Master's Thesis

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Control Engineering

Support for PREM on contemporary multicore COTS systems

Jan Doležal
Open Informatics, Computer Engineering
dolezj21@fel.cvut.cz

January 2019
Supervisor: Ing. Joel Matějka

Acknowledgement / Declaration

My gratitude for support during preparations of this work belongs to my supervisor Joel Matějka. I thank him especially for his guidance and also for his easy-going attitude that shifted the fun while creating this work to the completely different level. I am very grateful for the support provided by people from the CIIRC's Industrial Informatics Research Center. Namely I want to thank to Michal Sojka for all the valuable advices.

I hereby declare that the submitted thesis is exclusively my own work and that I have listed all used information sources in accordance with the Methodological Guideline on Ethical Principles for College Final Work Preparation.

In Prague, 8. 1. 2019

....

Abstrakt

Práce přináší podporu pro vyvíjené výkonné aplikace reálného času, které pro svůj běh využívají vícejádrové běžně komerčně dostupné systémy.

Hlavním cílem práce je shrnout metody pro zvýšení předvídatelnosti doby běhu programu a provést testy nástrojů a hardware, které tyto metody využívají. Metody diskutované v této práci řeší problémy při konfliktech při přístupu ke sdíleným prostředkům na těchto platformách.

Konflikty při přístupech do hlavní paměti jsou řešeny za pomoci Predikovatelného výpočetního modelu (PREM). K vylepšení jeho spolehlivosti jsou zkoumány a vyhodnoceny možnosti omezení šířky pásma klientů paměti na úrovni paměťového kontroléru a také s využítím nástroje MemGuard na úrovní jader procesoru. Dvě ze zmíněných metod k omezení šířky pásma paměti jsou testovány na energeticky účinné vestavné platformě NVIDIA TX2.

Dále jsou prozkoumány efekty hardwarového dělení sdílených vyrovnávacích pamětí (Intel RDT) na platformě s procesorem řady Intel Xeon.

Výsledky ukázaly, že omezení šířky pásma na úrovni paměťového kontoléru není použitelné, protože není možné korektně separovat klienty. Naopak implementace MemGuardu byla úspěšně otestována na více testovacích případech. Testované hardwarové dělení sdílených vyrovnávacích pamětí zlepšuje předvídatelnost dle očekávání. Dále poskytujeme informace jak vyladit parametry nástrojů využívaných implementací PREM.

Klíčová slova: Systémy reálného času; WCET; Predikovatelný výpočetní model; omezování šířky pásma klientů paměti; Využití dělené skryté paměti.

Překlad titulu: Podpora PREM na současných multicore COTS systémech

Abstract

This work brings support for high-performance real-time applications that run on generic contemporary multicore systems referred to as Commercials Off-The-Shelf (COTS) platforms.

The main goal is to summarize the methods improving the predictability of program execution time and to test the tools and hardware utilizing these methods. Methods discussed in this work address issues of competition for shared resources in multicore COTS platforms.

Memory contention is resolved by PRedictable Execution Model (PREM) that schedules memory accesses in the system. To improve the reliability of PREM, we describe and employ bandwidth limiting capabilities to the memory clients. We evaluate throttling of memory controller clients and the MemGuard tool, which allows throttling of CPU cores, and profile the implementation to show its overhead. Tests are performed to see the effects of throttling on the memory controller level. The two presented methods to throttle bandwidth of the memory clients are tested on the low-power embedded platform NVIDIA TX2.

Further, we address shared cache contention using methods for partitioning shared cache (Intel RDT, Page Color). We showed the effects of the hardware partitioning using RDT on an Intel Xeon series processor.

The results have shown that throttling of the memory controller is not usable since we cannot separate particular clients properly. Further, MemGuard was successfully tested on multiple use cases. Tested hardware cache partitioning improves the system performance as expected. Moreover, we provide details about the fine-tuning of the tools used by PREM.

Keywords: Real-time systems; Worst-Case Execution Time; Predictable Execution Model; throttling memory clients bandwidth; Partitioned Cache Utilization.

Contents

Tables / Figures

Chapter 1
Introduction

The requirements of real-time embedded systems on their computational power increase as they are expected to solve the new feature-rich challenges with higher and higher complexity. There are ongoing efforts to simplify everyday life tasks. The example of such effort is research in the area of autonomous transportation. To cover the market, the devices that carry out such tasks need to be distributed in high numbers. Therefore, the price of the underlying devices used in such solutions becomes increasingly important. The custom hardware designed for any particular real-time application cannot provide the low price, so the attention shifted to the use of generic multi-core computation platforms referred to as Commercials of The Shelf (COTS).

COTS platforms bring high performance and low price. However, their properties are not optimized for critical real-time applications. Programs driving critical real-time systems have to provide their result within the specified deadline. Otherwise, an event threatening human life or property may occur. Real-time systems and their parts are therefore subjected to the worst-case execution time (WCET) analyses to be sure no deadline within the system is exceeded. The hardware of the multicore COTS platforms is targeting primarily non-real-time segments, and it is optimized for high average performance (i.e., low average execution time) by utilizing complex superscalar CPUs and multi-level caches. Sparse random occurrences of the significantly lower performance are not considered to be an issue. Therefore, use of such systems in real-time applications lead to pessimistic estimations of worst-case execution times, and significant underutilization of the designed system.

In multicore COTS platforms the possibility of the state that leads to the occurrence of unusually high execution time is caused by shared resources. Shared resources are contention points such as shared last level cache and shared main memory bus. Competition of parallelly running CPU cores for these shared resources can introduce significant timing delays and increase execution times of all contenders.

The focus of this work is on a research of the methods to decrease WCET by increasing the predictability of program execution time, and on evaluating some of these methods using the COTS platforms. The effort of these methods is to moderate quality of service provided by shared resources so that it corresponds to the system's requirements.

PRedictable Execution Model is a mechanism addressing competition for shared main memory. PREM coschedules all active components (e.g., CPU cores, I/O peripherals) in the system. Each component has a time slot in a static schedule where it can access the memory. The scheduler notifies components about beginning and end of their memory phase. If the task executed by certain component is aware of PREM, it is called predictable task. If not, it is called legacy task. The predictable task is prepared in a way it contains memory phases and executes phases that are in line with the static schedule. In memory phase the, data necessary for the execute phase are loaded to the component-local memory (cache for CPU). During execute phase the component performs no memory accesses leaving the memory bus free to be used by other components. It is not possible to premize every task because many of them have complex memory access patterns. Such legacy tasks could unpredictably interfere with the memory phases of predictable tasks.

The memory phase of predictable task suffering from negative interference might not be completed by the static schedule and expected execution time of such task could be crossed. This could lead to serious consequences if the task employed was a real-time task. Moreover, this could start chain reaction breaking down the static schedule and crashing the whole system in the end.

To ensure that memory phase finishes within its deadline, a certain level of the bus bandwidth has to be guaranteed for the usage by the statically scheduled memory phases. Such a guarantee is provided when the bandwidth of all legacy components is limited. This way the memory bus bandwidth is partitioned among predictable and legacy components. This motivated the exploration of methods to limit the bandwidth of legacy tasks.

There exist methods to partition shared cache among contenders leading to improved predictability of their execution time.

This work addresses the following methods or employes the following tools:

- PRedictable Execution Model (PREM) that addresses competition for main memory
- MemGuard which is the tool to limit memory bandwidth of the CPU cores utilizing Performance Monitor Counters
- memory controller as the mean to limit memory bandwidth of its clients
- cache partitioning as a method addressing contention for the space in a cache shared among CPU cores
  - in software with page coloring
  - in hardware with Intel Resource Director Technology
- Jailhouse hardware partitioning hypervisor
  - version with the implemented MemGuard support
  - version with the support for RDT

There are hardware and software solutions that we tested. Two tested software solutions are related to the throttling bandwidth of memory clients. First, we test throttling on the memory controller level. Next, the MemGuard tool is profiled. Both aforementioned solutions are evaluated using low-power embedded platform NVIDIA Tegra X2. Lastly, we tested the behavior of the hardware cache partitioning with RDT. This feature is available on some of the Intel Xeon processors.

Memory controller and its throttling mechanism are complex devices with hierarchical organizations. Moreover, NVIDIA does not provide a complete description of the controller in its manuals. The setup of test incorporating memory controller is, therefore, challenging task.

The work partly originated as a contribution to the European project HERCULES [1].

[1] https://hercules2020.eu/

Chapter 2
Background

In section 2.1 are stated main objectives of embedded systems and real-time systems along with their analysis. Section 2.2 briefly introduces multicore systems and problems of their usage in the area of real-time applications. The methods increasing predictability within multicore systems were briefly introduced in the section 2.3. In section 2.4 the chapter continues with the memory hierarchy description, highlighting especially caching mechanism. The memory hierarchy is followed by the section 2.5 that describes Performance Monitor Unit allowing to profile, among others, characteristics of the memory subsystem. The section 2.6 talks about hardware partitioning with the utilization of hypervisor. Next follows the section 2.7 describing PRedictable Execution Model. Section 2.8 describes MemGuard – tool to reservate memory bandwidth for execution unit. Finally there is an example of software based cache partitioning in section 2.9.

2.1 Real-time embedded systems

In our daily lives, people use computer systems that have dedicated purpose and that are controlling some larger system. These are called embedded systems. Several examples follow: ticket vending machine, cash register, digital watch, mobile phone, microwave oven, washing machine, medical imaging, GPS navigation, anti-lock braking system or infotainment system.

There are several highly important objectives for current embedded systems [1]:

- low development costs
- short time-to-market
- dependability
- temperature efficiency
- energy efficiency
- average-case performance
- worst-case performance

To ensure safety, the devices are required to response to an event within the certain time constraint. System having this property is referred to as a real-time system. My favorite example is the airbag responding to a collision. If the system responses to the collision too late, the airbag inflation might not help and it could even cause additional injuries to the passenger. This example shows hard real-time system. That is a system where missing the time constraint (so called deadline) can lead to damage of property or loss of lives.

There exist also soft real-time systems. If the deadline is missed in these systems, the result of computation invoked by the event can still be utilized, but the usefulness of such a result degrades as the time advances further beyond the deadline. Very frequent example of soft real-time systems are live audio-video systems, where late data delivery/processing leads to degraded quality of the output. [2]

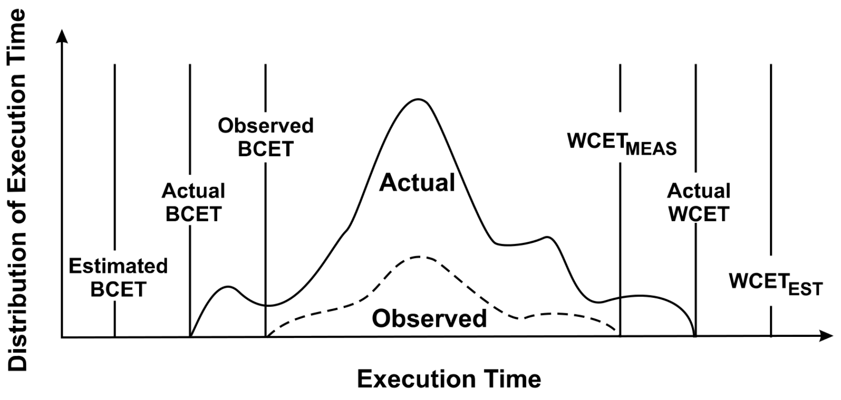


Figure 2.1. Execution time distribution. Best/Worst-Case Execution Times. [1]

2.1.1 Worst-case execution time (WCET)

To ensure that the time constraint is fulfilled whenever the code is invoked, there is need to know the longest possible time interval during which the given code can run from its invocation. Such a period is referred to as worst-case execution time (WCET).

To find the maximum execution time it is not feasible to exhaustively explore all the possible execution times since software for contemporary architectures exhibit a too large state space [1]. Thus measured WCET, WCET$_{MEAS}$, is not sufficient parameter for the verification of hard real-time systems. However, it can be utilized for more relaxed real-time systems, where there is added a safety margin to WCET$_{MEAS}$.

The more scientific approach includes usage of formal methods to reason about WCET. With this approach, the code is analyzed statically to determine properties of its temporal behavior [1]. Nevertheless WCET cannot be derived from every program due to undecidability. Analyzed programs must meet constraints such as guaranteed termination, bounded recursion depths and loop iteration counts.

2.1.2 WCET and contemporary hardware

Since embedded systems are getting more and more complex, there is a need to use high-performance commercials off the shelf (COTS) hardware [3]. There is need to account for architectural features of such hardware. Common architectural features such as superscalar pipelines and caches are targeting reduction of average-case execution time (ACET). It is however not possible to precisely analyze these features using static methods [1].

Furthermore, a sound approximation of the actual WCET demands an abstraction of possible inputs and initial states of the system, which introduces another source of imprecision [1]. As a consequence, the determination of the actual WCET has to be relaxed to the derivation of an upper bound on the execution time of the task. These bounds represent the estimated WCET$_{EST}$. The process of estimating the WCET is called timing analysis. [1]

The relation among mentioned worst-case execution times is as follows

$$WCET_{MEAS} < WCET_{Actual} < WCET_{EST}$$

. Example that helps to understand the execution time distribution is in the figure 2.1. To ensure safeness, $WCET_{Actual} \leq WCET_{EST}$ must hold. The estimated WCET gets more precise when $WCET_{EST} - WCET_{Actual} \rightarrow 0$. This is referred to as tightness.

2.1.3 Timing constraints in industry

The availability of timing-aware software development tools is insufficient. As a result trial-and-error approach is often used in practice. The software is tuned until $WCET_{MEAS} + Safety\ margin$ is less than timing constraints. [1] If the optimizations of WCET create enough time reserve before the deadline either cheaper hardware may be utilized, or the software may be modified to do more work.

2.2 Multicore systems and real-time applications

Historically the performance of silicon system increased with usage of higher clock frequencies. Higher frequency allows more operations to be performed within the same time. With higher frequency there is produced more heat and if the frequency is increased too much, the limit may be reached, where too much heat can damage the system.

Since the performance couldn't be increased by scaling frequency anymore, another way to increase performance came to broad use. There are used multiple computational cores i.e., multiple CPUs in one system.

Within real-time domain, one of the main issues with multicore systems is the main memory shared among the cores. When multiple cores share the main memory it leads to contention over access to the memory. In turn this increases the timing unpredictability. Existing WCET analyses are extremely pessimistic for these configurations. Since industry relies on those analyses, the usage of multi-core systems in these areas is limited unless new approaches are utililzed. [4]

In this work the approach to address memory contention described in previous paragraph is PRedictable Execution Model (PREM). Further details on PREM can be found in section 2.7.

Memory organization is thoroughly described in section 2.4.

2.3 Methods increasing predictability of execution time within multicore system

In the area of our interest – real-time embedded systems – there is an aim to utilize simple ideas so the whole system is deeply understood. This way people can more easily reason about the guarantees within the system and potentially provide certified system.

With PRedictable Execution Model (PREM) basically the main memory accesses are scheduled among the clients of the memory. This way the timing of memory accesses is more predictable and theoretical WCET bloated by possible clashes of concurrent accesses to the memory can be decreased. PREM is described in detail in the section 2.7.

With PREM each task is divided into memory phases to retrieve some data and computation phases to work on those data. In memory phase the data are prefetched to the local memory of an execution unit. Tasks that may be divided into these phases are called predictable intervals. PREM employs static scheduling of intervals. To achieve predictability it is necessary to shedule the memory phases so that they follow up each other. This is easily done utilizing static scheduling. In practice this means that there is one unit accessing memory and all other execution units are either in their compute phase or stalled. This way WCET in execution phases can be decreased, since each data access is faster.

Not all tasks may be easily separated into memory and computation phase due to its complexity. These undividable tasks are called compatible or legacy intervals. A compatible interval may seriously interfere with predictable interval when they run concurrently. To regualte the level of interference there is required certain level of memory bandwidth for the predictable intervals. This allows to guarantee WCET.

In this paragraph there are presented methods to limit the bandwidth of the computation unit. Some platforms utilize memory controllers that have its clients attached in a way their bandwidth can be limited directly. For example when the GPU, as the computation unit, is one of the MC's clients, its memory accesses can be limited on the MC level to ensure it does not interfere with CPU. The CPU could be approached if only each CPU core was a client to the memory controller. However the whole CPU cluster is typically attached as one MC client. To limit memory bandwidth of certain CPU core there may be utilized Performance Monitor Unit (PMU). The PMU captures various statistics related to the performance of CPU. It may also essentially count memory accesses. The memory bandwidth can be limited in the following way. In the given period of time certain amount of the memory accesses is allowed to be executed. The execution of the core that exceeded the limit is suspended until the period times out. The method using Performance Measuring Counters was introduced by MemGuard [5].

Any task requires certain resources i.e., hardware to achieve its goal. This includes certain portion of memory, execution unit and it may require further devices such as timers, DMAs and many others. Hardware often needs to be shared among tasks as the number of its instance is limited. In general however, sharing resources may lead to timing unpredictabilities originating in the manager of such shares.

Consider the hardware is partitioned to certain bundles. Above each bundle there runs specific task. With such organization the simple model is obtained. It is possible to reason about such model more easily and easily ensure the bundels mutual negative interference is mitigated. An example follows: the reserved resources for certain task could include few regions in the physical memory address space, CPU cores 4 & 5 and memory mapped PCI device. No other task would get access to the resources mentioned in the previous sentence. More on this in section 3.4.

CPU clusters have typically some very fast per core memory as well as fast per cluster memory. These memories are meant to capture part of the data backed by main memory and provide execution unit that reaches for the same data repeatedly with fast accesses to such data. Described on-chip memories are called caches and they are hidden from direct access by the programmer. Since per cluster memory is shared among cores it may occure that one core thrashes data of the other core in such a cache. This behavior is undesirable since it is source of unpredictability. Hence some manufacturers equipped the shared caches with the ability to be partitioned. When partitioned, certain part of cache belongs exclusively to certain cores. Besides the trend of manufacturing processors with this effective cache management [6] there also exists software method to partition the cache called Page Coloring [7].

2.4 Memory hierarchy

Nowadays, computer systems utilize fast processors with high frequencies and fairly slower main memories. Figure 2.2 compares the evolution in speeds of processor and main memory at the end of last century. When the processor executes code, it first needs to load the executed code itself and then to load the data that are referenced by the code. Between main memory and processor there are present devices called caches.

Cache is much faster memory compared to the main memory. If the caches were not present in the system, it would take hundreds of processor cycles for the main memory to load a data item. That would lead to the most of the processor time to be wasted.

Cache is basically small piece of memory but much faster than the main memory. Different technology is used for the main memory (Dynamic RAM: capacitor), and for caches (Static RAM: flip-flop). The price is the main reason why the main memory is not made with the same fast technology as the one used in caches. Another reason is the space requirements for SRAM technology. Besides the fact that for storing one bit DRAM requires only one transistor along with a capacitor whereas SRAM requires as much as six transistors, the wire capacities charge faster with larger transistors. Therefore the speed depends on the time needed to charge these wire capacities, and the speed and space on the die required have the inverse relationship. I.e., having two caches of the same byte capacity the one taking up more space on the die (larger transistors) is faster. Further, the high amount of memory requires more complex controllers and typically when the complexity rises the speed goes down.

When writing code, programmer ponders over virtual memory model, and in general, he does not interact with caches directly. Detailed information can be found in a serial about memory at the Linux Weekly News [8].

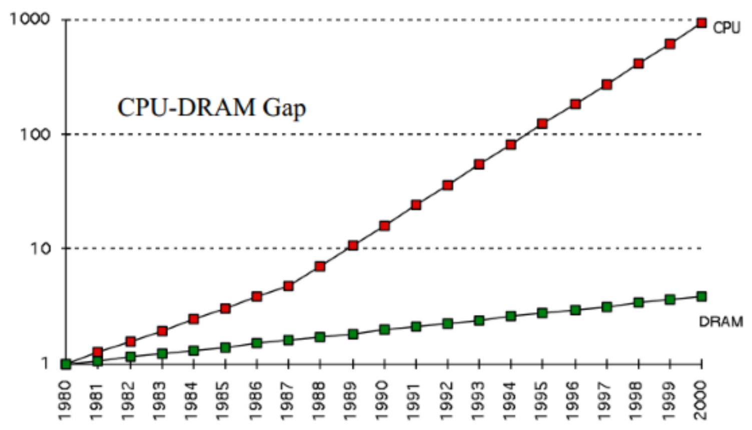


Figure 2.2. Processor and memory performance comparison throughout the years.

2.4.1 Data request propagation

In memory hierarchy, there is typically more than one cache level where the cache level closer to the CPU is smaller and faster than the level farther from the CPU. Smaller cache has simpler control circuits hence it is faster (relates to the propagation times).

When CPU accesses the data item, it starts by searching the item in the first level cache. Cache either returns an item, in case of the cache hit, or informs the CPU about cache miss, in case the item is not present in the cache. In the event of a miss, CPU tries to find the data item in the next cache level. Eventually, if the item is not found in any cache level it is retrieved from the main memory. The example of the cache hierarchy in Sandybridge architecture from Intel is in the figure 2.3.

Cache miss typically introduce a time miss penalty, because cache first checks whether it contains the item and only then advances to the next level in the hierarchy. Time to retrieve uncached data item is the sum of time miss penalties in each cache level and time of the memory access. Due to this some synthetic programs could perform better

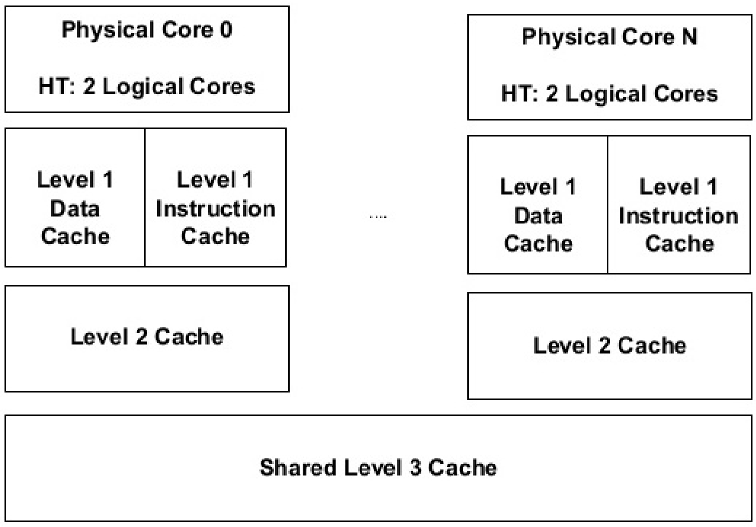


Figure 2.3. Multi-level caching on Intel Sandybridge.

Level 1 cache is the closest to the execution unit, it has the shortest access time, but it is the smallest cache. Caches farther from the execution units are slower, but they can hold more data.

with caches turned off completely. The potential increase in data access time is a risk factor for the real-time systems.

The power of caches lies in the locality of reference. Temporal locality describes phenomena where the same values or memory locations are frequently and repeatedly accessed. Spatial locality assumes that when one memory location is referenced it is likely that nearby locations will soon be also referenced. To exploit spatial locality, the bigger chunk of memory is copied into cache than what was requested. The chunk size closely relates to cache organization which is described in the next section.

2.4.2 Cache organization

The cache purpose is to capture useful part of the main memory at the given moment. Memories are organized in sequences of data elements – bytes and one byte ($1B = 8b$; B: byte, b: bit) is the basic addressable unit of memory. In the following, address having the size of 32 bits is considered. The cache needs to know the memory address of the captured element. When the CPU asks for certain address, the cache compares it with addresses associated to the stored elements.

In this paragraph simplistic example is considered, where one cache element allows to store 1B. For such an element there are needed 40 bits (5B) – 4B of information to store 32b address and 1B of data. This would be an inexcusable waste of transistors. Instead, for given address, there are stored multiple data bytes of the memory surrounding such an address. Multiple bytes in cache line is a setup that exploits the spatial locality.

The cache line is the term referencing to basic copiable unit between levels of hierarchy. There are several information fields associated with each cache line such as flags including line validity and bits determining line usage frequency, and further, there is a *tag* as a part of a memory address allowing to refer to the cache line.

Consider cache line with the size of 64B. To refer to the specific byte in the cache line, there is need for 6 addressing bits (indeed $2^6 = 64$). These 6 bits are cut from the end of an address, and they are called the *offset*. The rest 26 bits of the address are used to reference the cache line, and they are called the *tag*. This is the example of **fully associative cache**. The cache stores each tag that is not yet in it to the next

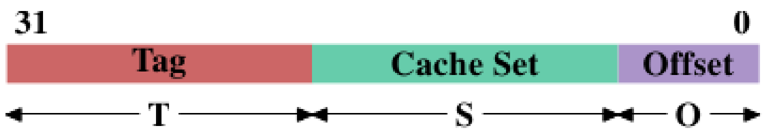


Figure 2.4. Address parts driving cache data element choice. [8]

empty cache line along with its data. Retrieving cache line requires comparison of the input tag to each cache line tag, so the comparator is needed for each cache line.

Besides *offset* another portion of bits can be cutted from the address and use to address cache lines. Such a portion of bits is called *index* or *cache set*. If each *set* addresses exactly one cache line, then the cache is called **direct-mapped cache**. For example, if the *offset* is 6 bits long yielding 64B long cache line and the *set* is 8 bits long yielding 256 cache lines the size of the direct-mapped cache is 16kB. Considering 32 bit addresses the size of *tag* is 18 bits long, and only one comparator is necessary to compare *tag* of selected cache line with the input address tag. Every two addresses having same bits in the field *set* end up in the same cache line and replace each other's data. To address this problem, the associativity of the cache must be increased as described in the next paragraph.

By combining the two previous approaches the **set-associative cache** is constructed. The field *set* in the address refers to multiple cache lines. The cache lines in the given set are called ways. According to the number of ways this type of cache is also called **n-way associative cache**. If there are 2 ways in the cache, where for certain *set* one way has already valid cache line, and there is need to store another cache line to the same *set*, it is stored in the second, empty way. Now if the third request comes that has different *tag* but the same *set*, the cache line in one of the two ways have to be replaced. If the cache had more ways than 2 the replacement would not be necessary, cache line would be stored in any empty/invalid way. There are n comparators used in this cache type corresponding to number of ways. Modified example from the previous paragraph (64B long cache line and 256 cache lines) with 2 ways would have the size of cache equal to 32kB. Note that none of the address parts (offset, set, tag) sizes have changed.

Caches which have associativity greater than one uses various replacement strategies to a select cache line to be sacrificed when lines in all ways are valid. Among strategies mentioned in literature belong random replacement, Least Recently Used (LRU), Least Frequently Used (LFU), Adaptive Replacement Cache (ARC) combining previous two or Not Recently Used (NRU). These strategies developed in the era of single-core processor. In the multi-core processors, it seems that for per-core private caches the best replacement policy is LRU [9]. There is ongoing research targeting replacement strategies for caches shared by multiple cores in multi-core processors [10–11].

2.4.3 Writing data back to main memory

When the processor writes data to the given address, it might not end up in the main memory right away. Data typically stay in the cache for the moment and get to the main memory later.

- *write through*: policy writes data to the cache, and to the write buffer. From the write buffer data are moved to the main memory asynchronously.
- *write back*: policy writes data to the cache line marking them with flag *Dirty*. When the cache line is canceled the data are moved to the main memory.

2.4.4 Cache inclusion policy

Inclusion policy characterizes relation between adjacent cache levels. It specifies requirements on the presence of the cache line in higher level of hierarchy when it is present in lower hierarchy level.

There are three possible policies:

- *inclusive*: higher level cache is inclusive of lower level cache if the cache line present in lower level cache have to be presnt also in higher level cache
- *exclusive*: higher level cache is exclusive of lower level cache if the cache line present in lower level cache must not be present in the higher level cache
- *non-inclusive non-exclusive*: neither of the above two is required

2.4.5 Cache coherency

Since there are per-core private caches in multi-core systems, with parallel programming, it might occur that same data are loaded into two or more caches associated with different cores. If one core modifies data in its cache, it is necessary to inform other cores' caches about the change. Otherwise the incoherency makes the whole system fairly unusable.

If the private caches are interconnected with higher cache level using common bus, the invalidation may be utilized. When one core writes to given address, this address is announced as invalidated on the bus. Cores monitor events on the bus. If the core is not the originator of the announcement, it checks whether its cache contains cache line with the announced address and in case it does the cache line is invalidated.

Another approach is to send updated information. Cores read update requests on the bus, and if the request contains common data, the core updates them in its cache.

Further, more sophisticated protocols such as MOESI and friends may be utilized. Each cache line has information about its coherency state. The possible states in MOESI protocol are briefly described in table 2.1. The protocol then assures that the caches are coherent given the set of allowed states for any pair of caches in the system.

state	brief description
Modified	This cache has the only valid copy of the cache line.
Owned	There are several caches with the valid copy, but this one has exclusive write permission.
Exclusive	This cache has the only copy.
Shared	This cache line can be only read.
Invalid	This cache line is invalid.

Table 2.1. MOESI cache coherency protocol states. The validity relates to the given cache line.

2.4.6 Caches incorporated in virtual address model

Up until now the caches were described as a mean to mediate faster access to the portion of memory. Caches are also utilized to speed up some other mechanisms used in current systems.

Nowadays there is widely present memory virtualization in the contemporary systems. With this mechanism it seems to an application as it has whole memory just for itself. It is achieved by providing each app with its own virtual memory (VM). Only small, currently used, portion of program and its data located in VM is actually mapped to the physical memory. The currently unused portions of program and its data are stored in the secondary device (hard drive). The mechanism simplifies management of concurrenly running applications as their address spaces do not collide with each other.

Virtual memory is typically implemented using paging which is managed by OS. With paging the physical address space (main memory) is divided to the regions of equal size called frames. Virtual memory is also divided to the regions of equal size, but these are called pages. Size of frame and page is equal. The size of page in current systems ranges from traditional 4kB all the way up to 16GB. It is always power of 2 as this choice allows to divide the address space equaly.

Simple synthetic example that follows will clarify the behavior of the virtual memory. Consider our virtual space consists of 100 pages and physical memory is divided into 100 frames. Now consider applications – A1 utilizing 60 pages within its 100 page VM and A2 utilizing 70 pages within its 100 page VM. OS starts A1 which uses 60 pages that are mapped (loaded) to arbitrary 60 frames. While A1 is running it needs only 20 pages, because there was possibly only initialization code in other 40 pages. Now the OS starts A2. A2 needs 70 pages to overcome initializations and to get to its regular operation mode. There are still 40 empty frames in physical memory. Further there are 40 frames loaded with 40 pages belonging to A1 that were not used for the long time. 30 of these pages may be stored to the hard drive so that frames become free for use by A2.

Continuing in previous example the sizes of virtual and physical spaces may vary. By lowering the amount of physical memory to 10 frames, the mechanism would still work, but it would be considerably slower due to the necessity to move the pages between frames in main memory and the secondary device. On the other hand if there were 1000 frames available there could be loaded basically 10 applications without need to use of secondary device given the apps utilize fully its 100 pages virtual memory.

Virtual memory wouldn't bring such improvement without support for management structures in hardware. There is need for page tables allowing translation from virtual to physical address. Each process has its own page table structure. When translating virtual memory address to physical memory address, part of the virtual address is used as an index to select entry within page table. The remaining part of the virtual address is an offset within page/frame itself. Among others the page table entry may contain following informations: frame number, where the page is loaded, presence of the page (in memory or swapped out to the secondary device), some privilege information (writable, executable), and some usage statistics.

When pages have relatively small sizes, the page table would be very big. To prevent this, page table is split to the hierarchy of page tables. In figure 2.5 is an possible example of hierarchical paging on x86 platform.

As can be seen from the x86 example to read from or write to physical address there are necessary three accesses to the memory. First it is necessary to read page directory entry, after that page table entry, and finally physical address is obtained and it is accessed. Memory accesses take a long time therefore the translations from virtual page number to the physical frame number are cached in so called Translation Lookaside Buffer (TLB) which may be implemented as CAM. On inserting virtual address it returns physical address or reports cache miss. In figure 2.6 is depicted translation of virtual address to physical address utilizing TLB. The TLB may be divided to instruction cache – ITLB and data cache – DTLB which increases hit rate [13]. There are compared caches of contemporary systems in the table 2.2. Both virtual and physical

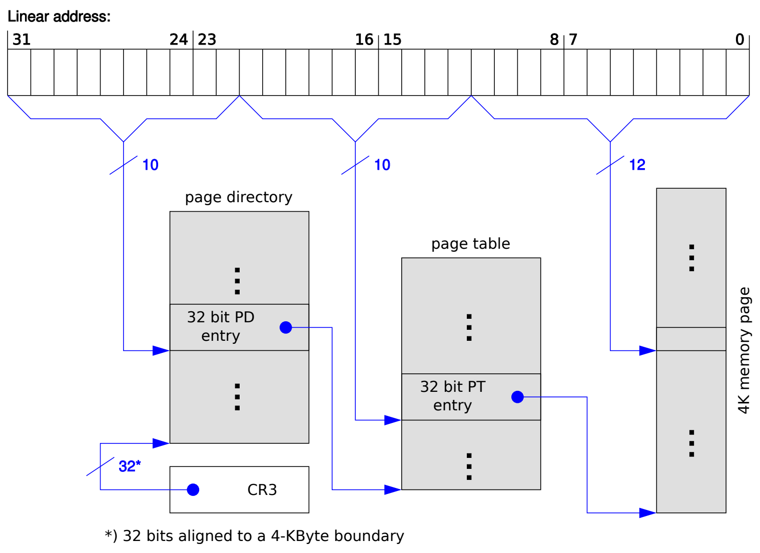


Figure 2.5. Paging with 4kB long pages on x86 architecture in 32-bit protected mode. [12]

The OS fills CR3 register with the base address of application's page directory. Linear address is Virtual address after applying x86 segmentation mechanism. Segmentation nowadays typically maps addresses one to one and we won't further elaborate on this. First 10 bits of the Linear address are used to point to entry within page directory, which effectively select the page table – the base address of the page table is stored in the entry. Next 10 bits of the Linear address are used as an index that selects an entry within the page table. PT entry contains frame number within main memory where the program's page is loaded. Remaining 12 bits of the Linear address points to the byte within 4kB long page. Notice the PDs and PTs are also 4kB long therefore they fit in the frames. PDs and PTs may also be swapped out to the secondary device.

addresses are used to select L1 cache item in the Intel platforms. This way cache set may get ready during the time physical address is retrieved from the TLB.

2.4.7 Caches in contemporary hardware

The comparison of currently used caches is best made based on their parameters, which can be seen in table 2.2.[1][2][3][4]

Interesting Intel's feature is the usage of adaptive cache allocation mechanism for their shared cache. Marketing name is Intel Smart Cache. If only one core is active, it can utilize the whole shared cache.

2.4.8 Quality of service in relation to cache

In general the possibility to monitor the cache utilization on the per-thread basis allows the operating system (OS) or hypervisor (HV) / virtual machine monitor (VMM) to reschedule tasks in a way to favor high-priority task. Incremental step is allocating the portion of the cache for such high-priority task.

[1] http://www.7-cpu.com/

[2] https://stackoverflow.com/questions/33974193/does-x86-64-cpu-use-the-same-chache-lines-for-communicate-between-2-processes-vi

[3] http://www.realworldtech.com/sandy-bridge/7/

[4] https://superuser.com/questions/745008/whats-the-difference-between-physical-and-virtual-cache

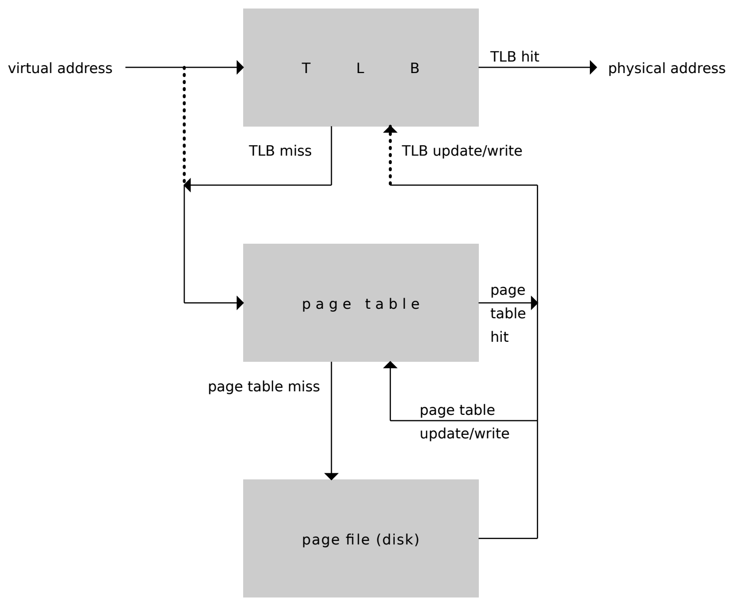


Figure 2.6. Principle of TLB and illustration of Page Table miss. [12]

When TLB has virtual address cached it responds with physical address. If the physical address can't be obtained from the TLB there is need to do a page walk. If either page table or target page is not present in the memory, OS loads them from the hard drive to the main memory. Then the physical address is obtained using the paging mechanism, the record is created in the TLB (i.e., VA is cached) and the physical address is passed to the next stage.

An implementation of cache QoS is available for example in the Intel Resource Director Technology RDT. The details on the RDT are available in section 3.2.3. Another example utilizing QoS in relation to cache could be Qualcomm's Centriq 2400 [1)].

[1)] `https://www.anandtech.com/show/11737/analyzing-falkors-microarchitecture-a-deep-dive-into-qualcomms-centriq-2400-for-windows-server-and-linux/2`

	Intel Haswell	Intel Skylake	IBM POWER8	ARM Cortex-A1	AMD Jaguar
L1 data	32 KB	32 KB	64 KB	32 KB	32 KB
	64 B/line	64 B/line	128 B/line	64 B/line	64 B/line
	8-WAY	8-WAY	8-WAY	2-WAY	8-WAY
- indexed	virtually	virtually			
- tagged	physically	physically			
L1 instr.	32 KB	32 KB	32 KB	—	32 KB
	64 B/line	64 B/line	—		64 B/line
	8-WAY	8-WAY	8-WAY		2-WAY
- indexed	virtually	virtually			
- tagged	physically	physically			
L2 cache	256 KB	256 KB	512 KB	1 MB	2 MB
	64 B/line	64 B/line	128 B/line	64 B/line	64 B/line
	8-WAY	4-WAY	8-WAY	16-WAY	16-WAY
	NINE	NINE			
- indexed	physically	physically			
- tagged	physically	physically			
L3 cache	8 MB	8 MB	8 MB	—	—
	64 B/line	64 B/line	128 B/line		
	—	16-WAY	8-WAY		
	inclusive	inclusive			
- indexed	physically	physically			
- tagged	physically	physically			

Table 2.2. Cache parameters of contemporary hardware platforms.
Note: some specific model parameters may vary.

2.5 Performance Monitor Unit (PMU)

PMU is a logical component of the processor. PMU serves the purpose of capturing and reporting summaries of various performance events. With this component it is possible to profile applications. It is possbile to measure characteristics such as cache misses, cache accesses, memory accesses, and others.

The value of the measured event is counted in the performance monitor counter. There is limited amount of the counters available on platforms and each counter has configuration register. To start counting event its number needs to be set in the configuration register of the selected counter.

When reading core specific value such as L1 cache characteristics, on typical platform it is necessary to read the register using given core.

2.6 Hypervisor

A hypervisor (HV) is virtual machine manager and monitor. Typically the hypervisor manages multiple instances of virtual machines running on one physical computer. Such a computer needs to support hardware virtualization (otherwise the virtualization would be significantly slower). Virtual machines called guest machines run operating systems and guest OSes run in turn the target applications.

Hypervisor running on host machine manages the hardware and provides guests with access to the hardware. Since there exist cases of one instance of hardware (for example there is typically only one instance of the ethernet interface) the HV provides guests with virtualization of such a hardware so that each can access such a scarce resource. Typical HV may also schedule accesses to the shared resources. When such a resource is currently assigned to one VM another VM has to wait until the scarce resource becomes available.

Historically hypervisors were classified as Type-1 for native hypervisors running on bare metal and Type-2 hypervisors as an application under host operating system [14]. Figure 2.7 highlights the difference.

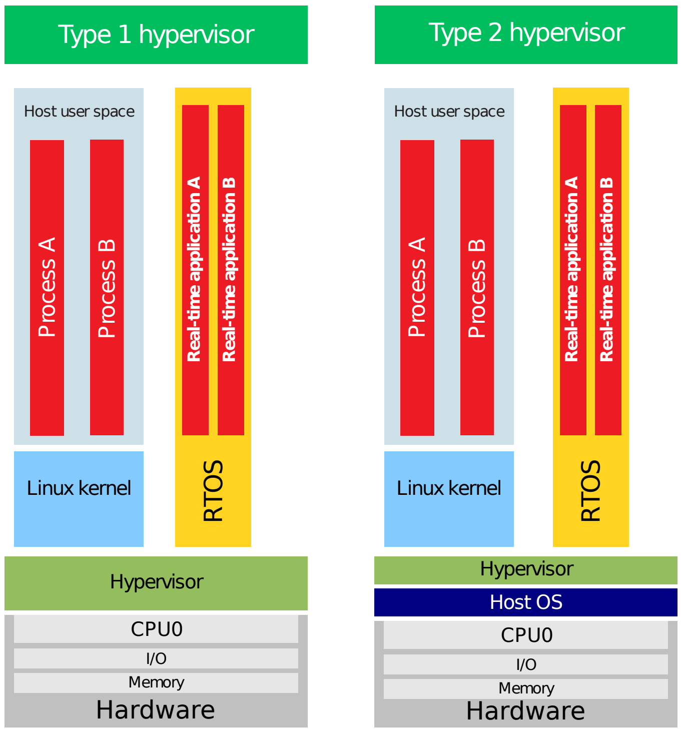


Figure 2.7. Hypervisor types comparison. [15]

2.6.1 Hardware partitioning

With the generic hypervisor there are hardware resources shared among VMs. The scheduling of access to the hardware is a contention point and potentially source of time unpredictability. The solution to overcome such unpredictabilities that may be employed is partitioning hardware with the supervision of hypervisor. The contention point is removed by dedicating certain hardware to certain VM. With this setup there can be a set of real-time tasks, each in its own VM with preallocated hardware resources necessary for the task's operation. The tasks are separated from each other and the risk of one task interfering with another is lowered significantly.

To perform beneficial task, the machine, in our case virtual machine, needs at least some memory for data and code, execution unit, and output device. If supported by the HV (and by HW), the following resource may be partitioned and assigned to virutal machines:

- processor cores
- cache
- memory
- input/output ports (on x86)

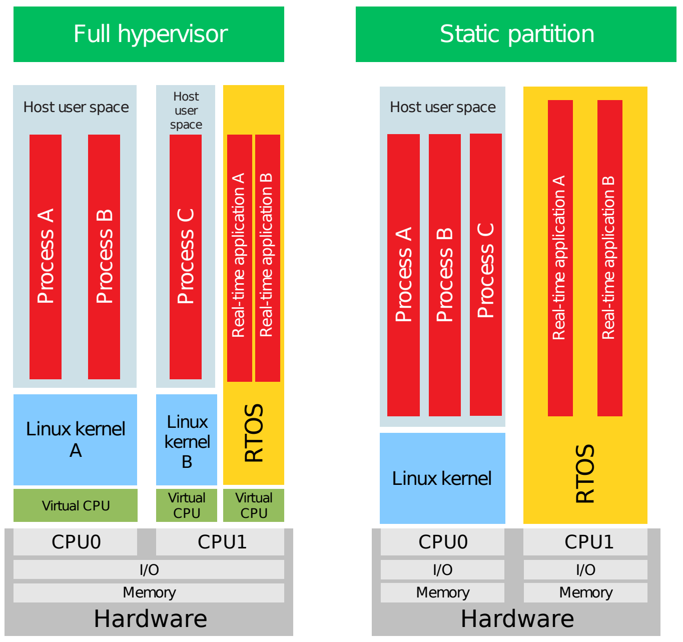


Figure 2.8. Comparison of virtualization types. [15]

- memory mapped input/output devices
  - PCI devices
  - UARTs
  - VGA text console
- IRQ chips

The hardware partitioning may be also called static allocation of resources. Compare full hypervisor with static partitioning hypervisor in figure 2.8. [16]

2.7 PRedictable Execution Model (PREM)

This section is based on the knowledge of Real-time systems that were described in section 2.1 and contemporary memory systems described in section 2.4.

In a typical computer system, there are several execution units – processor cores and other subsystems that share the main memory and contest for the main memory bandwidth. If two or more resources request access to the main memory at the same time, one of them gets the access while the others are stalled. Considering real-time systems requirement for predictability this behavior is undesirable due to its unpredictability. An execution unit might miss its deadline while waiting for the data. To address this problem, time slots could be allocated for each device in which it can access memory so that the access is dedicated and schedulable. That is exactly the main idea behind PRedictable Execution Model (PREM) – the ability to coschedule all active components (e.g., CPU cores, I/O peripherals) in the system at a high level. [17]

Each device has its time slot where it can access the needed data for further processing, but obviously, the device itself must have some local storage to bridge the time interval when it has no access to the main memory. There are two local storage options discussed in the literature that can be used for this purpose in CPUs – scratchpad memories (SPMs) and caches.

Scratchpad memory is fast and small on-device memory (SRAM) addressable directly from the device itself. Unlike cache, SPM is not transparent to the programmer, thus either programmer or compiler must issue instructions to load data into the scratchpad memory before processing them. SPM may be either mapped to global or separate address space, where the latter is more likely [18]. SPMs are not part of the mainstream hardware, so we will not discuss them further.

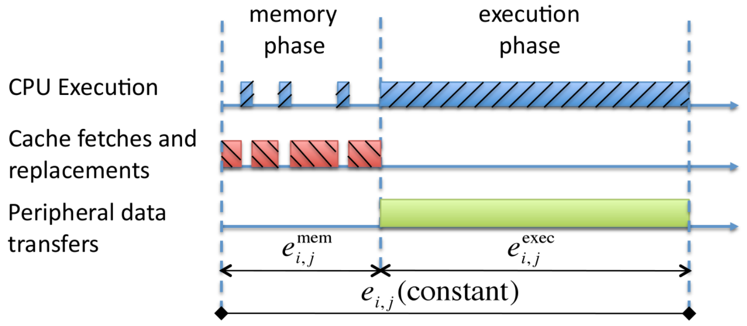


Figure 2.9. Model of a task in task-based execution system.

There is predictable time interval with constant execution time. In the memory phase CPU loads all data it needs and processes them in the execution phase. While CPU core is processing data memory is free to be accessed by peripherals (e.g., GPU) or other cores. $e_{i,j}^{mem}$ is the time duration of memory phase and $e_{i,j}^{exec}$ is the time of execution phase, where index i marks task number and since each task is divided into scheduling intervals those are numbered with index j. [17]

Pellizzoni et al. [17] describe how to schedule tasks with I/O operations, so that whole system execution is predictable while using COTS platforms. Two classes of intervals are scheduled. Compatible intervals are compiled and executed in no particular manner. Within compatible intervals I/O data flows are not allowed, there are allowed only I/O interrupts of peripherals associated with given task. Predictable intervals have two phases as depicted in figure 2.9 – during memory phase data are exchanged between main memory and cache of the corresponding CPU, and during the execution phase, the data loaded into caches are processed. The I/O operations accessing main memory are also scheduled in the execution phase.

There was created a PREM real-time C compiler [17]. Macros are used by the programmer to mark parts of predictable interval function. With the help of these annotated intervals compiler passes transform the interval to be predictable. The PREM compiler source code is not publicly available.

2.7.1 WCET-aware compiler

A compiler is a program that translates executable source code in one language to an executable source code in another language. Traditional compilers have two parts – frontend and backend. In this setup, the frontend translates source code into intermediate representation (IR), IR is optimized in case of optimizing compiler, and finally, the backend translates IR to the machine code.

With the IR in the middle the compiler becomes extensible more easily. To add support for another language only the frontend needs to be modified. Similarly to add support for the different target machine, the new backend needs to be added. This

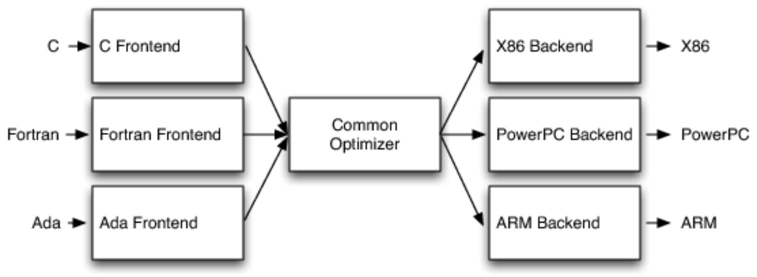


Figure 2.10. Multi-target compiler.
On the left, there are frontends for different languages that get transformed to the compiler intermediate representation (IR). IR is optimized and transformed to the target machine code by the appropriate backend.

approach makes the IR important part of the compiler. Insightfully designed IR can be optimized well, and that leads to somewhat better output programs than the original.

The goal of optimizations is to make the code better in some way. The programs can be optimized for speed when running, for the size of the resulting code, for energy efficiency, and for other purposes. IR is not the only stage that can be optimized. The resulting machine code is another stage where the optimization opportunity occurs. However, these optimizations require more effort since the stage needs to be reimplemented for each backend.

2.7.2 Decoupled Access Execute (DAE)

The approach that divides the execution into phases that are called *access phase* and *execute phase* is described in work of Koukos et al. [19]. These phases are analogic to the *memory phase* and *execution phase* in PREM [17]. Authors of DAE use the separated phases to optimize the energy efficiency of general purpose programs. They utilize hardware capability Dynamic Voltage and Frequency Scaling (DVFS) to decrease frequency during access phase when the data are preloaded from the main memory to the cache. The slow down in this phase is negligible since the processor waits for the data to be fetched anyway. Execute phase runs using high frequency. The results show that for memory-bound applications there are improvements not only in energy efficiency but the applications even run faster.

2.8 MemGuard: limiting memory bandwidth of CPU cores

MemGuard is a tool for memory performance isolation. It serves as an efficient memory bandwidth reservation system. “MemGuard distinguishes memory bandwidth as two parts: *guaranteed* and *best effort*.” [5] “It improves system throughput by exploiting best effort bandwidth after each core satisfies its guaranteed bandwidth.” [5] As MemGuard operates with available resources it needs to be implemented within OS or VMM.

Memory performance isolation means “that the average memory access latency is no larger than when running on a dedicated memory system” [5]. Potential delay in the DRAM controller can be minimized by regulating the per-client aggregated requests to the DRAM controller. This comes from the fact that thanks to MG the sum of bandwidth from all the clients is less than or equal to the possible memory bandwidth and so the memory requests are likely to be processed immediately.

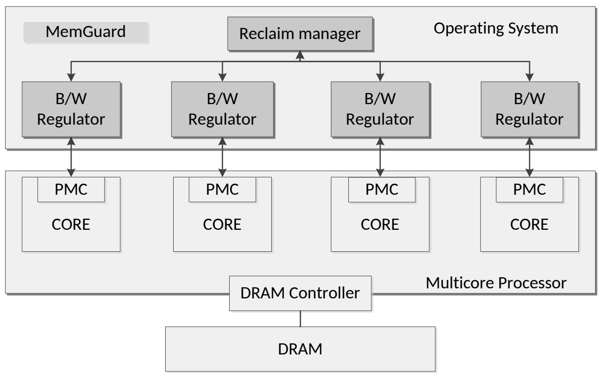


Figure 2.11. MemGuard system architecture. [5]

"The per-core regulator is responsible for monitoring and enforcing its corresponding core memory bandwidth usage. It reads the hardware PMC to account the memory access usage. When the memory usage reaches a predefined threshold, it generates an overflow interrupt so that the specified memory bandwidth usage is maintained. Each regulator has a history based memory usage predictor. Based on the predicted usage, the regulator can donate its budget so that cores can start reclaiming once they used up their given budget. The reclaim manager maintains a global shared reservation for receiving and re-distributing the budget for all regulators in the system. " [5]

MemGuard consists of two main components: the per-core regulator and the reclaim manager. Its design is described [5] along with the design outline in figure 2.11.

The budget assigning example and its cited [5] description in figure 2.12 helps to understand how MemGuard operates.

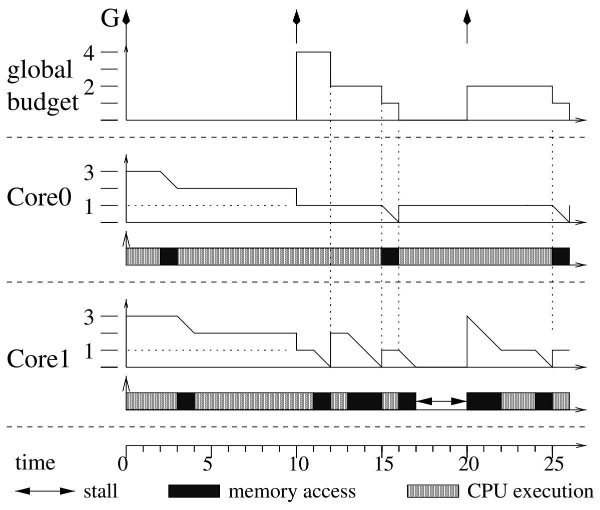


Figure 2.12. MemGuard budget illustrative example with two cores. [5]

"Each core has assigned static budget 3 (i.e., $Q_0 = Q_1 = 3$). The regulation period is 10 time units and the arrows at the top of the figure represent the period activation times. The figure demonstrates the global budget together with these two cores.

When the system starts, each core starts with the assigned budget 3. At time 10, the prediction for each core is 1 as it only used budget 1 within the period [0,10], hence, the instant budget becomes 1 and the global budget G becomes 4 (each core donates 2). At time 12, Core 1 depletes its instant budget. Since its assigned budget is 3, Core 1 tries to reclaim 2 from G and G becomes 2. At time 15, Core 1 depletes its budget again. This time Core 1 already used its assigned budget, only a fixed amount of extra budget (Q_{min}) 1 is reclaimed from G and G becomes 1. At time 16, Core 0 depletes its budget. Since G is 1 at this point, Core 0 only reclaims 1 and G drops to 0. At time 17, Core 1 depletes its budget again then it dequeues all the tasks as it can not reclaim additional budget from G. When the third period starts at time 20, the $Q_1^{predict}$ is larger than Q_1. Therefore, Core 1 gets the full amount of assigned budget 3, while Core 0 only gets 1, and donates 2 to G. At time 25, after Core 1 depletes its budget, Core 1 reclaims an additional budget Q_{min} from G." [5]

2.9 Page coloring as a software solution addressing cache contention

One cache contention solution was addressed in the section 3.2.3 dedicated to cache partitiong supported in hardware using Intel RDT. Another purely software method to partition cache that I want to mention is Page Coloring [7, 13]. An example explaining the principle of the matter is in figure 2.13. This software based cache partitioning may be enabled when there is support in the OS or the Hypervisor.

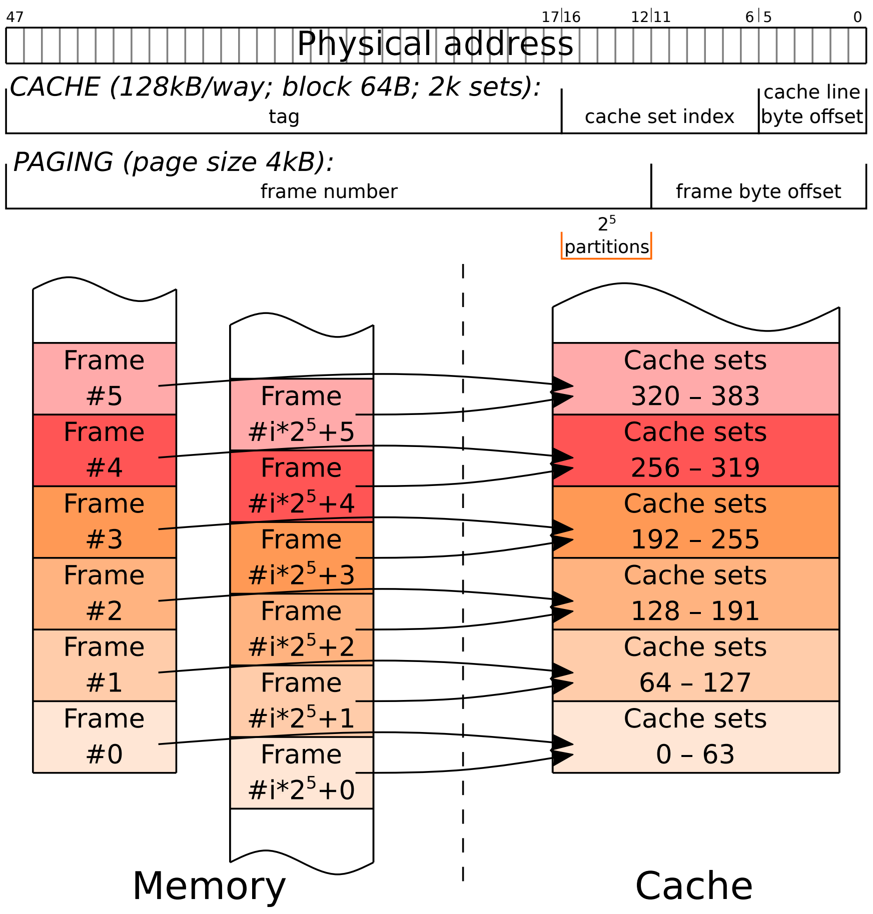


Figure 2.13. Example of page coloring used to partition cache.[7]

Consider system with the following parameters. Physical address is 48-bits long. Pages of virtual memory are 4kB in size. Cache is indexed using physical address. It has 128kB per way, its block size is 64B, and it has 2048 sets.

All the frames that have low 5 bits of frame number in common are cached in specific portion of 64 cache sets. In this example there may be total of $2^5 = 32$ partitions. The bits that determines partition number are referred to as *page color*[7]. Virtual memory manager has to ensure client of the partitioning has its virutal pages mapped into frames with numbers following this formula: $i * 2^n + p$, where $i \in \aleph_0$, $n = log_2(number\_of\_cache\_sets) + log_2(cache\_line\_size) - log_2(page\_size)$, and per client chosen partition $p \in \langle 0; 2^n)$ Notice the page size have to be less than cache size in one way for the partitioning described to be possible.

Chapter 3 Methodology

The chapter describes:

- used hardware platforms
- evaluation hypervisor Jailhouse
- design of experiments

The description of evalutation platforms includes NVIDIA TX2 and Intel Xeon W. System on chip Tegra X2 made by NVIDIA is to be found in section 3.1. The platform to evaluate cache partitioning effects is Xeon W processor from Intel described in 3.2.

3.1 NVIDIA Tegra X2

The primary platform used to evaluate methods increasing time predictability is the Jetson Tegra X2 made by NVIDIA. At a time, it was marketed as "the fastest, most power-efficient embedded AI computing device" [1]. There is a summary of parameters of the TX2 SoM in the table 3.1.

A variant of this platform is used in the applications such as Tesla Motors' self-driving capability [2] or infotainment system of the Mercedes-Benz [3] and others.

parameter	value
Power consumption	< 15 Watts
Main memory	8 GB LPDDR4
	128-bit bus
Peak memory bandwidth	59.7 GB/s
CPU	NVIDIA Denver2 (dual-core)
	ARM Cortex-A57 (quad-core)
GPU	Pascal (256-core)
SW support	Ubuntu 16.04
	Kernel 4.4

Table 3.1. Selected parameters of the Tegra X2 system on module.

TX2 platform is used in this work to see the effects of memory bandwidth throttling on the level of memory controller. We want throttle especially the GPU to ensure certain level of memory bandwidth is available for the CPUs. The throttling of the GPU on the memory controller level was previously done and used on the TX1 platform. Further we evaluate the concept of MemGuard on this platform. The MemGuard implementation is dependant especially on the CPU's Performance Monitor Unit (PMU). Therefore

[1] `https://developer.nvidia.com`
[2] `https://electrek.co/2017/05/22/tesla-nvidia-supercomputer-self-driving-autopilot/`
[3] `https://blogs.nvidia.com/blog/2018/01/09/mercedes-ces-2018/`

futher sections focus especially on the description of memory controller of the TX2 platform and the PMUs used.

Note: The documentation [20] of the TX2's memory controller is rather sparse whereas the TX1's memory controller is documented in its manual [21] substantially better. From the comparison of register descriptions in both documentations follows that the MC of TX2 is heavily based on the one of TX1. Therefore we use TX2 documentation as an incremental documentation above TX1 docs. Some MC registers are completely left out of the TX2 documentation. As an example of the omitted registers those related to PTSA could be mentioned. There exists another source allowing to check which registers are available on the platform and it is the Linux kernel headers provided by NVIDIA.

For the overlapping MC design of the TX1 and TX2 discussed in previous paragraph, sources from both platforms' documentations are compiled to describe memory controler in coming sections.

3.1.1 System organization

TX2 platform is based on the Parker series processor. Its block diagram may be seen in the figure 3.1.

The module utilizes shared memory over all data consumers i.e., CPUs and GPU access the same physical memory, meaning there is no dedicated memory for the GPU.

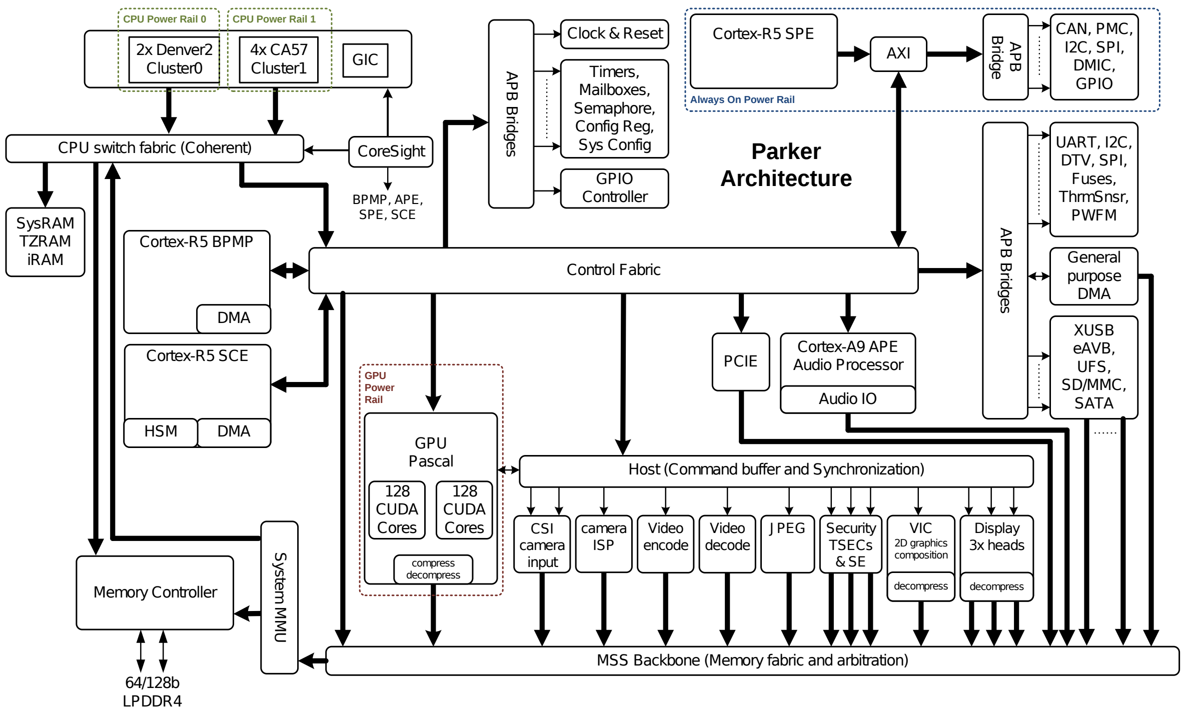


Figure 3.1. Block Diagram of a Parker series processor. [20]

3.1.2 Memory subsystem

There are several technical means utilized when moving data along the path from the main memory (DRAM) modules to the consumers of the data and back. It includes components that are described in the following paragraphs. The scheme of the memory controller components is in the figure 3.2. Following terms are taken over from [20].

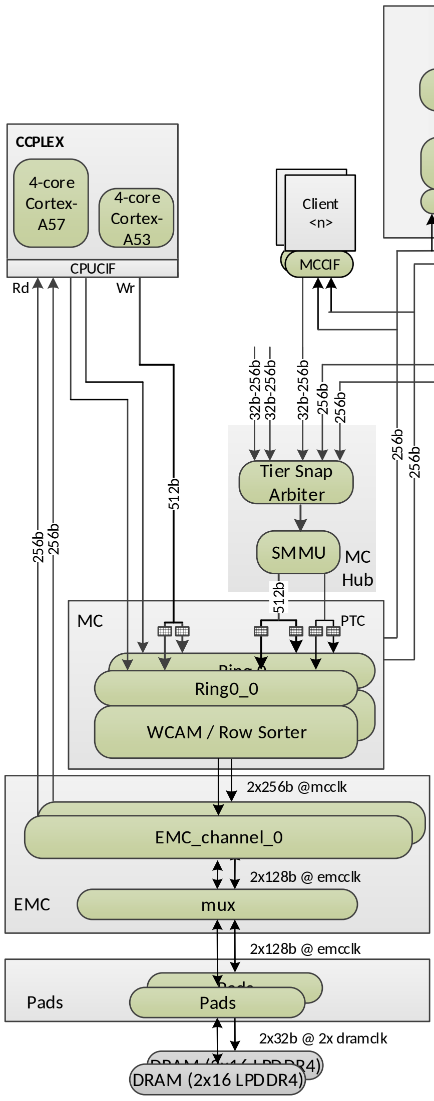


Figure 3.2. Data path from the memory modules to the memory controller clients in Tegra X1. [21]

External Memory Controller (EMC) is a module that interfaces with external DDR devices.

Memory Controller (MC) is a module that handles requests from internal clients and arbitrates among them to allocate memory bandwidth.

Write Content Addressable Memory (WCAM) presents a point of global visibility for writes within a single channel, but read-after-read ordering is not guaranteed by WCAM.

Row Sorter reschedules reads and writes. It is a pending request buffer that sorts requests by the DRAM row it refers to. Write rescheduling is hidden behind WCAM.

System Memory Management Unit (SMMU) is a block within the memory controller used to map from a virtual address space to physical addresses for device DMA.

Ring arbiter is a type of round-robin arbiter. Memory client requests are arbitrated through a sequence of ring arbiters.

Priority Tier Snap Arbiter (PTSA) is a rate control mechanism above each ring.

Memory Controller Client InterFace (MCCIF) is the standard interface block between the memory controller sub-system fabric and the client device.

3.1.3 Throttling memory controller clients

Our main focus in relation to the memory controller is exploration of the ability to throttle the bandwidth of memory controller clients. First there is described the throttle infrastructure and then MC related registers that allows throttling of clients.

The figure 3.3 below is a view of memory client request datapaths through the PTSA rate control mechanism in the Tegra X1. Memory client requests are arbitrated through a sequence of ring arbiters which perform a type of round-robin arbitration. There are three ring arbiters referred to as ring 0, ring 1, and ring 2. Each ring has a rate control mechanism referred to as Priority Tier Snap Arbiter (PTSA). The client's bandwidth guarantee is specified by the PTSA rate. [21]

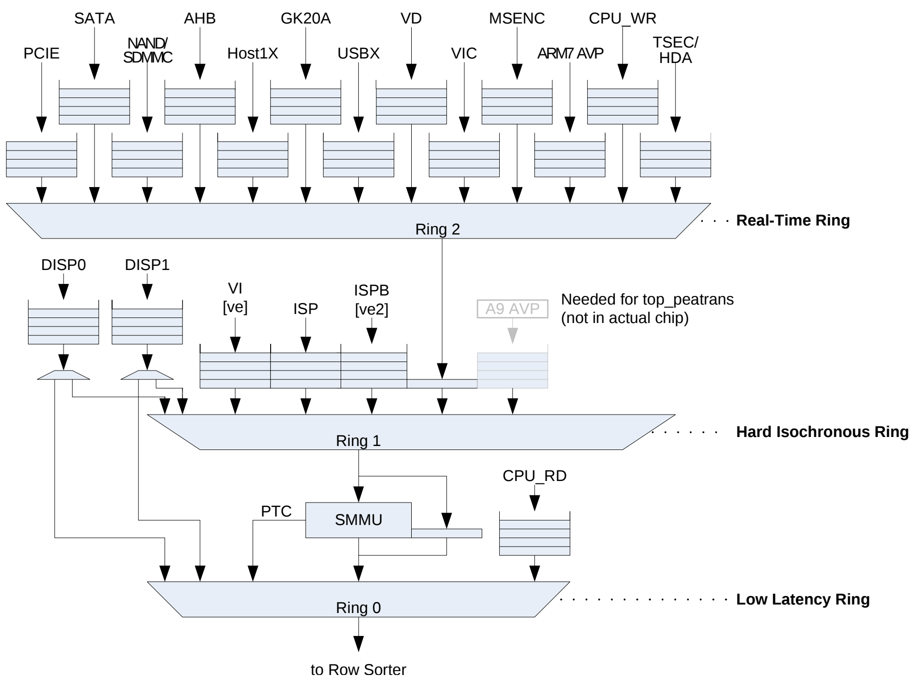


Figure 3.3. Snap arbiter tree of the memory controller clients in Tegra X1. [21]

Ring 0 serves the following clients:

- CPU reads (mpcorer)
- L2 cache (ftop) – CPU writes through L2
- Page table cache (ptc)
- Raw output from ring 1
- Output from ring 1 translated through SMMU

Ring 1 has the following clients:

- Ring 2
- Display
- Camera

Ring 2 serves every other client including e.g., two graphics card client instances. The clients connected here are of non-isochronous nature. Since requests of this ring goes through both lower rings there are higer time jitters to be expected between processed requests.

The following knobs affect the fairness algorithm in snap-arbitration [20]:

- Client PTSA (DDA) settings
- RING1_THROTTLE, MAX_OUTSTANDING, RING0_THROTTLE_MASK
- NISO_THROTTLE, MAX_OUTSTANDING_NISO, and NISO_THROTTLE_MASK
- BLOCK_LP_CPU_RD_IF_SMMU_INP_HP

It is possible to select which clients get throttled above ring 0 and ring 2. The MASK registers listed in table 3.2 serves this purpose. On top of ring 2 the group of clients to be throttled is called non-isochronous (NISO) meta-client group. The THROTTLE registers listed in the same table allows to set number of stall cycles that get inserted at the input to the ring after every request from any of the clients included in the meta-client group in question. The number of requests pending in the row sorter can affect whether ring1 or ring2 arbiters are throttled (slowed down) based on thresholds (OUTSTANDING_REQ registers). There are two values settable in THROTTLE register. One for the case when number of requests is below OUTSTANDING_REQ threshold and another for the case above the threshold.

Ring	Grouped Clients	Wait Cycles	Trigger Requests
2 input	NISO_THROTTLE_MASK, NISO_THROTTLE_MASK_1	NISO_THROTTLE	OUTSTANDING_REQ_NISO
2 output		RING3_THROTTLE	OUTSTANDING_REQ_RING3
0 input	RING0_THROTTLE_MASK	RING1_THROTTLE	OUTSTANDING_REQ

Table 3.2. Registers used to throttle groups of clients.
Note: registers named RING3 are not a mistake, it is more of a naming legacy.

PTSA registers are used to set the bandwidth of clients. Client snap levels are expected to be programmed statically based on client type and its needs. Boot code programs and locks the override control register for certain clients. Registers may be configured dynamically to provide bandwidth guarantees to ISO clients.

The important statement of this section is the possibility to throttle group of MC clients by inserting certain number of stall cycles after each memory request. During stall cycles no request is served. That is how bandwidth should get throttled.

3.1.4 Processing units

In TX2 there are two CPU clusters contained in TX2 platform and one GPU. There is present Quad-core Cortex-A57 cluster, which is an implementation of ARM intelectual property, along with Dual-core Denver2 cluster created by NVIDIA. Denver uses a combination of simple hardware decoder with software-based dynamic binary recompilation storing the code it optimized in a portion of main memory. From the programmer perspective the Denver core implements ARMv8 instruction set architecture following the ARMv8 programming model and it should effectively behave as an A57. There is slightly better performance in Denver compared to Cortex-A57.

Both clusters has L1 data and L1 instruction per core caches and L2 unified per cluster cache. Cache line size is 64B. There are listed details of the TX2 CPU caches parameters in the table 3.3.

Cluster	Level	Type	Size	Ways	Sets	Write Policy
A57	L1	Data	4x 32K	2	256	Write Back
	L1	Instruction	4x 48K	3	256	
	L2	Unified	2M	16	2048	Write Back
Denver2	L1	Data	2x 64K	4	256	Write Through
	L1	Instruction	2x 128K	4	512	
	L2	Unified	2M	16	2048	Write Through

Table 3.3. Cache parameters of TX2 CPU clusters [20]

Each core may operate on certain exception level (EL). There are four EL in ARMv8 marked EL0 to EL3. Higher values of EL indicate increased software execution privilege. Execution at EL0 is called unprivileged execution. EL1 is the level meant for execution of operating system kernel. EL2 provides support for virtualization and lastly EL3 provides support for switching to Secure state and back to Non-secure state. When EL3 is implemented there exists portion of physical address space available only from the Secure state.

GPU is based on Pascal microarchitecture. There are two streaming multiprocessors each consisting of 128 CUDA cores. GPU is attached to the PTSA of the memory controller through two interfaces. These are present to favor GPU data retrieval over other clients also attached to the ring 2. Duplicating MC interfaces is a simple mechanism how to favor GPU which is expected to have much higher data consumption over other clients.

3.1.5 Performance Monitor Unit (PMU)

There may be implemented up to 31 event counters PMEVCNTR<n> each with the length of 32 bits. Each of these registers may be configured to count certain event type with the PMEVTYPER<n>_EL0 of the same number <n>. These counting registers overflow when they wrap. [22]

To give an example there are a few performance events in the following list:

- Software increment – architecturally executed instruction
- Level 1 instruction cache refill
- Level 1 data cache refill
- Level 1 data cache access
- Level 2 data cache refill
- Level 2 data cache access
- Bus Access
- Data memory access
- ...

PMU allows to raise an interrupt to notify software when the counter overflows. If the software wants to count specific amount of performance event occurrences it has to preset the counter register with the maximum value the counter register can hold decreased by that specific counted amount. The PMU can filter events by combinations of Exception level and Security state. Filtering is settable in the PMEVTYPER<n>_EL0 registers.

Compare with the Intel's PMU in section 3.2.4.

3.2 Intel Xeon W

This section briefly describes Intel Xeon W-2133. The processor is employed by this work to see the effects of cache partitioning technology. The cache partitioning is available in Resource Director Technology under the term Cache Allocation Technology (more in section 3.2.3).

Xeon labeled with W is a succesor to the E5 series. The Intel Xeon W-2133 is made in 14nm litography technology and belongs to Skylake microarchitecture. It has integrated 6 cores and can handle 12 threads at a time. Operation frequency moves in range 3.6 – 3.9 GHz. Processor's thermal design power is 140W.

3.2.1 Memory controller

Parameters of memory controller integrated in Xeon W-2133 [1]:

- Maximum Memory Size: 512GB
- Maximum # of Memory Channels: 4
- Memory Type: DDR4
- Maximum Memory Frequency: 2666 MHz
- Maximum Memory Bandwidth: 85.3 GB/s

3.2.2 Cache organization

Cache line has size of 64B. Table 3.4 gives an overview of the CPU's cache organization.

Level	Type	Size	Ways	Sets	Write Policy
L1	Data	6x 32K	8	64	Write Back
L1	Instruction	6x 32K	8	64	
L2	Unified	6x 1024K	16	1024	Write Back
L3	Unified	8448K	11	12288	Write Back

Table 3.4. Cache parameters of Xeon W-2133 CPU [Linux sysFS records]

3.2.3 Resource Director Technology (RDT)

Resource Director Technology brought possibility to monitor and allocate cache. RDT was first introduced with Xeon E5-2600 v4 and it operated above LLC. Since than it is available on certain Xeon series e.g., Intel Scalable Processors family. The technology is the mean to assure Quality of Service. It targets shared resource contention between co-running applications. [23]

Intel RDT consists of following technologies:

- Cache Monitoring Technology (CMT)
- Cache Allocation Technology (CAT)
- Memory Bandwidth Monitoring (MBM)
- Code and Data Prioritization (CDP)
- Memory Bandwidth Allocation (MBA)

RDT above L3 cache gradually developped with Xeon versions as shown in table 3.5. The technology or its parts might not be available on all Xeon processors.

[1] `https://www.intel.com/content/www/us/en/products/processors/xeon/w-processors/w-2133.html`

Xeon Version	Microarchitecture		RDT increment
v2	Ivy Bridge	22nm Tick	*N/A*
v3	Haswell	22nm Tock	CMT
v4	Broadwell	14nm Tick	MBM, CAT, CDP
v5	Skylake	14nm Tock	MBA
v6	Kaby Lake	14nm optimized	

Table 3.5. RDT support that appeared on certain Xeon processor of the given versions. [20]

To use RDT there must be Intel VT-x feature enabled in the BIOS setup.

The desription of the individual technologies associated under RDT follows. [1] [24]

Cache Monitoring Technology allows the OS or VMM to recognize whether some thread is negatively interfering with another high priority thread, application, or virtual machine. Misbehaving thread may be then migrated or postponed.

Cache Allocation Technology is an incremental step over CMT. “CAT is a way-based hardware cache-partitioning mechanism for enforcing quality-of-service with respect to LLC occupancy.” [25] There can be allocated certain portion of the total cache size for thred, application, or VM. Effect of the CAT is depicted in the figure 3.4 that depicts CAT effect.

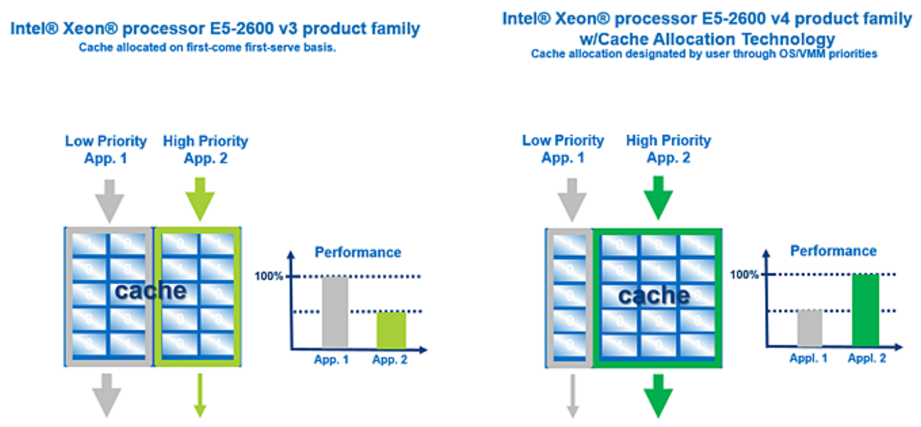

Intel® Xeon® processor E5-2600 v3 product family
Cache allocated on first-come first-serve basis.
Low Priority App. 1
High Priority App. 2
cache
Performance
100%
App. 1
App. 2
Intel® Xeon® processor E5-2600 v4 product family w/Cache Allocation Technology
Cache allocation designated by user through OS/VMM priorities
Low Priority App. 1
High Priority App. 2
cache
Performance
100%
Appl. 1
Appl. 2

Figure 3.4. Intel RDT Cache Allocation Technology. [24]

Another technology from the bundle, Memory Bandwidth Monitoring, which allows the per-core or thread memory bandwidth to be monitored. Based on the information from monitoring can OS or VMM adjust scheduling accordingly.

Code and Data Prioritization is an extension to the CAT. CDP allows to optimize LLC layout for the given workload characteristics as it can prioritize separately data and instruction.

Lastly Memory Bandwidth Allocation technology help enforce a limit on the memory bandwidth each thread or VM can use. MBA uses credit-based throttling mechanism.

CMT, MBM and CAT are configured using Model Specific Registers (MSRs). MSRs are part of many control registers implemented on the x86 platform.

In this work we are most interesed in the CAT. There is certain number of classes of service (COS) defined. In relation to cache the COS is basically a bitmask where each bit corresponds to the portion of cache. It is possible to define COSes with shared regions or completely isolated regions. To prevent cache sharing among clients the

[1] https://wiki.xenproject.org/wiki/Intel_Platform_QoS_Technologies

COSes are set in a way that when bit at position i in the mask of one COS is set to one, then in any other COS there must be bit at the same position i set to zero (given bit value equal to one means the corresponding portion of cache is included in the COS and value equal to zero means the cache portion is not included). The bitmask based COS provides an abstraction independent of the partitioning scheme. When the COSes are constructed they can be groupped into threads, applications, or VMs. There is a hardware register known as "PQR" into which scheduler of the OS or VMM writes required COS of the unit being scheduled. [26]

Since CAT is way-based partitioning and our Xeon W has 11-way associative LLC, the COS has 11 bits.

3.2.4 Performance Monitor Unit (PMU)

The principle is the same as for TX2's PMU described in 3.1.5. Xeon W-2133 identifies itself with CPUID signature DisplayFamily_DisplayModel of 06_55H. It corresponds to the Intel® Xeon® Processor Scalable Family based on Skylake microarchitecture. Appropiate section is available in the Intel software developer's manual combined volumes describing this family's Performance Monitoring [27].

"Performance monitoring was introduced in the Pentium processor with a set of model-specific performance-monitoring counter MSRs. These counters permit selection of processor performance parameters to be monitored and measured. The information obtained from these counters can be used for tuning system and compiler performance." Intel processors newer than Intel Core Duo generation support enhanced architectural performance events and non-architectural performance events. [27]

There are finite number of performance event select IA32_PERFEVTSELx MSRs and corresponding number of performance monitoring counter IA32_PMCx MSRs to capture selected events. These performance related MSRs are available per core. Besides selecting the performance event to be measured allows PERFEVTSEL register to enable/disable evenets counting, enable/disable interrupts on overflow, and it contains some further settings.

In the table 3.6 there are few of the events that can be measured. The Event Number and Unit mask Value choose the event and they are filled in the specific IA32_PERFEVTSELx to measure the chosen event.

Event Name	Event Num.	Unit mask Value
Instructions Retired	0xC0	0x00
LLC Reference	0x2E	0x4F
LLC Misses	0x2E	0x41
Branch Instruction Retired	0xC4	0x00
Branch Misses Retired	0xC5	0x00
...		

Table 3.6. Selected Performance Events measurable using the PMU on Intel platform. [27]

3.3 Measuring performance events under Linux

Linux kernel has support for measuring performance events utilizing PMU. There needs to be enabled option CONFIG_PERF_EVENTS. On Intel we also need to include support for intel uncore performance events provided as kernel module.

```
CONFIG_PERF_EVENTS=y
CONFIG_PERF_EVENTS_INTEL_UNCORE=m
```

Further there is a system call **perf_event_open** as an interface to set up performance monitoring. The call returns a file descriptor created above an event that was passed as argument to the **perf_event_open**. The event may be restricted to certain processes or processors and it may also be grouped with other events under one fd. To control event there are **ioctl** operations called on the corresponding fd. With the linux command **man perf_event_open** the details may be obtained.

There is a security measure that, when set to certain value, limits the ability of processes to access the perf events. [1][2] To allow use of all events we need to set following to -1:

```
echo -1 > /proc/sys/kernel/perf_event_paranoid
```

The **perf_event_open** receives as a first argument description of the event to be measured. Among other its type may be set to the PERF_TYPE_RAW to select arbitrary performance event to be measured. There are though some events predefined with type PERF_TYPE_HARDWARE, PERF_TYPE_SOFTWARE, and PERF_TYPE_HW_CACHE, and others. These are specified per architecture. For intel the predefined values are in the file *arch/x86/events/intel/core.c* and contains e.g., following:

```
PERF_COUNT_HW_CACHE_MISSES = 0x412e
PERF_COUNT_HW_CACHE_REFERENCES = 0x4f2e
```

Compare with table 3.6.

3.4 Jailhouse: HV partitions HW

Jailhouse is a static partitioning hypervisor. It aims to be a minimal hypervisor and basically it is a monitor only. This HV allocates hardware exclusively for its guests. Thanks to this exlusivity, there is no need for scheduling (which means less overhead in HV) and it also means no contention for access to partitioned hardware (optimally guests would not interfere among each other). Both features are interesting from the point of view of the real-time applications. It only virtualizes those resources in SW, that are essential for the platform and cannot be partitioned in HW.

The minimal code base of Jailhouse is reached thanks to exploiting the Linux in essence as the Jailhouse bootloader. Before the JH is started, Linux configures the hardware. The approach greatly simplifies operations that has to be performed by the hypervisor. Image 3.6 shows the loading process of the Jailhouse. [16]

Jailhouse management relies strongly on the Linux infrastructure. Hypervisor provides linux kernel module which is the main component to control the hypervisor through '/dev/jailhouse' device. Jailhouse also comes with the 'jailhouse' utility, that allows to easily issue commands to control the hypervisor. The command completion is also included.

There are used special names in Jailhouse. The *cell* is a configuration that describes partitioned hardware for certain VM. The *inmate* is the code loaded to the *cell*. When enabling the Jailhouse the configuration containing extensive system resources description has to be provided. The *cell* that starts with all the resources assigned is called

[1] https://www.kernel.org/doc/Documentation/sysctl/kernel.txt – section perf_event_paranoid
[2] https://lwn.net/Articles/696216/

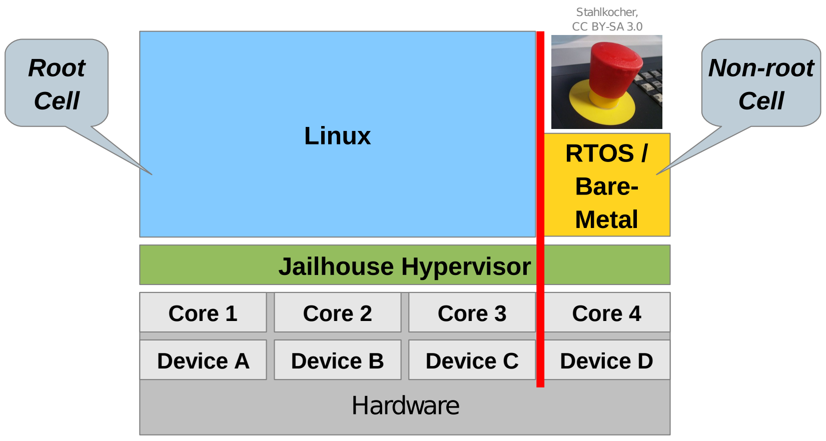


Figure 3.5. Statically partitioned hardware with Jailhouse. [28]

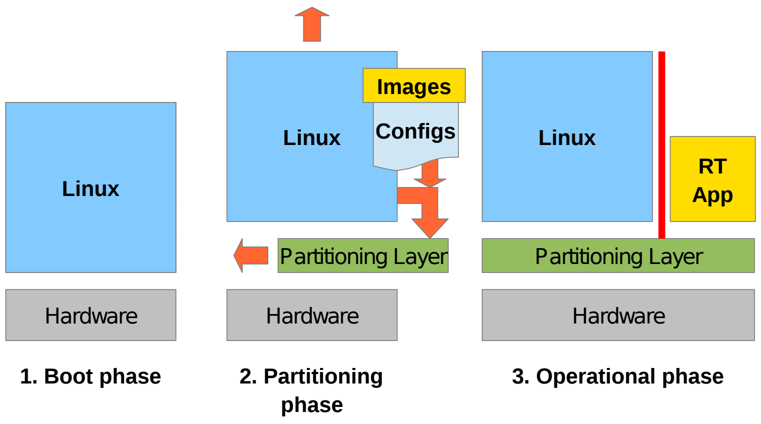


Figure 3.6. Phases in the Jailhouse loading process. [28]

root-cell. The HV takes control over these hardware resources and assigns them immediately back to the Linux *root-cell*.

The first step to create a guest (to run an inmate) within Jailhouse is creation of the cell configuration with description of hardware which the inmante will utilize. Creating new cell cuts away resources from the root-cell. When the cell is ready we may bring in the inmate. In this step the code of the guest is *loaded* to the proper virtual memory locations. After that the cell is *start*, which effectively begins the execution of the loaded code.

In the following listing is an example of starting Jailhouse with one demo non-root cell. Cf. figure 3.6.

```
    # insert jailhouse kernel module
insmod driver/jailhouse.ko
    # enable partitioning layer with available hardware description
jailhouse enable configs/root.cell
    # allocat some hardware for new cell (including memory)
jailhouse cell create configs/non-root-demo.cell
    # load code to the allocated partitioned memory
jailhouse cell load non-root-demo inmates/demos/demo.bin -a 0xf0000
    # start the execution of the new machine
jailhouse cell start non-root-demo
```

3.4.1 Setup

Root Linux settings

The Jailhouse has few requirements on the Linux that is used for JH bootstrap. Jailhouse needs some space in memory where it will load itself. We allocate the space for JH with Linux kernel option 'memmap'.

On x86 platform we need to turn off VT-d IOMMU with the option 'intel_iommu'. After disabling this for Linux, IOMMU can be fully in hands of Jailhouse when it is enabled.

In our setup we also disabled kernel address space layout randomization with the option 'nokaslr'. ASLR is a security mean that loads sections of binary to the random addresses within memory. With that turned off we get stable root-cell description of the system.

The options are passed to the Linux kernel at the boot time.

```
vmlinuz ... nokaslr intel_iommu=off memmap=82M$0x3a000000
```

Jailhouse contains command to generate root-cell description. For this 'intel_iommu' option needs to be turned on so that Linux loads right modules to enumerate all the devices.

different versions for TX2 and Xeon

I disabled HyperThreading

3.5 Benchmarks

The benchmarks we used in the experiments can be categorized into synthetic benchmarks and semi-real-world benchmarks.

The synthetic benchmarks are convenient from the perspective of results evaluation. They are simple so their behavior is predictable which makes it easier to interpret the results. We used synthetic benchmarks with the aim of maximally utilizing memory bandwidths by the computation unit that performs such benchmark. In our tests the synthetic benchamrks included sequential reads and sequential writes of an array. Since the accent of our tests is on the memory bandwidth behavior, the stride, when accessing arrays, is set to the cache line size. That allows CPU core to generate top amount of memory requests. As an synthetic benchmark for the GPU we employed *saxpy* – "Single-Precision A.X Plus Y". This GPU kernel performs simple operations above arrays in parallel which leads to very big memory bandwidth utilization.

There is often need to get closer to the real world applications, but also to retain the ability to derive the meaning of the results with ease. We used the Polyhedral Benchmark suite to design tests that model the real world applications closer. "PolyBench is a collection of benchmarks containing static control parts. The purpose is to uniformize the execution and monitoring of kernels, typically used in past and current publications." [1] Table 3.7 shows the complexity of the kernels used in experiments.

benchmark	operations	memory	O(Ops)	O(Mem)
gemm	$3n^3 + n^2$	$3n^2$	$O(n^3)$	$O(n^2)$
doitgen	$2n^4$	$n^3 + n^2 + n$	$O(n^4)$	$O(n^3)$

Table 3.7. Complexity of the used PolyBench benchmarks. [29]

[1] http://web.cs.ucla.edu/~pouchet/software/polybench/

The complexity of the benchmark helps to understand the results of experiments.

Since we can understand why the benchmarks behave certain ways we use them to understand aggregated behavior of the real world application. The real world application that we compared to synthetic benchmark is the KCF tracker, which is the tool for tracking objects visually.

3.6 Design of experiments

3.6.1 Throttling memory clients

We are especially interested in the effects of the throttling mechanism when there is high utilization of the memory bandwidth. That is the essence of our test scenario – we use threads executing on CPU and GPU that highly utilize the memory bandwidth. As the main memory is shared accross the system we should observe contention over the bandwidth. Thanks to the throttling mechanism we should be able to moderate some and prefer other clients.

We will take a look here at the CPU and GPU tasks utilizing memory bandwidth.

The memory intensive task on the CPU is mere array traversing. The array is prepared, where each element has size same as the size of the cache line. This means that with each access to an element of the array we transfer 64B (cache line size) between cache and the memory. Each element of the array contains pointer with an address to the next element of the array. Pointer of the last element points back to the first element. Now the benchmark can only access next pointers to perform sequential memory access and count the accesses. We can easily measure the bandwidth. Number of elements accessed multiplied by cache line size (item size) yields the total number of bytes transfered. The utilized bandwidth is then obtained as the total number of transfered bytes divided by the time it took to perform those transfers. Described benchmark is referred to as pointer chasing [8]. The size of the array is chosen in a way it is bigger than LLC so that the array cannot be cached and there are indeed performed memory accesses.

The core of the task for GPU is CUDA kernel *saxpy* – "Single-Precision A.X Plus Y". It is an operation over arrays defined with following formula $z[i] = a * x[i] + y[i]$, where x, y, and y are arrays of floats each of length N. Computation of each $z[i]$ requires 3 memory accesses: 2 reads ($x[i]$, $y[i]$) and 1 write ($z[i]$). Even though the GPU does not have dedicated memory it is necessary for the GPU to have knowledge about the data on which it should operate. One of the ways to do that is by utilizing cudaMemcpy that creates a copy of the data to be used solely by the GPU. To compute bandwidth we use the following formula: $\frac{3*N*sizeof(float)}{time\_in\_ns} = bandwidth[GB/s]$ There is actually performed a batch of these kernels to increase precision of measurements.

Part of the codebase used in work of Houdek et al. [30] was used as a start point to evaluate these tests.

The memory controller configuration was discussed in the section 3.1.3. We just note here the THROTTLE_MASK registers serve to select clients to be throttled and THROTTLE registers serve to set number of cycles to be inserted after each performed memory operation. THROTTLE registers are associated with arbitration point and their values may be set on the ring 0 and on the NISO group to 0 – 63 cycles and on the ring 3 to 0 – 31 cycles.

3.6.2 Profiling MemGuard implementation

Concept of MemGuard was introduced in the section 2.8. Typical length of the memory phase in PREM on this platform is in range 100–500us and length of the compute phase moves in range from 100us to 10ms [31]. Compared to memory controller level, where can be throttled various clients including GPU, MemGuard can throttle specific CPU cores. On the other hand, the memory controller allows to throttle whole CPU cluster and not the single cores. We evaluate MemGuard implementation in the Jailhouse hypervisor. The MemGuard's integration to the Jailhouse was done as a part of the efforts within HERCULES project which is an innovation action under EU's Horizon 2020 programme. Evaluation platform employed is NVIDIA's Tegra X2.

The MemGuard variant implemented utilizes per-core memory bandwidth limiter only. It is available to use by the applications through hypercall interface. For the purposes of profiling there is available patch for Linux kernel enabling the syscall as an MG limiter's interface. The tests were performed using this Linux interface.

Limiter in the implementation allows to setup memory budget for the calling CPU core plus the budget's replenish time period. Memory budget is the amount of memory acceesses that may be depleted during the time period. Time period is implemented using hardware timer and memory budget monitoring is implemented using PMU. Whenever the timer expires the memory budget is replenished. If the budget is depleted before the timer expires an PMU's interrupt is triggered. Within the corresponding ISR the core is then blocked until the timer expires and unblocks the core by replenishing the budget. The instruction wait for event (*wfe*) is employed during blocking to activate low-power standby mode of the core.

There were tested multiple performance monitor events that seemed suitable for measuring the amount of memory accesses. 'HW_EVENT_L2_CACHE_REFILL' seems most suitable. This event increases the performance counter anytime the cache line is loaded from the main memory to the LLC i.e., memory access is performed.

The interface further allows to mask low priority interrupts. With interrupts masked the profiling is shielded from distortion of the results as there are expected certain outcome values from the profiled tasks. Another option the interfece provides is periodic or non-periodic mode. The periodic mode was described as a standard mode of operation. In non-periodic mode there is occuring no limiting of the memory bandwidth. This is used for profiling.

MemGuard measures certain statistics. When its interface is called it reports the amounts measured since the last call. The statistics include:

- number of cache misses measured (performance events that occured)
- total time that passed
- memory budget overrun
- time budget overrun

Memory budget overrun in effect means that the ISR handling overflow of the performance monitor counter was invoked at least once. Similarly the time budget overrun means the ISR handling replenish time period was invoked at least once.

In the HERCULES project there was a need to see behavior of the KCF tracker. It is a tool to track objects using visual data. The aim of measured KCF implementation is to comply with PREM. [1)] An illustrative example is in figure 3.7.

[1)] `https://github.com/CTU-IIG/kcf`

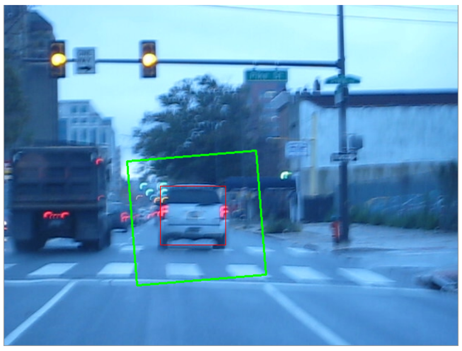


Figure 3.7. Visualization of the profiled KCF tracker object tracking. Red square is reference value. Green square is the tracker's outcome.

In this work the tool was used as a blackbox. We compare performance of this application with the synthetic task performing sequential array reads. The performed measurements utilize memory bandwith limiting with the MemGuard.

3.6.3 Hardware cache partitioning

The experiment described in this section aims to show the behavior of hardware last level cache partitioning. The LLC is typically shared among CPU cores. The partitioning of the LLC is available in some of the Intel Xeon series processors. Jailhouse hypervisor includes software support for partitioning of the Xeon's LLC and assigns cache partitions to cells according to their configuration.

As a benchmark we use kernels from the PolyBench suite with various dataset sizes. The performance of the benchmark is measured under Linux within root cell. There was utilized the *perf* tool that allows to report task statistics. It uses PMU's counters internally.

We utilize Jailhouse hypervisor to create cells with bare-metal interfering tasks. The interfering tasks perform sequential array reads with stride of a cache line size.

The experiment compares shared and partitioned cache setups. For single benchmark there are measured multiple test cases in both cache setups. The test cases are depicted in the figure 3.8. The total number of utilized cache ways is set to be equal to the number of cells in the test case.

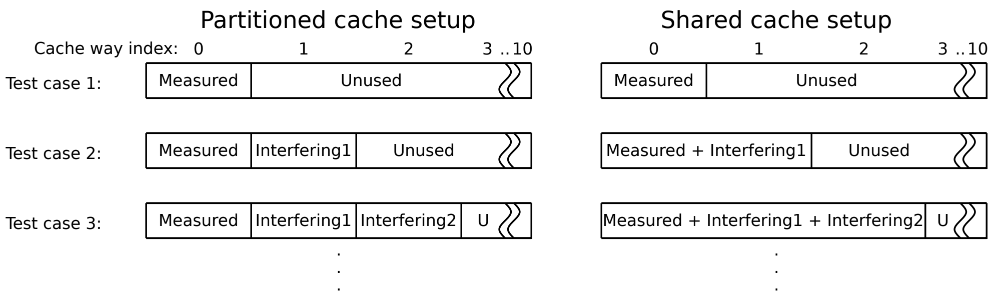


Figure 3.8. Cache partitioning experiment.

Root-cell with Linux has assigned one cache way. Every interfering cell has also assigned one cache way. Unused cache ways and CPU cores are assigned to the dummy cell. The dummy cell does not run any code and its CPU cores are off.

For this experiment the HyperThreading (HT) feature of the Intel Xeon processor was disabled. HT is hardware level virtualization that allows to interact with one physical

CPU core as if there were two cores present. We experienced crashes of Jailhouse due to misconfigured cell access to the cores so the HT was disabled in the BIOS setup. I believe both virtual cores belonging to one real core would have to be assigned to the single cell for proper functioning with HT.

Chapter 4
Evaluation

There are used two platforms to perform tests. I find it interesting to see the relative performance comparison of the platforms employed in relation to caches. The comparison is plotted in the figure 4.1. There is employed PolyBench GEMM benchmark with the same dataset sizes on both platforms. For the test we limited Xeon LLC size to 3MB so that we get closer to the TX2's 2MB LLC. The test for TX2 uses MemGuard to measure cache misses and execution time. Measured and interfering tasks run all under Linux. The test for Xeon uses *perf* utility to measure cache misses and execution time. Measured task runs in the Jailhouse root-cell and interfering tasks run in the non-root cells. In the figure, in the point where there is no interfering task, the number of cache misses on the Xeon is ten times lower than on the TX2. The main reason for this is the Xeon has three level cache whereas TX2 has only two level cache. Xeon's L2 cache has the size of 1MB.

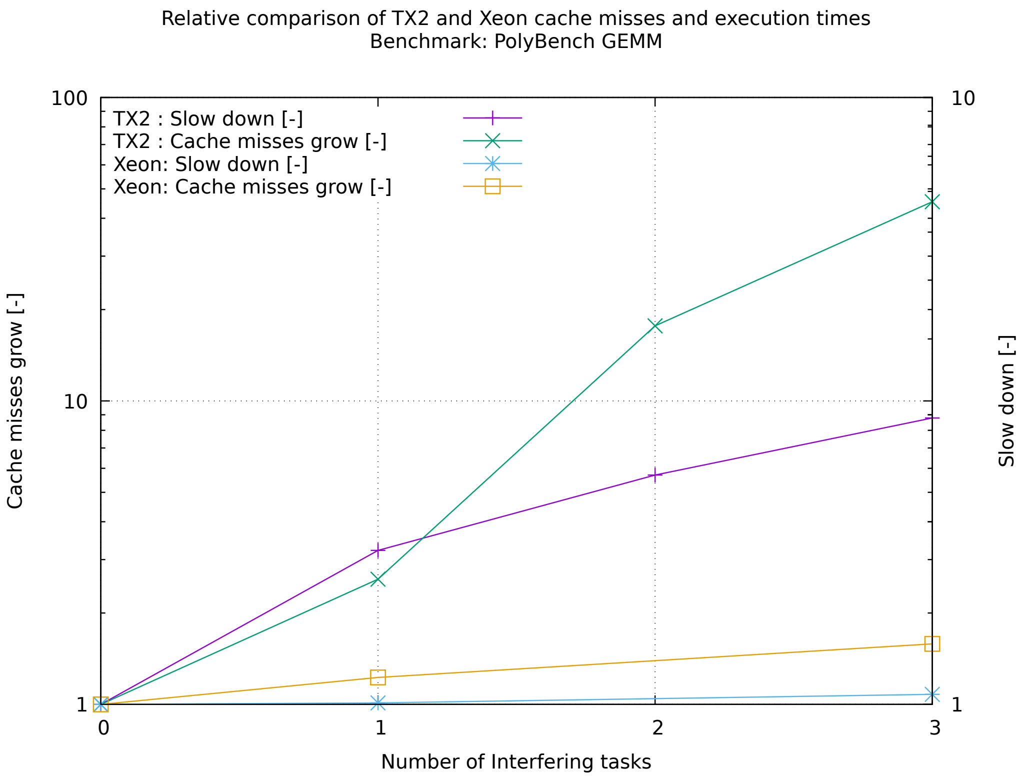


Figure 4.1. Relative comparison of benchmark performance on the TX2 and Xeon W in relation to caches.

There is employed PolyBench with the same dataset size on both platforms.

4.1 Throttling memory clients

4.1.1 Measurement results

Our measuring is performed above combinations of up to two CPU threads and a GPU thread. There are first measured CPU thread alone and GPU thread alone. Later the measured GPU thread is combined with one or two interfering CPU threads. The interfering thread starts before the beginning of measured thread and finishes after the measured thread execution is done. This way the measured thread should have more precise results. Number of cycles to throttle clients is chosen approximately as powers of two.

The setup that follows depicts the effects of setting different amounts of throttle cycles applied to different rings (i.e., to the THROTTLE registers), but without enabling any client within mask registers at all.

When examining Ring 2 we see, as we would expect, that in the figure 4.2 the memory bandwidth of our threads is not influenced.

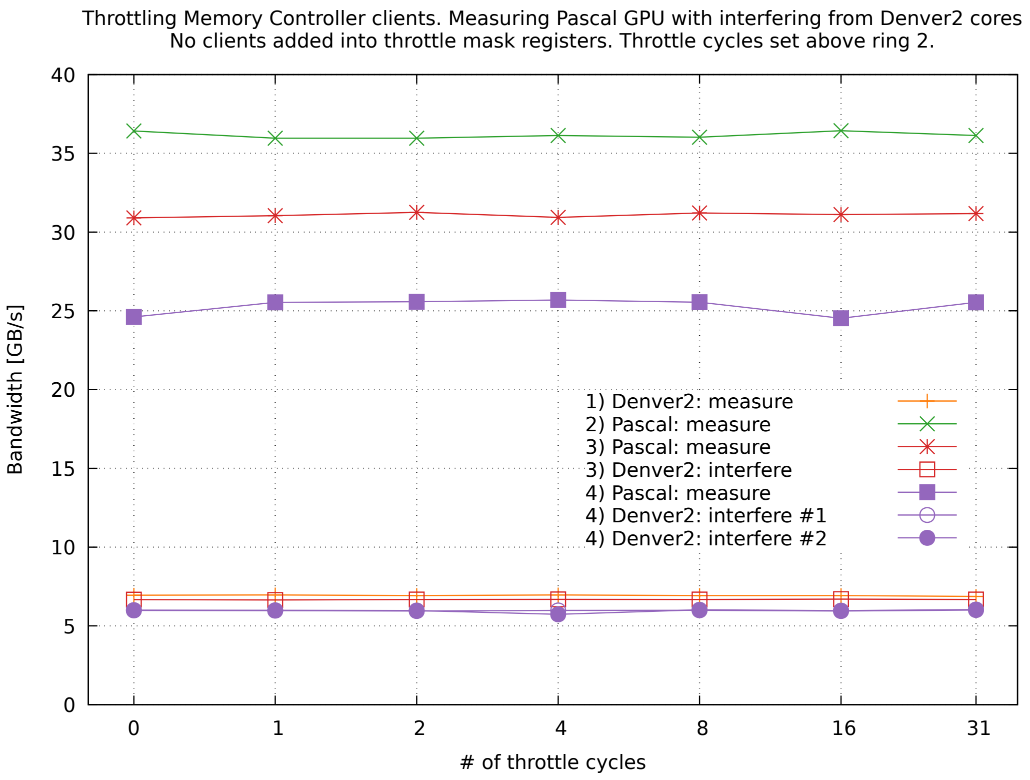


Figure 4.2. Examining behavior of the THROTTLE register above ring 2.

That is however not the case for Ring 0. In the figure 4.3 it seems clients that are used by our threads gets throttled anyway no matter they are not set to be throttled. I suspect the Ring 0 does not mind its mask register.

Finally NISO client group is portrayed in the figure 4.4.

It seems the CPU client is selectively influenced as soon as 3 throttle cycles count (in these cases our memory intensive task for CPU [pointerChasing] reads from a cache-line size cell at a time and also writes into it). This looks like the CPU reads are set to be throttled even though all clients in the NISO meta-group are disabled (i.e., zeroed mask register).

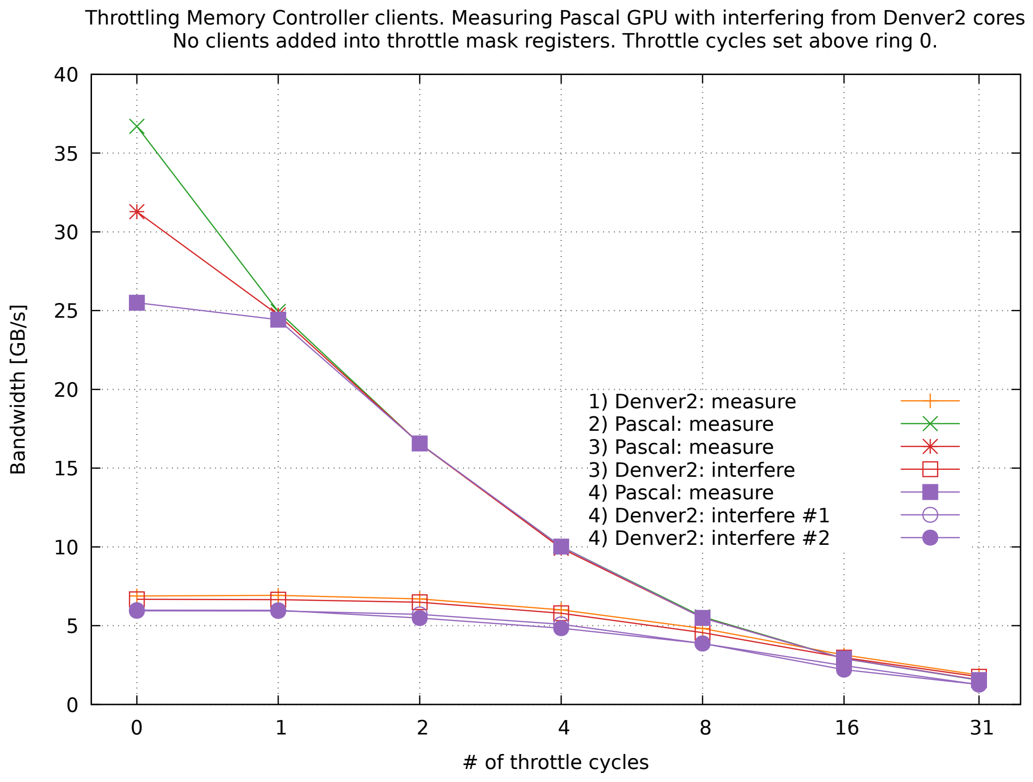


Figure 4.3. Examining behavior of the THROTTLE register above ring 0.

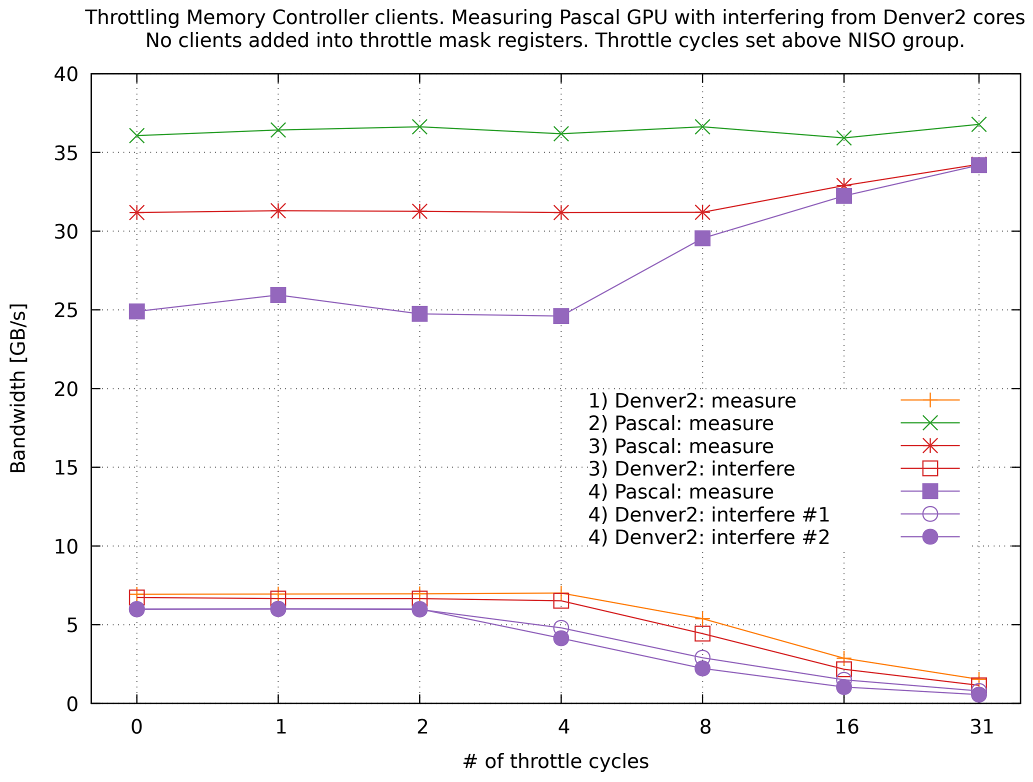


Figure 4.4. Examining behavior of the THROTTLE register above NISO client group.

We show only the results for Denver2 core, but the analogic results are yielded by A57 core. The trends are less striking with A57.

As the specific test case we throttle GPU memory client (GK and GK2 in the masks registers). The resulting bandwidth is depicted in chart 4.5.

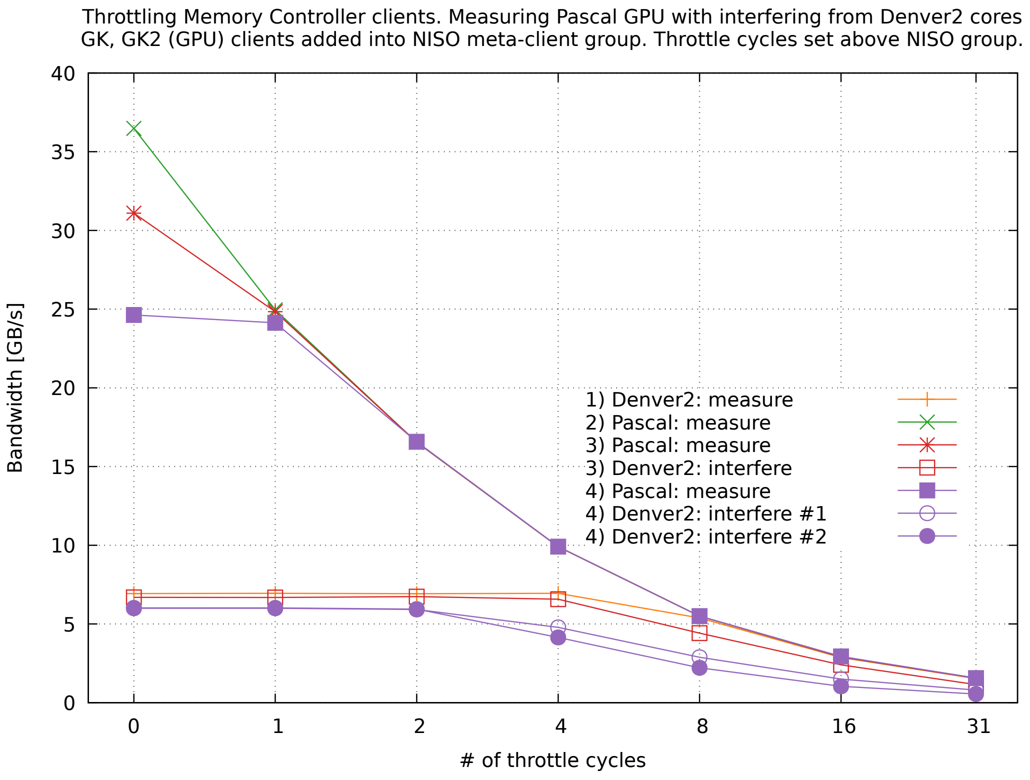


Figure 4.5. Throttling GPU within NISO client group.

The GPU client memory accesses are obviously well throttled there, but unfortunately the CPU accesses gets throttled as well. I wasn't able to find any further (register) settings that could allow the CPU bandwith to stay intact. In this form is the GPU throttling on the TX2's MC level unusable for increasing the time predictability of the system.

Further we measured the delay between time the new values are set in the throttle related registers and the time same values are read back. The delay was in range from 20ms to 100ms. Such high values would make the throttling on the MC level for the purpose of increasing the system time predictability also unsusable since the task running on CPU that should be protected from GPU interference would finish its execution by the time this throttling was operational.

4.2 Profiling MemGuard implementation

4.2.1 Profiling basic MG parameters

The synthetic application used to profile MemGuard performs sequential reads or writes above an array. The array is accessed with stride of cache line size. It should be noted that when the application is memory bound, decreasing memory bandwidth of the app

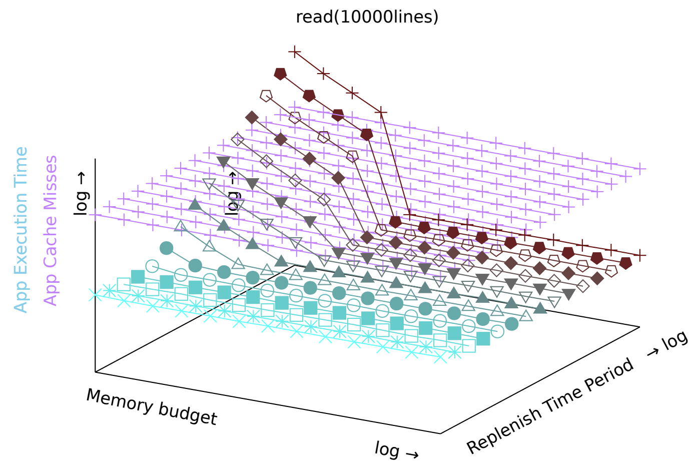


Figure 4.6. Application performance with varying MemGuard parameters.
The arrows indicate direction of increase of the quantities associated with the axes. Scale of the axes is logarithmic.
Points having the same application execution time corresponds more or less to the same bandwidth.

results in increased execution time and vice versa. The application employed is designed to be memory bound.

The figure 4.6 depicts behavior of the synthetic sequential read application under MemGuard. We observe the app's response to various combinations of memory budgets and replenish time periods. For each combination the total execution time and total cache misses amount the app needed are recorded and plotted. We can see the number of cache misses stays more or less constant. It is expected as the application should have more or less constant number of memory accesses in each run. The execution time however changes as we moderate memory bandwidth the app can utilize.

The figure 4.7 is a two dimensional projection of the figure 4.6. It highlights application's response to the changes in the memory budget. When the amount of memory budget is lower than the amount that would saturate the task bandwidth needs, we need more replenishes to finish the task. That results in longer total execution time of the task. The MemGuard stops to limit core's bandwidth at the point where the task cannot deplete all the memory budget given for the current time period. In chart 4.7 it happens at around 10000 cache misses and it is directly related to our task reading 10000 cache lines, because one cacheline miss means one access to the main memory. At the same time if we decrease replenish time period enough (128us and less in the chart) the bandwidth of the app is increased. This results in the decrease of total execution time.

We can also make projection of the figure 4.7 so that the application's response to the changes in the replenish time period is higlighted. It is shown in figure 4.8. The description of behavior from the previous paragraph holds. The interesting part of the chart is on the far left.

4.2.2 Reliability of MemGuard

In the figure 4.8 we see the total cache misses counts slightly decrease on the very left. After reading the ARMv8 documentation thoroughly it is revealed that the PMU architecture does not require event filtering to be accurate. For most events, it is

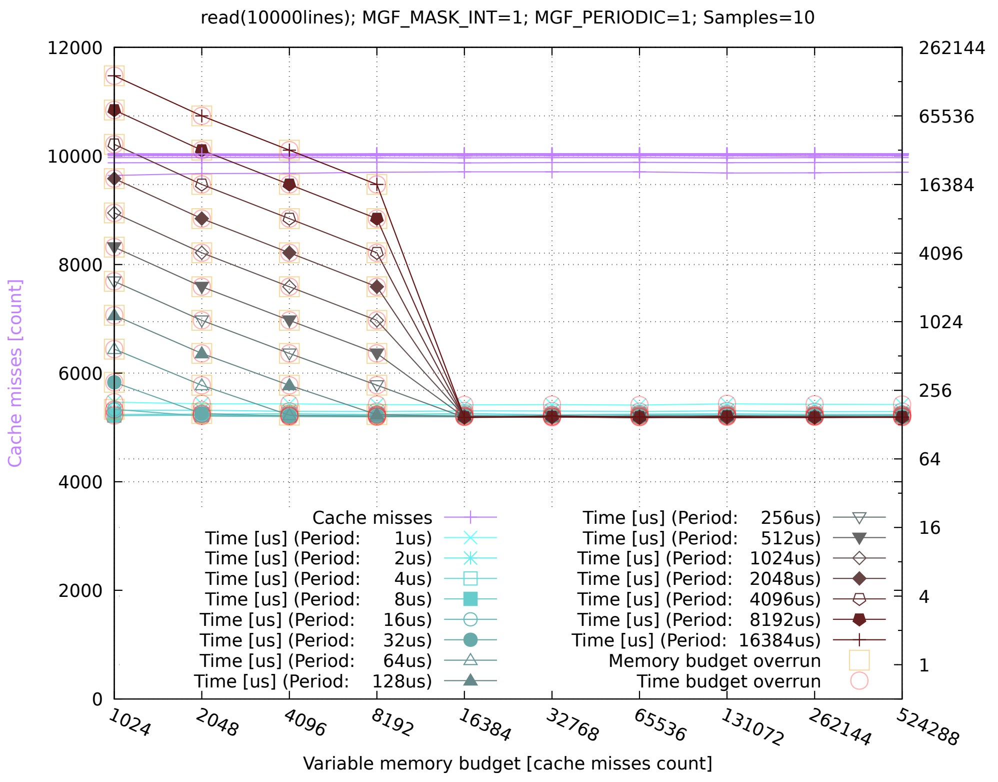


Figure 4.7. MemGuard: focus on memory budget behavior.

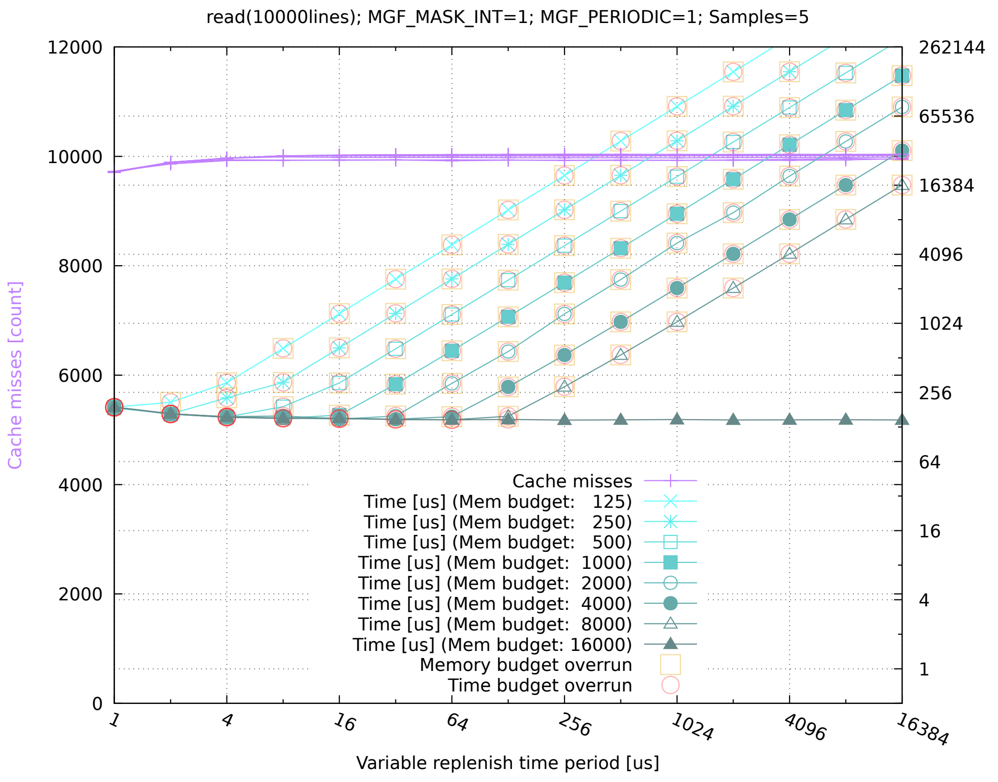


Figure 4.8. MemGuard: focus on replenish time period behavior.

acceptable that, during a transition between states, events generated by instructions executed in one state are counted in the other state. [22] Since we filter the PMU events only by exception level 0 and 1 we miss those that are reported in the EL2 (hypervisor). That is the cause for the slight drop in cache misses.

In the left part of the chart there is also the slight increase in the total execution time. When testing various performance events behavior we discovered this increase is connected to the increase in TLB cache misses. The value of TLB cache misses was around 4 times higher with 1us period compared to for example 64us period. EL2 events were included when performing the TLB measurement. This is one of the factors contributing to the MemGuard overhead in the system. The real world application probably would not thrash the TLB that much so our results are getting close to the worst-case scenario.

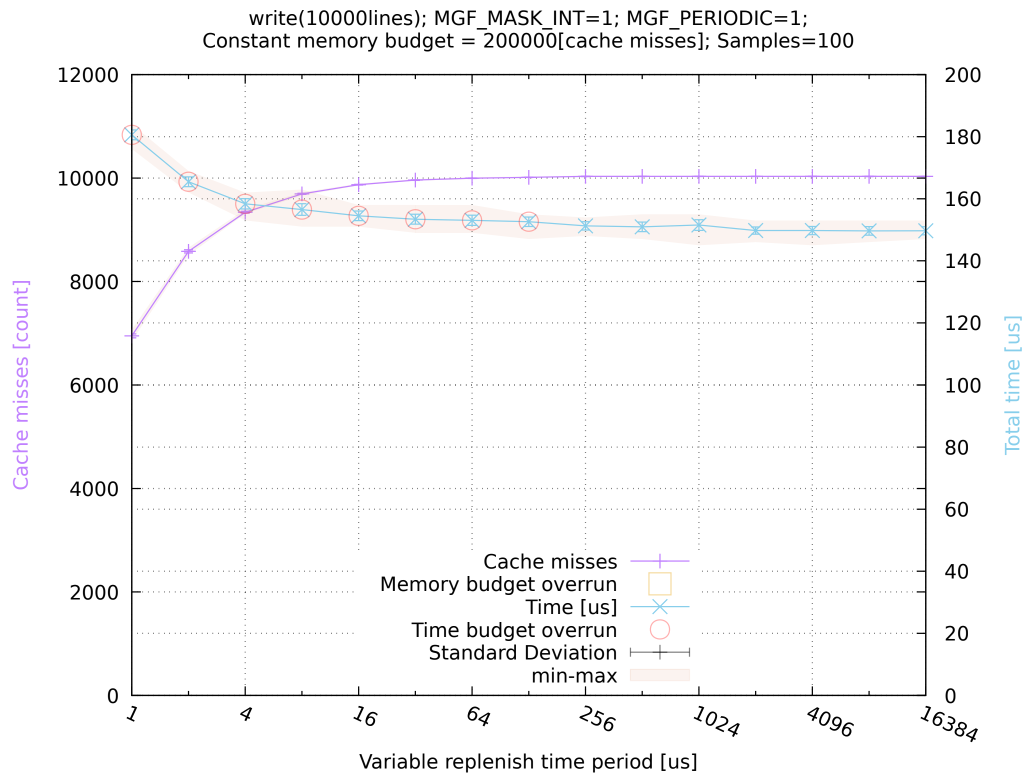


Figure 4.9. Profiling MemGuard's timer ISR overhead.
One CPU enabled at measure time. Events counted only from EL0 and EL1.

The experiment with its outcomes in figure 4.9 was constructed to measure overhead of the MemGuard. Since memory budget is higher than the task could ever need we know ther won't be executed performance event ISR. The experiment then depicts only the overhead of the timer ISR that handles replenishes of the budget. We see that the low amounts of replenish time period result in higher execution times.

With this chart we can decide how high overhead would be acceptable. Unfortunately there is relatively high deviation in the execution time. If we could partition TLB and reserve some records for the MemGuard's usage, the experiment's execution time would persumably get more stable.

The percentage overhead of the MemGuard's timer ISR is listed in table 4.1. It compares overheads for various replenish time periods. To compute the overhead we use the following formula: $Overhead = (\frac{exec\_time}{150.36} - 1) * 100[\%]$, where $exec\_time$ is the total execution time for the replenish time period. *exec_time* is the total execution time yielded for the given replenish time period with the active MemGuard (i.e., time budget overrun occured; periods 1 to 32 in the chart). Value in the denominator was counted as an average of the total execution times with MemGuard inactive (i.e., time budget overrun did not occur; periods 256 to 16384 in the chart).

Replenish time period [us]	Timer Overhead [%]
1	20.10
2	10.04
4	5.32
8	4.10
16	2.74
32	2.03
64	1.79
128	1.48

Table 4.1. Precentage overhead of the MemGuard for various replenish time periods.

4.2.3 MemGuard as a profiler

When memory budget of MemGuard is set substantially higher than where are the app needs and also when the replenish time period is higher than the app execution time we may use reports from MemGuard to profile the app. We use statistics reported by MG to profile synthetic application that reads array with cache line stride. With this we experimentally explore the behavior of the app on the TX2 platform. To get significant results we perfromed the measurement million times.

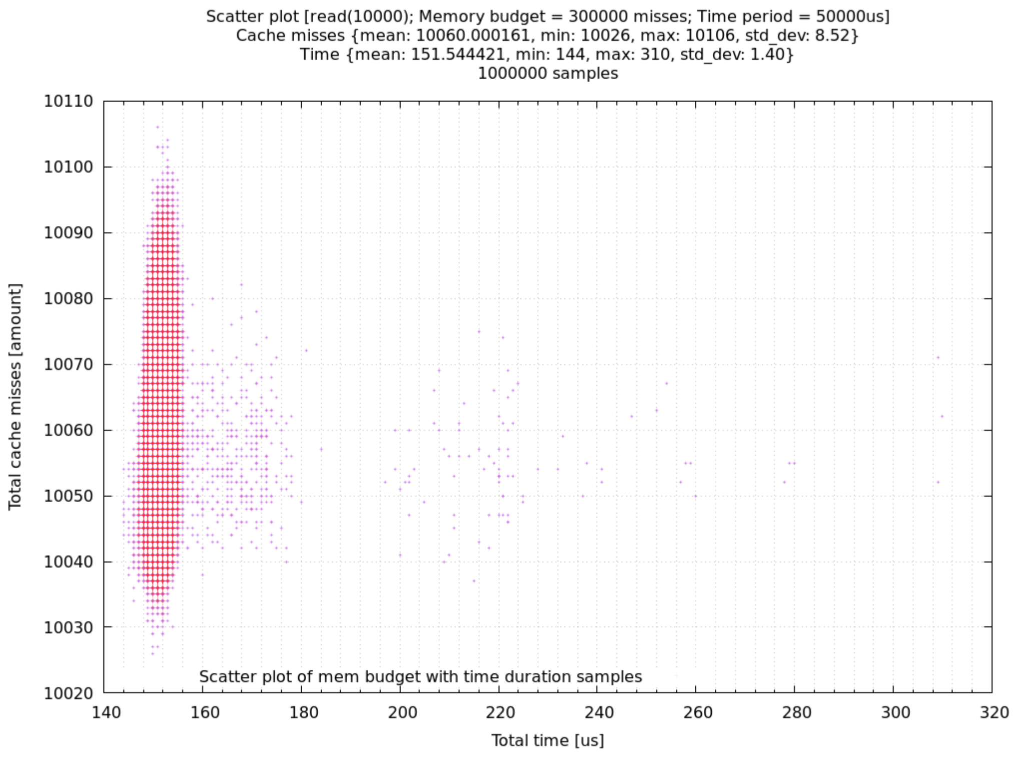


Figure 4.10. Scatter plot of million measurements that involve task reading ten thousand cache lines.

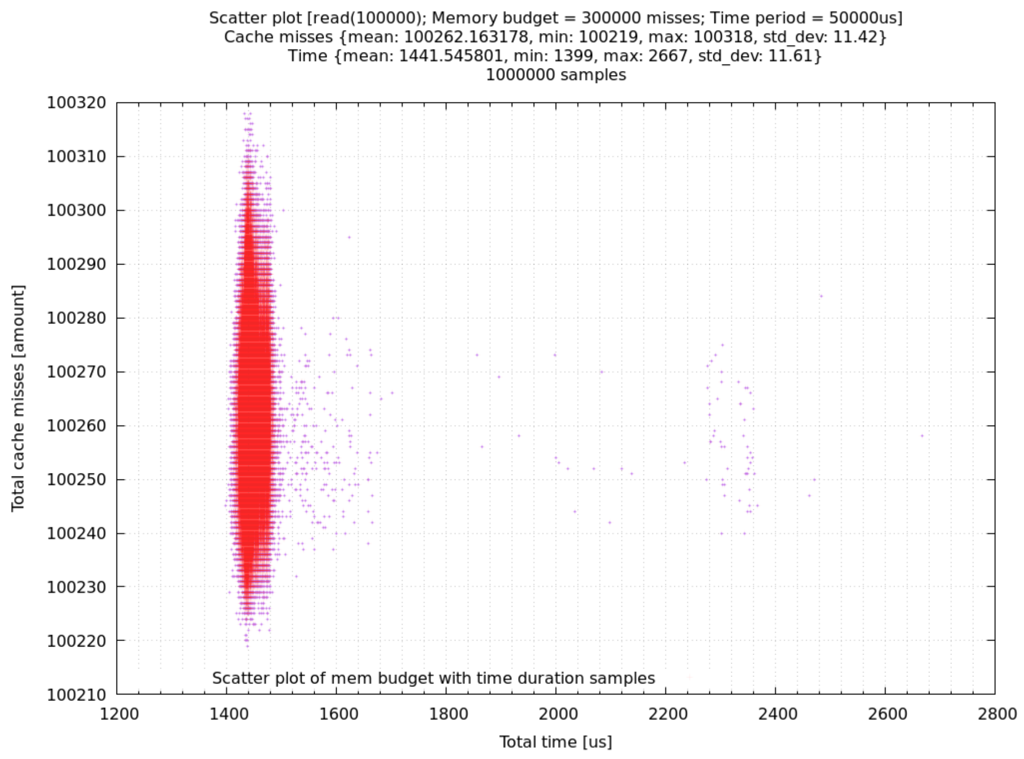


Figure 4.11. Scatter plot of million measurements that involve task reading hundred thousand cache lines.

There is a scatter plot of ten thousand array reads in figure 4.10 and hundred thousand array reads in figure 4.11. In the scatter plot, the more points are in the same spot the more red that spot turns. We see the cache misses are all in one cluster. Full cache misses range is about 100 misses for both measurements performed. Total execution time is also in one cluster, but its variation depends on the number of operations performed by measured task. Further there exist outliers with significantly higher times. Since we run these experiments under Linux root-cell there may occur some high priority interrupts. Future work could execute this or similar test on the bare metal Jailhouse cell to see if the time outliers disappear in such setup.

The cumulative distribution function of the cache misses formed from the results presented in figure 4.11 may be seen in the figure 4.12. Similarly, we can see the cumulative distribution function of the execution time in figure 4.13. Based on these profiling results we may allocate the memory bandwidth for the task.

With the initial results from the profiling we were able to recoginze issues in the MemGuard statistics reporting and based on that we fixed the tool.

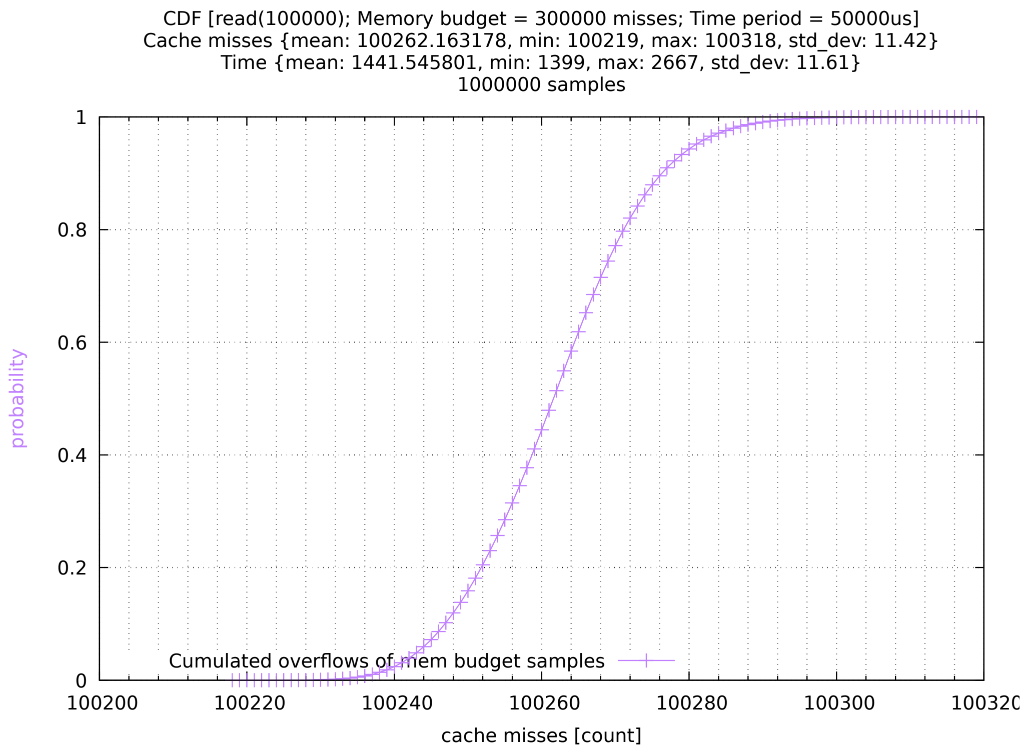


Figure 4.12. CDF: cache misses plot of million measurements that involve task reading hundred thousand cache lines.

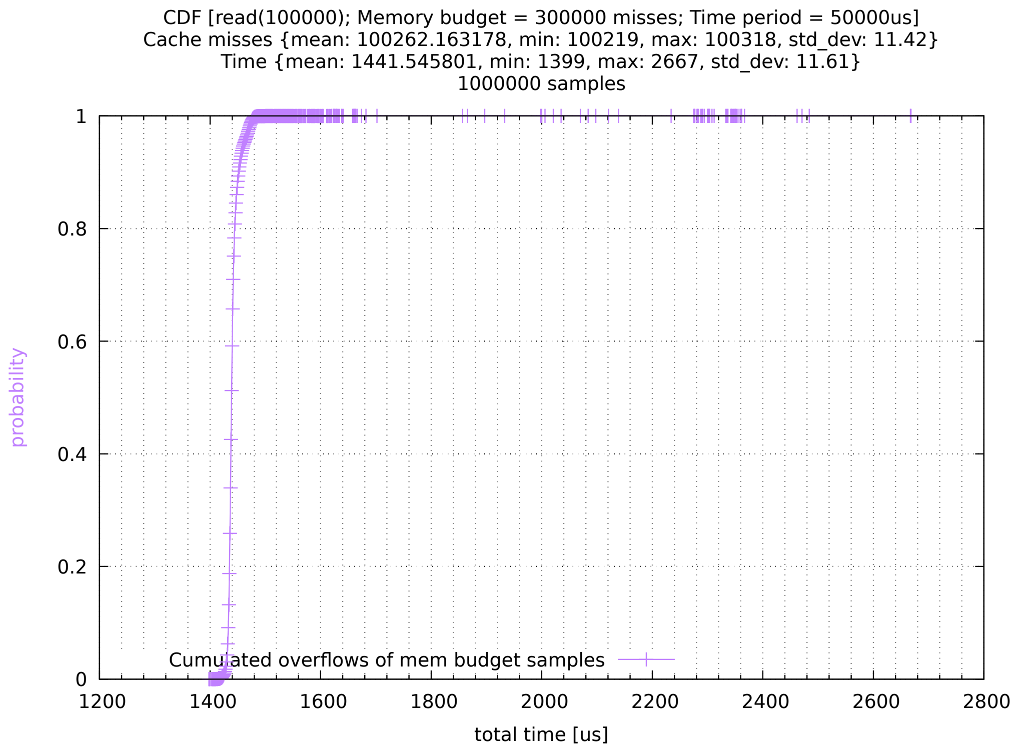


Figure 4.13. CDF: execution time plot of million measurements that involve task reading hundred thousand cache lines.

4.2.4 MemGuard upon real-world application: KCF tracker

We compare the KCF tracker performance with the synthetic task executing sequential array reads. For each task there are measured different values of memory bandwidth preset with MemGuard. The figure 4.14 depicts slowdown of applications with decreasing bandwidth.

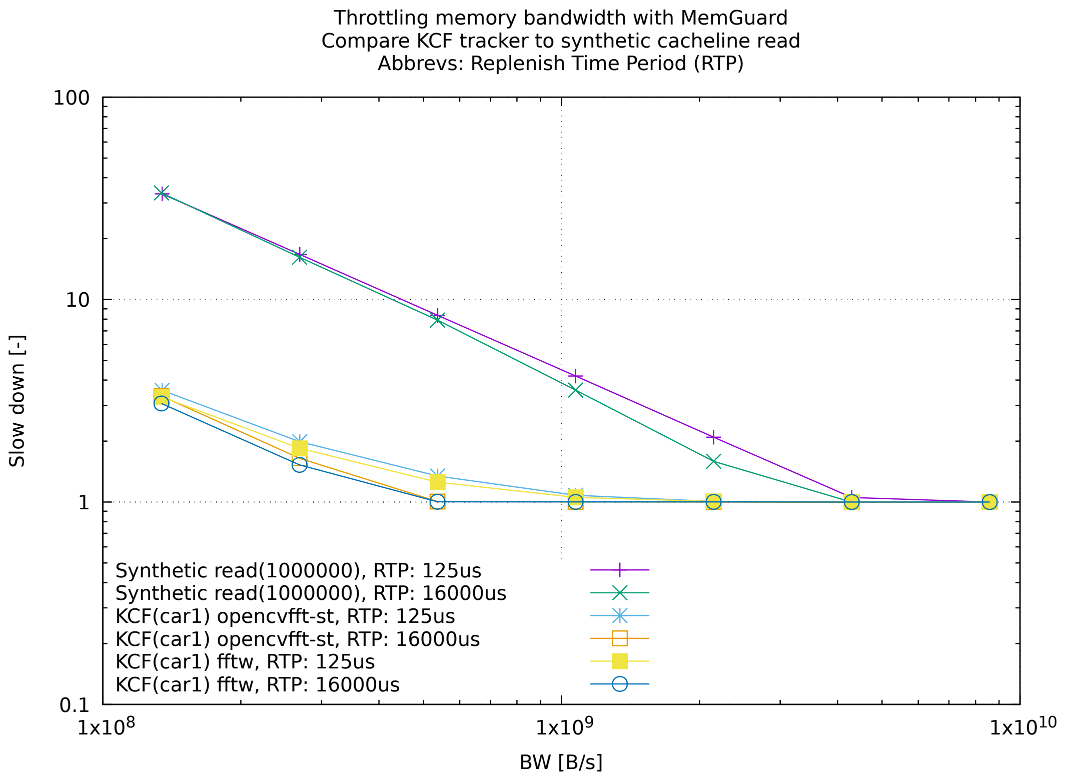


Figure 4.14. MemGuard used to compare real-world and synthetic applications.

car1 is the name of the test based on dataset within VOT2016 [1)] datasets. The KCF codebase comes with multiple FFT implementations. We evaluate OpenCV FFT single threaded and FFTW library implementations.

4.3 Hardware cache partitioning

The design of the experiment is described in section 3.6.3.

We measured setups with various dataset sizes utilizing GEMM and DOITGEN benchmarks from the PolyBench suite. There is presented only one setup with GEMM kernel as all results tend to have similar behavior.

In the figure 4.15 there are compared characteristics of setup with partitioned cache and setup with non-partitioned (shared) cache. The number of interfering tasks in the test varies. Each point is an average of either cache misses count or the total execution time of the measured benchmark.

With the partitioned setup the total number of cache misses that the benchmark suffers is constant. In the shared cache setup the cache misses rise as the cache lines belonging to the benchmark are evicted from the cache by accesses of the interfering task.

The execution time in case of one interfering task is better for the paritioned cache setup compared to the shared cache setup. With the number of interfering tasks higher than one the performance of the paritioned cache setup is worse than for the shared cache setup. This was not expected, the time in the partitioned cache case should outperform shared cache case. I believe the reason for this behavior is that the experiment was not designed optimally. I suppose there is employed LRU replacement policy as it is mentioned in Intel manuals for earlier processor versions platforms [32]. The measured benchmarks keeps its cache lines hot leaving multiple interfering tasks compete among each other. Therefore in the shared cache setup the benchmark is limited significantly less compared to what we expected.

The problem could also lie in the small size of the array used in the interfering application. Since Jailhouse runs inmates with EL2 level MMU enabled and maps only the first 2MB for the inmate, we employed just 1.75 MB array in the interfering tasks to fit the available mapped memory. If this was the issue the mapped memory could be increased explicitly with calling Jailhouse function *map_range*.

For the reasons stated above we may consider only test cases with zero and one interfering task to be significant.

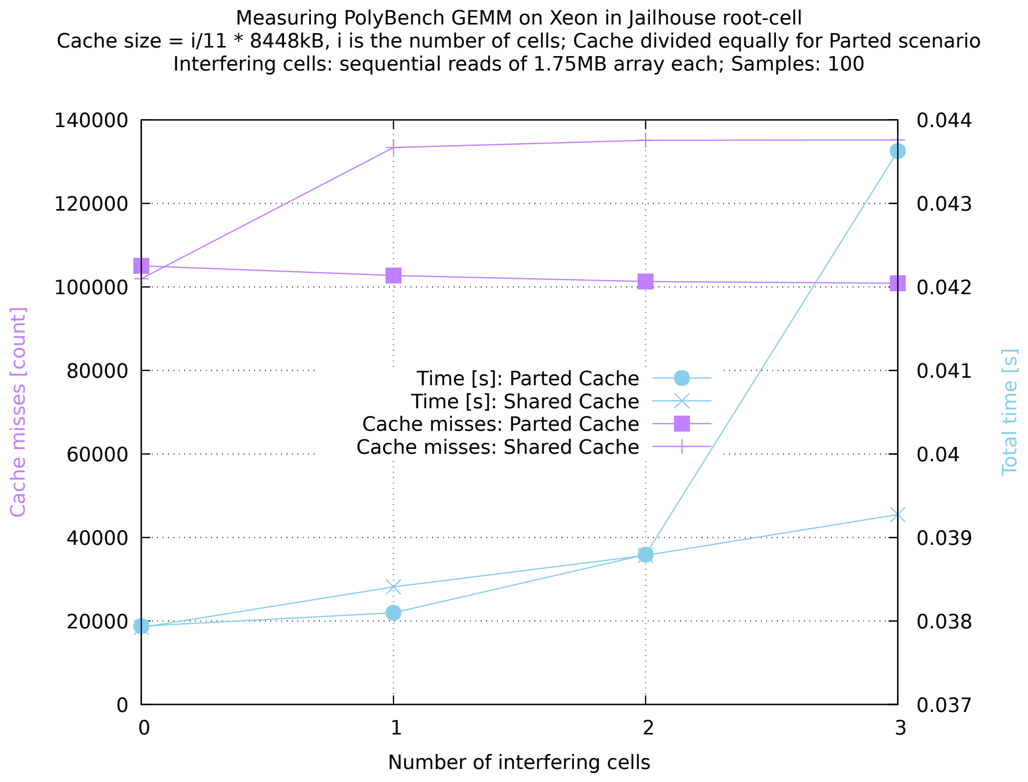


Figure 4.15. Benchmarking shared and partitioned cache: averages of cache misses counts and total times.

To be complete, the figure 4.16 depicts scatter plot of the datapoints measured in presented experiment. The outliers are expected as we run the task under Linux without protection from the possible system's interference.

I'd like to point out, that the evaluation platform arrived late. Therefore there was not time to design better experiments.

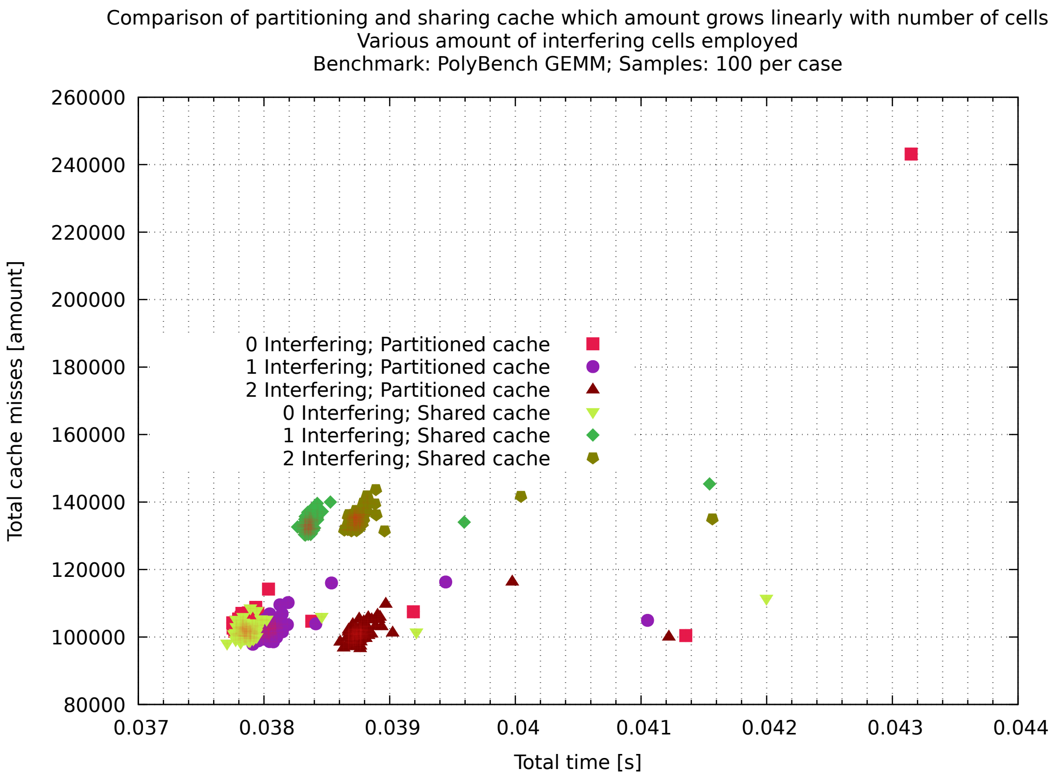


Figure 4.16. Benchmarking shared and partitioned cache: scatter plot of measurements.

Chapter 5 Conclusion

The work presents methods that improve predictability of program execution time and describes the results of the experiments.

We evaluated methods to limit the bandwidth of memory clients, which are neccessary for the operation of the PRedictable Execution Model.

We wanted to use the memory controller of the TX2 platform to throttle the GPU memory bandwidth as this was previously done for the TX1 platform. We showed that on the TX2 the CPUs are throttled along with the GPU even though the CPUs are not set to be throttled. This makes the throttling of the GPU on the TX2's memory controller level unusable.

We verified that the behavior of MemGuard is correct. We found a small issue in statistics reporting and submitted patch that fixed the issue. We showed that MemGuard's overhead is 2% when the replenish time period is set to 32 us. This is acceptable overhead.

The results of profiling the KCF tracker show that the application uses approximately 1.8 GB/s when it is not throttled. However, we set the tracker's bandwidth to the 100 MB/s and show that this slows the tracker by the factor of 3.5.

We showed that hardware cache partitioning improves the performance of the system. However, the test was not designed ideally. Details may be found in section 4.3.

References

[1] Paul Lokuciejewski, and Peter Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Springer Netherlands, 2011. ISBN 978-90-481-9928-0.

[2] Albert M. K. Cheng. *Real-Time Systems: Scheduling, Analysis, and Verification*. John Wiley & Sons, Inc., 2003. ISBN 9780471224624.
`http://dx.doi.org/10.1002/0471224626`.

[3] Renato Mancuso, Roman Dudko, and Marco Caccamo. *Light-PREM: Automated software refactoring for predictable execution on COTS embedded systems*. In: *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications [online]*. IEEE, 2014. pp. 1-10. ISBN 978-1-4799-3953-4.
`http://ieeexplore.ieee.org/document/6910515/`.

[4] P. Burgio, A. Marongiu, P. Valente, and M. Bertogna. *A memory-centric approach to enable timing-predictability within embedded many-core accelerators.*. In: *2015 CSI Symposium on Real-Time and Embedded Systems and Technologies (RTEST)*. IEEE, 2015. pp. 1–8.
`https://doi.org/10.1109/RTEST.2015.7369851`.

[5] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha. *MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms*. In: *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2013. 55-64.

[6] Sparsh Mittal. A Survey of Techniques for Cache Partitioning in Multicore Processors. *ACM Comput. Surv.*. 2017, 50 (2), 27:1–27:39. DOI 10.1145/3062394.

[7] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. *Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems*. In: *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 2008. 367-378.
`https://doi.org/10.1109/HPCA.2008.4658653`.

[8] Ulrich Drepper. *What every programmer should know about memory [online: accessed 11.2.2017]*. 2007.
`https://lwn.net/Articles/250967/`.

[9] Matthew Lentz, and Manoj Franklin. *Performance of Private Cache Replacement Policies For Multicore Processors [online]*. 2014.
`http://doi.org/10.5121/csit.2014.4708`.

[10] Yuejian Xie, and Gabriel H. Loh. PIPP: Promotion/Insertion Pseudo-partitioning of Multi-core Shared Caches. *SIGARCH Comput. Archit. News*. 2009, 37 (3), pp. 174–183. DOI 10.1145/1555815.1555778.

[11] M Geanta, L. Ghica, and N. Tapus. *Leverage cache replacement policy in multicore processors*. In: *2016 IEEE 12th International Conference on Intelligent Computer*

Communication and Processing (ICCP) [online]. 2016. pp. 417–424. ISBN 978-1-5090-3899-2.
https://doi.org/10.1109/ICCP.2016.7737182.

[12] *Wikimedia Commons.*
https://commons.wikimedia.org.

[13] George Taylor, Peter Davies, and Michael Farmwald. *The TLB Slice&Mdash;a Low-cost High-speed Address Translation Mechanism.* In: *Proceedings of the 17th Annual International Symposium on Computer Architecture.* New York, NY, USA: ACM, 1990. 355–363. ISBN 0-89791-366-3.
http://doi.acm.org/10.1145/325164.325161.

[14] Gerald J. Popek, and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM.* 1974, 17 (7), 412–421. DOI 10.1145/361011.361073.

[15] Ellen Kou Texas Instruments. *Virtualization for embedded industrial systems.*
http://www.tij.co.jp/jp/lit/wp/spry317a/spry317a.pdf.

[16] *Jailhouse, project home [online].*
https://github.com/siemens/jailhouse.

[17] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. *A Predictable Execution Model for COTS-Based Embedded Systems.* In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium [online].* IEEE, 2011. pp. 269–279. ISBN 978-1-61284-326-1.
http://ieeexplore.ieee.org/document/5767117/.

[18] Rakesh Komuravelli, Matthew D. Sinclair, Johnathan Alsop, Muhammad Huzaifa, Maria Kotsifakou, Prakalp Srivastava, Sarita V. Adve, and Vikram S. Adve. *Stash: Have your scratchpad and cache it too.* In: *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA) [online].* ACM Press, 2015. pp. 707–719. ISBN 978-1-4503-3402-0.
http://dx.doi.org/10.1145/2749469.2750374.

[19] Konstantinos Koukos, Per Ekemark, Georgios Zacharopoulos, Vasileios Spiliopoulos, Stefanos Kaxiras, and Alexandra Jimborean. *Multiversioned Decoupled Access-execute: The Key to Energy-efficient Compilation of General-purpose Programs.* In: *Proceedings of the 25th International Conference on Compiler Construction – CC 2016 [online].* New York, NY, USA: ACM Press, 2016. pp. 121–131. ISBN 978-1-4503-4241-4.
http://doi.acm.org/10.1145/2892208.2892209.

[20] NVIDIA. *NVIDIA Parker Series SoC, Technical Reference Manual [online].* 2017.
https://developer.nvidia.com/embedded/downloads#?search=X2.

[21] NVIDIA. *NVIDIA Tegra X1 Mobile Processor, Technical Reference Manual [online].* 2016.
https://developer.nvidia.com/embedded/downloads#?search=X1.

[22] ARM Limited. *ARM Architecture Refernce Manual [online].* 2018.
https://static.docs.arm.com/ddi0487/da/DDI0487D_a_armv8_arm.pdf.

[23] Intel Corporation. *Intel RDT in Linux [online].*
https://01.org/intel-rdt-linux/blogs/fyu1/2017/resource-allocation-intel%C2%AE-resource-director-technology.

[24] Intel Corporation. *Intel Xeon Processor E5-2600 V4 Product Family Technical Overview.*
`https://software.intel.com/en-us/articles/intel-xeon-processor-e5-2600-v4-product-family-technical-overview.`

[25] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. *CATalyst: Defeating last-level cache side channel attacks in cloud computing.* In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 2016. 406-418.

[26] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. *Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family.* In: *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 2016. 657-668.

[27] Intel Corporation. *Intel® 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4 [online].*
`https://software.intel.com/en-us/articles/intel-sdm.`

[28] Jan Kiszka Siemens Corporate Technology. *Bootstraping the Partitioning Hypervisor Jailhouse.* 2016.

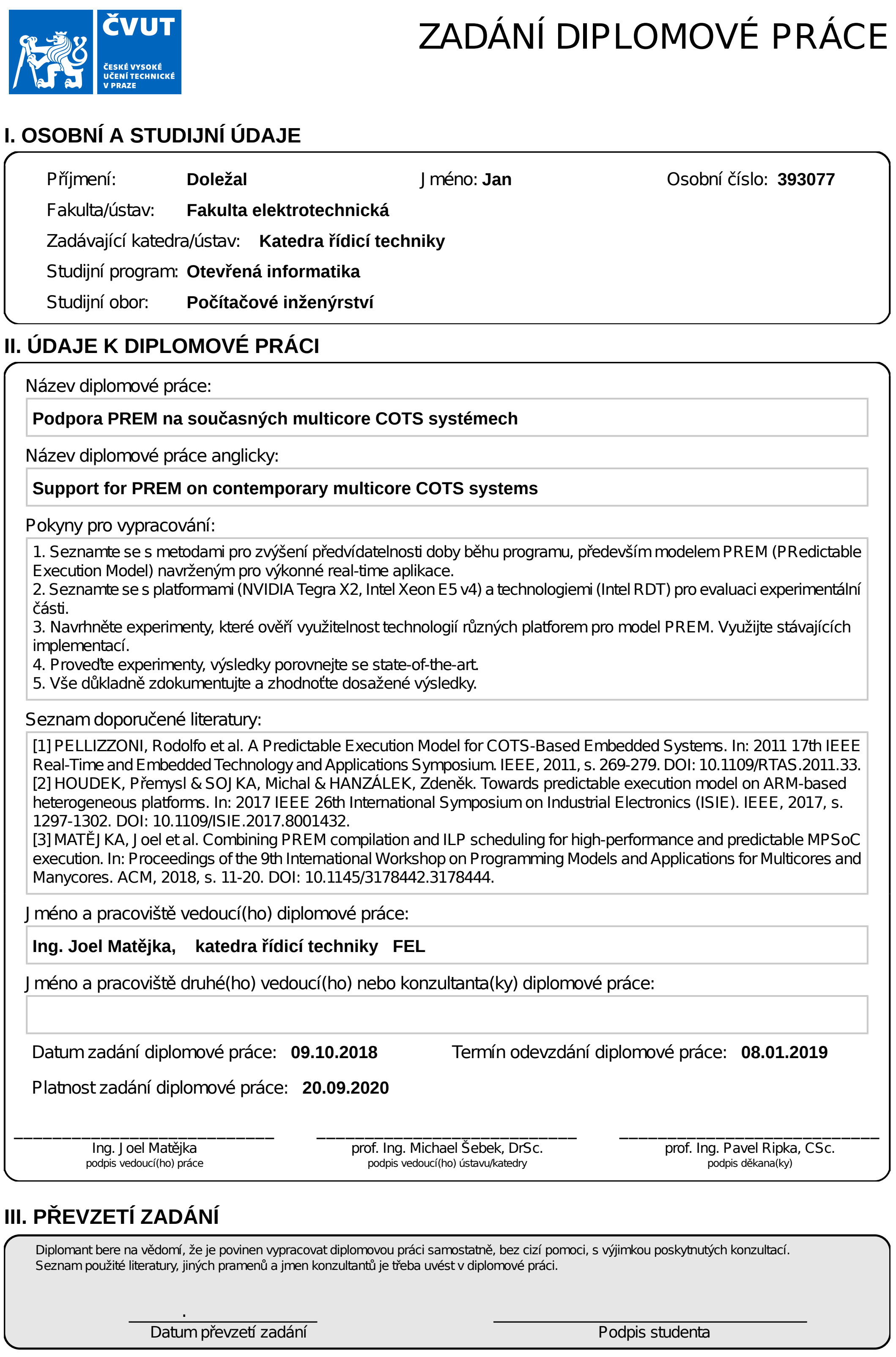
[29] Tomofumi Yuki, and Louis-Noël Pouchet. *PolyBench 4.0.* 2015.
`http://www.cs.colostate.edu/AlphaZsvn/Development/trunk/mde/edu.csu.melange.alphaz.polybench/polybench-alpha-4.0/polybench.pdf.`

[30] Přemysl Houdek, Michal Sojka, and Zdeněk Hanzálek. *Towards predictable execution model on ARM-based heterogenous platforms..* In: *2017 26th IEEE International Symposium on Industrial Electronics (ISIE) [online].* IEEE, 2017. pp. 1297–1302.

[31] Joel Matějka, Björn Forsberg, Michal Sojka, Zdeněk Hanzálek, Luca Benini, and Andrea Marongiu. *Combining PREM compilation and ILP scheduling for high-performance and predictable MPSoC execution..* In: *Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores. [online].* ACM, 2018. pp. 11–20.

[32] Intel Corporation. *Improving Real-Time Performance by Utilizing Cache Allocation Technology.* 2015.
`http://www.cs.colostate.edu/AlphaZsvn/Development/trunk/mde/edu.csu.melange.alphaz.polybench/polybench-alpha-4.0/polybench.pdf.`

Appendix A
Specification



ZADÁNÍ DIPLOMOVÉ PRÁCE

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Doležal**	Jméno: **Jan**	Osobní číslo: **393077**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra řídicí techniky**		
Studijní program:	**Otevřená informatika**		
Studijní obor:	**Počítačové inženýrství**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Podpora PREM na současných multicore COTS systémech

Název diplomové práce anglicky:

Support for PREM on contemporary multicore COTS systems

Pokyny pro vypracování:

1. Seznamte se s metodami pro zvýšení předvídatelnosti doby běhu programu, především modelem PREM (PRedictable Execution Model) navrženým pro výkonné real-time aplikace.
2. Seznamte se s platformami (NVIDIA Tegra X2, Intel Xeon E5 v4) a technologiemi (Intel RDT) pro evaluaci experimentální části.
3. Navrhněte experimenty, které ověří využitelnost technologií různých platforem pro model PREM. Využijte stávajících implementací.
4. Proveďte experimenty, výsledky porovnejte se state-of-the-art.
5. Vše důkladně zdokumentujte a zhodnoťte dosažené výsledky.

Seznam doporučené literatury:

[1] PELLIZZONI, Rodolfo et al. A Predictable Execution Model for COTS-Based Embedded Systems. In: 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium. IEEE, 2011, s. 269-279. DOI: 10.1109/RTAS.2011.33.
[2] HOUDEK, Přemysl & SOJKA, Michal & HANZÁLEK, Zdeněk. Towards predictable execution model on ARM-based heterogeneous platforms. In: 2017 IEEE 26th International Symposium on Industrial Electronics (ISIE). IEEE, 2017, s. 1297-1302. DOI: 10.1109/ISIE.2017.8001432.
[3] MATĚJKA, Joel et al. Combining PREM compilation and ILP scheduling for high-performance and predictable MPSoC execution. In: Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores. ACM, 2018, s. 11-20. DOI: 10.1145/3178442.3178444.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Joel Matějka, katedra řídicí techniky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **09.10.2018** Termín odevzdání diplomové práce: **08.01.2019**

Platnost zadání diplomové práce: **20.09.2020**

______________________	______________________	______________________
Ing. Joel Matějka	prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

A.1 Specification in English

Master's Thesis Topic: Support for PREM on contemporary multicore COTS systems
Instructions for elaboration:

- Apprise yourself of the methods for increasing the predictability of program execution time, especially of model PREM (PRedictable Execution Model) designed for high-performing real-time applications.
- Apprise yourself of platforms (NVIDIA Tegra X2, Intel Xeon E5 v4) and technologies (Intel RDT) for evaluation of experimental part.
- Design experiments, which verify usability of different platform's technoligies for model PREM. Use existing implementations.
- Realize experiments, compare the results with state-of-the-art.
- Document everything thoroughly and evaluate achieved results.

Appendix B
Abbreviations

ACET	■ Average Case Execution Time
ADT	■ Abstract Data Type
ARC	■ Adaptive Replacement Cache
ASLR	■ Address Space Layout Randomization
AST	■ Abstract Syntax Tree
BCET	■ Best Case Execution Time
BIOS	■ Binary Input Output System
CAM	■ Content Addresable Memory
CAT	■ Cache Allocation Technology
CCPLEX	■ CPU Complex
CDP	■ Code and Data Prioritization
CFG	■ Control Flow Graph; Context Free Grammar
CMT	■ Cache Monitoring Technology
COS	■ Class of Service
COTS	■ Commercials Off-The-Shelf
CPU	■ Central Processing Unit
CUDA	■ Compute Unified Device Architecture
DAE	■ Decoupled Access Execute
DDA	■ Digital Differential Analyzer
DDR	■ Double Data Rate
DMA	■ Direct Memory Access
DRAM	■ Dynamic Random Access Memory
EL	■ Exception Level
EMC	■ External Memory Controller
fd	■ file descriptor
FFT	■ Fast Fourier Transform
GCC	■ GNU Compiler Collection
GNU	■ GNU's Not Unix!
GPS	■ Global Positioning System
GPU	■ Graphics Processing Unit
HERCULES	■ High-Performance Real-time Architectures for Low-Power Embedded Systems
HT	■ Hyper Threading
HV	■ Hypervisor
HW	■ Hardware
I/O	■ Input/Output
IR	■ Intermediate Representation
IRQ	■ Interrupt Request
ISR	■ Interrupt Service Routine
JH	■ Jaihlouse
JIT	■ Just-In-Time compiler

KCF	Kernelized Correlation Filter
LFU	Least Frequently Used
LLC	Last Level Cache
LPDDR	Low Power DDR
LRU	Least Recently Used
LTO	Link Time Optimization
LWN	Linux Weekly News
L4T	Linux for Tegra
MBA	Memory Bandwidth Allocation
MBM	Memory Bandwidth Monitoring
MC	Memory Controller
MCCIF	Memory Controller Client InterFace
MMU	Memory Management Unit
MSR	Model Specific Register
MSS	Memory Subsystem
NRU	Not Recently Used
OS	Operating System
PCI	Peripheral Component Interconnect
PM	Physical Memory
PMU	Performance Monitor Unit
PREM	PRedictable Execution Model
PTC	Page Table Cache
PTSA	Priority Tier Snap Arbiter
QoS	Quality of Service
RAM	Random Access Memory
RDT	Resource Dirctor Technology
saxpy	Single-Precision A.X Plus Y
SMMU	System Memory Management Unit
SoC	System on Chip
SoM	System on Module
SPM	Scratchpad Memory
SRAM	Static Random Access Memory
SSA	Single Static Assignment
SW	Software
TLB	Translation Lookaside Buffer
UART	Universal Asynchronous Receiver/Transmitter
VGA	Video Graphics Array
VM	Virtual Machine; Virtual Memory
VMM	Virtual Machine Monitor
VOT	Visual Object Tracking
VT-d	Intel Virtualization Technology for Directed I/O
WCET	Worst Case Execution Time