hillenbrand and Jens Kehne, who, while proofreading the poster and the paper, provided invaluable suggestions which had significant influence on this document. I would also like to thank Yannick Hörstensmeyer, who pulled an all-nigher to provide feedback as quickly as possible when I sent him a draft of this work.

Contents

Chapter 1

Introduction

In the future, server systems are going to use more and more aggressive power management. At smaller and smaller chip feature sizes, CPU performance is mainly limited by thermal dissipation, so methods to improve power efficiency are increasingly beneficial for performance [56]. Whenever possible, the CPU will therefore be placed in deep sleep states to reduce power. The most important power management methods utilized by these sleep states are clock gating [57] and power gating [49]. Whereas the former simply removes the clock signal from inactive parts of the chip, the latter also disconnects the supply voltage. The important advantage of power gating is that leakage power is completely removed when the supply voltage is disconnected. However, the affected parts of the chip lose their state.

This loss of information is especially problematic for the CPU caches. Modern processors contain large on-chip caches, which are flushed when power is removed. As a result, when the processor resumes operation, the server application suffers from high cache miss rates and therefore high average memory access latency while the working set is loaded into the cache.

In a server system, cold caches cause increased response times for the network requests which are processed directly after the system has resumed from a deep sleep state. The affected network requests significantly increase the tail latency of the system. As operations are often parallelized on hundreds of machines (e.g., database shards), the tail latency of each single system is critical, though, as the final result of a request by a user will be delayed even if only a single sub-operation experiences increased latency [27].

In this thesis, we present a solution to reduce the response time of those network requests which are processed with cold caches due to the effects of deep CPU sleep states. Our design analyzes the working set of the server application and estimates the memory locations which are required while a network request is processed. Once sufficient information about the application's working set is

available, the design uses a predictable network architecture with a central arbiter to predict the arrival time of future network requests. Whenever a network request is expected to arrive while the CPU is in a deep sleep mode, the system is configured to wake up earlier and the working set estimate is loaded into the last-level cache before the network request is processed. As a result, the server application requires fewer accesses to external memory, so its response time is reduced.

In the following chapters, we provide a detailed description of the design, and we also show that the design is viable and provides the desired response time reduction. To do so, we have implemented the main techniques in a prototype based on Linux, and benchmarked that prototype with several common server applications. Our benchmark results show that the effect of cold caches on the response time is reduced by 80% on average. Although our prototype system frequently requires too much time to preheat the caches, we show that cache preheating is a viable technique on modern server hardware.

The following sections are organized as follows: First, in Section 2, we give more details on the problem we are trying to solve and on the technical background to our solution. Also, we describe other related work and show the differences to the design presented in this thesis. Afterwards, in Section 3, we analyze the properties of a generic cache preheating solution and describe the hardware and software properties as well as the assumptions which lead to our specific design. Then, in Section 4, we describe our proposed specific cache preheating solution and describe how it mitigates the effects of deep CPU sleep states on response times. We have implemented a prototype of our design and, while doing so, have encountered several practical problems which are specific to the underlying hardware. Our implementation and these problems are detailed in Section 5. Section 6 contains an evaluation of our design, with a focus on response time reduction and general viability. Finally, in Section 7, we conclude the thesis and lay out a plan for future work.

Chapter 2

Background and Related Work

In this chapter, we present technical background and related work of our proposed cache preheating solution. Our design is supposed to counteract the negative side effects of cache flushes due to aggressive power management. We expect such aggressive power management to become increasingly common in server systems. In Section 2.1, we give an overview of the most important power management mechanisms, and we explain why future server processors will frequently flush the CPU caches. As our solution makes use of predictable network architectures to predict future network packets, we describe such architectures in Section 2.2. The rest of this chapter is dedicated to related work: In Section 2.3, we describe mechanisms to record the cache content, because our solution includes working set prediction based on cache content analysis. Our design utilizes the predicted working set in order to reduce the number of cache misses after a cache flush. Similar techniques have been proposed to reduce the number of page faults after virtual machine migration. We present these techniques in Section 2.4.

2.1 Dark Silicon and Power Management

The main problem we are trying to solve is that aggressive power management leads to a significant response time overhead in server systems. Until now, server administrators have only sparingly used power management techniques in their systems, especially if the servers perform performance-critical tasks, because there are a wide range of reports that power saving options hurt the overall performance [14, 15, 46]. In the future, however, a situation will emerge in which aggressive power management is crucial for good CPU performance, because the latter will be limited not by the number of available transistors but rather by the power density of the chip.

For a long time, the required power density stayed constant despite rising fre-

quencies and increased numbers of transistors, because constant-field scaling (also called *Dennard Scaling*) provided lower and lower supply voltages [28]. This trend stopped, however, because further threshold voltage scaling would have lead to excessive leakage power and because the gate oxide had reached a size where no significant further thickness reduction was possible [23]. Without voltage scaling, every feature size reduction leads to a significant power density increase. The dissipated power of a chip is generally limited by the cooling solution though. The result is a problem known as *Dark Silicon* (or, originally, as a reduction of the *Simultaneously Active Fraction* [24]): Because the thermal budget is limited, only a fraction of the transistors can be active at any point in time. For a feature size of 8 nm, less than 50% of the chip area is expected to be usable [30]. Alternatively, *Computational Sprinting* [50] can be used to temporarily exceed the thermal budget at some points in time if the system conserves energy at others, so that the overall average power stays within the limits. Such a technique can be used to reduce the response time of a system when the workload allows for temporary low-power phases. In other words, Dark Silicon creates a situation where aggressive power management is beneficial for performance.

2.1.1 Power Management Techniques

There are various different methods available to reduce the dissipated power of a CPU. The power dissipation of a CMOS chip is generally dominated by dynamic power and leakage power. Dynamic power is dissipated when capacitive elements like wiring and transistor gates are charged and discharged. Many important techniques to reduce dynamic power therefore reduce switching activity. For example, *clock gating* [57] disconnects the clock signals from inactive parts of the chip, and *operand isolation* [38] disconnects unnecessary input signals to prevent spurious switching activity. Because all inactive parts of the chip are still powered, however, these techniques do not affect leakage power.

Leakage power is dissipated due to sub-threshold currents between source and train of the transistors as well as tunneling currents at the gate. At small feature sizes and especially at small threshold voltages, these currents are large enough that power consumption is dominated by the leakage power [39]. The leakage currents depend on the supply voltage, therefore dynamic voltage and frequency scaling (DVFS) can be used to significantly reduce the leakage power. DVFS reduces the operating voltage of the chip and, in order to guarantee correct functioning, also reduces the operating frequency. The result is a super-linear power reduction, whereas the performance loss is only approximately linear to the voltage scaling factor [25]. However, recent studies have found that the range of usable voltages has been significantly reduced for recent CPU generations [61], and that frequency scaling has become increasingly ineffective [41].

A technique which can provide far better leakage power reduction is *power gating*. Power gating, originally also called *multithreshold-voltage CMOS* [44], is a technique where sleep transistors are inserted between the power supply and the virtual power rails of parts of the chip. When these parts of the chip are inactive, they can be decoupled from the power supply. Especially if the sleep transistors have a comparably high threshold voltage, there is almost no leakage current in disabled regions [49].

All these mentioned techniques are available in current systems. For example, on x86 systems, power gating and clock gating are implemented in the form of ACPI CPU power states (C-states). Current Intel processors will power-gate individual cores and their L2 caches if the cores are placed in the ACPI C6 state. The L3 cache and parts of the chip uncore area are additionally power-gated when all cores are shut down and the package is placed in the C7 state [53]. Reducing the leakage power of the caches is especially important, as they occupy a large percentage of the chip area. For example, cache and other on-chip memory occupied 30% of the die area of the Alpha 21264 CPU and 60% of the StrongARM CPU [49], and die shots of more recent Intel Haswell CPUs indicate that approximately 20% of these chips is used for last-level caches [26, 54] (excluding processor graphics).

However, as described in the next section, power-gating the caches has the disadvantage that, once the supply voltage is removed, all cache content is lost. Therefore, a number of techniques have been proposed which can reduce the power dissipation of on-chip memories without losing significant amounts of data. Once such technique is the concept of *Drowsy Caches*, where the caches are not power gated, but instead the supply voltage is reduced to 1.5 times the threshold voltage [32]. At this supply voltage, the SRAM cells can keep their state, but no access to the data is possible. Although drowsy caches yield impressive energy savings and can in theory reduce the leakage power per bit by up to 85%, the caches of an idle system still need several times as much energy as if they were completely power-gated.

Alternatively, techniques have been proposed to power-gate only those parts of the cache which are not in active use. For example, the Dynamically Resizable Instruction Cache [62] can be shrunk if only parts of the cache are frequently used. The decision to shrink or grow the cache is made based on the number of cache misses in a given time interval. The unused parts of the cache can then be power-gated [49] in order to reduce power consumption. Dynamically resizable instruction caches can reduce the cache size by 62%, whereas the performance is reduced by only 4% overhead while the CPU is active.

This technique 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. Similarly, the application running after the idle period might not be the same application as before, so different cache contents should be restored. For a similar scenario, Zhu et al. have proposed stronger software engagement of the OS with sleeping CPUs. They show that the OS can efficiently implement predictive policies to reduce the performance impact of power management mechanisms [64]. In this thesis, we present such a predictive software technique which allows the cache to be completely power-gated during periods of inactivity, while mitigating the negative effects of the cold cache on system performance.

2.1.2 Response Time Overhead

As described above, aggressive power management saves significant amounts of energy and is essential for good performance in a scenario with large amounts of dark silicon. However, we have identified two main sources of overhead caused by deep CPU sleep states: First, the system needs some time to reactivate the CPU, and second, the following code runs with reduced performance because the sleep states cause the CPU caches to be flushed.

In current server systems, power saving mechanisms are usually implemented in the form of ACPI C-states. In [53], the authors have analyzed the wakeup time from various ACPI C-states. For example, a SandyBridge-EP CPU takes between 30 and 40 microseconds to wake up from the package ACPI C6 state. As a result, the system is slower to respond to incoming network requests or other types of events. Additional significant response time overhead is caused by the cache misses generated after the processor has been woken up. The reason for these cache misses is that the caches are power gated and therefore lose all their content when the processor enters a sufficiently deep sleep mode. For example, on modern Intel CPUs, the ACPI C3 state flushes the L1 and L2 caches, whereas the package C7 state additionally flushes and power-gates the L3 cache [53]. Depending on the implementation of the sleep states, most other caches are flushed as well, including branch prediction caches and translation lookaside buffers (TLBs), although these, as shown in Section 3.3.1, tend to have a significantly lower impact on performance.

The impact of a cache flush on system performance depends on the size of the cache, which roughly translates to the number of cache misses generated by the cache flush, as well as on the cost of each single cache miss. Figure 2.1 shows the memory hierarchy of an Intel Nehalem system. Each core has its own small exclusive L1 and L2 caches, and all cores together share one large L3 cache per chip. The access latency to each memory hierarchy level depends on the distance

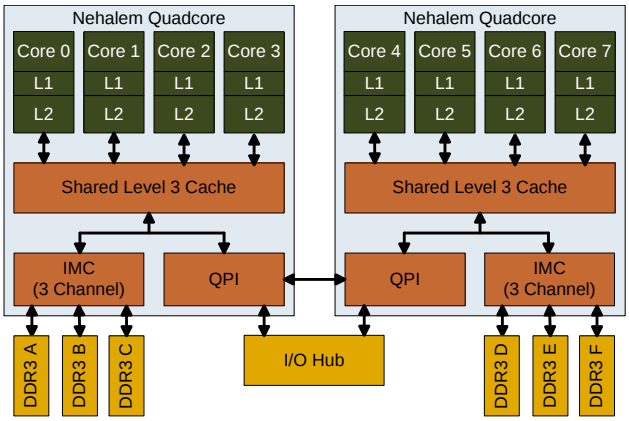


Figure 2.1: Memory hierarchy of an Intel Nehalem system with two CPUs [43]—each core has its own L1/L2 caches, but all cores of a CPU share one L3 cache.

to the CPU core: Whereas an access to the L1 cache requires only 1.3ns, an L2 cache hit requires 3.4ns. If the L3 cache is accessed, the access latency further rises to 13ns [43]. If all caches are flushed, however, the access latency increases to around 63ns because the data needs to be loaded from external memory. However, not only is the access latency of requests to external memory especially high, but the latency highly depends on the access pattern. External DRAM is organized in multiple banks, and each bank is organized in multiple rows. Access operations experience increased latency if they do not hit the currently selected row and instead trigger a bank or row switch [51].

Due to the high cost of access to external memory, we expect especially last level cache flushes to have significant effect on the response time to the following events. These effects vary depending on the working set size and the memory access pattern of the program. To measure the effect, we benchmark the response time of several network programs (Nginx, memcached, MariaDB). Before the network request, the benchmark flushes all cache levels. No actual processor sleep modes are active, as we wish to show the effect of memory access overhead in isolation. In Figure 2.2, we plot the resulting latency histogram for a simple static web page served by the Nginx web server. We compare it to the response time for the same setup, but without any cache flush. The benchmark shows that cold caches increase the latency of Nginx by 132 μs, or by about 36% of the original latency. Other benchmarks (memcached, MariaDB) also show significant response time overhead depending on working set size, total response time and memory access pattern.

As these results show, the overhead from cold caches is significantly larger than the wakeup latency of the CPU. Previous work, however, has only addressed

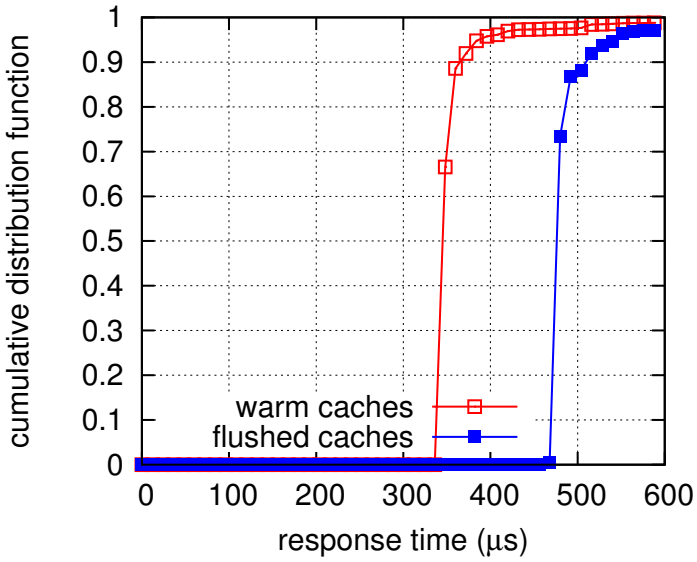


Figure 2.2: Cumulative histogram of the response time of the nginx web server with and without cache flushes between the requests.

the latter. Anticipatory wakeups [64] are a technique which, whenever a future event is known which will wake the CPU up, mitigates the wakeup latency by waking the CPU early in anticipation of the event, so that the system can quickly react because the CPU is already active. Anticipatory wakeups can effectively reduce the performance impact of most CPU sleep states. The implementation described in [64] has two significant limitations though, both which we try to solve in this thesis: First, anticipatory wakeups do not reduce the effect of cold caches on CPU performance, as the cache state is not changed by the early wakeup. Second, anticipatory wakeups are only viable if the arrival time of future events is well predictable. This limitation prevents the use of the technique to reduce the response time for incoming network packets. Web services, however, are especially affected by increased response times, even if most web services do not have to provide any hard real-time guarantees: It has been shown that excessive latency often affects the user experience and results in reduced user interaction with the web service [35].

Moreover, the user experience is even affected when only a small percentage of all network requests in the data center suffers from increased latency. Modern large-scale often parallelize incoming requests on hundreds of systems, and the final result is delayed when even a single of the sub-operations is delayed [27]. In other words, the tail latency of each system is more important than its average

response time. There are situations in which power management does not have any significant effect on the average response time, but it always increases the tail latency: We expect that each system processes network requests in bursts and puts its CPU to sleep between each two bursts, but the first request of each burst always hits cold CPU caches. The increased response time of this first request therefore affects the overall performance of the web service.

2.2 Predictable Network Architectures

If a system is supposed to wake the CPU up and to preheat the caches before an event causes an application to resume execution, the system needs to know the arrival time of this event. Most events caused by local hardware, like the completion interrupt of SSDs or GPUs, can be precisely predicted by analyzing the runtime of similar previous operations. For example, the size of a transfer to or from a block device is mostly proportional to the latency of the operation, and the execution time of GPU kernels can be predicted based on the history of previous launches of the same kernel [64]. Other sources of events, however, are more difficult to predict. Especially, it is hard to predict the time of future incoming network requests. A rather recent development in the area of data center networks could change this situation, namely the reintroduction of connection switching to data center networks.

Practically all current networks utilize packet switching to allocate physical connections on a per-packet basis between the various networked systems. However, packet switching requires queues in all intermediate switches, because otherwise temporary network traffic bursts can lead to increased packet loss if the capacity of a network segment is exceeded, even if the average throughput of that segment is sufficient over longer time periods [31]. Commodity switches generally have rather deep switch queues to handle bursty traffic patterns [42]. As traditional TCP congestion control, however, uses packet loss to determine link congestion, so flows with high throughput fill up the switch queues and affect the network latency of all other packets. There are alternative congestion control algorithms which reduce length of the switch queues (e.g., Data Center TCP [22]), as a practical way to reduce the end-to-end latency.

An alternative to these techniques is to use connection switching instead of packet switching, thereby removing the need for switch queues. If two systems communicate over a network with connection switching, they are temporarily allocated a physical connection. This connection is either exclusive to the two systems, or the systems are guaranteed a minimum network bandwidth. If the connection is exclusive to the two systems, no collisions can occur in the switches, so no queues are needed at all [34]. If the connection is not exclusive, but instead

some parts of the connection are shared with other systems, each communicating system is allocated a limited data rate so that the capacity of shared connection segments is not exceeded. Such a data rate limit can be enforced by scheduling individual packets in a way that the overall rate is limited [47]. When the total data rate of each network segment is below its maximum capacity, the switch queues are mostly empty.

The allocation of such connections and data rates is usually done by a central arbiter. This arbiter usually collects requests by the systems in the network and periodically, based on the requests, creates a schedule of future connections. The arbiter can then either directly mute those systems which are not supposed to send any data [58], or it can send the schedule to all affected systems which then send outgoing data according to the schedule [47]. The latter is a core component of the design presented in this thesis: In Section 4.1.3, we show how we use this mechanism to notify the receiver systems about future incoming packets, so that they can use anticipatory wakeups and cache preheating to reduce the response time to those packets.

2.3 Cache Usage Analysis

In this thesis, we present a cache preheating solution to reduce the effects of deep CPU sleep states on network applications. We base our design on an analysis of the cache contents after several network requests have been processed. Therefore, our design requires hardware facilities to read the cache state. Previous research on methods to dump processor cache contents has mostly attempted to improve the performance of post-silicon processor debugging: When test cases are executed on a processor to find faults in the hardware, the processor state is repeatedly saved. The dumped processor state serves both as a basis for later analysis of the faults as well as a starting point when the faulty parts of the tests are executed again with finer processor state dump intervals [59]. It is not conceptually difficult to construct facilities which allow to dump the cache contents. In fact, as used in the design presented in this thesis, current ARM processors already allow direct access to cache memories [5]. However, efficient usage of the provided data is difficult, both due to the large amount of data and because frequent snapshots slow down the running programs.

Our work only requires the cache tag bits, other parts of the cache contents like the actual cached data are not needed to reconstruct the cache state. Therefore, our design deals with a significantly reduced amount of data, and such partial cache dumps are not as expensive as a full dump would be. Still, techniques which further reduce the amount of data can be beneficial. One option is to implement cache dumping or restoring in hardware, in order to reduce power usage or so that the op-

erations could be executed in parallel with the running software (for example, [60] describes an implementation of the latter). Another potential optimization is to compress the dumped cache state. Vishnoi et al. present a technique to compress the cache state with a derivative of the LZW compression algorithm [59]. In this thesis, we have opted not to use any such compression algorithm. Instead, we rely on simple run-length encoding, because we optimize for a different use case: In the processor debugging scenario, cache analysis is time-critical, whereas in our situation decompression is more problematic, as cache preheating has to meet tight deadlines which are dictated by the network architecture.

2.4 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 [36]:

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*) [36]. 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 [63], 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.

Chapter 3

Analysis

In this work, we present a software design which employs cache preheating in order to reduce server response times. The goal of this design is to mitigate the effects of deep CPU sleep modes, as such sleep modes cause the caches to be flushed and power-gated. To the best of our knowledge, these effects of cache flushes on server application performance have not been analyzed in detail before. The development of our software design, however, required a good understanding of the problem: Strong timing and performance requirements made extensive scenario-specific optimizations necessary. Additionally, in some cases, additional knowledge about application behaviour either made significant simplifications possible or uncovered problems that needed to be handled by the preheating solution. At the same time, cache preheating can significantly profit from hardware support for various cache management techniques.

In this chapter, we start at a generic cache preheating system (Section 3.1) and we describe how this system is affected by the choice of hardware (Section 3.2). We also describe how the expected workloads interact with the hardware, and we show the resulting consequences for cache preheating (Section 3.3).

3.1 Situation and Requirements

After the CPU has entered a deep sleep mode, the next incoming network packet wakes the CPU up and triggers the server application. The server application then has to transfer the working set from RAM back into the cache. Cache preheating can reduce the response time by loading the working set into the cache in advance. Although there are many ways in which cache preheating can be implemented, we expect all potential solutions to follow the scheme shown in Figure 3.1.

In all cases, the system has to predict the future working set first. Afterwards, whenever the system wakes up from a deep CPU sleep state, the predicted working

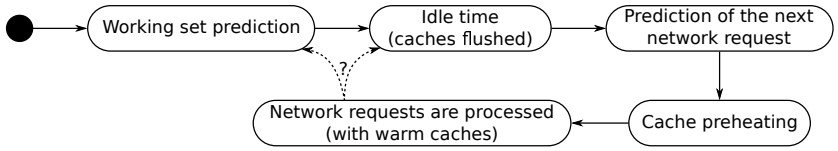


Figure 3.1: Generic steps which are required to preheat the caches for a single network requests. The dashed lines mark the possible transitions when the system becomes idle after the first network requests has been processed: Working set estimation does not have to be repeated for every network request, and our design skips it for all but the first requests.

set is loaded into the caches. The system can be woken up either after a network packet arrives, or it can be woken up in anticipation of the packet, so that the cache is warm when the packet arrives. The former significantly reduces the effectivity of cache preheating, as the server application is delayed while the working set is loaded into the cache. Therefore, we expect that all practical cache preheating solutions employ anticipatory wakeups and require a method to predict future incoming network packets. Before the predicted packet arrives, the system wakes up the CPU and loads the working set into the cache. When the cache has been preheated, the server application resumes execution and processes the incoming network packet. After the server application has processed one or more network requests and when the system is idle again, the cache preheating system can either reuse the previously computed working set to preheat the cache for future packets, or, as shown in Figure 3.1, it can repeat working set prediction to refresh the information.

Although every cache preheating solution encompasses the steps described above, each step can be implemented in various different ways. The specific design and implementation described in this thesis are based on a number of basic requirements which we have identified. As we show in the following sections, the generic requirements listed below significantly restrict the design space presented in the beginning of this section.

- **Decreased response times:** The main goal of a cache preheating system like the one described in this thesis is to reduce the tail latency of the server application. Especially, cache preheating should ensure that the caches are already warm whenever a network request arrives, so that the request can be processed with reduced response time. At the same time, cache preheating must not have any negative effects on tail latency. Such effects can be caused by any computational overhead connected to cache preheating. For example, working set estimation can be computationally expensive and

must not delay the response to any network requests arriving in parallel.

- **Reduced power:** Deep CPU sleep states are used to reduce the power dissipation of the CPU. Cache preheating compensates negative effects of sleep states on the system's latency, but the preheating code itself uses additional energy. Overall, the usage of cache preheating must not result in a significant net power increase. Ideally, the cost of cache preheating is offset by the server application which is able to process requests with increased efficiency due to the reduced number of cache misses.

- **Robustness:** Any cache preheating system must be robust in the presence of other hardware components which can potentially access memory (e.g., DMA controllers). Depending on the architecture, memory is temporarily treated as being non-cacheable by the operating system to enforce coherency between the CPU's and the components' view of memory. If not implemented carefully, preheating can potentially violate that promise of certain data not being loaded into the cache.

- **Application independence:** Ideally, the cache preheating system should be compatible to existing software. Modern systems consist of large amounts of code, and any requirement to modify parts of the software (e.g., to insert manual prefetching hints) will pose a significant hurdle to adoption. Especially, this requirement means that any kind of working set estimation needs to be performed dynamically at runtime by the operating system, because the memory layout of applications is partially unpredictable.

- **Commodity hardware:** Compatibility to existing hardware is similarly important as compatibility to existing software. Ideally, all mechanisms should work efficiently on current commodity hardware.

3.2 Hardware Support

Any cache preheating solution requires an estimate of the application's working set in order to decide which data is loaded into the cache. Usually, the working set is used to provide efficient page replacement [21] in a system which performs memory overcommitment. Working set estimation is therefore implemented with page granularity. Cache preheating requires a more fine-grained working set estimate, as data is loaded into the cache with cache line granularity, and any larger granularity would result in wasted memory throughput during preheating. We have identified three methods which can be used to track memory usage with finer granularity:

- **Tracing cache misses:** On some systems, the CPU can record all cache misses and the corresponding accessed memory locations. For example, recent Intel CPUs provide Processor Event-Based Sampling (PEBS) [37, 18-97]. PEBS can be configured to monitor a wide range of CPU events (e.g., cache misses). It counts the selected event and stores a copy of most CPU registers in a buffer whenever the counter reaches a configurable threshold. In addition to the CPU registers, PEBS can also store the accessed memory address for memory-related events [37, 18-66]. When configured to store a copy of the registers as well as the accessed address whenever a cache miss occurs, PEBS can in theory be used to trace all cache misses.

  In practice, however, one hardware limitation makes precise PEBS-based working set estimation difficult with current CPUs: Whenever PEBS records a cache miss, it loses information about cache misses which happen directly after the recorded cache miss. This loss of information likely stems from the fact that PEBS is implemented as a microcode assist which, in microcode, flushes the CPU pipeline before recording the register state. When the pipeline is flushed, the instructions which trigger the next cache misses are aborted, but the corresponding memory accesses are already executed. When the instructions are repeated, they therefore do not produce cache misses anymore because the data has already been loaded into the cache [40].

  Additionally, PEBS causes rather large overhead, because many CPU registers are saved whenever a cache miss is recorded [37, 18-66], even though the working set estimation code only requires the accessed memory address. Although other tracing mechanisms with lower overhead and better precision are certainly possible, we are not aware of any such mechanism being already available in current CPUs.

- **Analyzing the cache state:** As an alternative to cache miss tracing, the working set can also be estimated by analyzing the cache contents. Data in the cache is addressed by its index. As multiple memory addresses map to the same index, the cache additionally stores the higher bits of the address as the tag of the cache entry, so that the address can be reconstructed by concatenating the tag bits with the index. Therefore, the contents of the tag bit memory 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 contain support for the RAMINDEX register [5], 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 com-

plete and, unlike any simple PEBS-based tracing mechanism, does not miss some addresses which have been accessed. In Section 3.3.2, we show that it is safe to assume that there are rarely any capacity or conflict misses directly following a cache flush.

- **Instrumentation:** If no other hardware mechanism is available, the application can be instrumented to record all memory accesses. The original application binary must not be modified, therefore such instrumentation would be introduced by an emulation layer, by interpreting or dynamically translating the executable. An example for such a instrumentation frame work is Valgrind, which uses dynamic translation to provide instrumentation as an acceptable performance loss [45]. Especially, Valgrind already provides the Cachegrind plugin, which simulates CPU caches to profile the memory access behaviour of the instrumented application [19]. This plugin could be extended to record information on all cache misses.

  Instrumented emulation does not require any hardware support and yields a precise trace of all memory accesses. Of all described working set techniques, instrumented emulation is the most complex one though: Although emulating a single user space process is simple, many cache misses are actually generated by the operating system before it switches to the userland application. Emulating the operating system can only be done in a hypervisor, but that hypervisor must neither violate the timing expectations of the operating system, nor must it have a significant impact on the working set of the system while emulation is inactive. We believe that dynamic translation with instrumentation is a very promising technique to generate precise memory access traces. However, due to the complexity of the task, this thesis presents a conceptually more simple design based on cache tag bit analysis.

3.3 System and Application Behaviour

The presented design is not only affected by the available hardware. It is also tailored towards a specific class of applications: We envision cache preheating to be useful particularly for latency-critical server applications which are executed on server systems. We assume certain workload properties, as other types of applications (e.g., long-running data processing applications or applications without latency requirements) do not profit from cache preheating. The assumptions can significantly simplify the design and make it more efficient. In the following sections, we present the most important assumptions and show their effect on a cache preheating solution.

3.3.1 Only the Last-Level Cache Matters

Cache preheating mitigates the effects of cold caches on a server's response times by loading the working set and similar required data into the caches before a network request is processed by the server. As shown in Figure 2.1, a modern system contains many different types of cache memory with different access characteristics: Caches near to the CPU core tend to be very small and provide low-latency access, whereas larger last-level caches are placed further away from the CPU core and are significantly slower. Several additional small caches are placed in various parts of the CPU core to reduce the likeliness of pipeline stalls in certain scenarios (e.g., branch prediction caches or the translation lookaside buffer).

Flushed Caches	Latency	Relative Overhead
None	337.9 μs	
TLB	339.2 μs	0.4%
L1 (instructions)	337.5 μs	-0.1%
L1 (data)	339.1 μs	0.4%
L1, L2	516.8 μs	**53.0%**
branch prediction	338.0 μs	0.0%
all	526.5 μs	**55.8%**

Figure 3.2: Overhead for nginx when various types of caches are flushed. The system does not have any L3 cache, so the L2 cache is the last-level cache. Only flushes of the L2 cache seem to have any significant effect.

Our analysis shows that it is sufficient to focus on preheating the last-level cache, as flushes of all other caches have negligible impact on performance. We have measured the latency of the nginx web server for a static website after types of caches have been flushed (Figure 3.2). The latency is significantly increased if all cache levels down to the L2 cache are flushed, whereas TLB flushes and L1 cache flushes do not appear to have significant effect on the response time. This observation is easy to explain: As described in Section 2.1.2, the cost of a cache flush depends on the size of the cache as well as the resulting cost of cache misses, and the last-level cache is particularly large and accesses to external memory are particularly expensive.

The focus on the last-level cache significantly simplifies the implementation of the cache preheating code. Whereas regular load instructions can be used to preheat the last-level cache, other types of cache would have required more complex preheating methods. For example, loading code into the L1 instruction cache, in the absence of any specialized cache management engine, would require an ex-

pensive series of branches and hardware breakpoints to execute one instruction from every affected cache line in order to load the code into the instruction cache.

3.3.2 Large Last-Level Caches

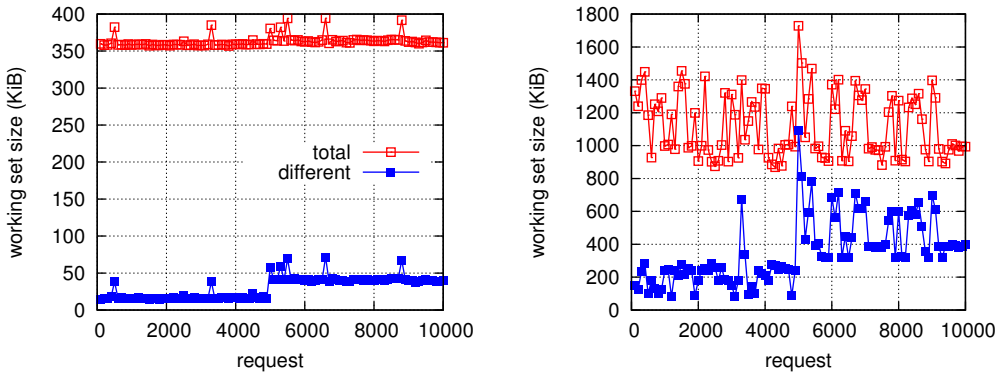
In the previous section, we have shown that only the last-level cache needs to be preheated. During working set prediction, the preheating design therefore analyzes the application's memory access pattern and constructs a list of the memory locations which are loaded from external memory. As described in Section 3.2, cache tag bit analysis is the most promising method for working set prediction. However, cache tag bit analysis is significantly affected by conflict and capacity misses, as every such cache miss replaces a cache line from the cache, so the corresponding memory location is missed when analyzing the cache state. The predicted working set needs to be as complete as possible, though, as any missing memory location can cause a potentially expensive cache miss.

We show that, despite their negative effects on working set prediction, neither conflict nor capacity misses have any significant impact in practical scenarios, because the working sets of the targeted server applications generally fit into the last-level cache: The targeted server applications usually perform rather simple tasks (e.g., serve static or dynamic web pages, or query databases). We have analyzed the memory access pattern of several applications (nginx, memcached, MariaDB) while they process a network request. To do so, we have extended the cache profiler Cachegrind [19] with code to flush the simulated caches before the network request is created and to log all subsequent cache misses. For every cache miss, our code also logs whether another cache line was evicted from the cache. We have configured Cachegrind to simulate a system with 4 MiB last-level cache. None of these applications causes any significant number of cache lines to be evicted the last-level cache.

3.3.3 High Temporal Correlation of the Working Set

Working set prediction is an expensive operation. Ideally, the resulting working set description can be reused to preheat the caches many times, thereby amortizing the prediction costs. Such reuse, however, is only possible if the working set is highly temporally correlated and each invocation of the server application has approximately the same working set.

We show that the working set is highly temporally correlated by submitting 10000 requests to a server application and measuring the working set of each request. Before every network request, we flush the cache, and after the request, the client reads the contents of the cache tag memory to retrieve the memory locations which have been loaded into the cache. Each resulting working set estimate



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

Figure 3.3: Total working set size of every 100th request submitted to the server application, compared to the size of those memory locations which are not accessed by the first request. The server application is restarted after approximately 5000 requests. The working set of the MariaDB benchmark varies significantly, because multiple different types of requests are processed. Even in this case, the effect of the server restart on the working set is clearly visible, though.

is compared to the working set of the first request. We compute the number of cache lines which are missing in the working set of the first request but have been used to process the later request.

We conduct the test for the nginx web server serving static web pages and for the DBT-2 MariaDB database workload (see Figure 3.3). In the case of the nginx web server, less than 5% of the working set is different between two arbitrary requests. This value is fairly constant and does not depend on the temporal distance between the requests. The majority of the working set is constant even over long timeframes. The working set of the MariaDB workload shows significantly more change, mostly because the benchmark executes various different types of database queries which access different data and cause different code paths to be taken. Even in this case, on average more than 80% of the working set of different requests overlap, though, and growing temporal distances between requests do not result in less correlation.

The benchmarks show that the working set prediction can be reused, although care must be taken to avoid preheating those parts of the working set which are different for every request. Those parts cannot be well predicted, so cache preheating is useless for them. All other parts of the working set, however, can be assumed to be constant.

3.3.4 Lower Temporal Correlation over Larger Time Frames

In the last section, we showed that the working set of common server applications is mostly constant. In our experiments, we did not observe any phase transitions which cause abrupt change to the working set. Such phase transitions are possible, though, as are slow changes to the working set over time.

As an example for slow change to the working set, the operating system could continuously defragment physical memory in order to introduce huge page mappings [52]. Additionally, many server applications maintain data structures in memory which represent the state of the server. Over time, these data structures are manipulated and might be moved around in memory by the server application. For example, a database server might keep parts of database indices in memory, these indices are modified whenever rows are inserted into or removed from the database, and each modification can potentially change the working set of the application. Because such slow change is hard to detect, we propose that working set prediction is periodically repeated to limit the negative effect of the changes.

Sudden phase changes are easier to detect if they are triggered by an external event. For example, the server application could simply be restarted, for example due to software updates, in which case its location in physical memory changes: Even a slightly different initialization sequence can cause vastly different memory allocator behaviour. Additionally, the OS can place mutable sections of the program's executable file at different physical addresses, even if their virtual address does not change during the restart.

To measure the influence of application restarts on the working set, we repeat the test from the last section with nginx, but stop and restart the server application in the middle of the test. As seen in Figure 3.3, the server restart introduces 7% change to the working set of nginx. Similarly, in the MariaDB benchmark, the average difference to the working set of the first request is more than doubled. Because this change significantly affects the effectiveness of cache preheating, our design contains a method to detect server restarts (Section 4.2) and renews the working set prediction after a restart.

3.3.5 Overlapping Working Sets

Although the working set of a single application shows a high degree of temporal correlation, the working sets of different applications are usually radically different. Even different server applications share some parts of the working set though. For example, the operating system's network stack is usually present in all working sets, and different applications might link to the same share libraries. This overlap in the working set is problematic for working set estimation: As we will show in Section 4.1.2, the working set estimation needs to be able to isolate a

single network application, and any overlap between the working sets makes this isolation more difficult.

To show that such overlap is significant even among radically different web services, we execute an instance of the nginx server and a MariaDB database server on the same system and submit network requests to both. We read the cache contents after both requests and compare the working set. Our experiment shows that 31% of the combined working set of the two applications is accessed by both applications. Running the applications in two separate virtual machines can slightly reduce the overlap, because each application then has its own separate network stack. However, the network stack of the hypervisor still generates significant overlap between the two working sets.

Chapter 4

Design

In this work, we show how cache preheating can reduce the latency overhead caused by deep CPU sleep states. The general idea is to wake the CPU up before a network packet arrives and to load the estimated working set of the server application into the cache. In Section 3, we have already sketched the structure of a generic cache preheating solution and have identified generic requirements as well as a number of properties of the targeted workloads. In this section, we describe a specific software design which implements cache preheating and fulfills the requirements listed above.

The resulting design consists of two parts described in Section 4.1: One part is a program which analyzes the cache usage of the server for one or more network requests (Section 4.1.1 to 4.1.2) and then uses that information to speed up subsequent requests by preheating the caches (Section 4.1.5). The caches have to be preheated in advance though, so the other part is a network arbiter which announces future incoming network requests to the server (Section 4.1.3 to 4.1.4). The remainder of the section then focuses on additional features: Our proposed cache preheating solution is able to detect server restarts (Section 4.2) and can also be extended for the use with other types of wakeup events besides incoming network packets (Section 4.3).

4.1 Components

The design which results from the requirements and assumptions listed above operates in two phases, as shown in Figure 4.1:

- **Working Set Prediction Phase:** The preheating system estimates the working set of a network application by analyzing the accessed memory locations after several requests have been processed by the application. The system

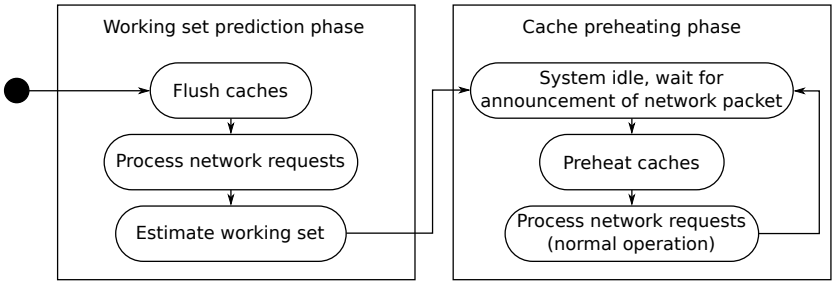


Figure 4.1: The two phases of our cache preheating solution: First, the working set of the server application is estimated, then the estimate is used to preheat the caches for all following network requests. Incoming network packets are announced by an external component (described in Section 4.1.3).

retrieves these accessed memory locations by reading the content of the cache tag RAM and calculating the physical addresses from the tag bits.

- **Preheating Phase:** As soon as a good working set estimate is available, the system switches to regular operation. In this phase, the system predicts future network requests through the help of a central network arbiter. Whenever the arrival time of the next incoming network request is known, the system wakes up in anticipation of the request and loads the estimated working set into the caches. The system switches back into the training phase when it detects that the server application has been restarted.

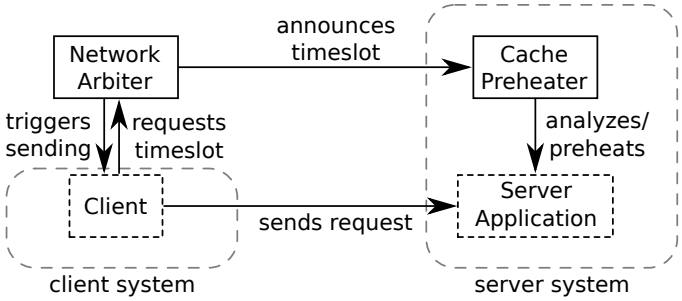


Figure 4.2: Components of the preheating solution and their interaction with the network service

The resulting software architecture is simple and consists of two major components, as shown in Figure 4.2: On the server system, the operating system is extended with a component which estimates the server application's working set and preheats the caches whenever a network request is announced. This announcement is implemented by another component, the central network arbiter, which is extended to predict future network packets and to notify the server system.

Because our design is supposed to work with arbitrary unmodified network applications, neither the server application nor the client application are modified. In both cases, the system is designed to work with arbitrary existing software, and software changes are limited to the network and the operating system of the server.

4.1.1 Working Set Estimation

The first execution phase of our preheating system is the working set prediction phase, because cache preheating requires a fine-granular description of the memory locations which are supposed to be loaded into the cache. The working set is only estimated once, and the result is reused throughout the cache preheating phase, because, as shown in Section 3.3.3, all network requests to the same server application have approximately the same working set.

Traditionally, working set estimation is conducted with page granularity. Access to pages is detected either via page faults or via hardware access bits. However, these mechanisms cannot be used if cache line granularity is required. Instead, the information about the working set can be either generated by a pure software solution, or it can be provided by the memory controller or the caches, as these are the parts of the CPU which operate with cache line granularity. In Section 3.2, we have identified several mechanisms which can be used to provide fine-grained information about the current working set.

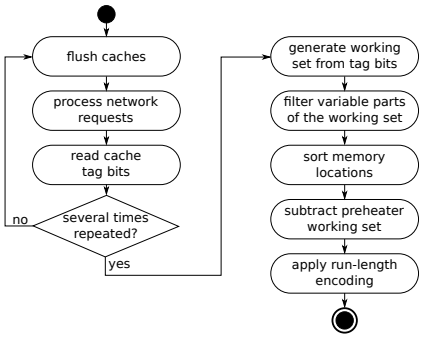


Figure 4.3: The various steps of our working set estimation phase: First, our software analyzes the cache state to compute a list of all memory locations accessed by the server application (left side). Then, the software postprocesses this estimate of the working set to improve preheating performance (right side).

Of these mechanisms, analysis of cache tag bits provides the easiest method to determine the working set of the system. Figure 4.3 shows how our design uses snapshots of the cache state to determine what memory locations are used

while the server application processes a network request: First, the caches are flushed before the network request arrives, in order to exclude all irrelevant cache lines from the results. Then, the network request arrives and is processed by the server application. Afterwards, our design loops over all cache lines and reads the corresponding tag bits. The tag bits are then concatenated with the index of the corresponding cache line and converted into physical addresses. If there are no significant numbers of conflict or capacity misses during the working set prediction phase, the resulting list of physical addresses is a superset of the system's working set.

However, the resulting cache state dump is not yet a good basis for cache preheating. Especially, it will contain a significant number of cache lines which do not need to be loaded into the cache by the preheating code, either because they are only infrequently (or never) required by the server application or because they are already in the cache when the preheating code is executed. Such memory locations must not be included in the resulting working set description, because loading them does not provide significant gain, whereas the memory accesses significantly increase the cost of preheating. As shown in Section 6.4, this preheating cost is critical.

To make sure that the result does not include more memory locations than accessed by the server application, our design disables any kind of hardware prefetching during the working set prediction phase. Otherwise, the cache would contain data which has been loaded by the hardware prefetcher but which is not in the application's working set. Disabling the hardware prefetcher can cause significant performance loss. However, the working set prediction phase can be kept rather short, and the resulting working set estimate can be reused for many requests, so the temporary performance loss does not have any significant effect on the overall performance. As a further improvement, we suggest an extension for the hardware prefetcher which marks cache lines which have been fetched by the hardware prefetching unit but have not been used by the application. Such marking can enable the working estimation code to distinguish whether a cache line was actually used by the server application, so prefetching does not need to be switched off. We are, however, not aware of any available CPU architecture which has such a hardware feature.

Even after the hardware prefetcher has been disabled, the resulting working set estimate contains unnecessary cache lines and can otherwise be optimized. Therefore, our design performs a number of post-processing steps to improve the quality of the predicted working set:

- **Removing highly variable parts of the working set:** Apart from hardware prefetching as described above, we have identified three important reasons why unnecessary cache lines are included in the dumped cache state:

First, the operating system scheduler can preempt the server application and schedule a different process with a distinct working set, and that process can pollute the cache. Second, the server application can process network requests of different types, and each type of request has a different working set, so some memory locations are not accessed by other types of requests. These locations are therefore unnecessary when the cache is preheated for these types of requests. Third, and most importantly, some data (e.g., network buffers) is placed in a different location for each network request.

In all these cases, our cache preheating solution purges all varying memory locations from the working set estimate by performing the analysis repeatedly on several consecutive network requests. The resulting cache dumps are intersected, and all memory locations which are not found in all cache dumps are removed from the final working set estimate.

- **Preventing memory locations from being loaded twice:** In our design, the cache preheating code is triggered by an announcement from a central network arbiter. If this announcement is sent via a network packet, then the CPU has already traversed the OS network stack by the time cache preheating is started. Therefore, the network stack as well as other parts of the OS (e.g., interrupt handling) are already in the cache at that point in time. Most likely, the locations are also part of the working set of the network application, so the preheating code tries to load them again. Although no second access to external memory occurs, the second load instruction increases the computational complexity of preheating.

  Therefore, for the first network request to be preheated, the system once dumps the cache state from within the preheating code instead of preheating the caches. This cache dump contains the cache lines which have already been loaded when the cache preheating code is executed, so the memory locations are removed from the working set estimate which is used for preheating.

- **Sorting memory locations by increasing physical addresses:** After the working set has been filtered as described above, it consists mostly of memory locations which are actually required by the server application. The order of the memory locations is rather random, though. Our design sorts the working set by increasing physical addresses to reduce the time required for preheating. The preheating code linearly iterates through the list of memory locations to be loaded, and the resulting sorted access pattern is significantly more efficient.

  Generally, DRAM is organized into many rows, and one row is held in the row buffer at a time. An access to a different row causes the content of the

row buffer to be written back into the memory matrix, and a different row is read into the buffer instead [51]. A sorted memory access pattern causes significantly less switching between rows compared to a random access pattern, because the spatial locality of the accesses is increased. As a result, the memory throughput is increased.

- **Compression of the working set description:** In order to maximize the memory throughput available for loading the working set, we must minimize the parts of the cache preheater's cache footprint which are not part of the application's working set. For example, the description of the working set is metadata which is loaded, but which is not required by the server application. As the working set description is highly regular (all addresses are aligned to cache line boundaries and are sorted), compression can significantly reduce the size of the data. Complex compression techniques, however, require significant amounts of decompression code in the preheater and increase the cache footprint again.

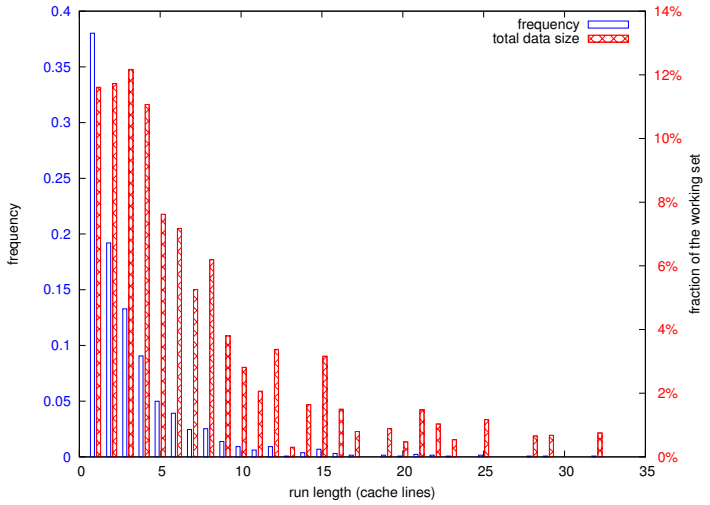


Figure 4.4: Relative frequency and combined data size of runs of different length in the working set of the nginx web server. Although very short runs are more frequent, longer runs describe a larger amount of data.

We have opted for a very simple run-length encoding scheme instead, as it provides very cheap on-the-fly decompression of the data, does not require any data structures which increase the cache footprint (e.g., dictionaries), and still achieves a good compression ratio. Most of the time when an application touches one cache line, it also touches surrounding cache lines, so memory locations in the working set description often form long runs of

consecutive cache lines. Figure 4.4 shows the distribution of the length of these runs for the working set of the nginx web server and a simple HTTP GET request. On average, 3.3 cache lines form one consecutive run. In our design, the working set estimation code counts the number of consecutive addresses and encodes this number into the least significant bits of the first address to maximize the compression ratio. These bits are guaranteed to be zero, because all addresses are always aligned to cache line boundaries. Each run of consecutive addresses is therefore packed into 4 bytes, and the size of the working set description of the example is reduced by 69%.

4.1.2 Isolating Single Network Requests

In the last section, we assumed that, as shown in Figure 4.3, the cache contains the memory locations of exactly one network request. In practice, the operating system might not have enough information to isolate a single request to a single network application, though. Multiple server applications might be executed in parallel, and multiple clients might send requests to the same application in parallel. Without application-specific knowledge, it is difficult for the OS to know when requests begin and end, and it is difficult to isolate the memory accesses of a single network request or a single network application. However, we argue that it is not necessary to distinguish between multiple network requests of one single network application, and we present mechanisms to differentiate between the memory accesses of multiple server applications which are running in parallel.

Multiple Active Server Applications

If multiple server applications are processing requests in parallel, the working set prediction method presented in the last section will generate the combined working set of the applications, because the cache is shared by all applications. As a result, too much data is loaded whenever the cache is preheated. As shown in Section 3.3.5, the overlap between different applications is significant, even if they applications are placed in different virtual machines. Any working set estimation based on cache content analysis would therefore only result in the combined working set.

In this work, we expect a system which only executes one single type of application. Although such a setup is common in some cases where other concurrent applications would reduce the performance of the executed application (e.g., dedicated cache servers or data storage servers), it often results in reduced utilization of the available resources (and therefore increased cost) compared to a setup where multiple different applications share one physical system. We therefore provide several potential solutions for future evaluation:

- **Physical separation:** One practical solution to isolate a single server application is run it on a core with a separate cache, so that all data in that cache was actually loaded by the server application. Other server applications can continue to run on cores with different caches without affecting working set estimation. The hardware platform of our prototype, the Samsung Exynos 5422, has such two separate L2 caches where each cache is used by a different set of cores [9]. We use this hardware feature to separate the benchmark client from the benchmark server. We could use a similar setup to distinguish between the working sets of two server applications. However, with the exception of multi-socket NUMA systems, real server systems might not cluster cores around more than one last-level cache. Therefore, such a technique is not universally applicable.

- **Other working set estimation techniques:** Trace-based or simulation-based working set estimation techniques (see Section 3.2) can be designed in a way in which they ignore all but one server application. For example, a simulator-based design can be configured to record memory accesses done by either the network stack or one server application, or a trace-based technique can trace all L1 cache misses from the server application's CPU core. Although these approaches are difficult to implement, they can likely produce a clean estimation of the working set of a single server application.

Multiple Requests to One Server Application

Even if the system can isolate server applications from each other, the working set prediction mechanism still does not know when the analyzed network request ends. There is little harm from combining the working sets of multiple subsequent requests into one working set estimate, though, because the working sets of the requests are basically identical. As described in Section 3.3.3, the working set of a single network service shows a high degree of temporal correlation. Therefore, after one request has been processed, further requests do not cause significant amounts of additional cache misses and therefore do not significantly change the cache contents. Some parts of the working set, however, are specific to a single request. For example, the operating system potentially places each incoming network packet at a different location in memory.

We measured the cache footprint of the nginx web server after executing one single request for a static web page and after executing 10 identical requests. On average, the cache footprint was 20% larger when multiple requests were processed before the cache state was analyzed. To measure how much of this increase can be attributed to data which is specific to a single request, we repeated the experiment, but applied additional filters to the result: Each cache footprint

analysis was repeated 10 times, and only memory locations were counted which appeared in all repetitions. Under these conditions, the difference is significantly smaller, and only 6% additional cache lines are included in the working set estimate when multiple requests are processed during working set estimation. We expect this overhead to be acceptable, therefore the working set estimation code does not need to distinguish between different network requests to the same server application. Instead, the cache usage can be analyzed either when the CPU is idle again or after a conservative time span has elapsed after the initial packet has been received, even when multiple network requests have arrived after the initial request.

4.1.3 Prediction of Future Events

When a good description of the server application's working set has been created, the system switches from the working set prediction phase to the preheating phase. In this phase, the system wakes the processor cores up before a request arrives and loads the working set into the cache. This behaviour means that the preheating code has to know in advance when network requests are going to arrive.

Previous work has shown that such predictions are possible for other types of events (solid state disks, GPUs), but network requests have generally been excluded [64]. Responses from network services with low response time jitter are likely well predictable. In general, incoming network packets cannot be predicted, though, because they are issued independently by other systems, outside of the control of the system trying to preheat the caches.

If the network performs access control via a central arbiter though, this central arbiter can provide information about future network packets. As described in Section 2.2, such network architectures have been proposed for low latency traffic in data center networks and show superior network latencies compared to networks with traditional decentralized packet switching and flow control.

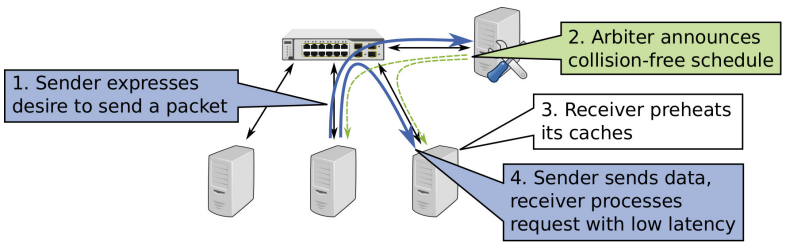


Figure 4.5: Future network packets are announced by the network arbiter in advance, so that the receiver can preheat the caches in anticipation of the packets.

Figure 4.5 shows how the central arbiter can also be used to predict future network packets. The figure shows the steps which happen when a sender system wants to send a network packet to a different target system. First, as always in such systems, the sender asks the arbiter for a timeslot to send a packet. The arbiter then schedules all such requests in a way that no collisions between different packets occur. The resulting schedule is sent to the sender, which then creates a timer to send the packet. Deviating from the regular behaviour in such network architectures, the arbiter however also sends the schedule to the target system, as an announcement of future incoming packets. When the target system receives the schedule, it can activate a sleeping CPU core and preheat its caches. By the time the packet reaches the target system, the CPU is fully awake with warm caches and can process the network packet with low latency.

If cache preheating is still in progress after the arrival time of the packet, the server application is delayed by the preheating code, and the effectiveness of cache preheating is reduced. Therefore, this design causes cache preheating to be highly time-critical, because the time between the announcement of the packet and its arrival time is very limited. As the presented network architectures are designed especially for low-latency traffic, packets must be scheduled as soon as possible after the sender system has requested a time slot. For example, Fastpass calculates schedules only 65 microseconds in advance [47].

4.1.4 Detecting the Targeted Server Application

Although the technique described in the last section can predict the arrival time of future network packets, it can not yet identify the targeted server application. However, depending on the server application, a different working set needs to be loaded into the cache, and in Section 4.1.2, we have presented a working set estimation mechanism for multiple concurrent applications. Therefore, the system must not only predict the time of the request, but also its type.

In existing centrally arbitrated networks, the announcement of a network packet usually only contains identifiers for the source and target systems [10]. The announcement does not contain any information which can be used to identify the type of the request. We propose that the network architecture is modified in a way that the announcement also contains the destination port of the TCP or UDP packet. Whenever a system needs to send a packet, it sends a request with the destination address and the destination port to the central arbiter, and the arbiter includes that port number when announcing the packet's allocated timeslot to the destination system. Depending on the port number, the preheating code then selects the working set of the corresponding server application and loads it into the cache.

4.1.5 Cache Preheating

Once an incoming network packet has been announced, the cache preheating code iterates through the run-length-encoded working set description and loads the working set into the cache. At this point, the memory access pattern has already been optimized as part of the working set estimation code (Section 4.1.1). The goal of the cache preheating code is therefore to execute this memory access pattern in the most efficient way possible. The resulting optimizations are very architecture specific, so we describe them in the implementation chapter (Section 5.3).

4.2 Detecting Server Restarts

As described in Section 3.3.4, the working set of a server application can gradually change over time, and there can be abrupt phase changes. We have not observed the former in practice, and most variable parts of the working set are already filtered out by the working set post-processing described above. In cases where the working set is gradually degrading, we propose that the working set is periodically refreshed to reduce the effect of such gradual change.

Unlike gradual working set changes, sudden phase changes are easier to detect, because they are usually triggered by external events. Most importantly, server restarts induce sudden change to the working set. However, server restarts are also rather easy to detect, even without any knowledge about the internal workings of the server application. A number of resources are acquired when the server application starts, and are released when the server application is stopped. Examples for relevant resources are processes, threads, network sockets or memory. The resources are allocated by the operating system, and so the operating system can analyze resource usage to detect server restarts.

Detection of restarts is made difficult by the fact that different server applications use drastically different architectures, and each server architecture shows different resource usage patterns. Our design therefore monitors the state of the server application's network sockets, because network sockets are the only resource which is used in a similar fashion by all common server architectures, whereas the number of processes and threads varies wildly [29].

For example, servers often maintain one process per connection, especially on Unix systems. Such multi-process servers can either fork the main process whenever a new connection is established, or a pool of processes can be initialized on startup. Even in the latter case, a high number of connections can require additional processes to be created. When a connection is closed, the corresponding process is then potentially destroyed to reduce memory usage. Similarly, multi-

threaded servers can either spawn a new thread for every incoming connection, or a pool of threads can be maintained. Therefore, for these types of server applications, neither processes nor threads can be used to identify server restarts.

The main network socket of a server application is usually created at startup, though, and is destroyed when the server application is stopped. Therefore, the socket provides a good indicator for server restarts. Our design assumes that the server application has been restarted whenever the preheating software notices that the server application's socket is destroyed. First, however, the system needs to establish a mapping between sockets and server applications. In our design, this mapping is determined during the working set prediction phase. In this phase, the preheating software memorizes the port number of the network requests which trigger an analysis of the cache contents. The port numbers are stored along with the resulting working set. Later, during the preheating phase, the system is notified by the operating system whenever a socket is destroyed. In that case, the preheating software compares the port number of the destroyed software to the recorded port numbers. If the port numbers match, the system returns to the working set prediction phase to regenerate the working set estimate.

Although this design is able to reliably detect restarts for most common server applications, it does not support applications which utilize socket activation to start the application on-demand when a network packet arrives. An example for such a socket activation system is `inetd` [20]. The `inetd` „super-server“ uses `select` (and, optionally, `accept`) to wait for incoming connections and packets. It then spawns a new server processes which processes all requests from the new connection, or—in the case of datagram sockets—processes all future packets arriving on the original socket [55, p.376]. In both cases, `inetd` maintains a handle to the original socket, so the socket can outlive the server application. Therefore, socket destruction is not a good indicator for a server restart. The situation is however special in that there is only one service like `inetd` which provides automatic restarts for many different server applications. Therefore, only one single service needs to be modified to manually notify the operating system about server restarts. Although such modifications are a clear violation of our requirement for the preheating system to work with unmodified server applications, the fact that only one application (inetd) needs to be patched makes such changes viable.

4.3 Other Types of Events

Above, we have described how cache preheating can be used to reduce the response time to network requests. Such a technique is not limited to network applications though. Similarly to how Zhu et al. use anticipatory wakeups mitigate the

effect of sleeping CPUs on the response time to various other types of events [64], cache preheating can be employed to reduce the effect of cold caches.

Cache preheating, just like anticipatory wakeups, only requires the source of wakeup events to be well predictable. Zhu et al. have shown that HDD transfers and GPU kernel invocations are well predictable, so cache preheating can likely improve the performance of the corresponding applications. The only limitation of cache preheating in these situations is that not all these asynchronous tasks take long enough that aggressive power gating of the CPU is power-efficient

Chapter 5

Implementation

In the last chapter, we have described a design which preheats the CPU caches before a network request arrives, in order to reduce the effects of cold caches on the response time of network applications. We have performed our evaluation with a prototype based on Linux and the Odroid XU3 single board computer. In this chapter, we describe this implementation, with a focus on those parts which are specific to the underlying hardware and operating system.

In Section 5.1 we describe our choice of hardware platform and the resulting limitations of our prototype. We show the resulting structure of the prototype in Section 5.2. In the following sections, we then describe how we implement and optimize cache preheating (Section 5.3, how we ensure that DMA transfers are not affected when data is loaded into the cache (Section 5.4 and how we detect server restarts in the Linux kernel (Section 5.5).

5.1 Hardware Platform

As shown in Section 3.2, current hardware platforms provide two basic methods to create a fine-grained working set estimate. The two possibilities are to read the contents of the cache tag RAM (supported by many modern ARM processor cores) and to trace all cache misses (supported by current Intel processor cores). As described in that section, our current design focuses on direct access to cache tag RAM as the method for working set estimation. Therefore, we based our prototype on a system with an ARM processor, the Hardkernel Odroid XU3.

The Odroid XU3 is a single board computer designed around the Samsung Exynos 5422 system-on-chip [9]. The system contains two clusters of CPU cores, one with four ARM Cortex-A15 cores and one with four Cortex-A7 cores. Of these cores, however, only the Cortex-A15 cores support direct access to cache memories through the RAMINDEX register, so our cache preheating prototype

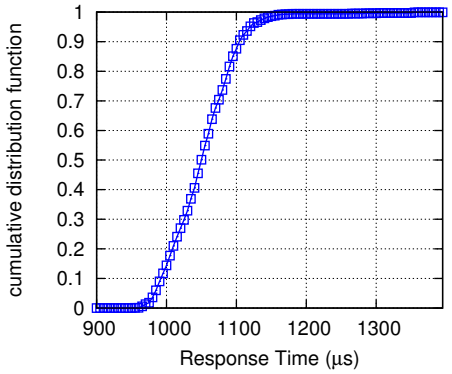
is limited to these cores. Each core contains 32 KiB L1 instruction cache and data L1 cache. The two clusters each contain their own independent unified L2 cache (2 MiB for the Cortex-A15 cores, and 512 KiB for the Cortex-A7 cores) [9]. There are two properties of the system which limit the scope of our prototype:

- **No advanced power management support:** Although the Exynos 5422 SoC was originally designed for smart phones and provides advanced power management functionality [6], we could not configure the system to flush its caches during idle periods. The kernel provided by Hardkernel does provide functionality to disable single cores or whole clusters of cores though. By disabling all Cortex-A15 cores, the power usage of the system can be significantly reduced. Our experiments show that in this situation the cache seems to be flushed. These findings are consistent with coarse-grained power gating of the cores as well as the cache, which is exactly the scenario targeted by our cache preheating design.

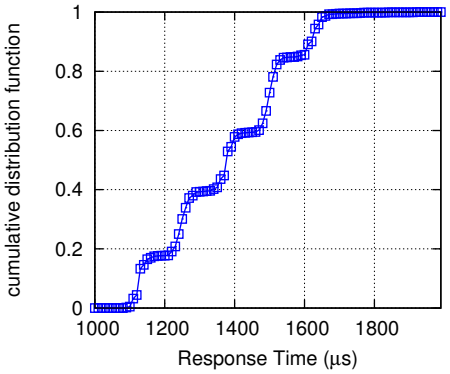
  However, the difference between this simple power management mechanism and the sleep modes found in current server and desktop CPUs is significant: Our experiments showed that reenabling a cluster of cores takes more than a millisecond, whereas current desktop and server CPUs usually require much less than 100 microseconds [53]. The available power management techniques are therefore not very representative for future server systems, and the system is not the right system to show interactions with power saving options. We therefore decided to ignore hardware-provided power management and instead simulate deep sleep modes by manually flushing the CPU caches whenever the CPU would have entered a deep sleep mode.

- **USB network adapter:** Like most similar ARM single board computers, the Odroid XU3 connects to the network through a USB network adapter. In various parts of this evaluation chapter, we are presenting response time measurements, because response time reduction is the main goal of cache preheating. The USB bus however is a single-master bus in which only the host controller can initiate transfers, without any means for the USB network adapter to trigger interrupts when network packets arrive [18]. Instead, the host seems to poll the device once per microframe, where each microframe is 125 microseconds long. In a lightly loaded network, the arrival time of packets at the CPU is therefore aligned to 8000 Hz intervals.

  As a result, the response time of the server can only be measured precisely if the packets are synchronized to the USB microframes. Otherwise, the response to the packet will be delayed by the USB bus. To demonstrate the effect, we have created a network latency benchmark in which the network



(a) Cumulative histogram of the response time.



(b) Sum of response time and previous random delay.

Figure 5.1: Cumulative histogram of the round trip time of a simple echo server. The irregular distribution shows the effects of the USB bus.

client wants a random time before sending a network packet. Figure 5.1 shows the resulting response time histogram, with and without the random delay. Figure 5.1a shows that the packets are randomly delayed by up to 125 microseconds, and if add the response time to the previous sleep duration, the resulting distribution shows that the arrival time of the packet is discretized by USB polling (Figure 5.1b).

As it is difficult to synchronize the benchmark client to the USB bus of the server system, the results of latency measurements are basically useless to analyze the effects of cache preheating, as the differences are mostly smaller than the USB microframe interval. Therefore, we do not use physical networks at all, but instead place the benchmark client on the same system as the benchmarked server and the cache preheating system. We use the two processor clusters to separate the client from the server. The two separate L2 caches allow us to flush only the cache of the emulated server system while leaving the cache contents of the client intact.

These two hardware properties and the resulting simplification to the design significantly limit the usability of the prototype. However, the main goal of our evaluation is to show the general viability of the idea of cache preheating. The prototype therefore only needs to show that the basic mechanisms work and are likely suited for the presented scenario. Therefore, realistic power management or networking is not strictly required, as long as its properties are taken into account when analyzing the results (for example, Fastpass only computes schedules 60 microseconds in advance before the network packet is sent, placing a strict timing requirement on preheating).

5.2 Components

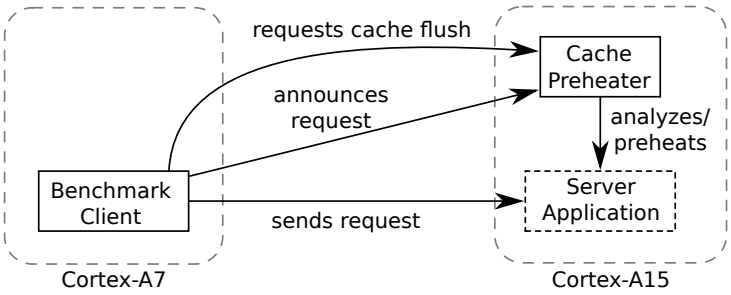


Figure 5.2: Components of our prototype setup. The benchmark client simulates both power gating and the predictable network architecture.

In Section 4.1 we have described the different components of our design. Our prototype follows this scheme, but, as described above, is slightly simplified. Figure 5.2 shows the components of the implementation:

- **Client application:** Because the prototype does not include any predictable network architecture, there is no central arbiter which can announce future network packets to the receiver. Instead, the benchmark client manually sends a UDP packet to the cache preheating server before sending any network request.

  This UDP packet includes the time at which the client sends the next network request. The client generally waits for 500 microseconds between the announcement and the predicted network request in order to give the server some time to preheat the caches. This timespan is significantly larger than what is provided by real predictable network architectures. However, we want to analyze the various effects and properties in isolation. If preheating takes too much time, we still want it to be finished when the network request arrives, in order to measure the potential response time reduction.

  In addition to the timestamp of the following network request, the UDP packet also contains an identifier of the requested service. As described in Section 4.1.4, these identifiers allow the preheating system to maintain a different working set estimate for every network service on the system. The identifiers are statically allocated in the client application. In a real-world setup, where the client application does not announce its requests, the network arbiter will announce the target port to the destination system instead.

- **Preheater:** The cache preheating system is divided into two components. A Linux kernel module implements all features which require elevated priv-

ileges. For example, working set estimation requires access to machine-specific registers, and cache preheating requires access to all of physical memory. A separate userspace program (the cache preheating server) implements a small UDP server and receives packets which announce future requests.

Whenever this server receives a packet announcing a future network request, it first calls the kernel module to flush the caches, in order to simulate deep processor sleep states. Then, the program issues a system call to the kernel module to either estimate the working set or to preheat the caches. Working set estimation is implemented as an analysis of the cache tag bits of the L2 cache with the RAMINDEX register [5].

- **Server application:** Whereas the client application has been modified to announce its network requests, the server application is not modified. It is executed on the same physical system (or, in our setup, the same core cluster) as the preheater, so that the working set prediction code captures the working set of the server application.

5.3 Cache Preheating

As soon as a good working set estimate is available, the system starts to preheat the caches whenever an incoming request is announced. As described in Section 4.1.5, the preheating code simply executes prefetch instructions on all required memory locations. The main challenge is that cache preheating needs to be complete when the network request arrives. Therefore, the achieved throughput needs to be maximized, and the code must not load more data than necessary, so we have implemented a number of optimizations.

- **Removal of high-memory support:** Our first implementation was severely slowed down by the way how Linux kernel code usually accesses physical addresses: Our hardware platform is a 32-bit system, and Linux partitions the virtual address space so that the bottom 3 GiB are accessible to the application, whereas the top 1 GiB is reserved to the kernel. As much as possible of physical memory is mapped into the kernel region at boot, in order to enable efficient access to arbitrary physical addresses [65]. However, the limited size of the kernel region means that only a certain amount of physical memory can be mapped. Therefore, Linux partitions physical memory in low and high memory. Low memory is directly mapped into the kernel region by default, whereas high memory is not mapped. Instead, whenever kernel code wants to access physical addresses in high memory,

the location needs to be manually mapped into the address space with the `kmap_atomic` function [65].

The cache preheating code needs to be able to load the data at arbitrary physical addresses into the cache. Therefore, the first version of our prototype used `kmap_atomic` and `kunmap_atomic` to make individual physical memory frames accessible whenever necessary. Doing so, however, proved to be prohibitively expensive, as each call to these functions causes significant overhead and little time was spent actually loading the working set into the cache.

However, the concept of high memory is only required on architectures with a small virtual address space. Future servers are universally expected to use 64-bit processors though. On these systems, the whole physical memory can be conveniently mapped into the kernel part of the virtual address space. To simulate this situation, and to achieve better cache preheating performance, our prototype configures the Linux kernel in a slightly unusual fashion. The virtual address space is split so that only 1 GiB is allocated to user space applications, and 3 GiB are available to the kernel. As the system only contains 2 GiB of RAM, all physical memory can be directly mapped into virtual memory, and no expensive page table manipulations are necessary while the cache is preheated.

- **Manual unrolling:** Most of the time during cache preheating is spent in two loops: The outer loop iterates through all runs of consecutive locations, and the inner nested loop iterates through all addresses in each of the loop to decode the run-length encoding. After, as described above, all page table manipulation has been removed, only code to read the working set description and to prefetch the working set is left. We have reimplemented the two loops in assembly, and we have manually unrolled the inner loop. Because the run length is encoded in the lowest six bits, the inner loop is repeated at most 64 times, so we have replaced the loop by a sequence of 64 prefetch instructions with different offsets. For each run, the code computes the instruction pointer so that the correct number of prefetch instructions is executed.

- **Parallel preheating:** Recent multi-core CPUs can often only completely utilize the available memory bandwidth when multiple cores are used to parallelize memory access. On our prototype platform, parallel access to memory from 6 cores is 40% faster than access from a single core, as measured with the *pmbw* parallel memory bandwidth benchmark. Because cache preheating is memory-bound, we have implemented a parallel version which

evenly divides the working set description into up to four parts and processes them in parallel on up to four Cortex-A15 cores. We have, however, found that the maximum throughput is achieved with only two cores, as otherwise the parallelization overhead reduces the overall speedup.

5.4 DMA Buffers

As described in the last section, the cache preheating system can load data from arbitrary physical addresses into the cache. There is one situation in which arbitrary access to memory can cause the system to misbehave, because the data is not supposed to be in any cache: If a device uses DMA to write to the memory location, the CPU might continue to see a stale copy of the data if the data is cached. Therefore, in some architectures, the preheating code might not be allowed to fetch data into the cache which is concurrently used for DMA transfers.

The access restrictions vary significantly depending on the architecture though. Current processor architectures and CPUs use several methods to prevent incoherent cache contents during normal operation. For example, x86 processors support the memory type range registers (*MTRR*) to mark segments of physical memory as non-cacheable, or support page attribute tables (*PAT*) to mark parts of virtual memory as non-cacheable [12], whereas other architectures allow the OS to mark individual pages as cacheable through configuration bits in the page tables [2]. However, marking large areas of memory as non-cacheable, in order to allow DMA operations on them, can cause a significant performance hit. Other mechanisms with higher performance include cache-coherent interconnects which automatically clean or flush cache lines when a DMA transaction touches the corresponding memory locations [3] and software-managed coherency where the operating system manually cleans or flushes parts of the cache whenever necessary [1].

In the Linux kernel, this architecture-dependent behaviour is hidden behind the DMA mapping API, and kernel code is supposed to use special functions (`dma_alloc_attrs` and `dma_alloc_coherent`) to allocate memory for DMA transfers [48]. A hook could be inserted into these functions to remove the corresponding parts from all existing working set estimates whenever a new DMA memory region is allocated or whenever the DMA mapping code flushes the caches, in order prevent the preheater from loading data into the cache which is not supposed to be cached.

However, our prototype is only required to work on the ARM architecture, so we chose a much less complex architecture-specific approach: Basically, the prototype completely ignores the problem of (temporarily) non-cacheable DMA buffers, because the underlying software-management coherency strategy already

expects that memory can be randomly fetched into the caches by the processor. The reason for this behaviour is that modern ARM processors implement speculative prefetching (readahead), which builds upon access pattern detection and already predicts the next access and fetches the corresponding data into the cache before the program executes the corresponding load or store instruction. Linux cleans the cache (writeback of all modified cache lines) before any DMA transfer to make sure that modified data is not lost during the transfer. After the DMA transfer has been completed, the kernel additionally discards any cached data [8]. This last step ensures that, whenever the cache might contain stale data because the data has been fetched into the cache before the DMA transfer has modified the corresponding memory locations, this stale data is discarded before the CPU can access it. Because of this mechanism, the system can use cache preheating without any additional mechanisms to provide cache coherency.

5.5 Detecting Server Restarts

As described in Section 4.2, the cache preheating system needs to repeat the working set prediction phase whenever the server application is restarted, because the restart significantly changes the working set. For multi-process applications, process restarts are not a good indicator for the application restart. Therefore, our design tracks changes to TCP and UDP sockets instead, and whenever a socket is destroyed, we expect the corresponding application to be shutting down. In our prototype, the restart detection logic is divided into two parts:

- **Working set prediction phase:** During working set estimation, the preheating system records the destination port of the next incoming IP packet after the estimated arrival time. As the working set prediction phase spans multiple incoming network requests, we track the destination port (and network protocol) for each of the requests. Ideally, the predictable network architecture announces the port of future network packets. In our case, no such announcement is available though, so we need to manually track the port of incoming packets via a hook inserted into the `tcp_v4_rcv` function [16]. Other protocols besides TCP are not tracked yet, although similar functions exist for these protocols as well. The hook is only activated when an incoming packet is announced during the working set prediction phase, and automatically deactivates itself once a packet is received in order to keep the performance overhead as low as possible.

  Not every recorded destination port is automatically associated with the predicted request type: In our prototype, we noticed that the arrival time of random network packets to other network applications (e.g., concurrent SSH

sessions) coincides with the predicted network packet, so the wrong port is recorded. Therefore, we only associate ports with the request type after they have been observed to be accessed at least twice. This modification might not be necessary in a realistic setup with a real predictive network architecture if those other random packets are announced in advance as well and the order of arrival is known.

- **Preheating phase:** During the preheating phase, the system tracks changes to network sockets and invalidates working set estimates whenever one of the corresponding sockets is destroyed. To detect when sockets are destroyed, we have inserted a hook into the `__inet_put_port` function which is called whenever a socket is destroyed and the corresponding port is deallocated [11].

Chapter 6

Evaluation

In this section, we present the evaluation of our cache preheating prototype described in Section 5. With our evaluation, we want 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? How much do the individual optimizations described in the previous sections contribute to the performance of the design?

The evaluation of our solution is divided into four parts: First, we describe our benchmark setup (Section 6.1) and describe the performance metrics which we use to check the performance of our prototype. In Section 6.3, we present the results of our performance evaluation to show whether cache preheating can significantly reduce response times. The effectiveness, however, is reduced if the cache is not preheated quickly enough. In Section 6.4, we therefore analyze whether our prototype meets the deadlines set by the network architecture. In Section 6.5, we show that no significant runtime overhead is caused by cache preheating. Finally, in Section 6.6, we discuss whether the presented solution is already viable and what further modifications and optimizations should be explored.

6.1 Benchmark Setup

Most parts of the evaluation share a common setup. In Section 5.2, we have described the components of our prototype, which differ slightly from the original design. Especially, our prototype does not involve any actual network between different physical systems. Instead, the components are only executed on separate processor cores.

Most benchmarks are executed on an Odroid XU3 with four Cortex-A15 cores running at 2 GHz and four Cortex-A7 cores running at 1.4 GHz. To reduce the

variance of the benchmark results, we have configured all cores to always run at their maximum frequency. The system's cooling fan was configured to always run at full speed in order to reduce the likeliness of thermal throttling. In some of the following sections we deviate from this benchmark setup, for example because some properties are demonstrated on an x86-based system. In these cases, the differences are explicitly mentioned.

Whenever the benchmarks are executed on the ARM system, the benchmark client is always executed on the first Cortex-A7 core, whereas the server application and the preheating server are always executed on the first Cortex-A15 core. The kernel module executes most functionality on one or more Cortex-A15 cores. The benchmark client generally waits 500 microseconds between announcing a request and sending the actual request.

6.2 Performance Metrics

As described in Section 3.1, a cache preheating system needs to fulfill a number of requirements. Most importantly, the system needs to reduce overall latency when the caches are cold, without sacrificing the power usage advantages of cache power gating. Also, the system needs to be robust even when DMA is not cache-coherent, and it needs to be compatible to existing hardware and software. Finally, the predictable network architecture places some timing requirements on the preheating system, as preheating needs to be completed by the time the network request arrives.

In this evaluation, we mainly focus on the overall latency reduction as well as the timing requirements caused by the network architecture, as these are the main quantifiable performance properties of our prototype. Latency reduction is measured by the benchmark client described in Section 5.2, which executes several requests with warm caches, cold caches and with cache preheating. In the following, we also analyze performance counters to show that the latency difference is caused by cache preheating. The Cortex-A15 core supports a number of performance counters which are usable for this task. Especially, the core can count the number of data L2 cache refill events, as well as the number of cache lines written back to RAM. Although these counters provide an approximation of the number of cache misses, they do not include instruction cache misses, and no other performance event seems to fill this gap [4]. More importantly, however, not every cache miss is detrimental for performance. In an out-of-order CPU, the latency of some cache misses can be hidden behind other instructions if those instructions do not depend on the result of the memory operation. Instead of counting cache misses, we would rather like to count the stall cycles caused by the cache misses. No such performance counter is available though, so we use the

cycles per instruction (CPI) as a metric which correlates with the number of stall cycles.

Cache misses are only prevented, though, if the corresponding data is fetched into the caches before the network request arrives. The time required to preheat the caches is critical and, as shown in the sections below, significantly limits the amount of data which can be loaded until the network request arrives. This time is simply measured in the preheating kernel module, along with the selected performance counters.

The fulfillment of other requirements is implicitly shown in our evaluation: We demonstrate the compatibility to existing infrastructure by evaluating our prototype with unmodified hardware and unmodified server applications. Also, our prototype has shown to be compatible to DMA from other devices, as we have not observed any crashes despite significant concurrent DMA activity during our experiments. Finally, we argue that cache preheating will not have any significant effect on overall power usage, because, as shown in Table 6.1, the response time reduction mostly cancels out the corresponding preheating costs. As the limitations described in Section 5.1 prevent any extensive power usage analysis, we defer such an analysis to future work.

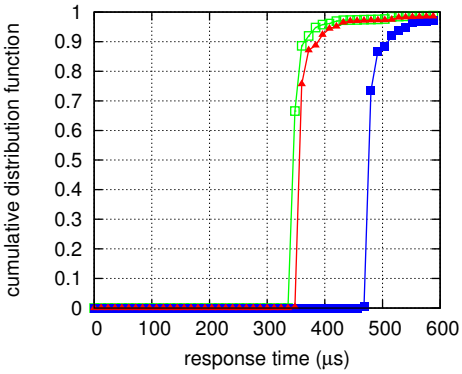
6.3 Performance Evaluation

The performance of a preheating system mostly depends on the achieved response time reduction, whereas the time required to preheat the caches is usually hidden behind the network latency. We analyze the response time reduction of our prototype for several real-world server applications to demonstrate the general viability of the design.

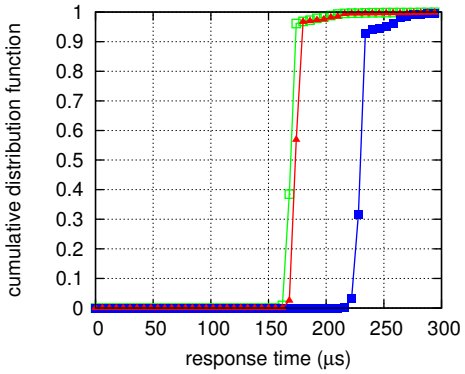
All benchmarks are only executed on one processor core, so we set the CPU affinity of the server applications when starting the benchmark. The nginx web server provides the `worker_cpu_affinity` to limit the cores on which the server is run. The other benchmarks provide no such option, instead they were pinned to one core with the `taskset` utility.

6.3.1 Response Time Benchmarks

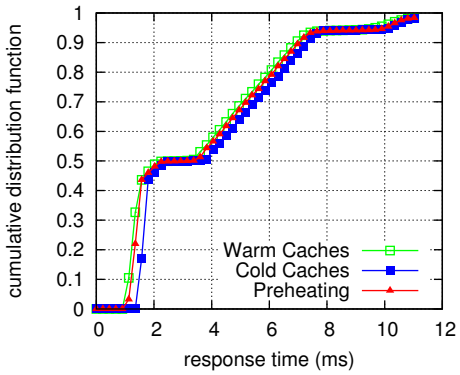
We have selected a number of representative server applications to demonstrate the effect of cache preheating on response times. These applications are the nginx web server, the memcached in-memory key-value store and the MariaDB database. The benchmarks have been selected because they are all typical latency-critical network services: For many web services, a single request can in turn



(a) nginx



(b) memcached



(c) DBT-2 (MariaDB)

Figure 6.1: Cumulative histogram of the response time of various real-world applications.

spawn many requests to databases or caches, and a single website can contain many small files which are fetched from a web server.

The nginx and memcached benchmarks use a static data set and the client simply fetches data from the server. The network requests are created with the libcurl HTTP library and the libmemcached library. The MariaDB benchmark is more dynamic, its network requests are copied from the DBT-2 benchmark [7], which is in turn a clone of the TPC-C database benchmark [17]. This benchmark simulates a warehouse management system and measures the performance of various database queries for order management and stock level status. However, originally, both TPC-C and DBT-2 were designed to measure the database throughput at maximum load. We have modified the DBT-2 benchmark to insert a short pause between the requests and have instrumented the code to announce the next request in order to simulate the predictable network architecture.

The complexity of the benchmarks differs significantly, and so do its working

	response time (μs)			preheated data	preheating
	cold	warm	preheated	(average)	cost
nginx	367.8	500.0	379.6	265.3 KiB	77.7 μs
memcached	174.5	236.4	180.4	154.9 KiB	52.6 μs
MariaDB	3959	4317	4057	657.1 KiB	154.0 μs
mixed	1567	1719	1569	360.2 KiB	95.7 μs

Table 6.1: Average response times of our benchmarks with and without cache flushes before each request as well as with cache preheating, and the corresponding time required to preheat the caches as well as the working set size of the benchmark.

	cache misses			CPI		
	cold	warm	preheated	cold	warm	preheated
nginx	267	6791	1392	3.61	5.80	3.95
memcached	249	4089	895	3.89	5.90	4.21
MariaDB	7597	22838	11733	2.26	2.47	2.31
mixed	3625	11423	4824	2.41	2.69	2.42

Table 6.2: Effect of preheating on the number of cache misses per network request and the average cycles per instruction

set size and the average response time. Figure 6.1 shows the average response time with and without preheating as well as the amount of preheated data and the time required to preheat the caches. In all cases, preheating causes a significant reduction of the response time, and we measure a significant reduction of the number of cache misses. We also measure a lower number of cache misses and reduced cycles per instruction (i.e., less stall cycles due to cache misses). These results show that the reduced number of cache misses is the main source of the response time reduction.

The DBT-2 database workload is especially challenging, because it executes a range of different network requests which have substantially different working sets and which access a large dynamic data set. Even in this case, cache preheating yields a significant performance improvement, even though only the commonly used memory locations are preheated.

6.3.2 Effects on Tail Latency

As we describe in Section 2.1.2, we expect servers to process network requests in bursts. Deep sleep modes are only activated between those bursts, so the first network request of every burst is processed with reduced performance. Depending

	Cold	Preheated
mean latency	1.9%	0.5%
50th percentile	0%	0%
90th percentile	6.3%	1.3%
95th percentile	23.9%	2.3%
97th percentile	19.0%	1.4%
99th percentile	0.6%	1.0%

Table 6.3: Overhead at selected response time percentiles when every 20th request hits a system with cold caches. In our setup, the 99th latency percentile is dominated by other effects which completely hide the effect of cold caches.

on the length of each burst, the average response time might not be significantly affected, but the tail latency is defined by the response time of the first request of each burst. In a modern large-scale web service, the tail latency of a single system however defines the overall latency for the end-user [27].

To show the effect of cache preheating on the tail latency in such a scenario, we simulate such burst processing: The benchmark client flushes the caches before every 20th network request, and optionally preheats the caches before this request. Table 6.3 shows the resulting response time overhead. Although the average latency is not significantly affected, cache flushes significantly increase the 95th and 97th percentile of the response time distribution, whereas cache preheating mostly mitigates this tail latency overhead. This result shows that cache preheating can efficiently mitigate the effect of cache flushes on the tail latency of web services.

6.3.3 Mixed Workloads

Our code detects the type of the network request based on the destination port of the network packet. Depending on the port, the working set of a different server application is loaded into the cache. To show that this differentiation between server applications is beneficial, we have benchmarked a mixed workload consisting of random requests to all the server applications described above. This benchmark is repeated twice, once with one working estimate per server application, and once with only one working set estimate for the whole system and without differentiation between different types of requests.

In the latter case, there are two potential strategies: The preheating code could either load all memory locations which are used by any of the application, or it could include only those locations accessed by all applications. Because preheating is time-critical and the amount of data loaded into the cache therefore needs

to be minimized, our code generally follows the latter strategy and removes all memory locations from the predicted working set which are not always required to process network packets. As a result, the working set is significantly reduced if the server applications are different, so the effectiveness of cache preheating is reduced. Our benchmarks show that, when the support for differentiation between multiple server application is removed, the overhead is reduced by only 23%, significantly less than the normal response reduction shown in Table 6.1.

6.4 Preheating Costs

Besides the effect on response time, we have also measured the overhead caused by preheating. Loading the working set into the caches is expensive, but also very time critical, because the time between the announcement and the arrival of a network packet is short. During all our performance experiments, we have also measured the time required to preheat the caches as well as the size of the working set. The results are shown in Table 6.1: The memcached server is the least complex application, as it only implements hash-table operations and memory management, whereas the MariaDB server implements execution of complex database queries, and therefore, according to our experiments, requires around 4 times as much data to process the average query. Similarly, the time required to preheat the caches varies between 53 and 154 microseconds. It can be seen that, even though the working sets are rather small, our prototype violates the deadline for cache preheating.

The numbers from Table 6.1 also show that the potential for further cost reduction is small on our hardware platform. On average, our prototype preheats the caches with an average throughput of 3.54 GiB per second. The achieved throughput depends significantly on the workload, with the preheating code transferring 2.94 GiB/s for the memcached workload and 4.27 GiB/s for the MariaDB workload. We have compared this number with results from the pmbw memory bandwidth benchmarking tool [13], which achieves up to 4.8 GiB per second when six cores are linearly accessing memory, and which achieves 4.3 GiB/s when—like in our prototype—two cores are accessing memory. Although the throughput differs between pmbw and our prototype, the difference is small enough to show that our prototype is already mainly limited by the available memory bandwidth.

6.4.1 Contribution of Individual Optimizations

As seen in the last section, the main challenge with cache preheating is that preheating needs to be completed by the time the next network packet arrives. The results shown above are the product of a number of individual optimizations. We

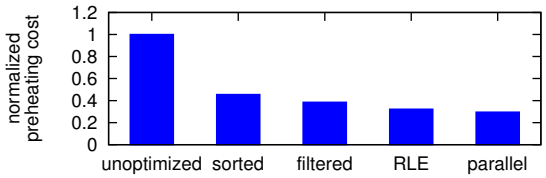


Figure 6.2: Normalized times to load the predicted working set of the nginx web server with different optimizations (normalized to the cost of unoptimized preheating)

have conducted an experiment to show that all of them have significant impact on preheating performance. Figure 6.2 shows the time needed to preheat the working set of the nginx web server when the optimizations below are successively enabled:

- **Sorted Cache Dumps:** Naturally, when the working set estimate is produced from copies of the cache tag memory, the list of memory locations is sorted by the indices of the corresponding cache sets. Addresses from different DRAM rows can map into the same cache set, so naive preheating utilizing this access order will lead to increased numbers of DRAM row switches. If we sort the memory locations by address, we improve the temporal correlation of the memory operations. As a result, the time required for preheating is reduced by 54.5%.

- **Working Set Filtering:** As described in Section 4.1.1, the initial working set estimate contains addresses which are only accessed during working set estimation (e.g., buffers into which the cache state is copied) as well as memory which has already been loaded into the cache when the preheating code is executed. Both can be removed from the working estimate. In the example of the nginx web server, this optimization reduces the size of the working set estimate by 65.5 KiB or 22%. As a result, cache preheating requires 15.3% less time.

- **Run-Length Encoding:** The list of memory locations in the working set is compressed using run-length encoding. Compression can reduce the time required to preheat the caches, as less data needs to be fetched from memory, and it can in theory reduce the potential for conflict cache misses because the cache footprint of the preheating code is reduced. In our case, the time required to preheat the caches is reduced by 6.9%.

- **Parallel Preheating:** The memory bandwidth of our test system cannot be fully utilized by a single core. If our preheating code is parallelized

on two cores, preheating requires 10.4% less time. We could not achieve any further improvements by adding further cores, although the results of the pmbw benchmark show limited potential for further optimizations. On more than two cores, the throughput improvement is likely offset by the parallelization overhead.

- **Manual Unrolling:** At this point, our preheating code is probably mostly limited by the available memory throughput. Nevertheless, there are situations during which the memory controller is not efficiently utilized. Therefore, we have manually unrolled the main loop which is used to preheat the caches, and have hand-optimized with manually written assembly code. This optimization further yields a 8.3% performance improvement.

6.5 Other Runtime Overhead

The last sections have shown the effects of the preheating solution on system performance during the working set prediction phase and while preheating is active. Preheating however only takes place while the system would otherwise be idle, so the potential negative impact is limited. It is still necessary to show that preheating does not produce any significant negative effect at other times while the system is active though. Especially, our solution introduces hooks in the kernel at two points, to check the target port of incoming packets and to invalidate working set estimates when the corresponding server socket is destroyed.

However, rarely any functions are registered for the first hook, and the second hook is rarely called during regular operation, so the induced overhead should be insignificant. In fact, we could not measure any response time difference compared to a system with an unmodified Linux kernel, because each reboot caused more variation to the response time than any of our modifications.

6.6 Discussion

As shown, cache preheating can reduce the response time overhead after a cache flush by on average 80%, almost as if the cache had never been flushed. As a result, the tail latency of our benchmarks is significantly reduced. Although the preheating system is limited to the last-level cache, this limitation is justified by the low performance impact when other types of caches are flushed. All in all, cache preheating is therefore a promising technique for response time reduction.

Our prototype, however, cannot preheat the caches quickly enough. The network arbiter announces future packets only approximately 60 microseconds in

advance, and this time includes the time required to wake the CPU up, so only approximately 30 microseconds are left to preheat the caches. Our cache preheating prototype exceeds this deadline for all presented benchmarks. In practice, cache preheating is still viable, due to two reasons:

1. **Increased preheating costs are compensated by the resulting response time reduction:** Even when preheating is not complete by the time the corresponding network packet arrives, we can naively continue to preheat the caches. The network request is then delayed until the cache has been preheated. Even this strategy results in a net response time reduction, as the delay is offset by an increased response time reduction.

   For example, our prototype overshoots the deadline by 37 microseconds, but the processing time is reduced by 120 microseconds, so the response time is reduced by 83 microseconds overall. Even the complex MariaDB workload would result in a net response time reduction.

2. **Server systems usually have significant higher memory bandwidth:** Our prototype only achieves a throughput of 3.54 GiB per seconds while preheating the caches, and synthetic memory benchmarks do not perform significantly better. Our prototype platform is not very representative for real-world server systems, though, which commonly employ more powerful CPUs and significantly faster memory. For example, we have measured the memory bandwidth of a system with an Intel i5-6500 CPU and two memory channels to be almost eight times as large as the one of our prototype system. With such increased memory bandwidth, all our benchmarks can likely be preheated within 30 microseconds, even the complex MariaDB workload which took four times longer in our test.

We conclude that cache preheating will result in good response time reduction on systems with high memory bandwidth. On systems with lower memory bandwidth, preheating is not yet utilized to its full potential. As a future optimization for this scenario, we propose a system which continues to preheat the caches even after the network packet has arrived, but which from this point executes preheating code in parallel to the server application.

The goal of cache preheating is that whenever the server application accesses memory, the accessed data is already in the cache. However, not all data needs to be in the cache when the network application is resumed. Only if a certain memory location is accessed right after the application has been resumed, the data needs to be loaded into the cache before any application code is executed. If data is accessed at a later point in time, it can be loaded into the cache while the server application is already running, as long as it is loaded into the cache before it is accessed.

Such concurrent preheating is possible, but requires detailed information about the order of the memory operations, so that data which is required early during execution of the server application is also loaded early during cache preheating. Concurrent preheating can be implemented either on a separate regular CPU core, on a separate thread on the same CPU core (e.g., Intel Hyperthreading), or on a specialized cache preheating controller. Using a separate CPU core wastes significant power, but provides good performance isolation between the preheating code and the server application. This performance isolation is not provided if the preheating code is executed on a separate hardware thread. In this case, if the CPU implements simultaneous multithreading, the memory accesses of the preheating code are interleaved with the server application. Although the power consumption is reduced compared to the usage of a second core for preheating, the two threads share one memory instruction pipeline and one thread can cause the other to stall due to resource conflicts. If, instead, cache preheating is implemented in a specialized cache preheating controller, for example as part of the memory controller, such resource conflicts cannot occur. Additionally, specialized hardware generally provides significantly higher power efficiency and often provides significantly higher performance.

In all three cases, the generated cache preheating memory accesses can block the memory bus, thereby causing a memory request of the server application to stall. Existing memory controllers implement heuristics to prioritize memory operations which cause pipeline stalls [33], and a similar technique should be used to prioritize „real“ memory accesses generated by server application code over any cache preheating operations.

Chapter 7

Conclusion

In this work, we have presented cache preheating as a technique to reduce the response time of server applications with cold CPU caches. Such cold CPU caches are frequently caused by power management mechanisms which disconnect a large fraction of the CPU area (including the caches) from the supply voltage. Although we would like to avoid the negative effects of such power management mechanisms, we have to use more and more aggressive power management in order to efficiently manage the increasing fraction of dark silicon.

Cache preheating is a technique which can reduce the effect of cold caches on CPU performance by loading the working set of the application into the cache before execution is resumed. The working set of the application can be predicted with fine granularity by analyzing the cache contents after similar requests are processed by the server application. Additional post-processing steps reduce the time required to load the working set, mainly by removing unnecessary data and by optimizing the memory access order. Ideally, in the case of server applications, such preheating is complete before the network packet arrives which triggers the wakeup from the CPU sleep state. Our design therefore also includes a mechanism to predict future incoming CPU packets with the help of a centrally arbitrated network architecture, as we modify the arbiter to announce future packets to their target systems.

In order to evaluate our design, we have implemented a prototype which implements cache preheating on top of simulated networking and simulated power management. The prototype is based on Linux and implements cache preheating on a system with ARM Cortex-A15 cores. Our evaluation is able to show that cache preheating is generally a promising and viable technique, even on hardware which is available today: With preheated caches, the benchmarked server applications perform almost as if the caches have never been flushed, and the response time overhead of the cache flushes is reduced on average by 80%. Our evaluation also shows that, on slightly more powerful systems, cache preheating

likely meets the timing requirements posed by common predictable network architectures, even though our prototype system does not provide enough memory bandwidth. Our evaluation also shows that cache preheating is useful when multiple concurrent server applications are running on the same physical system.

7.1 Future Work

Although our evaluation generally shows that cache preheating is a viable technique, we have not implemented a full working cache preheating solution, leaving significant space for future work. Especially, it is still necessary to evaluate the benefits of such a solution in a representative data center environment. Cache preheating has to be integrated with a real predictable network architecture, and has to be ported to more representative server CPUs. In such a more complete setting, the effects on overall power usage have to be evaluated.

Additionally, we have identified a number of places in the preheating software with further potential for optimization. Especially, as we expect large amounts of dark silicon to be available on future CPUs, some of these unusable transistors could be used to significantly improve the performance of cache preheating, for example by extending the memory controller to introduce more efficient methods to load the working set. Other optimizations might be possible purely in software. For example, preheating could be continued concurrently, on a different core, after the server application has resumed execution. Alternatively, the working set estimation could be used to identify parts of the application's memory which could be reordered to improve the cache preheating performance.

Finally, more research should be conducted on the various methods to create fine-grained predictions of the working set. Especially simulation-based techniques might be able to produce estimates of significantly higher quality with reasonable performance impact. Such techniques techniques can not only predict the working set but also yield information about the order of the memory accesses (as required by concurrent cache preheating as described above) and information about conflict cache misses.

Bibliography

[1] ARM Cortex-A Series Programmer's Guide for ARMv8-A: 14.3. Multi-core cache coherency within a cluster. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0024a/BABGJHAB.html.

[2] ARM926EJ-S™ Technical Reference Manual: 3.2.8. Second-level descriptor. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0198e/I16780.html.

[3] CoreLink CCI-400 Cache Coherent Interconnect. http://www.arm.com/products/system-ip/interconnect/corelink-cci-400.php.

[4] Cortex-A15 MPCore Processor Technical Reference Manual: 11.6. Events. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0438i/BIIDBAFB.html.

[5] 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.

[6] cpuidle-exynos5422.c. https://github.com/hardkernel/linux/blob/odroidxu3-3.10.y/arch/arm/mach-exynos/cpuidle-exynos5422.c.

[7] Database Test Suite. http://osdldbt.sourceforge.net/.

[8] dma-mapping.c. http://lxr.free-electrons.com/source/arch/arm/mm/dma-mapping.c#L100.

[9] en:xu3_hardware (Odroid Wiki). http://odroid.com/dokuwiki/doku.php?id=en:xu3_hardware.

[10] fastpass/fpproto.h – yonch/fastpass – Github. https://github.com/yonch/fastpass/blob/71107152dda6ed7eae9de7be12a63e6a0f4e2fb8/src/protocol/fpproto.h#L93.

[11] inet_hashtables.c. http://lxr.free-electrons.com/source/net/ipv4/inet_hashtables.c#L105.

[12] PAT (Page Attribute Table). https://www.kernel.org/doc/Documentation/x86/pat.txt.

[13] pmbw - Parallel Memory Bandwidth Benchmark / Measurement. https://panthema.net/2013/pmbw/.

[14] Poor virtual machine application performance may be caused by processor power management settings. https://kb.vmware.com/selfservice/microsites/search.do?language=en_US&cmd=displayKC&externalId=1018206.

[15] Slow Performance on Windows Server when using the „Balanced“ Power Plan. https://support.microsoft.com/en-us/kb/2207548.

[16] tcp_ipv4.c. http://lxr.free-electrons.com/source/net/ipv4/tcp_ipv4.c#L1539.

[17] TPC-C. http://www.tpc.org/tpcc/default.asp.

[18] USB Made Simple: Part 1 - Introduction to USB. http://www.usbmadesimple.co.uk/ums_1.htm.

[19] Valgrind User Manual – 5. Cachegrind: a cache and branch-prediction profiler. http://valgrind.org/docs/manual/cg-manual.html.

[20] xinetd(8) - Linux man page. http://linux.die.net/man/8/xinetd.

[21] Alfred V Aho, Peter J Denning, and Jeffrey D Ullman. Principles of optimal page replacement. *Journal of the ACM (JACM)*, 18(1):80–93, 1971.

[22] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). *ACM SIGCOMM Computer Communication Review*, 41(4):63–74, 2011.

[23] Mark Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *Solid-State Circuits Society Newsletter, IEEE*, 12(1):11–13, 2007.

[24] Koushik Chakraborty. *Over-Provisioned Multicore Systems*. PhD thesis, University of Wisconsin, Madison, 2008.

[25] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low-Power CMOS Digital Design. *IEICE Transactions on Electronics*, 75(4):371–382, 1992.

[26] Ian Cutress. The Intel Haswell-E CPU Review: Core i7-5960X, i7-5930K and i7-5820K Tested. `http://www.anandtech.com/show/8426/the-intel-haswell-e-cpu-review-core-i7-5960x-i7-5930k-i7-5820k-tested`.

[27] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[28] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[29] Benjamin Erb. Concurrent Programming for Scalable Web Architectures, April 2012. `http://www.benjamin-erb.de/thesis`.

[30] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the End of Multicore Scaling. In *38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376. IEEE, 2011.

[31] Greg Ferro. Switch Fabrics: Input and Output Queues and Buffers for a Switch Fabric. `http://etherealmind.com/switch-fabric-input-output-buffers-queues/`.

[32] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy Caches: Simple Techniques for Reducing Leakage Power. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*, pages 148–157. IEEE, 2002.

[33] Saugata Ghose, Hyodong Lee, and José F. Martínez. Improving Memory Scheduling via Processor-Side Load Criticality Information. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 84–95. ACM, 2013.

[34] Matthew P Grosvenor, Malte Schwarzkopf, and Andrew W Moore. R2D2: Bufferless, Switchless Data Center Networks Using Commodity Ethernet Hardware. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 507–508. ACM, 2013.

[35] James Hamilton. The Cost of Latency. `http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/`, 2009.

[36] Michael R Hines and Kartik Gopalan. 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)*, pages 51–60. ACM, 2009.

[37] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual – Volume 3: System Programming Guide*. Number 325384-058US. April 2016.

[38] Hema Kapadia, Luca Benini, and Giovanni De Micheli. Reducing Switching Activity on Datapath Buses with Control-Signal Gating. *IEEE Journal of Solid-State Circuits*, 34(3):405–414, 1999.

[39] Nam Sung Kim, Todd Austin, David Baauw, Trevor Mudge, Krisztián Flautner, Jie S Hu, Mary Jane Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. Leakage Current: Moore's Law Meets Static Power. *Computer*, 36(12):68–75. IEEE, 2003.

[40] Florian Larysch. Fine-Grained Estimation of Memory Bandwidth Utilization. Master Thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, March 2016.

[41] Etienne Le Sueur and Gernot Heiser. Dynamic Voltage and Frequency Scaling: The Laws of Diminishing Returns. In *Proceedings of the 2010 International Conference on Power Aware Computing and Systems (HotPower'10)*, pages 1–8. USENIX Association, 2010.

[42] Shuhao Liu, Hong Xu, and Zhiping Cai. Low Latency Datacenter Networking: A Short Survey. *arXiv preprint arXiv:1312.3455*, 2013.

[43] Daniel Molka, Daniel Hackenberg, Robert Schöne, and Matthias S. Müller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*, pages 261–270. IEEE, 2009.

[44] Shinichiro Mutoh, Takakuni Douseki, Yasuyuki Matsuya, Takahiro Aoki, Satoshi Shigematsu, and Junzo Yamada. 1-V Power Supply High-Speed Digital Circuit Technology with Multithreshold-Voltage CMOS. *IEEE Journal of Solid-State Circuits*, 30(8):847–854, 1995.

[45] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *ACM Sigplan Notices*, volume 42, pages 89–100. ACM, 2007.

[46] Brent Ozar. SQL Server on Power-Saving CPUs? Not So Fast. `https://www.brentozar.com/archive/2010/10/sql-server-on-powersaving-cpus-not-so-fast/`, October 2010.

[47] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. Fastpass: A Centralized „Zero-Queue“ Datacenter Network. *ACM SIGCOMM Computer Communication Review*, 44(4):307–318, 2015.

[48] Laurent Pinchart. Mastering the DMA and IOMMU APIs. `http://events.linuxfoundation.org/sites/events/files/slides/20140429-dma.pdf`, 2014.

[49] Michael Powell, Se-Hyun Yang, Babak Falsafi, Kaushik Roy, and TN Vijaykumar. 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)*, pages 90–95. ACM, 2000.

[50] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M. K. Martin. Computational Sprinting. In *18th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2012.

[51] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory Access Scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA'00)*, pages 128–138. ACM, June 2000.

[52] Theodore H. Romer, Wayne H. Ohlrich, Anna R. Karlin, and Brian N. Bershad. Reducing TLB and Memory Overhead Using Online Superpage Promotion. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 176–187. ACM, 1995.

[53] Robert Schöne, Daniel Molka, and Michael Werner. Wake-up latencies for processor idle states on current x86 processors. *Computer Science - Research and Development*, 30(2):219–227. Springer, 2015.

[54] Anand Lal Shimpi. The Haswell Review: Intel Core i7-4770K & i5-4670K Tested. `http://www.anandtech.com/show/7003/the-haswell-review-intel-core-i74770k-i54560k-tested`.

[55] W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. *UNIX Network Programming*, volume 1. Addison-Wesley Professional, 2004.

[56] Michael B. Taylor. Is Dark Silicon Useful? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*, pages 1131–1136. ACM, 2012.

[57] Gustavo E. Téllez, Amir Farrahi, and Majid Sarrafzadeh. Activity-Driven Clock Design for Low Power Circuits. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'95)*, pages 62–65. IEEE Computer Society, 1995.

[58] Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C Snoeren. Practical TDMA for Datacenter Ethernet. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys'12)*, pages 225–238. ACM, 2012.

[59] Anant Vishnoi, Preeti Ranjan Panda, and M Balakrishnan. Cache Aware Compression for Processor Debug Support. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'09)*, pages 208–213. European Design and Automation Association, 2009.

[60] Anant Vishnoi, Preeti Ranjan Panda, and M. Balakrishnan. Online cache state dumping for processor debug. In *46th ACM/IEEE Design Automation Conference (DAC'09)*, pages 358–363. IEEE, 2009.

[61] Yasuko Watanabe, John D. Davis, and David A. Wood. WiDGET: Wisconsin Decoupled Grid Execution Tiles. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 2–13. ACM, 2010.

[62] Se-Hyun Yang, Michael Powell, Babak Falsafi, Kaushik Roy, and TN Vijaykumar. Dynamically Resizable Instruction Cache: An Energy-Efficient and High-Performance Deep-Submicron Instruction Cache. *ECE Technical Reports*, page 22, 2000.

[63] Liang Zhang, James Litton, Frank Cangialosi, Theophilus Benson, Dave Levin, and Alan Mislove. Picocenter: Supporting Long-lived, Mostly-idle Applications in Cloud Environments. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*, page 37. ACM, 2016.

[64] Qi Zhu, Meng Zhu, Bo Wu, Xipeng Shen, Kai Shen, and Zhiying Wang. Software Engagement with Sleeping CPUs. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, May 2015.

[65] Peter Zijlstra. High Memory Handling. `https://www.kernel.org/doc/Documentation/vm/highmem.txt`.

All links were last accessed on 20. May 2016.