Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Algorithms for Memory-Bus Bandwidth Control on Multi-Core Embedded Platforms with GPU Accelerators

Anton Voznia

Supervisor: Ing. Michal Sojka Ph.D.
May 2020

Acknowledgements

I would like thank to my supervisor Michal Sojka for explaining me main ideas of CPU working, working benchmarks, and helping with the work, and Joel Matějka.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 22, 2020

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 22. května 2020

Abstract

In this work, I described a problem of inter-core interference in using one bus. And I showed different solutions which already exist. But the main focus in on the certain solution memguard. I described the algorithm and compared it with yet existing solutions PREM and Legacy as I showed in the next parts of this work. To demonstrate the differences, I expanded the already exists benchmark. All results of the benchmark's working are in graphs. And it would be clearly understood. Also, here is described any additional information which is useful for understanding the algorithm memguard: hardware description, compilers, CPU architecture, instruments that I used.

Keywords: CPU, PREM, memguard, jádro, algoritmus, srovnávací kritérium, paměť, integrovaný, sběrnice

Supervisor: Ing. Michal Sojka Ph.D.

Abstrakt

V této práci jsem popsal problém mezi jádry procesoru při použití jedné sběrnice. Také jsem ukázal různá řešení které už existovaly. Ale hlavně bylo řešení memguard. Popsal jsem tento algoritmus, a porovnal jsem ho s už existujicími řešení PREM a Legacy, jak jsem posal to dále. Abych ukazál odlišnosti, rozšiřil jsem už existujicí benchmarky. Všechny výsledky těchto benchmarků jsou ve grafů nahledné. Pro jednoduší pochopení. Také v teto práci jsem popsal doplňkovou informaci která je užitečná pro pochopení algoritmu memguard: hardware popis, kimpilatory, CPU architektura, nastroje použité mnou.

Klíčová slova: CPU, PREM, memguard, core, algorithm, benchmark, memory, embedded, bus

Překlad názvu: Algoritmy pro řízení šířky pásma paměťové sběrnice ve vícejádrových vestavěných počítačích s paralelními akcelerátory (GPU)

Contents

Figures

Tables

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Voznia Anton** Personal ID number: **474440**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Branch of study: **Computer and Information Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Algorithms for Memory-Bus Bandwidth Control on Multi-Core Embedded Platforms with GPU Accelerators

Bachelor's thesis title in Czech:

Algoritmy pro řízení šířky pásma paměťové sběrnice na vícejádrových vestavných platformách s GPU akcelerátory

Guidelines:

1. Familiarize yourself with the Memguard tool and its implementation for the Jailhouse hypervisor on NVIDIA TX2 platform.
2. Develop several benchmarks (both synthetic and realistic) and measure the overhead of the Memguard mechanism on those benchmarks.
3. Extend the Memguard with Memex – a locking protocol to avoid simultaneous memory accesses from multiple CPU cores at the same time – and evaluate its effect in explicit and automatic modes. Automatic mode means that Memex is locked automatically on Memguard budget overrun.
4. In cooperation with the supervisor, propose and implement a way of integrating Memguard/Memex with GPU workloads so that CPUs cannot "steal" memory bandwidth from the GPU.
5. Document the results.

Bibliography / sources:

[1] H. Yun et al., Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms
https://ieeexplore.ieee.org/document/7093151
[2] HERCULES: D4.5 Multi-OS integration and Virtualization
http://hercules2020.eu/wp-content/uploads/2016/04/D4.5_MultiOS_Integration_Final
.pd
[3] HERCULES: D5.3 Integrated Schedulability Analysis
http://hercules2020.eu/wp-content/uploads/2016/04/D5. 3_Integrated_Schedulability_Analysis_Final.pdf

Name and workplace of bachelor's thesis supervisor:

Ing. Michal Sojka, Ph.D., Embedded Systems, CIIRC

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **10.01.2020** Deadline for bachelor thesis submission: **25.05.2020**

Assignment valid until: **30.09.2021**

Ing. Michal Sojka, Ph.D.	doc. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____
Date of assignment receipt

_____
Student's signature

Chapter 1

Introduction

In this work I described the existing problems with multi-core systems and explained existed approaches and solutions for the problem. The main focus is on embedded and real-time systems. Where resources are limited. Similar systems are using in planes, autonomous cars, power stations. One of the solutions is algorithm memguard [HY].

Count of modern computers using multi-core processors increase. Also, computers are called multi-core systems. Multi-core systems allow increasing performance, and efficiency, and share any task between processors. Cameras in cars, robots or just video monitoring need to process too many images, Data Science and Machine Learning need many different computations, working with matrices and algorithms, Distributed systems need to confirm a transaction. The growing number of cores can accelerate a processing power.

But no always is reasonable to increase the count of cores or processors. One of the problems is a high cost for the systems. Also exist problems with the shared bus depicted on **Figure 1.1**. Modern programs and applications use different data structures, virtual machines, or just high-level programming languages. That all requires a lot of variables and memory, which means that most of the applications executed on the one machine use together common DRAM (Dynamic Random Access Memory). In this case, access of any processor impacts all over the system because the processor uses the common bus for transferring data from DRAM memory into registers. The problem can be solved by using a local memory. Most of processors hasn't own local memory. In stead of this they use cache. In this case cache for every core in the processor.

With using a cache exists a problem. The cores use the same bus for transferring data from DRAM into cache. Caches are used by core like local storage memory. They load from DRAM information from any address and continue their own working without access to the DRAM. That can increase performance, that L1 caches are near to their cores. The main idea is that reduce the number of accesses to DRAM and store data in "local storage". If some needed information isn't in the cache, the core must load new data from DRAM. The problem is that cores share the bus. If one core is using the bus to get new data from DRAM, other cores should be waiting for their turn.

Some types of tasks and problems can be solved in code by rewriting

structures or algorithms. For example, a matrix may be divided for different peace that can be placed in the memory cache. But the structures like binary trees have a random place in DRAM, and cannot be reordering for faster access.

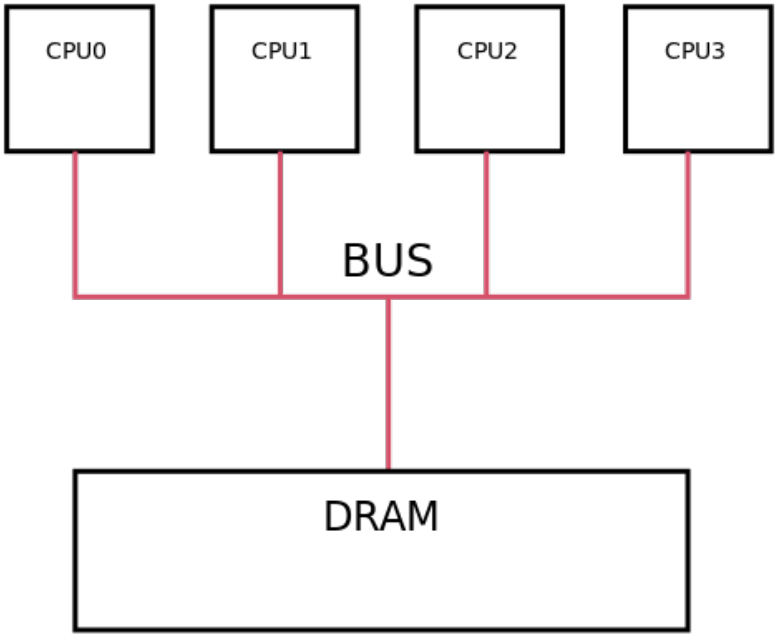

Figure 1.1: Cores are connected with DRAM by a bus

1.1 Goal: Time-deterministc execution, Problem: inter-core interference via memory subsystem

As was described above, with the bus connects the caches between the cores is a problem. The bus is one. And the cores share the bus for accessing DRAM. A core loads data from DRAM to its cache. The processor cannot load only necessary. Then the processor loads a part DRAM memory in the cache. The core is working with the data in the cache until the core doesn't

get a miss. Miss in the system means that data in the cache aren't equal to data in DRAM. And the core should bring the data from memory. And with the bus is a problem. Till any core brings necessary data, others cores are waiting in line.

1.2 Reduces inter-core interference

The problem of inter-core interference may solve in few ways.

- Preload all data will be in the cache before running computes. In this case, the first period will be a problem with inter-core interference. After, wouldn't access to DRAM, if all data fitted in the cache. But sometimes it is impossible because caches have a limit of memory.
- Rewrite algorithm, according to the computes of any cores and loading by another core don't happen at the same time. This approach can efficiently solve the inter-core interference. But it is not unique. Every issue and algorithm require their own solution. That leads to complications to predict behavior an algorithm. It requires many different tests to prove if it is effective. As was written before, exist issues that couldn't be predicted and rewrite (such binary search).
- Possible to restrict access time or quantity to DRAM memory for each core.

  In this work, the solution focuses on the last idea to restrict access for cores. Because it is important for real-time systems. All tests and experiments were on **NVIDIA Tegra X2 (TX2)** embedded system on a chip (SoC).

Chapter 2

Background

2.1 PREM

A predictable execution model (PREM), which splits execution into a sequence of memory and compute phases, and schedules them such that only a single core executes a memory phase at a time. The model is described in the article [PBB+11].

The PREM uses compiler. The compiler was developed to predict an optimal sequence for code execution. The compiler is written to split code in C and C++ into different intervals and execute it. the split code has 3 phases: prefetch, compute, and write-back. The compiler tries to split the program that when one core brings data from DRAM to cache, and other cores wait or do theirs computes.

One of the ideas of this work is to check if the different solutions work. For a reason were developed tests. The tests I got from the work

2.1.1 Tests description

The set of tests can simulate a real situation. Matrices multiplications are a usual part of Neural Networks, binary search is a common algorithm when working with trees, as is popular FFT. For the reasons were choose the tests.

The basic model of the tests may be divided into three algorithms:

- general matrix multiplication (GEMM)
- fast Fourier transform (FFT)
- binary search

The three algorithms can execute parallel. According to the PREM approach, the 3 algorithms were divided into jobs $I = \{I_1, ..., I_{16}\}$.

For GEMM:

1. I_1 part of matrices transposition
2. I_2, I_3, I_4, I_5 contains matrices multiplication

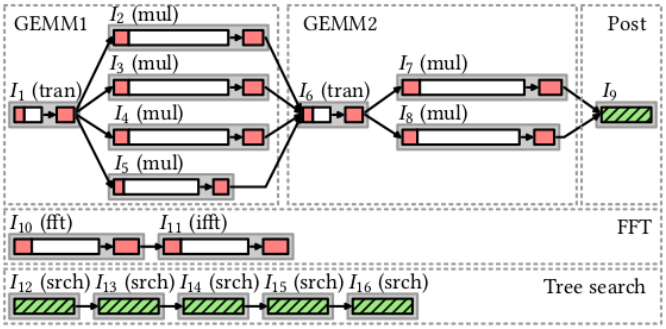

Figure 2.1: Example of the tests. The picture is taken from [MFS+18] and [MFS+19]

3. I_6 is matrices transposition
4. I_7, I_8 - multiplication
5. I_9 output results

For FFT:

1. I_{10}, I_{11} fast Furier transform
2. inverse fast Furier transform (iFFT)

For binary search

1. I_{12} just binary search
2. I_{13} just binary search
3. I_{14} just binary search
4. I_{15} just binary search
5. I_{16} just binary search

The programs for testing execute parallel every I_i in the list if they are in the same numbered line (for example, from the FFT task I_2, I_3, I_4, I_5 may be executed in parallel, or, in binary search no job which the code executes in parallel). The example of the tests in **Figure 2.1**

As a result of the PREM was to create an optimal sequence for the job's execution. For creating the chain was used integer linear programming (ILP).

2.1.2 The actual state of the tests

There are 3 actual test versions:

- **ILP** (file *worker-ilp.c* contains jobs in predicted sequence, as described above. A function pushes the jobs into a special list. Every core in CPU gets from the list a job that handles it. The function handles the job in

3 steps: prefetch (loading data in cache), compute (GEMM, or FFT, or binary search), and write-back (write the results in DRAM). The result of the program is time in milliseconds.

- **Mutex** (file *worker-mutex.c* contains jobs in any sequence. All jobs pushed into the list. Every core gets a job from the list and calculates it. The mutex solution also has prefetch, compute, and write-back phases. For avoiding interference, the tests use mutexes. When any core reaches the stage prefetch or write-back, before it blocks the part of code by the mutex. After prefetching or write-back, it unblocks. The solution hasn't an optimal timetable for the stages (prefetch, compute, and write-back), unlike the ILP solution. That means the first core will have gotten any jobs. Others should wait for it.
- **Legacy** (file *worker-legacy.c* contains the naive realisation. A function pushes all jobs into the list. In execution time, a function gets a random job from the list and compute it. Also, in the test aren't prefetch, compute, and write-back phases, only calculation.

2.2 Hypervisor Jailhouse

Here is described the concept of hypervisor and specifically is written why was used the Jaihlouse.

2.2.1 Hypervisor

Hypervisor [JAI] is computer software that allows us to execute a few virtual machines (VM) on the same hardware or share different parts of any equipment between VMs. That means if hypervisor starts a few VMs at once, the systems are in isolation. They don't know that they share the hardware. In hypervisors exit "bridges" which allows to transmit an information between VMs executed on it. Also, it allow to emulate any device in software.

2.2.2 Jailhouse hypervisor

Jailhouse is a hypervisor of Siemens company. It allows us to execute on the system programs written for Linux [JH3]. It configures CPU and different parts of a device in "cells" [Sin15]. Jailhouse doesn't have features for over-commitment ad CPU and RAM, and it doesn't have scheduling for processes and threads. It targets only on performance.

2.3 Memguard

Exists algorithm memguard [HY] which may optimize access to DRAM memory and solve the problem written above. The main idea is for a multi-core system to allow every core access to DRAM memory only exact limit

time or limit count of accesses. If a core does overhead execution than it would be turn in low power mode or stopped. In this case, the core cannot work as it was before and hasn't the same access to the DRAM. What gives other cores an opportunity to work with DRAM.

2.3.1 CTU memguard

CTU's memguard [JM] (written by Michal Sojka, Joel Matějka, Přemysl Houdek).

The algorithm is integrated into Jailhouse kernel and works on every core where we set it. The memguard has two budgets: time budget and memory budget. Every budget is a limit for a core. Memory budget means how much a core can make requests to DRAM memory (the request happens when a core gets miss in the cache memory). Time memory means how long the core has access to DRAM memory. If any budget will be empty, or in other words, a program will overrun the count of accesses to DRAM or overrun the time limit, than the core will turn to low power mode, or will be blocked. Also, memguard has 2 different options. In one option interruptions of CPU are allowed in second no. In case allowed interruptions when the Jailhouse has handle something, it can interrupt the execution of any program on a core. After any time, renew the execution. Another mode is banned interrupts. In this case, the program cannot be interrupted by the Jailhouse.

2.3.2 Memguard in code

To use the memguard is needed to set it on a core. For it using systemcall which has no. 793 with the next parameters:

1. budget_time – time how long can a core access to DRAM in microseconds (if set 0 monitoring is switched off)
2. budget_memory – count of accesses any core to DRAM, another word – counts of cache misses (if set 0 then memory access monitoring is switched off)
3. flag – special option which set ups memguard behavior
   - **MGF_PERIODCI:** when set, the memguard's timer is set to expire periodically every budget_time. Overrunning memory budget causes the CPU to block (enter low-power state) until the next expiration of
   - **MGF_MASK_INT:** when set, memguard disables interruptions that can be disabled and are not needed for proper memguard functionality. This is to ensure (almost) non-preemptive execution of PREM predictable intervals that are required to reduce the number of unpredictable cache misses.

The hypervisor gets a hypercall with the code 793 and handles the event with calling the function memguard_call.

```
long memguard_call(unsigned long budget_time,
unsigned long budget_memory, unsigned long flags);
```

Also, the function collects statistics (cache misses and time execution) from the previous call of memguard. All information pushes into structure memguard:

```
struct memguard {
    unsigned long start_time;
    unsigned long last_time;
    unsigned long pmu_evt_cnt;
    unsigned long budget_time;
    unsigned long budget_memory;
    unsigned long flags;
    bool memory_overrun;
    bool time_overrun;
    volatile u8 block;
};
```

where:

- **start_time** - time of first call memguard
- **last_time** - time of second call memguard
- **pmu_evt_cnt** - count of cache misses
- **budget_time** - time which we set it before
- **budget_memory** - memory which we set before
- **flags** - flag which defines mode of our memguard
- **memory_overrun** - info about if the budget memory was overran
- **time_overrun** - info about if the budget time was overran
- **block** - if we want to block a core

According to the parameters gotten in arguments, set ups new budget time, budget memory, and mode (MGF_PERIODIC, MGF_MASK_INT, or both). The result of the previous work memguard will return in *unsigned long variable retval*, where bit by bit is store information.

. Unsigned long varable has 64 bit (enumeration from zero, first it is 0):

- 0-31 bits contain a sum of memory events
- 32-55 bits contain the total time of performance
- 61 bit contains information if the set memory budget was overran

- 62 bit contains information if happened timeout (overrun time budget)
- 63 bit contains information if something was wrong

The code bellow switch on pmu count (count of cache misses) and timer if it is necessary:

```
if (budget_memory > 0)
    memguard_pmu_count_enable();
if (budget_time > 0)
    memguard_timer_enable();
```

At the end returns value retval.

PMEVCNTR5_EL0 RW Performance Monitor Event Count Register 5. Function
static inline void memguard_pmu_set_budget(u32 budget) is called from memguard_call and sets into the register (UINT32_MAX – budget) for generating interrupt after given certain number of events. The interrupt will be handled by function irqchip_handle_irq in a file irqchip.c

After, the function calls memguard_block_if_needed which to do a block of any core execution loop while if it was set to flag MGF_PERIODIC.

Shortly repeat the main stages of memguard:

1. set ups budgets of memory and time, it sets flag (MGF_PERIODIC or MGF_MASK_INT, may be set both) **memguard_call ??**.
2. enabling interrupts if set MGF_MASK_INT *memguard_mask_interrupts()*, otherwise disabling *memguard_unmask_interrupts()*.
3. Enabling pmu count if budget_memory more than 0 and timer if budget_time more than 0.
4. return statistics from previous calling memguard.
5. Waiting if any registered event happens (time overrun or memory overrun).
6. Handle certain function according with the event.
7. Block core if it is needed.

In step 7 may be blocked core if overrunning memory or time happened, and the core doesn't have access to DRAM memory.

Chapter 3

Memguard testing and evaluation

3.1 Initial state

It is the most important part of memguard. Here are 4 basic ideas:

- Does it work correctly? The main idea is to optimize any code in real-time. As was written above that, we cannot (or it is not easy) predict optimal execution of any task. And we should know if the memguard solution can help to decide the issue. For the reasons is needed tests. The tests must demonstrate that if we accelerated it.
- How to set up the memguard. In **CTU's memguard 2.3.2** section is description about two budgets (time budget and memory budget). And the budgets should be set reasonably. For this, experiments should be done to get the budgets.
- If we can improve it. It is also an important idea. One part of the work is research. According to the research should be answered the question about improving memguard. Or we will get new information about memguard's behavior. And we'll reach for any changes.
- Does it have errors or bugs? The tests should show us if we have any problem or bug in our memguard. It has less priority than 3 previous ideas but also an important part. Every software accompanied by different tests. But in our case, the tests are part of benchmark and knowledge-based we can check if the memguard has a logical problem or any bug.

3.1.1 Basic tests. Definition of the work

For testing, the memguard were chosen the same approach as described in the subsection **Test description 2.1.1**. In the same way here are 16 intervals (in future I'll call it jobs because it named according to code). Some of the jobs must be computed in a certain sequence as it is written at the page 4.

For a basis, I choose Legacy tests. Because there are no phases: prefetch, compute, and write-back (as it is in ILP and mutex solution). On the experiment is no influence from any optimal prediction. It gives more objective

to imagine how does it work. And I can compare the results gotten from memguard execution with the results from ILP, mutex, and legacy.

The results of the benchmarks are time in milliseconds. I measure the time from starting computing the jobs and at the end (when all jobs are computed). An exception is one more test which I wrote for testing every job execution (I explain it in the future of the work).

An additional, important part of the benchmarks is logical and optimal set up the memory and time budgets. I did different experiments and analyses with the budgets.

3.1.2 Hardware and software for the testing

For the benchmarks, I use **NVIDIA Tegra X2 (TX2)** embedded system on a chip (SoC). It provides high performance and power efficiency for execution on embedded systems software. Usually, the systems need to do many different calculations. And a similar computer TX2 is the right choice. The CPU of TX2 is AArch64 (ARM 64 architecture) [ARM].

The TX2 is connected to an internal local network. All manipulations such as starting, stopping, uploading I did by novaboot [NB]. The novaboo is using SSH protocol. Every booting the systems starts from uploading the Jailhouse hypervisor on the TX2. After, the hypervisor turns to boot state. It means that the TX2 has a specific configuration. The TX2 doesn't have persistent storage. If we reboot the system, all data will be deleted and will be nothing to start. We need again upload the Jailhouse. The approach is useful because it allows easier to manipulate it if we got any error is no problem to return back.

The SSH (Secured Shell) protocol is a network protocol for connection from any computer to another based on TCP. The SSH protocol allows remotely to execute different commands. And it is possible to work in an unsafe connection because the protocol uses encoding.

Also, it is possible to transfer files between machines connected by SSH. Here are a few opinions about why it is important:

- Jailhouse installation. By the SSH I install the hypervisor on the TX2.
- If I change the tests, I don't need to reload or restart the full system. I can upload the new version of the tests.
- Get data from the TX2. Since the testing is an important part of the work, I load the results of the benchmarks from the TX2 on my own computer.
- The TX2 hasn't any display, and the SSH is one possible possibility to interact with it.

Authorization

For connection via the SSH protocol is needed authorization. The protocol uses RSA encoding. According to the RSA I created two keys: a public key and private key. Public key was added in */root/.ssh/known_hosts*. Also, I added via *ssh-add* command my private key into automatic authorization. That means, when I use a ssh connection, I don't need to enter the password and user name. The system independently recognizes me and makes authorization.

Memguard in Jailhouse

The memguard is embedded in the Jailhouse **2.3**. And if we integrate any changes in the algorithm, we should recompile the full hypervisor. But it doesn't mean that every compilation phase will rebuild the full system. The compilation uses binary files *.o*, which contains data in binary form. Every *.o*, the file corresponds to any part of the code. If we change only memguard then it will be recompiled and updated only files concerning the memguard and integration part of the system. After that, the only linker connects the *.o* files and insert needed data.

Makefile

For the easier compilation were created makefiles. That is specific files which contain rules and steps for different compilation approaches. We don't need to enter every time the command for the compiler. We should predefine the rules, and the flags via the project would be built.

The typical commands is:

- **make** - to compile the project
- **make clean** - to delete all compiled files in the project
- **make run** - to start the program
- **make install** - to install the compiled program

Cross compiler

All changes with the memguard in stage of research I make on my local computer. Also, I carried out the compilation on my computer. For that approach, I use a cross compiler.

Cross compiler is a compiler capable of the system where is doing the compilation, but the output program is intended for another computer and architecture of CPU.

For the building, I use GCC compiler for arm64 architecture.

OS for development [Tan15] The project I did on Fedora 31. It is open-source distributed Linux from Red Hat company. I install on it cross compiler. And I make build for the ARM architecture.

Preprocessor The specific stage of compilation. It is important to explain because I use it in by implementation in the future.

The preprocessor is working after syntactic and lexical analyzer of a compiler. Every time when the compiler finds a symbol #, then it is a signal for preprocessing. The preprocessor replaces the insertion with according definition. It is useful because we can write once code and compile it with different flags and get different binary files.

Static library That is the library in compiled form. The usual name of the libraries ends with *.a*. That useful because it isn't needed recompile every time it. We can just insert it in our code. It is faster for big compilation.

Version system control (VSC) I use for working with different changes that I inserted in the code and revert if it needed to previous version. It allows us to avoid many errors and to see all changes integrated into the code. In this project was used GIT as VSC.

3.1.3 Structure of the benchmark in code

As was described before, the tests are divided into special jobs.

All files of the benchmarks are in *src* directory.

src contains the next subdirectories:

- *build* contains built files
- *lib* contains static libraries for the project
- *scenarios* contains the different configurations for the jobs.

Every job has the next structure:

```
typedef struct prem_function{
    void (*func)();
    int from;
    int to;
    uint32_t type;
    char name[16];
    int successors[8];
    int successor_quantity;
    volatile int predecesor_done;
```

```
    int predecesor_quantity;
    volatile uint32_t tsc[6];
    volatile uint64_t time[6];
    volatile uint32_t miss[6];
    volatile uint32_t core_id;
    uint32_t parallel_id;
    volatile uint32_t order_prefetch;
    volatile uint32_t order_writeback;
} prem_function;
```

The important parts of the code:

- **void (*func)()** - address of the function which is responsible for computing
- **successors** - the jobs which could be handled after the job ended. It corresponds the sequence described in subsection **Tests description 2.1.1**
- **order_prefetch** and **order_writeback** - it is predicted time according with the PREM approach (when the test will be prefetched - order_prefetch, and when the test will be written-back - order_writeback)
- **time[6]** - information how long the job worked
- **miss[6]** - how much misses were in the core of CPU when the job worked

I added into the structure the next lines:

```
#if defined(PREM_TARGET_MEMGUARD) || \
    defined(PREM_TARGET_JOBS_STATISTIC_MEMGUARD)
    volatile uint32_t budget_memory_memguard;
    volatile uint32_t budget_time_memguard;
    volatile uint64_t memguard_flag;
#endif
```

Here are two flags for different put builds. That means, in binary files, jobs will contain additional properties for the memguard.

- **budget_memory_memguard** - configuration for memory budget in memguard
- **budget_time_memguard** - configuration for time budget in memguard
- **memguard_flag** - MGF_PERIODCI or MGF_MASK_INT

In the *common* directory are files:

- *worker-ilp.c* tests configured with the ILP
- *worker-legacy.c* basic legacy tests

- *worker-mutex.c* tests configure with the mutex
- *worker-memguard.c* new added by me tests for memguard.

For the tests was set particular affinities by using function **CPU_SET**. That means, if we execute any thread on any core, the thread cannot migrate between the CPU's cores. If we don't do it, then certain thread with a job can migrate on other core with a different configuration for the memguard. And it won't be correct. Also, in the testing, I had a problem with the freezing of the core. When I didn't set flag *MGF_MASK_INT*, the interrupts in the CPU were banned. And if one core was blocked that any budget of memguard was empty. The task ended on another core due to migration thread between cores. In this case, the core is dead. Because it waits to end the task. And if it happens on all cores, the all system is frozen. I had to restart the whole system.

The file *main.c* contains few important moments.

```
#define PREM_CORES 4
```

Here is predefined the count of cores, which will be used for the computing. By default is set 4 because the TX2 has 4 core with the same architecture and frequency. The other 2 cores have a different rate. If we allowed all cores to work with the jobs, it wouldn't be a clear experiment. Some cores which are more powerful could get a job and execute faster. And the every time execution, the tests won't be exact. And, by default in Jailhouse configuration is set only 4 cores as active: CPU_0, CPU_3, CPU_4, CPU_5.

In code the CPUs are set in file *worker.c*.

```
const int affinity[4] = {0, 3, 4, 5};
```

The function **CPU_SET** uses the array **affinity** for setting affinities as it I described above, for banning the thread's migration between the cores.

The next function initializes tests. They are call according with compilation flags.

```
#if defined(PREM_TARGET_MUTEX)
    init_default_mutex();
#elif defined(PREM_TARGET_ILP)
    init_default_prem();
#elif defined(PREM_TARGET_LEGACY)
    init_default_legacy();
#elif defined(PREM_TARGET_MEMGUARD)
    init_default_memguard();
#endif
```

As a result of the compilation will be 4 binaries for ILP, mutex, legacy. And memguard, respectively. But outgoing binaries will be more because the tests have more variety of configurations.

In the initial functions setups the affinities creates 3 threads for testing. The last 4th thread is starting from the main function.

All jobs must be pushed in the queue. And for the act responds the code:

```
#if defined(PREM_TARGET_MUTEX)
    insert_mutex_tasks();
#elif defined(PREM_TARGET_LEGACY)
    insert_legacy_tasks();
#elif defined(PREM_TARGET_ILP)
    insert_ilp_tasks();
#elif defined(PREM_TARGET_MEMGUARD)
    insert_memguard_tasks();
#endif
```

Now I need to start the tests. The results of the tests is time. So I calculate the time before the starting, and after that, the tests end:

```
clock_gettime(CLOCK_MONOTONIC, &time_start);
```

- **time_start** - structure which has two fields: time in seconds and time in nanoseconds from the Epoch
- **CLOCK_MONOTONIC** - the flag ensures that there no jumps in system time

3.1.4 Memguard worker

The file *worker-memguard.c* was created by me for testing memguard. The basic part was copied from lagacy code. For easier call and using memguard here was function:

```
static __attribute__((always_inline))
    inline long memguard(unsigned long timeout, unsigned long
    memory_budget,unsigned long flags)
{
   return syscall(SYS_memguard, timeout, memory_budget, flags);
}
```

where parameter **SYS_memguard** is predefined 793 number code for hypercall.

Additionally, in the targets of the work is finding optimal values for the memguard's budgets I use the arguments **argv** in function **main(int argc, char *argv[])** for sending a quantity from 100% of the budget time and budget memory with which the tests will execute. The approach to how I got data for the default configuration of memguard is described bellow. I added the quantity for finding optimal values. I executed with the quantities: *0.1, 0.2, ..., 2.1.* And, for the reasons I added additional parts for memguard in structure:

```
typedef struct prem_resource{
    pthread_t thread;
    int core;
```

```
    uint32_t worker_done;
#if defined(PREM_TARGET_MEMGUARD) || \
    defined(PREM_TARGET_JOBS_STATISTIC_MEMGUARD)
    float time_memguard;
    float memory_memguard;
#endif
} prem_resource;
```

Because every job has its own values for time and memory budgets, I multiply the time budget for memguard by **time_memguard** from **struct prem_resource** and the memory budget for memguard by memory_memguard.

From the memguard's description is known that to set off it is needed to call with parameters **memguard(0, 0, 0)**. For the reasons I execute with normal parameters the memguard before the job execution, and with the 0 parameters after:

```
memguard(current_job->budget_time_memguard,
    current_job->budget_memory_memguard,
    current_job->memguard_flag);
    (*current_job->func)(current_job);
    memguard(0, 0, 0);
```

The last call with 0 should switch off the memguard. Also, we can get a statistics about cache misses, time execution as it is written in section **Memguard 2.3**.

In addition. The code I got in a state that memguard was set in ILP and mutex. Here was memguard using in prefetch, compute, and write-back phases. I deleted that and reversed to previous logic of the benchmark. Now, the memguard is using only for the **memguard-worker.c**. Also, in the code were data outputs in an inconvenient form. The outputs are using for debugging. Because on the system no easy way to debug. I rewrite it into the:

```
#define printf_message(fmt, ...) \
    do { if (TO_PRINT) printf(fmt, ##__VA_ARGS__ ); } while(0)
```

The style is using preprocessor is better. For using this we should push to compiler *-DTO_PRINT* flags.

```
gcc -DTO_PRINT ...
```

- **-DTO_PRINT** the flags set value 1 for the preprocessor and then the compiler in this case will understand it as truth condition

If we will not use the two flags, the our code will work as don't have any output for debug.

Chapter 4

Results

I this part of the work, I will describe how I tested the memguard. I was working with worker memguard (file *worker-memguard.c*). And after that, I compare it with other workers: ILP, Mutex, Legacy.

4.1 Configuration memguard

In theory, I described the memguard, its logic, and the main idea. Now is an important part to check if our assumption about the algorithm will be working. Firstly we need to set any optimal or just reasonable budgets for time and memory. In the first experiment, I set high enough values (maximal for type **unsigned int**) for memguard budgets. The reason is to get a statistic of how usual every job accesses to DRAM and how long it is going. I have got the next results. The worker was executed on 4 cores.

	Memguard stadard budgets	
Jobs id	Cache misses	Time execution
0	3539	189
1	3747	9104
2	4165	9093
3	4203	9099
4	2426	2486
5	1967	98
6	3232	9308
7	3501	7459
8	440	70
9	4332	4156
10	4316	4145
11	4198	4116
12	4153	4110
13	4837	873
s 14	3970	821
15	3915	812

4.2 Results of the tests

Now I will assume that is as quantity 100% for my future tests. I don't need to know precisely optimal values. Because it is the research part, and the values are at the start point. I will check it with quantity *10%, 20%, ..., 210%*.

4.2.1 Tests with different quantities

At first let's see on the graph **Figure 4.1**.

In this case, I set the memory budget to constant from the table to 100%. And on the axis X, I change only time budget. I set the flag for memguard to **MGF_PERIODIC**. In this case, when any budget (time or memory) will be empty after the time which I set for the time budget all the budgets will be recovered. On the axis Y I placed time in milliseconds. Here is showed time of execution all jobs. The benchmarks I executed 100 time for every quantity (for 10%, ..., 210%). Then I used boxplot from MATLAB to draw the boxes. The center red line indicates a median, the top, and bottom of the box indicate 25 and 75 percentiles. The '+' indicates outliers. And the top and bottom dashes indicate minimal and maximal values.

From the graph, we can see that the time execution grows. It means that if any budget becomes empty, then with the periodic flag, the memguard stops any core for a time, which is set in time budget. The higher the value of the time budget, the longer core is stopped and is in less power state. Hence the whole test executes slower. At any moment, the values of time budgets may be big enough that execution time goes down.

After that, I did the same experiment, but, I set the time budget to a constant value as it was before to 100% from the table. The results if graph **Figure 4.2**

Here the graph goes down. If to set enough small budget for memory (in our case it 10%). Then the memguard will faster use up the budget. And always will be called block for the core as it can be seen that from 150% is no big changes. Here it is different only median (red line). But the top and bottom borders are almost the same. That means if we set very big values for the budget memory than interrupts for stopping the core will not happen. And the CPU will works as we don't call the memguard.

The graphs **Figure 4.3** and **Figure 4.4** indicates the same as previous experiment but executing with the banned interrupts (flags **MGF_PERIODIC** and **MGF_MASK_INT** for the memguard). In this case, we can see that some tests execute faster than on previous graphs. And that confirms the working of banning interrupts and unbanning. The core is not interrupted until it is doing any job from the benchmarks.

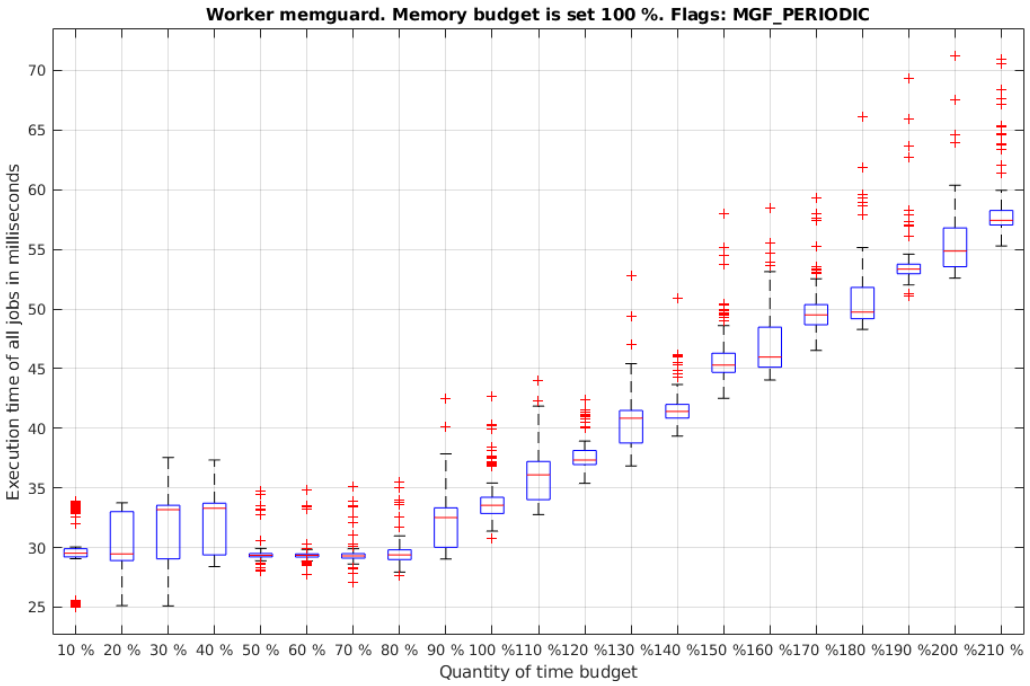

Figure 4.1: Testing memguard's time budget and flags MGF_PERIODIC

4.3 Comparing with PREM

As we can see on the graph **Figure 4.5**, here are results of 100 times execution the tests from PREM(ILP worker and Mutex worker), Memguard worker, and Legacy worker. For comparing the tests, I set the quantity for memguard based on previous experiments: 50% for time budget and 90% for memory budget. Because the budgets are reasonable enough as it is seen in graphs. For flag, I used the only **MGF_PERIODIC**. Because in case if the interruptions are allowed (flag **MGF_MASK_INT**) the tests are working slower. The Legacy tests show that sometimes it can be more optimal than ILP or mutex solution. But the top border also is so high. ILP solution is better than Mutex, because it has precomputed time for fetching data from DRAM to cache. And here are no intersections in the using bus between the cores. But, is it see that memguard is not such effective as other workers. But the top and the bottom borders are nearly to themselves. The memguard worker is more stable than Legacy worker. In memguard worker are no jumps in time execution. And the method allows more evenly to distribute a using the cores.

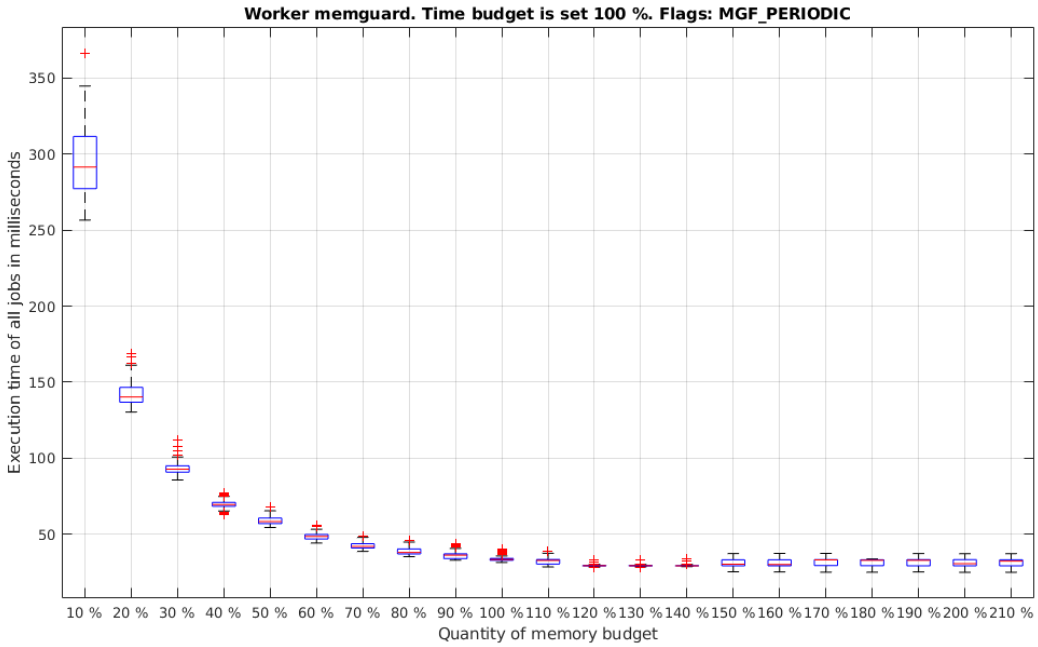

Figure 4.2: Testing memguard's memory budget and flags MGF_PERIODIC

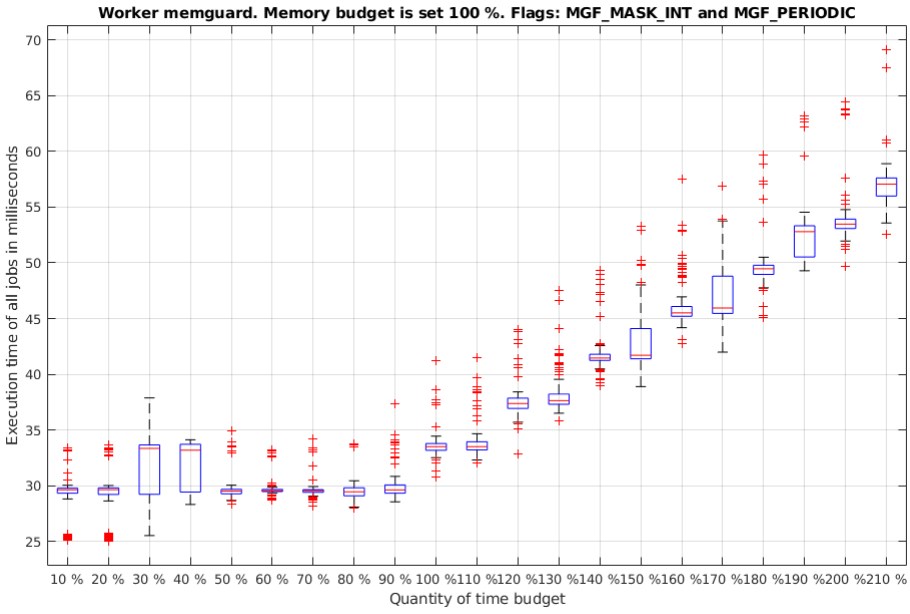

Figure 4.3: Testing memguard's time budget and flags MGF_PERIODIC and MGF_MASK_INT

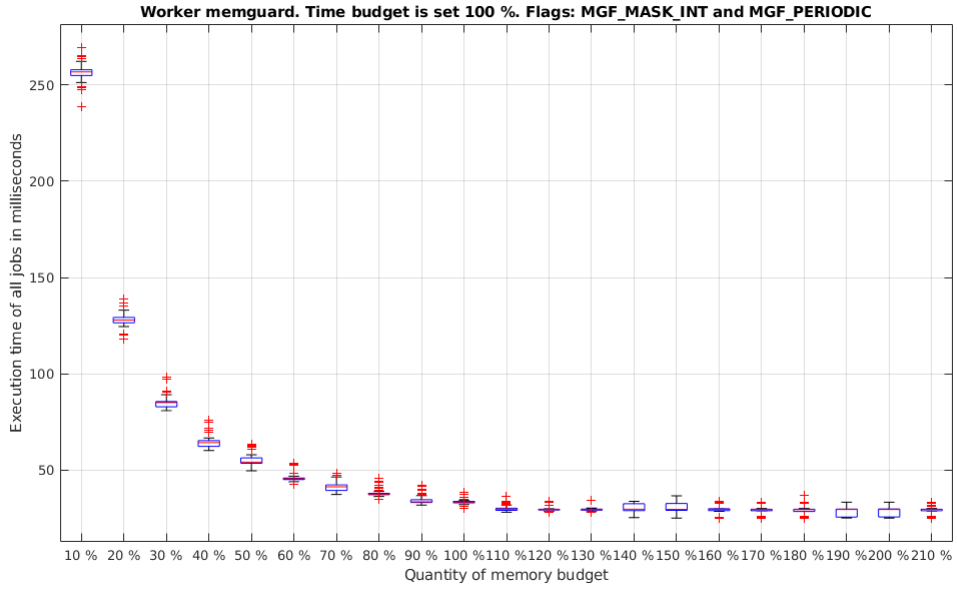

Figure 4.4: Testing memguard's memory budget and flags MGF_PERIODIC and MGF_MASK_INT

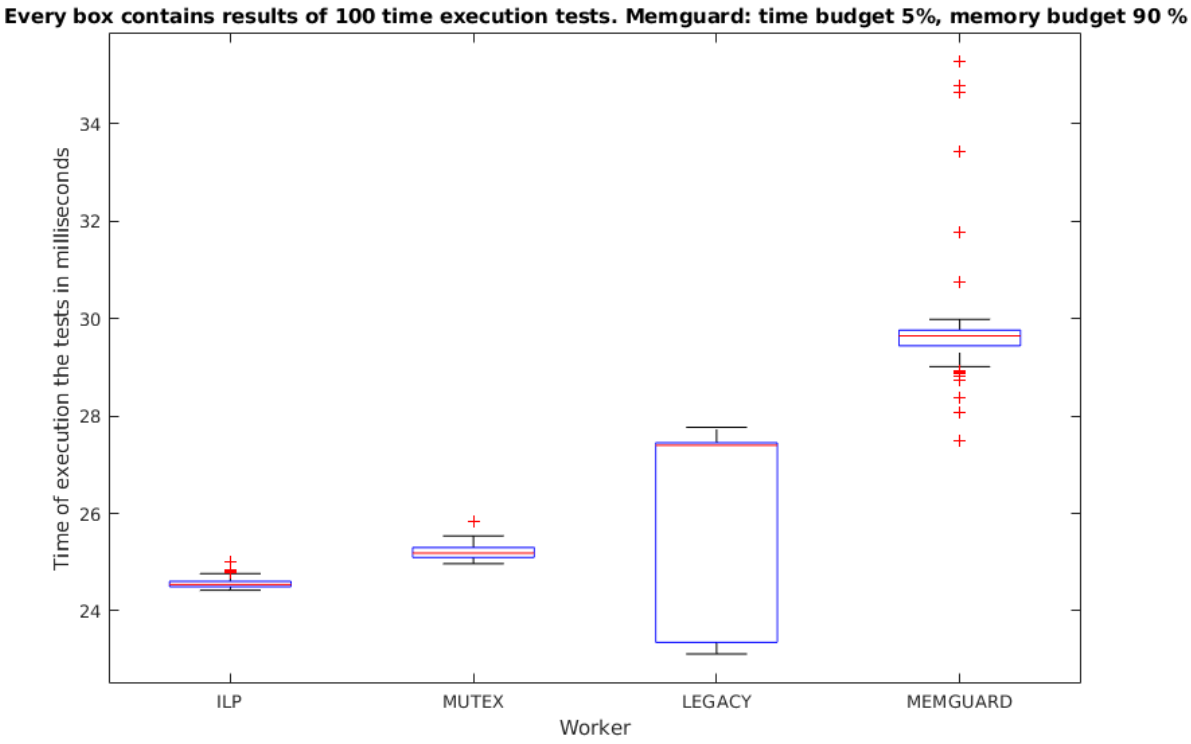

Figure 4.5: 100 time execution the PREM tests, Leagacy and Memguard. Axis x has a name of the test. Axis y has time of the execution in milliseconds.

Chapter 5

Conclusion

In this work, I figured out how works the algorithm memguard and explained the mechanism of the algorithm developed in CTU. And one of the main ideas of the work was to test the algorithm and to get any measures. As a result, I had to say if it works and I had compared with another exists method from refusing the inter-core interference. Basic tests PREM were developed before I started to work. And I spent time on research and understanding the logic of the tests and how they work. I wrote the tests based on PREM for memguard. In this work, I get to know technologies and additionally I explained any things which could be misunderstood for readers: basics architecture of CPU, caches and cores, bus, any tools in OS and hypervisors, why we use the Jailhouse, main things in compilers and why I choose the flags or predefined functions (preprocessing).

After that, I developed the new tests for memguard, which are based on PREM tests. I changed any function, deleted unused code.

I tested the memguard with different budgets and flags, compared it with yet existed results from the PREM and Legacy workers, and explained the behavior of the algorithm. For comparing with the Legacy worker I set parameters for memguard: 50% for the time budget and 90% for the memory budget. In the comparing memguard with the Legacy I established that sometimes the memguard is more stable than just Legacy worker. The memguard worker hasn't jumps in time execution like the Legacy. Of course, any experiments show that memguard may execute any program slower, but on that impact the parameters, which I tested. Also, at any time, it may be slower due to blocking. Also, The work shows that memguard is a working algorithm and can optimize any program's execution in embedded systems.

Appendix A

Bibliography

[ARM] *Arm64 documentation: http://infocenter.arm.com/help/index.jsp.*

[HY] Rodolfo Pellizzoni? Marco Caccamo‡ Lui Sha‡ Heechul Yun‡, Gang Yao‡, *Memguard: Memory bandwidth reservation-system for efficient performance isolation inmulti-core platforms*, University of Illinois at Urbana-Champaign, USA.heechul,gangyao,mcaccamo,lrs@illinois.edu?University of Waterloo, Canada. rpellizz@uwaterloo.ca.

[JAI] *Siemens jailhouse hypervisor https://github.com/siemens/jailhouse.*

[JH3] *Understanding the jailhouse hypervisor, part 1: https://lwn.net/articles/578295/.*

[JM] *Jailhouse with memguard. https://gitlab.fel.cvut.cz/sojkam1/jailhouse-with-memguard.*

[MFS+18] 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*, Proceedings of the 9th International Workshop on Programming Models and Applications for Multicores and Manycores - PMAM'18 (Vienna, Austria), ACM Press, 2018, pp. 11–20 (en).

[MFS+19] Joel Matějka, Björn Forsberg, Michal Sojka, Přemysl Šůcha, Luca Benini, Andrea Marongiu, and Zdeněk Hanzálek, *Combining PREM compilation and static scheduling for high-performance and predictable MPSoC execution*, Parallel Computing **85** (2019), 27–44 (en).

[NB] *novaboot - boots a locally compiled operating system on a remote target or in qemu: https://github.com/wentasah/novaboot.*

[PBB+11] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley, *A Predictable Execution Model for COTS-Based Embedded Systems*, 2011 17th

IEEE Real-Time and Embedded Technology and Applications Symposium, April 2011, ISSN: 1545-3421, pp. 269–279.

[Sin15] Valentine Sinitsyn, *Jailhouse: https://www.linuxjournal.com/content/jailhouse*, June 2015.

[Tan15] Andrew S. Tanenbaum, *Modern operating systems*, fourth edition ed., Pearson, Boston, 2015.