technische universität
dortmund

Bachelorarbeit

Improving the real-time properties of interrupt handlers by cache preloading

Phillip Alexander Goldap
25th September 2015

Betreuer:
Prof. Dr.-Ing. Olaf Spinczyk
M.Sc. Hendrik Borghorst

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Eingebettete Systemsoftware
http://ess.cs.tu-dortmund.de

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 25. September 2015

Phillip Alexander Goldap

Abstract

Today the industry offers a variety of modern common out of the shelve multi-core systems which provide high computational power for a low cost. However, the industry aims to improve the average case execution time by introducing architectural complex designs like branch prediction or shared caches. Therefore their timing behaviour is hard to predict. In a real time system the average case is not of interest, only the worst case execution time is of importance. Research tries to identify the different sources of unpredictability and searches for solutions to make them more predictable.

This work evaluates a proposed way to counter the unpredictability by using an operating system based cache management to preload and lock data in the level 2 cache. The proposed cache aware operating system has been implemented as a prototype and the impact on the execution time is examined and compared.

Contents

1 Introduction

In our modern world we rely more and more on computer systems which have to comply with real-time constraints and must show a predictable behaviour. This may be a computer which will trigger air-bags in case of an accident or fly-by-wire systems, where the computer must calculate, respond and react to external influences in real-time. Modern commercial off-the-shelf (COTS) multi-core hardware often does have low cost computational power, but lacks in predictability to ensure the real-time behaviour due to e.g. concurrent access to shared hardware resources. Thus, special and expensive hardware is often necessary.

The aim of this thesis is to improve the predictability of interrupt service routines on COTS hardware by implementing a software mechanism which reduces the parallel access on shared hardware resources.

The proposed idea is to transfer the control of the level 2 cache to the operating system and preload parts before use to avoid the slower and unpredictable shared bus level 3 DRAM memory access. To achieve this, the operating system is divided in small preloadable operating system components (OSCs) which will be preloaded and locked in the cache as long as they may be used.

This thesis consists of 4 sections and is structured in the following way: Section 2 explains real-time properties, gives an overview about the sources of unpredictability in modern multi-core systems and introduces previous work on cache preloading to counteract the unpredictability. In section 3 a proposed operating system model, which uses the cache preloading functionality, will be introduced and the hardware platform of an prototype implementation described briefly. This section is followed by an evaluation of a prototype implementation of the operating system model. Additionally the impact of the preloading functionality will be compared. In the end will be a short summary, a conclusion and an outline of possible future work.

2 Previous Work

2.1 Real-time Properties

The correctness of a computation in a real-time system depends not only on the correct result, but also on the time when the results are produced [1]. In a soft real-time system it is acceptable that a deadline occasionally can be exceeded. On the other hand, if the requirements of the system cannot tolerate any passing of a deadline, because this could lead to a catastrophe, it is a hard real-time system. This could for example be an emergency system of a nuclear power plant.

The key problem of a (hard) real-time system is now to define a maximum execution time in which the computation must be completed. Due to various sources of unpredictability, which will be explained in detail in the next section, the execution time cannot be determined and fluctuates. This fluctuating deviation is called jitter and must be reduced to a minimum. The average execution speed, on the other hand, does not matter. It is rather important to give the real-time system engineer the ability to calculate the worst-case execution time (WCET). A distinction is made between a complex and precise WCET analysis and an estimation.

2.2 Sources of Unpredictability

Todays COTS hardware is optimized for a good average case, whereas in real-time systems only the WCET matters. A previous work by Dakashina et al. [2] collected sources of unpredictability in multi-core systems. This work concentrates to tackle the following sources:

The first bottleneck is the shared front-side bus. Every memory access request which results in a level 2 cache miss must go over the bus. It is impossible to know when the bus is free or, when concurrent requests occur, which one will be prioritized by the memory controller.

Another important source of unpredictability are shared caches between CPU cores as simultaneously running tasks may mutual evict data from the other cache. This data must then be fetched from the front-side bus, which is also shared on COTS.

The DRAM architecture is also highly unpredictable, particularly because the data path is bi-directional and it takes several CPU cycles to switch from read to write mode. A refresh of the DRAM memory must occur in defined cycles, which will delay a read or write request. And although different DRAM banks can be accessed parallel, a row inside a bank must be opened before access.

2.3 Impact of Shared Resources

The impact of shared resources on different modern COTS multi-core architectures has been analysed by Radojković et al. [3]. For that purpose they designed benchmarks which stressed a single shared resource like the front end of the pipeline, the level 1 data cache, the level 1 instruction cache, the level 2 cache or the main memory. To evaluate the slowdown three benchmarks have been executed together with the resource stressing benchmarks on different Intel processors, like the Atom Z530, the Pentium D and as well on the Core2Quad. The first of the three benchmarks is the Space Time Adaptive Processing Radar, which detects moving objects with the data of an echo, the second is the CoreMark, a benchmark to measure the CPU performance, and at last, is the encoding with the H.264 compression standard.

The results show a notable impact when the level 2 cache and memory stressing benchmarks have been executed, with a slowdown of up to 15.3 times for the Atom and 14.4 times for the Core2Quad. On the Pendium D , however, the slowdown was only 10% for the level 2 stressing and 30% when the main memory has been accessed intensively. The impact of the level 1 cache is only notably on the Atom, which, however, is only capable of simultaneous multithreading, a form of hardware multithreading, where also the level 1 cache is shared. This work focuses on multi-core systems with a dedicated level 1 cache for each core.

The impact of a stressed memory bus on the Pandaboard ES, the hardware is described in section 3.1, has been analysed by Borghorst et al. [4] and can be seen in figure 2.1. It shows the access time without advanced preloading to a data array in CPU cycles. Four tasks accessed 12,000 words randomly. The bottom line shows level 1 cache hits with a stable access time of 7 cycles. All words that have been in the level 2 cache can be found in the second line, also with stable access times between 33 and 37 cycles. However, all other words are located in the level 3 DRAM and have therefore an access time of more than 100 CPU cycles. Most are around 140, but some are even higher than 250 cycles and thus make the data access unpredictable for systems with real time constraints. 566834 CPU cycles have been necessary in total to access the 12,000 words.

On the other hand, if the data is preloaded in the level 2 cache as we can see in figure 2.2, we get better access times. Because all data is preloaded, there is always a cache hit and no level 3 DRAM access. The level 1 cache access time is now between 7 and 10 CPU cycles instead of just 7, which is still a good result. The level 2 access time is still between 33 and 37 cycles. In total there were 262,361 cycles needed instead of 566,834 cycles without preloading, which is more than half the time. So the preloading guaranteed that every memory access is a level 2 cache hit, no more than 37 CPU cycles were necessary for one access.

A crucial assumption for an efficient cache preloading is a stable preload time per byte. It has also been shown that the preload time in CPU cycles per byte is stable with 8 CPU cycles. The results can be seen in figure 2.3 for a bulk transfer from a preload size of 1024 bytes on if only one CPU is preloading at a time .

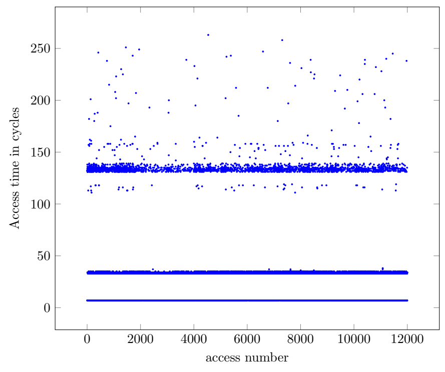


Figure 2.1: The figure shows the random access of 10,000 values. The access number is depicted on the x-axis and the y-axis shows the time for an access in CPU cycles. Source: [4]

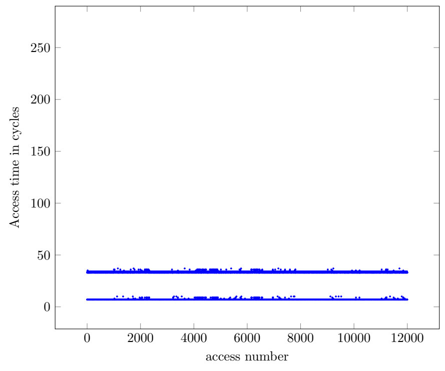


Figure 2.2: The figure shows the random access of 10,000 values which have been preloaded and locked in the level 2 cache. The access number is shown on the x-axis and the time for an access in CPU cycles on the y-axis. Source: [4]

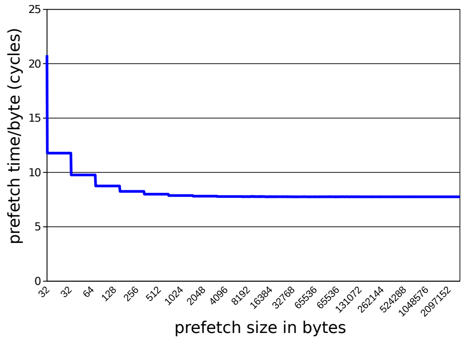


Figure 2.3: The x-axis is the preload size in bytes and the y-axis represents the preload time per byte in CPU cycles. Source: [4]

2.4 Proposed Techniques

Different solutions have been proposed in the literature to counteract the unpredictability caused by the memory in multi-core systems with the level 2 cache. According to Mancuso et al. cache partitioning, page colouring and cache lockdown are three categories for solutions today [5].

Cache partitioning uses features of the cache controller to prevent mutual eviction and give tasks or cores an exclusive access to parts of the cache. It is also possible to implement the method in software, but it is very complex. Plazar et al. [6] presented a technique to partition the instruction cache in software and a method to determine the optimal partition size for each task.

Page colouring influences the mapping of virtual memory to physical. Because caches use physical instead of virtual addresses today, it tries to ensure that virtually contiguous pages are also adjacent in the physical address space. Otherwise the page could cause an eviction of a whole cache way with already relevant data inside. Proposed solutions improved the average case performance, but cannot guarantee worst-case execution times.

Cache lockdown, on the other hand, uses features of the cache controller to lock a cache way or cache line and except them from the replacement policy. So it is possible to statically preload lock important data at system startup in the cache or dynamically preload necessary parts before use, e.g. the scheduler always preloads and locks the next task in the cache. The next section introduces an operating system model which uses the dynamic and static cache lockdown technique.

3 Operating System

The ARM prototype implementation of the cache-aware operating system model presented by Borghorst et al. [4] has been extended with the proposed cache management functionality at operating system level, which allows to dynamically preload necessary parts in the cache. The operating system takes therefore control over the level 2 cache replacement policy at way level.

Today a common level 2 cache consists of lines which are grouped in ways of the same size, where the count of the ways is called associativity. So each memory address can only be cached in one of the ways and there only in one of the lines. In other words, the memory is divided in parts with the size of a way and all of the parts can be mapped in the cache. The cache controller decides for every cache miss with a replacement strategy, which way will be mapped to the newly accessed memory part. For the operating system it is possible to lock and unlock cache ways and except them from the replacement strategy and controls in the end which data gets evicted or cached. So when every but one way is locked and data from the main memory is accessed, it will be preloaded in the specific unlocked way. The way will be locked right after the preloading process and the data cannot be evicted anymore.

The operating system will be separated in operating system components (OSCs), a highly modular and fine granular component. To achieve the separation, the data of each OSC which has to be preloaded, must be isolated and each OSC will have its own code, data and stack segment. However, a separate stack for each component requires major changes of the prototype operating system and is not implemented here. Right now the system has one system-wide stack for each core which will be locked in the cache permanently. Additionally, important parts of the operating system, like the interrupt handling and interrupt handlers, will be grouped in OSCs and a way will be implemented to dynamically preload the interrupt handlers in the level 2 cache right before use.

3.1 Hardware Platform

A Pandaboard ES, based on an OMAP4460-SoC from Texas Instruments, with two Cortex-A9 cores and 1 GB DDR2 RAM, has been used as a test platform for the prototype operating system. Additionally to the 32 KB data and 32 KB instruction cache it features a L2C-310 level 2 cache controller with 1 MB in total. The cache is divided in 16 ways with 64 KB each and supports different methods to influence the cache replacement policy, specificly locking by line and way. It is also possible to lock a way or line for a specific CPU core, so the specific locked core cannot cause any eviction.
The L2C-310 cache controller features two replacement strategies. The default one is

round-robin, but it is possible to use a pseudo-random replacement linear feedback shift register (LFSR) algorithm instead [7]. LFSR consists of a chain of bits, which represent the pseudo random output, and the bits will be shifted from the left to the right or the other way round. The new input bit is calculated every shift from the last state of the bits. Because the alternation of the bits is finite, the cycle will repeat itself. For the evaluation the default round-robin strategy has been used, which will choose the next unlocked way when there is no more invalid available. The pseudo-random LFSR algorithm, on the other hand, chooses, if no invalid way can be used, a pseudo random way instead. However, it has been shown that caches with a pseudo-randomized replacement have a worse hit rate probability [8].

3.2 OSC Segmentation

The operating system is designed for embedded systems, so the complete behaviour, like all running tasks, is well known at compilation time and no dynamic changes occur later on and thus the linker can be used to group and separate the data and instruction segments of an OSC. For that purpose the operating system has custom data structures named *preload_information*, which contain all information about the data and code segments of the OSCs that should be preloaded as it is shown in figure 3.1.

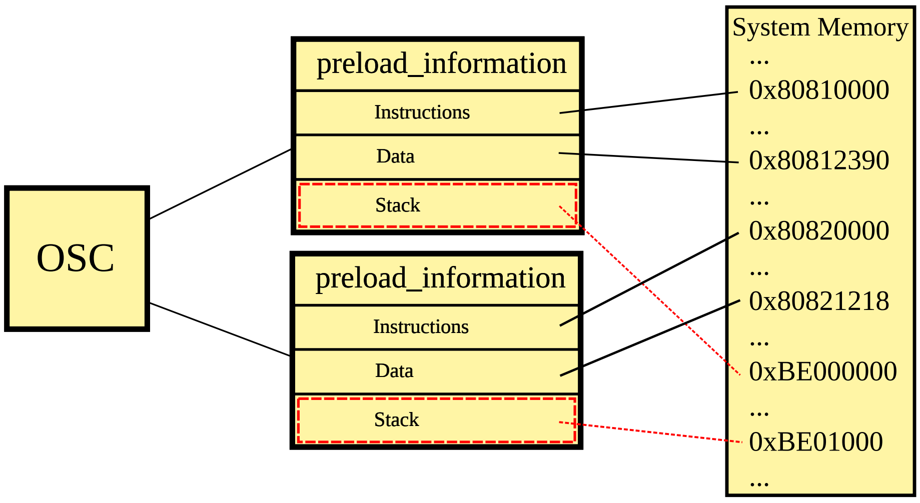


Figure 3.1: An example of an OSC with references to two prelaod_information data structures, which hold information about the place where the data and instructions of the OSC are located in the memory. As this work does not provide that each OSC has its own stack, it is marked red.

Every *preload_information* structure represents one cache way, so the addresses must be way aligned and have a maximum size of one way. If an OSC does not fit in one cache way, the component must be structured in the linker script to use as many *preload_information* structures as necessary. The instruction section is way aligned and the data sections is directly after the instruction section. Both sections must not exceed the size of a way in total. To get the addresses and size of those sections, we define symbols in the linker script which are also defined as external variables in the source. Figure 3.2 contains a possible definition of an OSC in the linker script.

```
.text.preloadTemporary0 :
{
    . = ALIGN(0x10000);
    temporary0CacheInstructionsStart = .;
    OSCCacheInstructionsStart = .;
    ./build/object_file0.o(*text*)
    ./build/object_file1.o(*text*)
    OSCCacheInstructionsEnd = .;
}
.data.preloadTemporary0 :
{
    OSCCacheDataStart = .;
    ./build/object_file0.o
    ./build/object_file1.o
    OSCCacheDataEnd = .;
    ASSERT( . - temporary0CacheInstructionsStart <= 0x10000 ,
        "Preloading0␣section␣>64k");
}
```

Figure 3.2: Example excerpt from the linker script which defines and groups a cache way aligned OSC. The script contains two sections, the first one is for instructions and the second one for data, and they must not exceed the size of a cache way, here 0x10000 bytes or 64 KB. All symbols are declared as external variables in the source code to determine the start and end address of the sections. All object files which have dependencies with the OSC must be included.

3.3 OS Cache Management

As every *preload_information* represents data and instructions that fit exactly in one way, the cache management provides a method to check whether the corresponding data and instructions pointed to by the *preload_information* are already preloaded and locked in the cache. If not, the best suitable way will be chosen by a cache replacement algorithm and the preloading process starts. LRU has been chosen as the replacement algorithm, because it provides good results and is fast and easy to implement. It is also possible to preload and lock critical OSCs permanently in the cache, so they are exempt from the replacement policy.

LRU has been implemented as a linked list with the least recently used way as the tail and the recently used as the head. A stable execution time is guaranteed because the linked list is manipulated without dynamic loops. The only pitfall is a spinlock to prevent mutual access to the linked list. The preloading process occurs per definition only on one core at the same time, but a way must been marked as used for every interrupt. So without the spinlock it would be possible that two interrupts occur at the same time on two cores and corrupt the linked list by trying to move the used item to the head.

Critical parts are indispensable OSCs, which are always involved in the interrupt handling and a negative performance would impact the real-time behaviour of the interrupt handling. In the current prototype the interrupt handling system and the cache management itself, which will be descriped in the next section, have been grouped in two OSCs and will be preloaded and locked permanently in the cache at boot time. Additionally two OSC have been created which only reference the stacks of the two cores and will also be preloaded at boot time. The size of each stack is equally to one cache way, here 64 KB.

It has been shown and can be seen in figure 2.3, that the preload time is stable as long as only one core is accessing the main memory. Thus it must be guaranteed that only one preload process takes place at the same time. Right know the software does not prohibit two simultaneous preload processes. However, the interrupts are configured to only target one core. It is possible to lock the preloading function, so only one preload process can start and the others have to wait, and the WCET will be adjusted accordingly. But this is a waste of CPU time.

The ARMv7 architecture supports memory hint instructions, which tell the CPU that a specific part of the memory will be used. As the instructions are only a hint to the memory system they may be ignored and treated like a *NOP*, but the previous work had shown that this is not the case on the Pandaboard ES and the data will actually be preloaded in the cache. For an instruction prefetch, the operating system uses the *PLI* instruction and for a data prefetch the *PLDW* instruction.

3.4 Interrupt Handler

Like the cache management functionality, the OSC, which holds the general first stage interrupt handling part of the operating system, is an essential and critical part, hence it must reside in the cache permanently and will be preloaded at boot time. The second level interrupt handler is an OSC of the operating system model called *gate* and may have of one or more *preload_information structures*.

Figure 3.3 gives an overview of the first stage interrupt handler. Whenever an interrupt occurs, the small assembler subroutine *guardian asm handler* will be invoked, which calls a handler written in c. This handler calls the *plugbox*, which is basically an interrupt vector table, to get the approprieate gate OSC to handle the interrupt. Before the *plugbox* returns the gate OSC, the *preloading controller* will be invoked to preloade and lock the OSC in the cache.

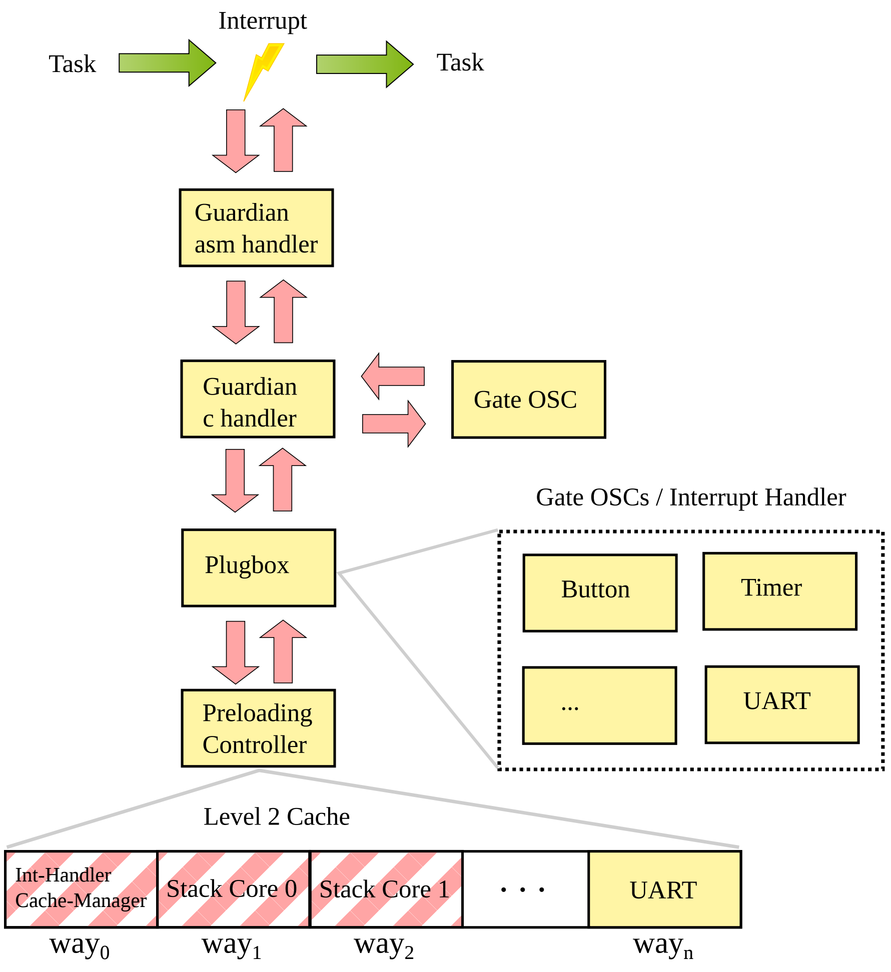


Figure 3.3: The figure gives an overview of the interrupt handling in the operating system. The level 2 cache is managed by the *preloading controller*, where the ways with the red stripes are locked permanently at boot time and the content of the other way is only locked temporarily and may be replaced.

4 Evaluation

As the previous work suggests a significant improvement of the real-time properties, the presented cache-aware prototype operating system has been used to perform various benchmarks on the Pandaboard ES. This section describes the benchmarks which have been executed, postulates assumptions and evaluates the results.

4.1 Benchmarks

To analyse the impact of the cache management functionality of the operating system, three interrupt handlers have been evaluated:

1. A pseudo software generated interrupt handler
2. An UART interrupt handler
3. A digital protective relay algorithm in combination with the UART interrupt handler

Each benchmark has been executed under three different scenarios, where no preload functionality stands for a hardware based cache management and preload functionality for the proposed operating system based cache management:

1. Operating system with no preload functionality and no interferences to get an impression of the fastest possible execution time.
2. Operating system with no preload functionality, where the hardware controls the cache replacement strategy, and interferences to outline the impact of unpredictable behaviour of multi-core systems.
3. Cache-aware operating system with preload functionality and interferences to show improvements of the real-time properties.

The cache controller has been configured to use round-robin as the replacement strategy as it has been described in section 3.1.

To analyse the impact of the interference that occur on shared resources in multi-core systems as described in section 2.2, it is necessary to simulate system load. This can be achieved by either stressing several resources at once or put a maximum of load on a single one as it has been presented by Radojkovic et al. [3]. However, the main focus of this work is to counteract the impact of shared resources with the level 2 cache, so it does make sense to focus on the caches and the DRAM. The benchmarks did also not access the memory all the time, so it is not desired that the interference task stresses another resource at that time. Therefore a task copied 2 MB of data in a loop from one address to another and the data size is large enough to cause an eviction of the level 1 as well as the level 2 cache and main memory access. The task which triggered the software interrupt, on the other hand, ran together with the interference task on one core. Due to the non pre-emptive scheduler, it has also been guaranteed that the interference task will complete the copy of the 2 MB and cause an eviction if the cache is unlocked.

4.1.1 Assumptions

It is to assume that the results will fulfil the following expectations:

1. The jitter of the execution time should be lower in the cache-aware system than the one with hardware cache management due to the stable access time of the level 1 or level 2 cache as it has been evaluated in section 2.3. This results in a better predictability of the interrupt handlers, because it is possible to narrow the execution time.

2. The average execution time of the interrupt handler should be significantly better than the one where the cache management is under hardware control, because it is not necessary to access the slower and shared main memory with all unpredictable side effects [2.2]. For interrupt handlers which just have few data accesses and do calculations most of the time, the improvement may be smaller. Due to the overhead of the cache-aware operating system, the dynamic cache management may also increase the average execution time of OSCs with few memory accesses if it is not locked permanently in the cache.

3. It is to be expected that the average execution time increases if the OSC spans across multiple ways and is not locked permanently. On the one hand, the cache-aware operating system works on a way basis, so additional ways increase the overhead. On the other hand, if the hardware controls the cache replacement strategy, it is more likely that one cache way will be evicted and the impact is more extreme.

4. If the hardware takes control and no interferences occur, the behaviour will be similar to a uniprocessor system and have less unpredictable influence. So it is assumed that this constellation will provide the best results and therefore defines an optimal boundary.

4.2 Software Generated Interrupt

A good way to outline the impact of a cached interrupt handler is the use of a pseudo software interrupt. Therefore a task runs on one core alternating with the interference task, generates software interrupts and measures the CPU cycle count directly before and after the interrupt will be triggered. The OSC of the interrupt handler is preloaded once in the cache when the first interrupt occurs and will not be evicted, because in this scenario, there are more available ways than OSCs which could cause a mutual eviction. The interrupt handler accesses the memory 1,000 times in random way confined to later specified data ranges. A linear access would cause the automated preloader of the cache controller to preload a cache line, which has here a size of 256 bits and results in a cache hit in 7 of 8 accesses.

4.2.1 Setup

To compare the performance of the pseudo software interrupt handler in the cache-aware operating system with the non cache-aware system, three different set-ups have been chosen. Each of them has been run three times with the configurations described above in section 4.1, so there are nine benchmarks in total and each of them triggered the interrupt handler 10.001 times. That is, because the OSC is not preloaded in the cache on the cache-aware operating system and has been preloaded when the interrupt has been triggered the first time. But as the preload functionality would increase the execution time dramatically, the first run has not been considered. Whether the jitter of the execution time is still lower when the OSC must be preloaded every time, will be evaluated later. All in all there are 10.000 measurements and the interrupt handler's OSCs have been configured in the following way:

- Uses one cache way, 63 KB data
- Spans across two cache ways, 127 KB data
- Spans across three cache ways, 191 KB data

So the interrupt handler's OSC fits in exactly one, two or three cache ways. The way size is on the Pandaboard ES 64 KB, the OSC's instruction size is 1 KB and is in the first part of the way, so 63 KB are left for the data field.

4.2.2 Results

The results are promising and the standard deviation, see table 4.1b, and the average, see table 4.1a, of the execution times for each benchmark have been calculated for comparison. Additionally the plots in figure 4.1 and the histograms in figure 4.2 give a good graphical representation of the results and the assumptions from earlier can be confirmed:

1. Lower jitter:
   Table 4.1b supports the first assumption. The standard deviation of the benchmark with interferences and preloading is with 249, 215 and 212 for the data field size of 63 KB, 121 KB and 191 KB significantly lower than without preloading. Without preloading the standard deviation varies between 1,466, 1,721 and 1,799, which increases with the size of the data field.

Benchmark	1 way	2 ways	3 ways
no interference, no preloading	32,769	33,368	33,568
interference, no preloading	63,741	79,554	86,877
interference, preloading	36,721	37,225	37,450

(a) Average

Benchmark	1 way	2 ways	3 ways
no interference, no preloading	127	139	171
interference, no preloading	1,466	1,721	1,799
interference, preloading	249	215	212

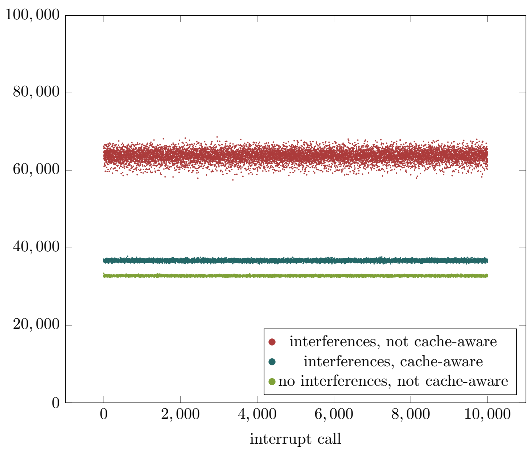
(b) Standard deviation

Table 4.1: The average and standard deviation of the benchmarks with the software generated interrupt. The OSC with the interrupt handler has 1 KB instrucions and either 63 KB data, 127 KB data or 191 KB data, so it fits in either one, two or thee cache-ways.

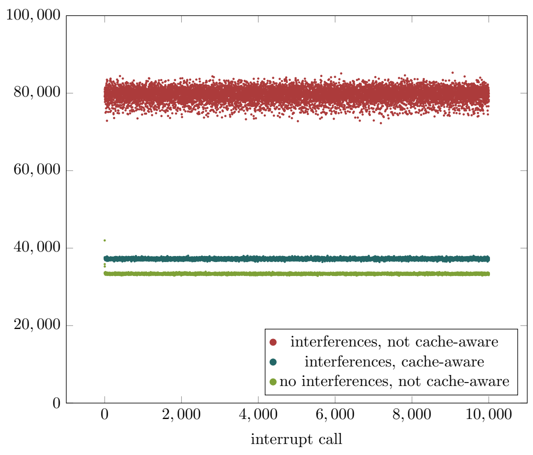
2. Better average execution time:
   The execution time is significantly better if the interrupt handler has been preloaded. Table 4.1a shows that the average for the not cache-aware system is 63,741 CPU cycles for the OSC which uses one way, 79,554 for two ways and 86,877 for three ways. In contrary the execution time of the cache-aware operating system is stable with an average of 36,721, 37,225 and 37,450. The plots in figure 4.1 illustrate the improvements.

3. OSCs which span across multiple ways:
   Table 4.1a shows that the average case of the preloaded scenario is more than half the time if the OCS spans over two or three cache ways compared to the not preloaded one. Even if the data field is only 63 KB in size, the execution is 74% slower if the OSC is not locked in the cache. The cost to check if an OSC is in the cache increases with each cache way the component spans across, which explains the higher execution time of the cache-aware.

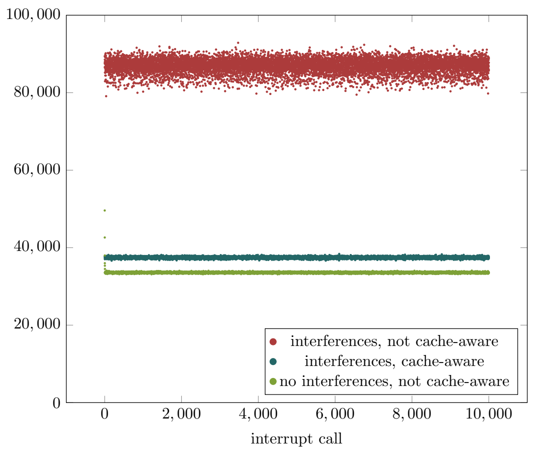
4. Optimal results:
   The best results have been measured when no interferences occurred, so it defines a good upper boundary.



(a) 1 cache way and 63 KB data

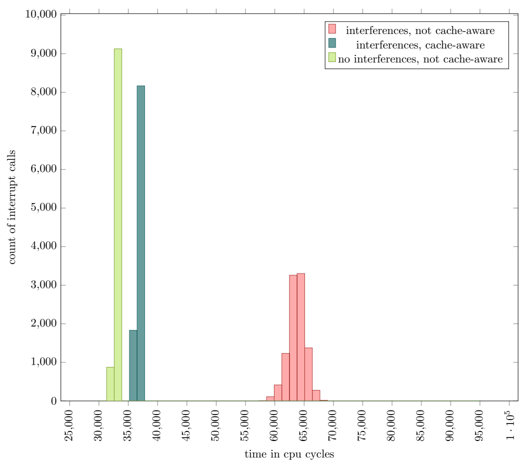


(b) 2 ways and 127 KB data

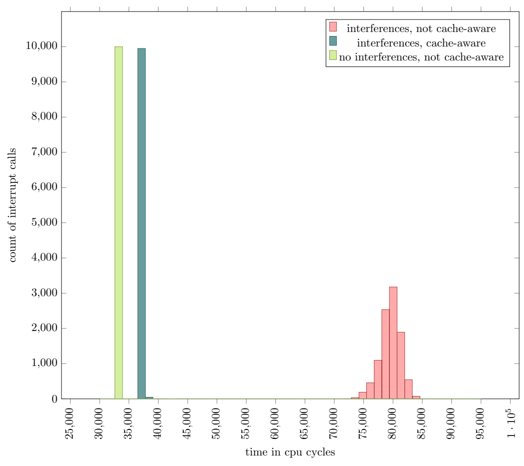


(c) 3 ways and 191 KB data

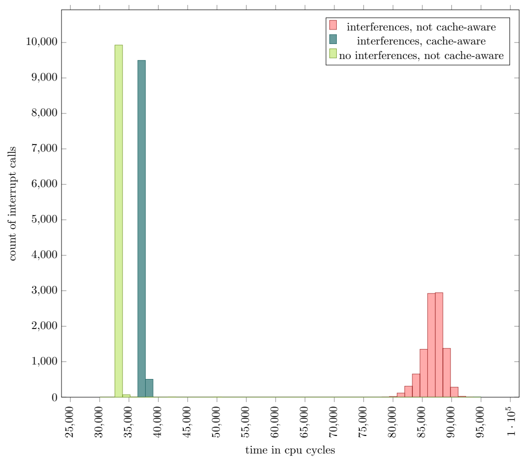
Figure 4.1: Results of the software generated interrupt benchmarks, where the x-axis represents the 10,000 triggered interrupts and the y-axis the execution time in CPU cycles. The three plots show the results for the three different set-ups of the experiment. Plot a) shows the results when the OSC uses one cache way and accessed 63 KB of data, b) shows the results when the OSC spans across two cache ways and the data range is 127 KB and c) shows the results when the OSC spans across three cache ways with a data range of 197 KB.



(a) 1 way and 63 KB data



(b) 2 ways and 127 KB data



(c) 3 ways and 191 KB data

Figure 4.2: The figures show the distribution of the results from the benchmark of the software generated interrupts [4.2], where a) shows the distribution of the experiment which used one way and a 63 KB data range, b) shows the distribution of the experiment which spanned across two ways with a data range of 127 KB and c) shows the distribution of the experiment which spanned across three ways and had a data range of 191 KB. The x-axis represents the execution time in CPU cycles and the y-axis the count of the triggered interrupts. All measured execution times are greater than 30,000 and smaller than 95,000 and have been binned into 50 groups with a range of 1.300 CPU cycles each.

4.3 Overhead of the Cache-Aware Operating System

A crucial function of the cache-aware operating system is to check, whether an OSC is in the cache or not. Hence, this function is executed every time when an interrupt occurs and causes a major overhead. It is assumed here that the OSC has already been in the cache, so the overhead is the check and to mark the OSC as used in regards of the LRU replacement algorithm. As the check is for every possible interrupt handler the same, the software generated interrupt with the same handler from the previous benchmark has been used again. The question here is how this function will behave and big the overhead is.

4.3.1 Setup

The setup is also like the previous one of the software generated interrupt benchmarks. The only difference is, that only the part of the cache-aware operating system has been measured, which checks whether the OSC is preloaded or not. Therefore the CPU cycle counter has been read in the *plugbox* right before and after the call to the *preloading controller*, see figure 3.3 for details.

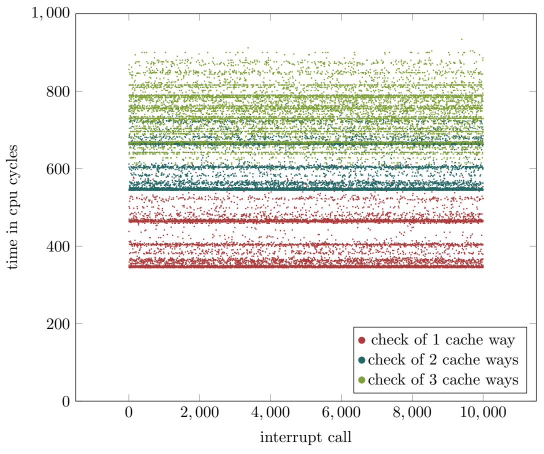
4.3.2 Results

The results in table 4.2 show a very low standard deviation of around 60 CPU cycles and an average of 400, 601 and respectively 740 CPU cycles. So the overhead can be considered insignificant, because a pseudo interrupt handler with 1,000 memory accesses already needs around 33,000 CPU cycles in the best case.

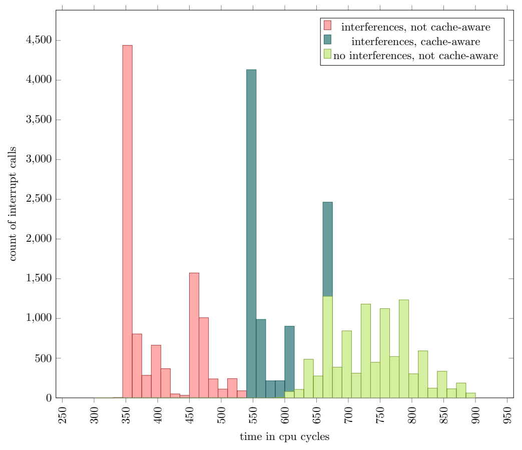
The average execution time does not increase linear with more cache ways, which may be caused by level 1 cache hits. A level 1 cache eviction happened between the interrupt calls, so the check of the first way results in a level 2 cache hit because the interrupt handling system is locked permanently in the cache. After the first check the appropriate data and instructions have been stores in the level 1 cache and the check for more ways is accordingly faster.

Benchmark	1 way	2 ways data	3 ways
Standard deviation	59	59	63
Average	400	601	740

Table 4.2: Results of the overhead of the cache-aware operating system. The standard deviation and average over 10.000 values in CPU cycles.



(a) Results of the overhead benchmark. The interrupt calls are represented on the x-axis and the execution time in CPU cycles on the y-axis.



(b) Distribution of the results. The x-axis represents the execution time in CPU cycles and the y-axis the count of the triggered interrupts. All measured execution times are greater than 300 and smaller than 800 and have been binned into 40 groups with a range of 20 CPU cycles each.

Figure 4.3: Results of the overhead of the cache-aware operating system. a) shows the plot and b) the distribution of the execution time for the check, whether the OSC is already cached.

4.4 OSC Eviction

The first experiment evaluated the performance of the software generated interrupt handler without the preloading time. In a realistic scenario it is likely that the size of the operating system is larger than the cache and preloaded OSCs get evicted from time to time. So this benchmark used the same interrupt handler as in section 4.2 and evaluated the scenario when the OSC gets preloaded every time.

4.4.1 Setup

The setup has been like the other one of the software generated interrupt with two differences. Firstly, all ways which cached parts of the benchmarked OSC have been cleand and the OSC marked as not in the cache, so it has been preloaded again on the next interrupt. And secondly, the interference task has been modified to only access data that is at least in the level 2 cache and was necessary, because the preloading process can only be done on one core per definition. So the task causes only level 1 cache evictions and load on the level 2 cache.

4.4.2 Results

Table 4.3 shows the results which support the assumption of a stable preloading and execution time. The standard deviation with 344, 313 and 315 is negligible in comparison with the average of more than 200, 400 and 600 thousand CPU cycles. And if it is compared with the results from the first experiment without the preloading functionality, the standard deviation only increased by around 100 cycles. In comparison with the standard deviation of the non cache-aware operating system from the first benchmark, the results here are still around 4 times better. If the measured execution time of the interrupt handler of around 37,000 CPU cycles from the first benchmark is subtracted from the average here, there is a preloading time of slightly less than 200,000 cycles per way. So the cache-aware operating system may have a worse average execution time, but it is possible to provide a more precise WCET estimation due to the lower jitter.

Benchmark	1 way	2 ways	3 ways
Standard deviation	344	313	315
Average	235,159	433,942	631,942

Table 4.3: Results of the overhead of the cache-aware operating system. The standard deviation and average over 10.000 values in CPU cycles.

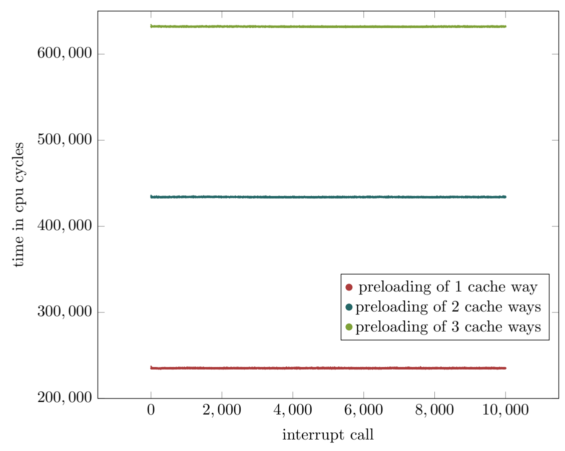


Figure 4.4: Results of the OSC eviction benchmarks, where the x-axis represents the count of interrupt calls and the y-axis the execution time in CPU cycles. The cache has been cleaned after every interrupt which caused a preloading process every time. The experiment has been done for an OSC which uses one, two and three ways.

4.5 UART Hardware Interrupt

The UART has been chosen to measure a hardware initiated case. Therefore the UART has been configured to trigger an interrupt for every character that arrived and the handler stored the received data in a buffer.
An eviction of the OSCs from the cache, like in the first benchmark of the software generated interrupt, also did not happen here.
Interesting will be the impact of the cache-aware operating system, because the interrupt handler is very small and does not perform massive memory accesses. It may be possible that the overhead of the check, whether the OSC is locked in the cache or not, is far greater than the benefit of guaranteed level 2 cache hits.
Another question is the impact of the delay between interrupts. If the characters are send as fast as possible, it is likely that the interrupt will be triggered right after the end of the last one, which could reduce the impact of the interference on the level 1 cache, as every core has a private one.

4.5.1 Setup

The UART interrupt handler and all necessary dependencies have been grouped in a OSC which fits in one cache way. To evaluate the performance of the interrupt handler, 10.001 characters with 1 byte each have been sent via to the UART with 115.200 Bps from an external device. Each character triggered an interrupt, so there are 10.000 measurements beside the first one, when the cache-aware operating system performs a preload operation.

The CPU cycles have been read once at the beginning and once at the end of the function in the *guardian c handler* of the interrupt handling system, as presented in figure 3.3. So a handful of assembler instructions in the *guardian asm handler* function have not been included in the measurement, but they can be considered as irrelevant and not influential.

The measurement has been repeated three times with different delays between the interrupts:

- The characters have been sent as fast as possible
- The characters have been sent with a delay of 0.1 seconds
- The characters have been sent with a delay of 0.2 seconds

As described in section 4.1, there are nine benchmarks in total.

4.5.2 Results

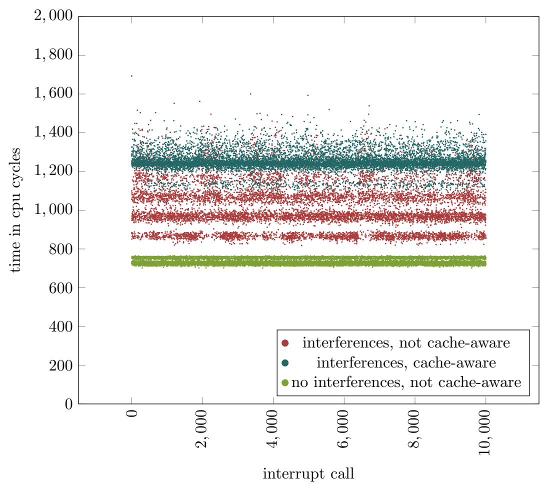
The standard deviation and the average of the execution times for each benchmark are in table 4.4a and 4.4b, whereas figures 4.5 and 4.6 give a good graphical representation with plots and histograms for the results.

1. Lower jitter:
   The data, especially visualized in figure 4.5 as well as 4.6, shows clearly that the preloaded OSC has a notably smaller standard deviation, which results also in a smaller jitter, in the two configurations with interferences.

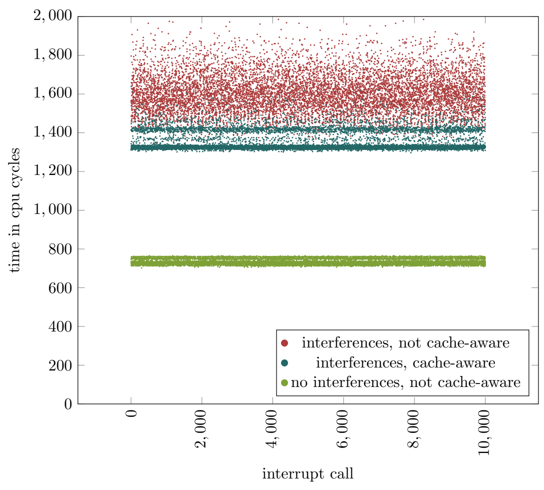
2. Better average execution time:
   The average execution time of the benchmarks with the cache-aware operating system is lower compared to the benchmarks where the hardware takes control when the interrupt has been triggered every 0.1 and 0.2 seconds. Though the cache-aware system has been slower when the characters have been sent as fast as possible, so the same interrupt occurred very fast or consecutively. That may caused the task, which causes the interferences, to be interrupted a lot and makes eviction from the level 1 cache unlikely. Another major impact is the check whether the OSC is preloaded or not. This increases the average execution time about a more or less constant time [4.3] and is especially notable when the interrupt handler is small and compact like this one.

3. OSCs which span across multiple ways:
   No OSCs span across multiple ways in this setup.

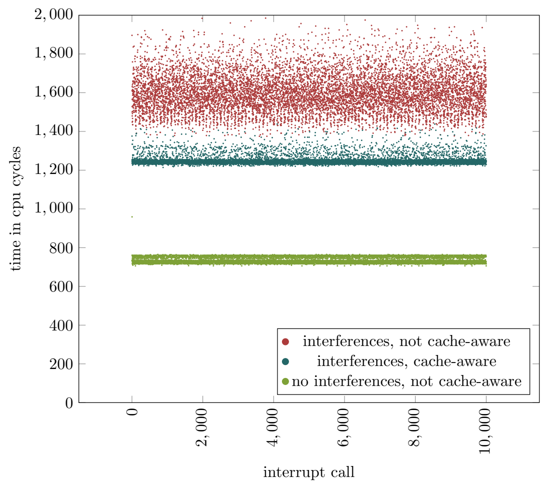
4. Optimal results:
   The results support clearly assumption 4, that the configuration with no interferences behaves most predictable, as the standard deviation is, in this case, the lowest and is also the fastest with the lowest average in all three set-ups. Only the first interrupt call has a significantly higher execution time for accessing the DRAM, because the OSC has never been used and cached before.



(a) UART triggered as fast as possible



(b) UART triggered every 0.1 seconds



(c) UART triggered every 0.2 seconds

Figure 4.5: Results of the UART hardware interrupt benchmarks, where the x-axis represents the 10,000 triggered interrupts and the y-axis the execution time in CPU cycles. The three plots show the results for the three different set-ups of the experiment. Plot a) shows the results when the interrupt has been triggered as fast as possible, b) shows the results when the interrupt has been triggered every 0.1 seconds and c) shows the results when the OSC has been triggered every 0.2 seconds.

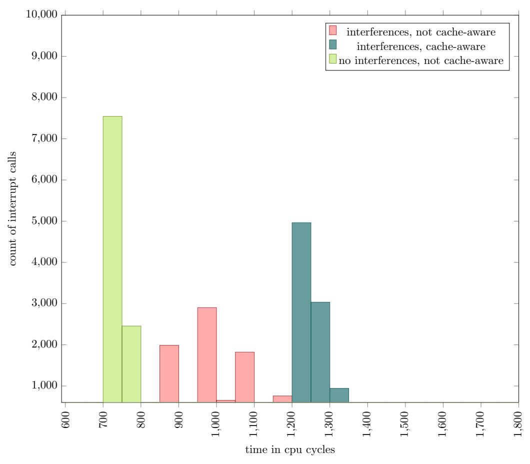
Benchmark	fast	0.1 seconds	0.2 seconds
no interference, no preloading	735	735	736
interference, no preloading	1,014	1,603	1,602
interference, preloading	1,253	1,355	1,250

(a) Average

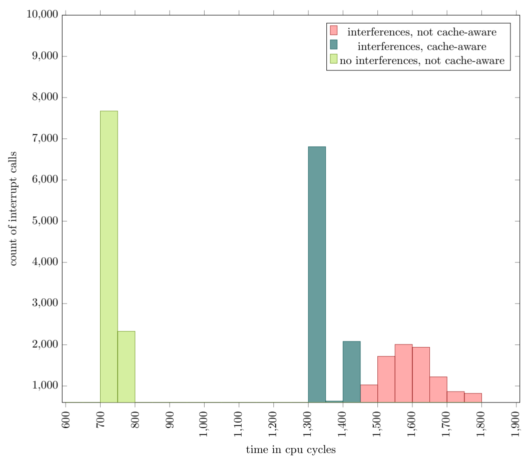
Benchmark	fast	0.1 seconds	0.2 seconds
no interference, no preloading	15	14	15
interference, no preloading	120	100	100
interference, preloading	49	49	24

(b) Standard deviation

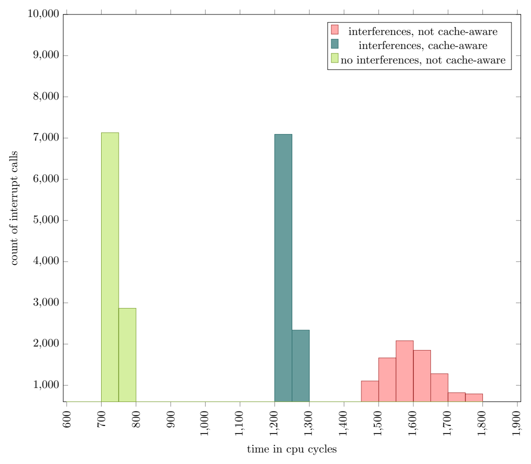
Table 4.4: The average and standard deviation of the benchmarks with the software generated interrupt. The OSC with the interrupt handler has 1 KB instructions and either 63 KB data, 127 KB data or 191 KB data, so it fits in either one, two or thee cache-ways.



(a) UART triggered as fast as possible



(b) UART triggered every 0.1 seconds



(c) UART triggered 0.2 seconds

Figure 4.6: The figures show the distribution of the results from the benchmark of the external triggered UART hardware interrupts [4.5]. Histogram a) shows the distribution of the experiment in which the interrupt has been triggered as fast as possible, b) shows the distribution of the experiment in which the interrupt has been triggered every 0.1 seconds and c) shows the distribution of the experiment in which the interrupt has been triggered every 0.2 seconds. The x-axis represents the execution time in CPU cycles and the y-axis the count of the triggered interrupts. All measured execution times are greater than 700 and smaller than 1.800 and have been binned into 24 groups with a range of 50 CPU cycles each.

4.6 Digital Protective Relay

The evaluation of the software and the UART interrupt handler has shown good results. However, it is in that configuration not a realistic application. For a more realistic benchmark the software of a digital protective relay has been implemented in the prototype operating system.

A digital protective relay is a power-system protection, which detects faulty parts and separates them from the rest of the system to keep it stable. Hence, the system constantly gets an input of the amperage and voltage from various meters and calculates the phasor, which indicates if there is a short circuit or not.

Björn Keune developed such a system for his master thesis [9]. The software part has been implemented in C++ and the aspect-orientated hard real-time operating system CiAO [10] has been chosen with two tasks running on it. The first task runs for 250 μs and constantly receives the amperage and voltage of the power-system over an ethernet bus, whereon the second tasks runs for 1000 μs and tries to calculate the phasor. For that the algorithm needs enough value pairs. The execution time increases strikingly when enough values are available to determine the phasor due to intense calculations. For testing purpose, he also developed a random input data generator.

It is to assume, that the benefit of the preloading will not be so drastic, because the algorithm does not perform many memory accesses but rather does heavy computation in the phasor part.

4.6.1 Setup

As the digital protective relay algorithm has been programmed in C++, it was possible to extend the UART OCS with some minor changes. However, the prototype operating system for the Pandaboard does lack an ethernet driver, so the UART has been used as the data source. The random data generator has been used to generate the amperage and voltage data, 80 values each, for the benchmarks. One pair of the amperage and voltage data has been send over the UART every 0.2 seconds and the time in CPU cycles has been measured for those interrupts, where the transmission of the pair has been complete and the algorithm has been called. The 80 value pairs have been sent two times, always in the same order, and the measurement has been repeated five times for each benchmark, so the algorithm has been invoked 800 times in total for each of the configurations described in section 4.1. All floating point calculations have been emulated in software because of problems with hardware floating point support.

4.6.2 Results

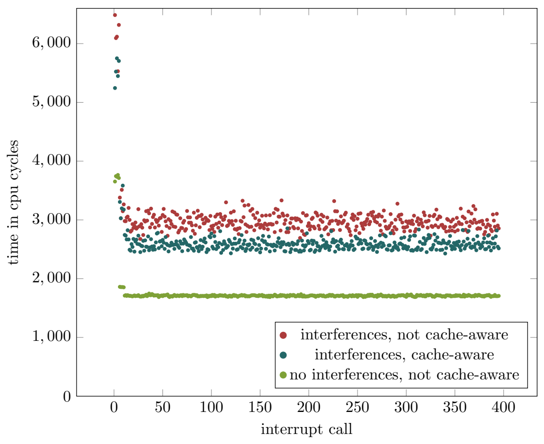
The threshold, after which the algorithm is able to calculate the phasor, is past the first 79 value pairs. Due to the fact that the execution time is notably higher when the algorithm was able to calculate the phasor, to the results will be referred as pre-phasor and phasor results.

1. Lower jitter:
   On the pre-phasor algorithm calls is the standard deviation of the cache-aware operating system according to table 4.5a slightly lower than the system with hardware cache management. In contrary to all benchmarks before, is the standard deviation with the phasor results with 64,477 to 64,362 slightly higher, but the difference is with 115 CPU cycles insignificant. But it is not clear why the standard deviation, and so the jitter, is worse. Theoretically it should at least be the same or better.

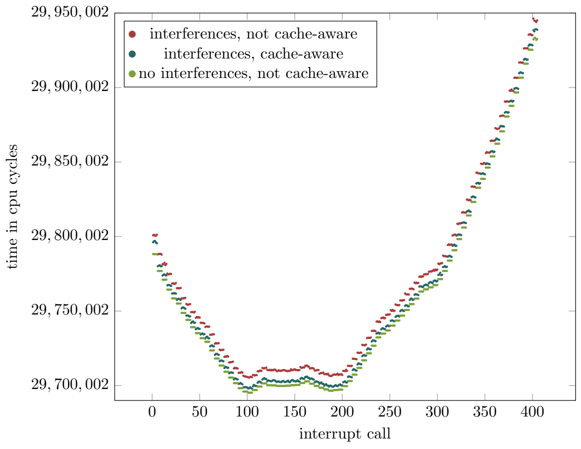
2. Better average execution time:
   The average execution time of the cache-aware operating system is lower than the system with hardware cache management. For the pre-phasor results, the execution time is in the average 149 CPU cycles or 12.5% faster. The phasor results on the other hand, have been calculated 7,252 CPU cycles faster on average, which is not that much as the average execution time is over 29 million cycles. This can be explained with the intensive calculations that have primarily been done and almost no memory access occurred. The instructions and data for the calculations have probably been in the level 1 cache, so the interference from the other core had no effect. Figure 4.7 gives a nice graphical representation. Interesting is the dent in figure 4.7b, with a sudden decrease of the execution time. It is probably caused by the calculations of the algorithm, because there it is unlikely that the cache causes this phenomenon as it appears on the cache-aware as well as on the not cache-aware operating system.

3. OSCs which span across multiple ways:
   No OSCs span across multiple ways in this setup.

4. Optimal results:
   The configuration which behaves like a single-core setup has again achieved the best results. However, the phasor results are quite alike, the average and standard deviation is just slightly better. This supports the assumption from above that extensive mathematical operations with almost no data access are not affected by interferences.



(a) Before the calculation of the phasor.



(b) With the calculation of the phasor.

Figure 4.7: Results of the digital protective relay benchmarks, where the x-axis represents the count of algorithm calls and the y-axis the execution time in CPU cycles, whereas the execution time of the algorithm depends on the calculation of the phasor and the results will be compared independently. Plot a) shows the results when the phasor could not be calculated because of too few input values and b) shows the results of the experiment with the calculation of the phasor. The algorithm has been called five times with the same 160 amperage and voltage pairs as input and to get a better overview, the results have been reordered: The execution time of the first five algorithm calls on the x-axis correspond to the execution time of the first calls of the five different runs, the second five to the second calls and so forth. Where the first 79th calls are shown in a), are the other calls, where the phasor could be calculated, shown in b).

Benchmark	Standard deviation	Average
no interference, no preloading	226	1736
interference, no preloading	375	3007
interference, preloading	349	2630

(a) Results of the execution time before the phasor could be calculated.

Benchmark	Standard deviation	Average
no interference, no preloading	64.298	29.756.230
interference, no preloading	64.362	29.766.565
interference, preloading	64.477	29.759.313

(b) Results of the execution time once the phasor could be calculated

Table 4.5: The standard deviation and average in CPU cycles of the results of the distance protective relay benchmarks. They have been calculated over the 400 measurements from before and after the phasor could be calculated.

5 Summary and Conclusion

The sources for unpredictable behaviour in COTS multi-core systems and a proposed operating system model to counteract these with cache preloading have been described. Later the interrupt handling of the cache-aware operating system has been intensively evaluated with software and hardware triggered interrupts and the results have been compared with a not cache-aware version of the operating system. The interrupts have been handled by a realistic application like the digital protective relay algorithm and pseudo handlers.

The evaluation has shown that there is indeed an improvement of the real-time properties. The jitter has been reduced extensively compared to the not cache-aware system and it was also possible to measure a notably better average execution time. However, the effects are not so drastic when the interrupt handler does not perform intensive memory accesses but rather calculations. Additionally it has been shown that the performance of interrupt handlers which are locked in the cache permanently respectively do not get evicted, is comparable to a uniprocessor system. Just the average execution time is slightly increased. It was also possible to show that the jitter of an interrupt handling is better on the cache-aware system for OSCs of different size. This makes it easier to provide a more precise WCET estimation, as long as a higher average execution time can be tolerated.

All in all it is possible to use the advantages of modern COTS multi-core systems for real-time applications as a low cost alternative to expensive custom hardware. But therefore it is necessary to consider the pitfals like the unpredictability and counter them with for example a cache-aware operating system.

6 Ongoing Work

Several interesting questions for a future research arise. Various hardware platforms lack for example a functionality to affect the cache replacement policy of the controller, like locking data inside the cache and prevent it from eviction. It has been shown that it is complex but possible to implement software based cache partitioning [6] and the operating system concept could be adjusted to support such software based cache partitioning.

According to the cache-aware operating system concept, every OSC should have a separate stack which will be preloaded too. As this has not been implemented and evaluated here, and can be done in the future.

It is also interesting for research how the preloading behaviour is when multiple cores preload at the same time. Additionally, it is not necessary to preload a complete OSC if only a small subroutine will be required. So an intelligent preloader could determine which part of the OSC is actually going to be used and preloads just that part. This would reduce the possible WCET time if the parts which are really used are significantly smaller then the whole OSC.

The cache replacement policy is also a critically part. Other policies may have a serious impact on the system behaviour and the resulting WCET, so it should be evaluated which policy is the best suitable for which setup.

Bibliography

[1] Kopetz, Hermann: *Real-time systems: design principles for distributed embedded applications.* Springer Science & Business Media, 2011

[2] Dasari, Dakshina ; Akesson, Benny ; Nelis, Vincent ; Awan, Muhammad A. ; Petters, Stefan M.: Identifying the sources of unpredictability in cots-based multicore systems. In: *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on* IEEE, 2013, S. 39–48

[3] Radojković, Petar ; Girbal, Sylvain ; Grasset, Arnaud ; Quiñones, Eduardo ; Yehia, Sami ; Cazorla, Francisco J.: On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 8 (2012), Nr. 4, S. 34

[4] Borghorst, Hendrik ; Spinczyk, Olaf: Increasing the Predictability of Modern COTS Hardware through Cache-Aware OS-Design. In: *Proceedings of the 11th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT '15)*, 2015

[5] Mancuso, Renato ; Dudko, Roman ; Betti, Emiliano ; Cesati, Marco ; Caccamo, Marco ; Pellizzoni, Rodolfo: Real-time cache management framework for multi-core architectures. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th* IEEE, 2013, S. 45–54

[6] Plazar, Sascha ; Lokuciejewski, Paul ; Marwedel, Peter: WCET-aware software based cache partitioning for multi-task real-time systems. In: *Proceedings of the International Workshop on Worst-Case Execution Time Analysis*, 2009, S. 78–88

[7] ARM: *AMBA Level 2 Cache Controller (L2C-310) Technical Reference Manual.* r3p1. 3 2010

[8] Reineke, Jan: Randomized caches considered harmful in hard real-time systems. In: *Leibniz Transactions on Embedded Systems* 1 (2014), Nr. 1, S. 03–1

[9] Keune, Björn: *Realisierung eines Distanzschutzes mit Methoden der Industrieautomatisierung.* Germany, TU Dortmund, Masters's Thesis (Diplomarbeit), September 2012

[10] LOHMANN, Daniel ; HOFER, Wanja ; SCHRÖDER-PREIKSCHAT, Wolfgang ; STREICHER, Jochen ; SPINCZYK, Olaf: CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems. In: *USENIX Annual Technical Conference*, 2009

List of Figures