SECURE SEPARATION OF SHARED CACHES IN AMP-BASED MIXED CRITICALITY SYSTEMS

P. Schnarz, C. Fischer, J. Wietzke*, I. Stengel†

* *Faculty of Computer Science, University of Applied Sciences Darmstadt, Germany, E-mail: pierre.schnarz@h-da.de*

† *Centre for Security, Communications and Network Research Plymouth, United Kingdom, E-mail: ingo.stengel@plymouth.ac.uk*

Abstract: The secure separation of functionality is one of the key requirements particularly in mixed criticality systems (MCS). Well-known security models as the multiple independent levels of security (MILS) aim to formalise the isolation of compartments to avoid interference and make them reliable to work in safety critical environments. Especially for in-car multimedia systems, also known as In-Vehicle Infotainment (IVI) systems, the composition of compartments onto a system-on-chip (SoC) offers a wide variety of advantages in embedded system development. The development of such systems implies often the combination of pre-qualified hardware- and software components. These components are CPU subsystems and operating systems, for example. However, the required strict separation can suffer due to the pre-qualified and therefore not reconfigurable hardware components. Particularly, this is true for shared cache levels in CPU subsystems. The phenomena of interference in the concurrent usage of shared last-level caches, are exploitable by adversaries. Therefore, this article identifies the attack surface and proposes a mitigation to prevent from the intentional misuse of the fixed cache association. Generally, the solution is based on a suitable mapping scheme in the intermediate address space of an asymmetric multiprocessing environment which implements the MCS. Furthermore, we evaluate the strength of the approach and show how the solution contributes to a separation property conformal system.

Key words: security architecture, shared cache, denial of service, proof of separability

1. INTRODUCTION

Partitioning and combining different software components into a shared platform is an ongoing trend in embedded system development. These partitions provide the opportunity to separate applications having different functional or non-functional requirements. Among other purposes this fosters the development process or certification needs. Particularly non-functional requirements as e.g. for safety or security are mandatory in critical environments like avionics or automotive. Those software compositions are referred to as Mixed Critical Systems (MCS). Especially in the automotive sector the safe and secure composition of MCS is pushed due to the rising number of functions of in-car multimedia systems, also known as In-Vehicle Infotainment (IVI) systems. From a technical view, there are several possibilities to partition or separate applications. The focus in this work is set to the combination of multiple operating systems on a common System-on-Chip (SoC) for automotive purposes. Multi-operating systems (multi-OS) are going to be established in future [1].

The development of systems and software in the automotive environment implies special requirements and challenges [2]. In particular, one of the challenges is to integrate the wide-spread functions onto one single head unit [3]. Furthermore, tightly network-coupled (cloud) applications, such as social networks will gain increased entry into that environment. The utilization of multiple OSs on one platform provides the opportunity of gaining advantage of their capabilities. An example is the execution of a real-time OS parallel to a mobile OS or a general purpose OS. The loosely-coupled component structure of SoCs offers the possibility to implement a multi-OS following the asynchronous multiprocessing (AMP) paradigm rather than following the classical virtualization schemes implemented in commodity desktop architectures. The AMP paradigm implies the total split of every resource in the system. Specialized hardware extensions introduced with current RISC processor architectures such as the ARMv7 [4] enable the assignment of devices and resources of the SoC to single OSs. The asymmetric paradigm is not yet applicable to every part of current system-on-chip (SoC) architectures. Caches are often shared between multiple processor cores on multi processor SoCs (MP-SoC). According to their fixed association-scheme to the main memory, code running on the processor cores is naturally vulnerable to DoS attacks. Generally spoken, due to the exploitation of the caching infrastructure an starvation of an OS-partition is possible. Adversaries could explicitly aim for this weakness.

Hence, in this work we introduce and provide an empirical analysis of the separation capabilities of a indirect memory mapping method. The proposed memory mapping technique aims to mitigate the surface of interference on shared caches.

The latency for each memory access of a certain processor has been chosen to quantify the effects of our proposed

Based on: "On a Domain Block Based Mechanism to Mitigate DoS Attacks on Shared Caches in Asymmetric Multiprocessing Multi Operating Systems", by P Schnarz, C Fischer, J Wietzke and I Stengel which appeared in the Proceedings of Information Security South African (ISSA) 2014, Johannesburg, 13 & 14 August 2014.

method. Because this value represents the time consumed by the cache subsystem to transfer the issued data. It can be assumed that a heavy usage of a shared cache will increase the access latency of a co-CPU on average. Therefore, in order to evaluate the separation capability, we applied measurements to quantify the latencies. These measurements are executed using certain memory access patterns which are supposed to provoke, intended interference or starvation effects. This pattern is derived from the specific cache implementation of our experimental platform.
The outcome of this measurement-tooling is twofold. First, it is possible to show that the issue of denial-of-service attacks in such environments is true, as well as the starvation. Second, the output of the data shows the effectiveness of our solution to this particular problem. A detailed description of the quantitative methodology in the particular setup is given in Section 6.

Additionally, this article maps the novel mapping method to security models which are commonly applied for mixed criticality systems.

1.1 Threat Analysis

The security threat discussed in this work is specific to this particular environment. Thus, this affects the consideration of threat sources, actors and the foci of interests/assets. Since this article focuses on the denial of service attacks we only consider the circumstances that attack the availability of the asset.

Threat Sources/Actors Commonly, threat sources and actors are distinguishable. Briefly, the source of an threat is not necessarily the actor or the actual attacker in the end. What is more important are the capabilities and the motivation of the actors. Therefore, in this case, attackers are assumed to be very *knowledgeable insiders*. This ends in a relative high capability considering the substantial skills in embedded engineering. Furthermore, we assume actors have access to the non-disclosure documentation of the hardware platform. As an example, the motivation to attack the availability could be to prevent important safety messages from being displayed. This would result in the safety relevant partitions of the system being threatened.

Threat Vector In this work we consider availability attacks at system-level. This means we assume an attacker has already succeeding in compromising and controlling an OS-domain. This is reasonable due to the attack surface of highly Internet-coupled mobile-OSs.

Focus of Interest As a result, the actors aim for vulnerabilities at system-level. Despite privilege escalation attacks (horizontal or vertical) this article focuses on DoS-attack-surfaces in the shared last level cache (LLC). The attacker's aim is to overcommit the cache from its compromised OS side in order to degrade the memory access performance on the target's side. According to the cache associativity, the attacker is able to aim for a memory access to a specific PA in the system. As an example, it is feasible for adversaries to aim at a co-OS-domain which computes the cluster device (speedometer) for the driver of a vehicle. The target would need to fetch data from the memory in a strict timing order. If the memory access is delayed by the attacker, the displaying of the data could also be significantly delayed as well. This has an impact on the reliability and availability of the target system.

1.2 Related Work

A common known phenomena dealing with caches in multi-core processors is *cache thrashing*. This work aims generally to the intentional interference of the cache system.
Similar approaches to implement multi-OS are shown in [5], [6], [7] and [8]. However, the approaches focus on *IA-32* multicore architectures which are used for desktop computers. The difference to the architecture proposed in this work lies in the design paradigm, protocols and hardware compilation, which can not simply be transferred to SoC architectures. Nevertheless, in [9] Crespo et al. shows an hypervisor meeting the requirements of high-assurance mixed critical systems. The intended solution to manage the cache allocation indirectly is introduced in [10], [11] and [12]. The authors propose the solution of coloring caches in order to avoid interference between applications. They also deal with the problem of the dynamic allocation and assignment of cache colors to applications. The solutions are implemented into the memory allocation mechanism on OS level.

In [13] and [14] the authors claim approaches for the prevention of starvation in multinode systems or cache coherence protocols. Those approaches might be applicable in AMP based systems. However, they have to be implemented into the hardware platform, which is not in scope for the consideration in this work. Not in every embedded systems development project it is even possible to introduce changes in pre-qualified hardware blocks, like the CPU subsystems are.

The approach proposed in this article differs substantially from this related work. In the case of AMP-based MCS, the memory segmentation must be enforced on system level to have the capacity of configuration protection. Furthermore, since the system setup is statically defined, no dynamic allocation page coloring algorithms can be implemented.

1.3 Structure

In Section 5. a method is proposed which addresses the possible surface for interference. Furthermore, implementation issues for the solution are given in Section 5.2. To verify the concept, a quantitative measurement method is introduced in Section 4.. The impact is shown with an experimental setup examined in Section 6.. All measurement results are given in Section 7.. The remainder of this article is organized as follows: In Section

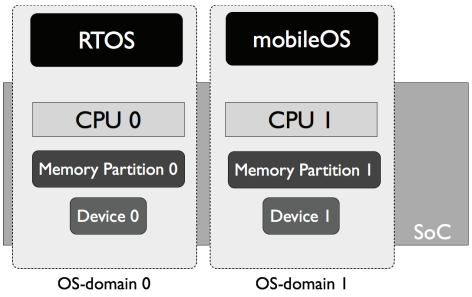

Figure 1: Multi-OS composition and resource assignment

2. the Multi-OS environment and it's practical realization is introduced. In Section 3. the organization of the memory is examined. Lastly, a conclusion is presented in 8..

2. AMP BASED MIXED CRITICALITY SYSTEMS

As previously mentioned, many ways exist to construct a Mixed Criticality System. In this work we consider MCS separated into multiple instances of operating systems (OS). According to their capabilities, they can, but are not obliged to, be of different types. In this work, OSs are divided into three categories: real-time, general purpose and mobile. Each OS maintains its own memory as well as the physical devices provided by the hardware platform. The combination of OS, memory and devices builds an independent and self-organizing OS-domain. Figure 1 shows a semantic system overview.

The difference to other multi-OS approaches (compare the related work in 1.2) is that OS-domains are statically bound to a processing unit, which is usually one of the central processing units (CPU) or CPU-cores of the hardware platform. This is led by the intention of achieving a static system configuration. It is not intended to expand domains in the main memory or to reorganize the device assignments during runtime. During the boot phase all necessary initializations and configurations are set up. In comparison to classical virtualization examples, this approach avoids interfaces to manage the configuration once the system is running. The intention is to minimise the attack surface.

The proposed approach is based on asynchronous multi processing. AMP systems are not new in certain domains. Primarily an AMP system provides asymmetric access to hardware. The term asymmetry refers to the separation of hardware resources core-by-core in the system. Each core has access to and works on a different partition of the main-memory and hardware devices. To maintain and handle the different core partitions and their applications, each partition must run an OS. In this way it is also possible to run different OSs on an AMP multicore system. This approach for encapsulation is hardware-based and requires that fundamental hardware functions to be adapted or configured to run multiple OSs. The approach is contrary to symmetric multiprocessing (SMP) which runs a single OS on all CPU-cores and maintains all hardware resources. AMP based systems can be categorized as *mixed critical system* [15] because of the independency of all subsystems running on the multiple CPUs.

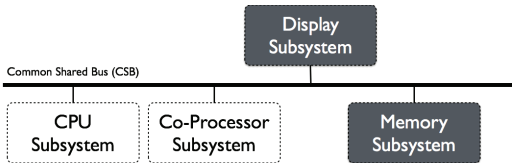

Figure 2: Generalized SoC-structure

2.1 System on Chip Structure

SoC is an integrated circuit that combines all components of a computing platform or other electronic systems into a single chip. Designing SoCs is a very demanding area in embedded system development. The platform will be constructed for their intended environment. The architecture of the system is generally tailored to its application rather than being general-purpose. This means there is usually no common structure for such platforms. Nevertheless, most SoCs are developed from pre-qualified hardware blocks (compare [16]). These blocks are connected through an on-chip network, often referred to as system-bus. In Figure 2 a generalized structure for SoCs is presented. For this work, the focus is set to a single subsystem which implements CPUs. Furthermore, the CPU-subsystem's connection to the main memory is considered.

2.2 Security Model

The here presented Multi-OS applies a security model which fits with the mixed criticality of the system. *Multiple Independent Levels of Security* (MILS) has been introduced to prove the security of high assurance systems especially for aircraft, space, and defence purposes [17] [18]. The model incorporates the following properties [19]:

- **Data isolation:** The isolation of data does not only affect the confidentionality of information, it is also aimed at the separation capabilities of the system design.
- **Control of information flow:** The location of the information has to be defined. Furthermore, access control has to be enforced.
- **Periods processing:** This aims for single core or classical virtualized systems which share one or more processors. In this case, it has to be proven that each system can meet its timing requirements.
- **Fault isolation:** Failures are detected, contained and recovered locally [19].

Security measures to enforce the aforementioned properties need to implement the following attributes:

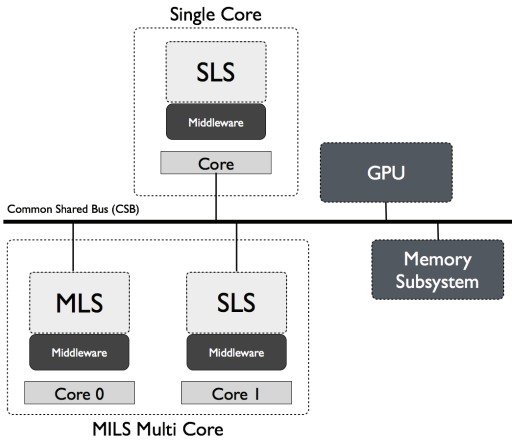

Figure 3: MILS AMP system overview

non-by-passable, evaluable, always invoked, tamper proof. These are commonly known as the acronym *NEAT* [20]. In Figure 3 the AMP system which is based on a SoC is mapped to the MILS model. Single subsystems are either defined as multi-level secure (MLS) or single level secure (SLS). Hence, each compartment has its own multiple security levels or they are considered to be single levelled. There is no dependency between the security levels of each compartment. The highest security level of system 1 is unequal to the highest level of system 2, etc.

One of the key elements of the MILS model in contrast to our work is the *data isolation* property. Sometimes referred to as *proof of separability* [21], this aims to assure a multi-compartment system by avoiding surfaces for interference. Therefore, the method we are going to introduce must fit with the *NEAT* attributes in order to be compliant with the security model.

2.3 AMP Multi-OS Realization

This section will examine issues associated with realizing an AMP-based multi-OS. In this work hardware architectures are considered which use memory maps for hardware accesses (memory mapped i/o). Each device connected to the common shared bus (CSB) is accessible by a statically defined physical address. These addresses are bundled in an i/o-address space or configuration address space if there are any configuration-registers of the device. Processors map those physical addresses to their virtual address space using a memory management unit (MMU). The MMU itself uses a translation table (TT) to match and redirect accesses to peripherals connected to the CSB. The translation table itself either resides somewhere in the physical memory space or is implemented into the MMU hardware. The whole system configuration is set during the boot phase within the privileged hypervisor mode. Figure 4 shows the CPU subsystem.

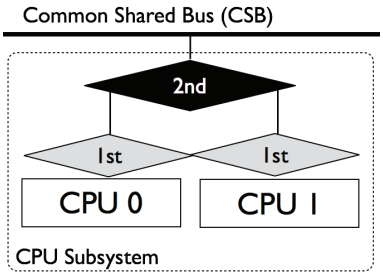

Figure 4: First and second stage MMUs in the CPU centric memory management

2.4 Centralized Memory Mapping and Peripheral Assignment

The segmentation of the addressable space as well as the assignment of certain resources, peripherals or devices is an integral part necessary for the creation of an AMP multi-OS.
In this context, assignment means the device is only accessible by a single, defined OS-domain. As mentioned, the assignment will be enforced by the second stage MMU. To bind a resource to an OS-domain, it must provide an interface (configuration registers) in a dedicated address area (configuration-space). This includes clock assignments, MMU activation and signals, etc. In order to assign a resource to an OS-domain all of the configuration-registers will be mapped to its address space. The example in Figure 5 shows two OS-domains and two devices. Based on the full addressable space, each device or memory partition is assigned to a domain. As an example, if 4GB of main memory is given, the main memory could be mapped from the physical address *0x80000000* to *0xbfffffff*. If an address space is shared the associated addresses are multiply assigned.

2.5 Address Spaces

As a result of the two staged address translations, the system deals with three different address space types, which are briefly introduced in this section.

- **Virtual address space (VA):** This space is typically maintained by the OS-domain. The addresses used in this space are referred to as virtual addresses. An address used in an instruction, as a data or instruction address, is a VA [4] and has a space of up to 32 bits.
- **Intermediate physical address space (IPA):** The IPA is the output of the stage 1 translation and the input of the second stage. If no stage 2 translation takes place, the IPA is the same as the physical address.
- **Physical address space (PA):** PA contains the address of a location in the memory map, which is an output address from the processor to the memory system.

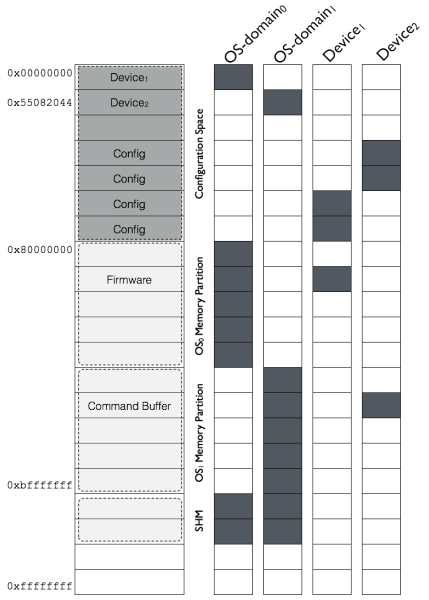

Figure 5: Example for a system memory map

The translation process works as follows:

$$(VA)\xrightarrow{\texttt{stage1}}(IPA)\xrightarrow{\texttt{stage2}}(PA) \tag{1}$$

The proposed method introduces a suitable mapping hat the second stage.

2.6 Translation Tables

The MMU utilizes a translation table to convert an input address to a corresponding output address. Depending on the implementation, these translation tables are located in the PA address space of the SoC. In order to manage a huge address space, the translation tables are divided into different levels. Typically, there are three translation Levels. According to the ARM reference [4] *Level 1* maps 1GiB blocks, *Level 2* 2MiB blocks and *Level 3* 4KiB pages respectively. The input address, particularly the IPA, indexes the position in the table. Each entry points either to a memory region or to the next corresponding translation table level. The suitable construction of these translation tables is further described in 5.1.

3. MEMORY ORGANIZATION

This work develops general models. Nevertheless, the organization of memory, cache subsystems, their protocols and hierarchy are based on the ARMv7 architecture specification [4]. This specification is the foundation for a significant amount of SoC platforms.

3.1 Caches

The general intention of the integration of caches to processors is to speed up access to frequently used memory. The memory in computing systems is hierarchically organized [22]. Regardless of the highest orders, which are the processor's registers, there are one or more levels of cache, which are denoted as *L1*, *L2*, etc. In multi processor (MP) systems some levels are private to the processor and some are shared between multiple processors. In SMP based systems a coherence protocol maintains the synchronization of shared data. Caches expand from the lower to the higher levels. The smallest addressable entity within a cache is a cache-line (CL), which has a fixed CL size, such as 64 Byte.
The last level cache (LLC) before the main memory often has an associativity scheme. The scheme describes which CL in main memory, the memory line (ML), is loaded to a specific location in the cache. The location where a ML is loaded to, is denoted as CL_ID. The associativity between the LLC can be fully associative or could be organized into associativity-cache-way sets. Fully associative means each CL can be loaded to all possible CL_ID positions in the LLC. In most cases caches are divided into way-sets (WS). Thus, a specific ML is associated with a specific WS in cache. If a WS has a size of 8, it is called an 8-way-set associative cache. When a ML is loaded to a CL into the WS, a replacement algorithm determines the specific location. Upon implementation, this could be done by a *least recently used* algorithm or could be totally randomized. The CL that gets replaced, will be written back to the main memory. The number of WSs in the cache can be calculated by:

$$WS_{count} = \frac{cache_{size}}{(CL_{size} * WS_{assoc.})} \tag{2}$$

Table 1: Caching terminology

Sign	Description
WS	Way Set
CL	Cache Line
ML	Memory Line
CL_{Size}	Size of single cache line
ML_{Size}	Size of single memory line
WS_{ID}	A specific WS
DB_{Size}	Size of a Domain Block
WS_{Count}	Amount of way sets
CL_{Count}	Amount of CLs

3.2 Addressing Scheme

Addressing is the fundamental part of memory access. Usually the smallest addressable entities in computer systems are 32Bit. As a result, a single CL contains 16 addressable locations. The data or instructions loaded into the cache can be logically/virtually indexed or physically indexed. In case of physically indexed caches the PA of a memory location identifies CLs in the cache system. As a result, VA or IPAs have to be translated through the MMUs

before the data can be loaded into the cache. If a processor accesses a particular memory location on main memory, the VA will be translated into a PA. In equation 3 how to determine a specific WS in cache to a given PA is shown.

$$WS_{ID} = \frac{PA}{CL_{size}} \bmod WS_{count} \tag{3}$$

4. QUANTITATIVE INTERFERENCE

In this section, the method of how to achieve interference between the two co-OS-domains will be described. Furthermore, we show how to implement the method. For our consideration, we have assumed a two-leveled cache hierarchy and 16 WS associativity of the L2 cache to the main memory.

4.1 Method

In order to introduce performance impact on co-OS-domains, an attack vector is examined which aims to overcommit a certain WS in the LLC. The memory mapping introduced in Section 2.4 shows that the memory partitions are assigned in two big consecutive blocks to the OS-domains, which are denoted as *Memory Partition 0* and *Memory Partition 1* respectively.

As shown in Figure 6, each ML in the main memory is assigned to a specific WS in the LLC. Since the main memory is bigger than the L2 cache, the pattern repeats every time WS_{count} has been reached. The method to fill a specific WS in the cache is to compute WS_ID according to the following equation:

$$Block_{size} = WS_{count} * CL_{size} \tag{4}$$

Algorithm 1 Fill specified WS

Require: $PA \geq 0; Step_{size} > 0$
  $NextPA \Leftarrow PA$
  for $i = 0; i \leq WS_{size}; i++$ **do**
    $AccessNextPA$
    $NextPA \Leftarrow NextPA + Block_{size}$
  end for

If the attacker aims to interfere with the victim, he just has to use the same WSs as the victim. Aiming for a specific WS, and by doing this very frequently the victim's memory accesses to this WS can be significantly delayed. According to the replacement strategy in the WSs, this effect can be deterministically predicted in case of least recently used (LRU) or statistically measured in case of random replacement.

5. COUNTERMEASURE

The attack vector shows that it is possible to interfere with an co-OS domain by attacking specific WSs. Since this would lead to unpredictable memory access execution delays, a method is introduced to prohibit this effect. The cache covers the whole main memory. Attributes like the associativity commonly cannot be changed in the system. Hence, a method is required which is applicable without the introduction of architectural hardware changes.

5.1 Domain Block Memory Mapping

The general strategy for the countermeasure is to invert the DoS method. The method assigns WS in the cache to OS-domains. This is achieved by the introduction of Domain Blocks (DB). The DBs are later mapped to the particular OS-domains. A DB is a memory region that is assigned to a specified region of a set of WSs in LLC. In Figure 7 the method is depicted. The example shows a DB mapping for two memory partitions. As a result, there are two different "colors" for DBs in this case. A single DB consists of a set of MLs. The DBs describe an alternating pattern within the main memory. In the example, a cache with 2048 WS is assumed. To split the cache literally into two halves, a DB consists of 1024 ML. Generally, the size of a DB is calculated through:

$$DB_{size} = \frac{WS_{count}}{Domains} * CL_{size} \tag{5}$$

The DBs will be mapped to the OS-domains using the second stage MMU. The mappings in the *Level 3* descriptors are generated as follows: The input address space, represented by the IPA, must be consecutive for a proper operation of the OS. The output addresses (PA) are generated with regard to the proposed pattern. Since each entry in the TT describes a 4096 Byte page in the main memory, each page contains 64 MLs with a size of 64 Byte. To contain 1024 MLs, 16 pages form a single DB. The mappings are generated using the Algorithm 2. The PA space for the main memory starts at address 0x80000000. For $OS - domain_1$ IPA space starts at 0x80000000 and for $OS - domain_2$ at 0xA0000000. The algorithm iterates through the whole address space of the main memory.

Algorithm 2 Generate Level 3 TT

  $IPA_1 \Leftarrow 0x80000000; IPA_2 \Leftarrow 0xA0000000$
  $PA_1 \Leftarrow 0x80000000, PA_2 \Leftarrow 0x80010000$
  $Pagesize \Leftarrow 0x1000$
  for $j = 0; j \leq MainMemoroy_{size}; j+ = DB_{size}$ **do**
    for $i = 0; i \leq 16; i+ = Pagesize$ **do**
      $IPA_{[1,2]} \Leftrightarrow IPA_{[1,2]} + = Pagesize$
      $PA_{[1,2]} \Leftrightarrow PA_{[1,2]} + = Pagesize$
    end for
    $IPA_{[1,2]} \Leftrightarrow IPA_{[1,2]} + = Pagesize$
    $PA_{[1,2]} \Leftrightarrow PA_{[1,2]} + = DB_{size} * 2$
  end for

There are multiple possibilities to map these pages. For example, in the introduced algorithm, the DBs are optimized to the maximum size. To some extent, this avoids a high fragmentation of the PA memory.

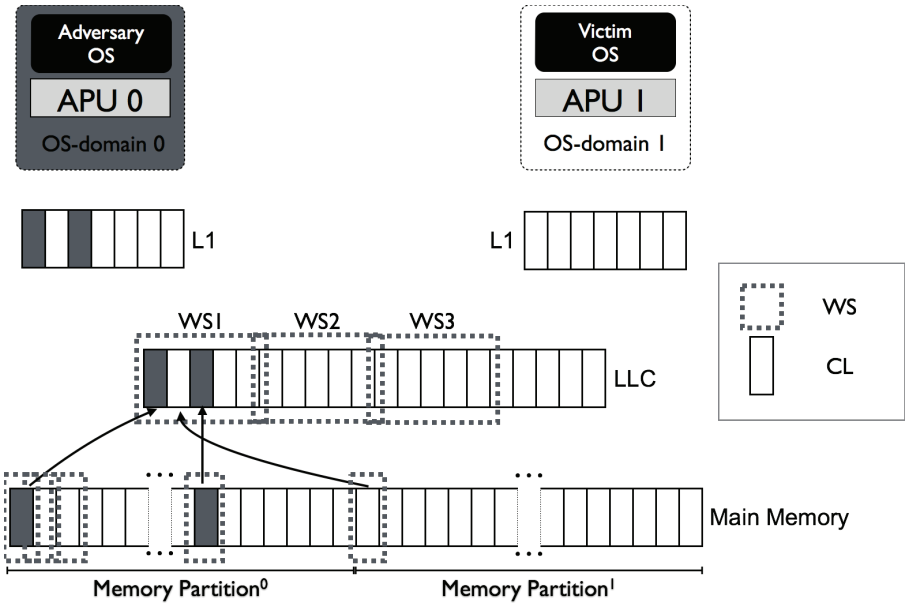

Figure 6: Exploiting the cache way-set association

5.2 Implementation Issues

The proposed mapping theme shows how to separate former shared resources like the set associative cache. In the results we show the applicability to avoid DoS attacks. However, the implementation of the scheme involves certain challenges in system and architectural improvements.

As mentioned previously, the system setup is statically defined and initialized during bootup. To implement the proposed mapping, the initialization of the second stage MMU must be completed before the OS images or configuration files such as Device Tree Blobs are loaded into the main memory. This is problematic, since the second stage MMU is commonly initialized using an identity mapping, which means there is no remapping of memory pages. Since, we divide the main memory into DBs, this scheme has to be considered on loading the OS-images. Basically the images are bigger than the 64KiB DBs, so they have to be loaded with respect to that pattern.

Furthermore, the proposed memory mapping scheme demands a special treatment of direct memory accesses of other subsystems connected to the common system interconnect. Direct memory accesses are preformed on consecutive PA memory areas. In order to use DMA capabilities of certain resources, the maximum transfer size needs to be limited and aligned to the DBs. Otherwise, the IPA address space needs to be established to the DMA capable resource. This requires that the DMA device is aware of the mapping scheme, and is able to translate the IPA addresses which are communicated by the OS-domains into the real PA residing in the main memory.

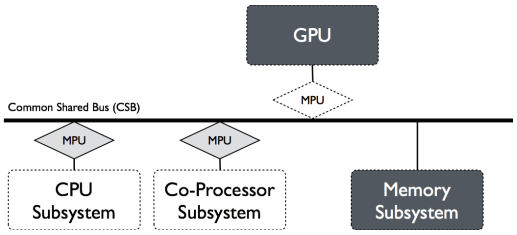

Figure 8: Proposed Architecture

5.3 Properties for a System Architecture

In order to incorporate the outcomes of the key findings in this paper, we propose the properties for a secure integration of the domain block mapping.

There are secure integration of two levelled memory mapping or a memory protection unit. As shown in Figure 8, dedicated MPUs for each master subsystem are proposed. The MPUs must be implemented in a multi-level privilege scheme. This means the configuration is only accessible by the highest privilege level (hypervisor level).

The maximum number of OS-domains, respectively the CPUs, supported by this method is dependent on the cache size, CL_{Size} and on the minimum pagesize supported by the translation table. Assuming an ARMv7 architecture, the minimum pagesize is set to 4KiB and the CL_{Size} is 64 Byte.

$$Domains = WS_{Count} / \frac{Pagesize}{CL_{Size}} \tag{6}$$

In the example configuration, 32 OS-domains or CPUs

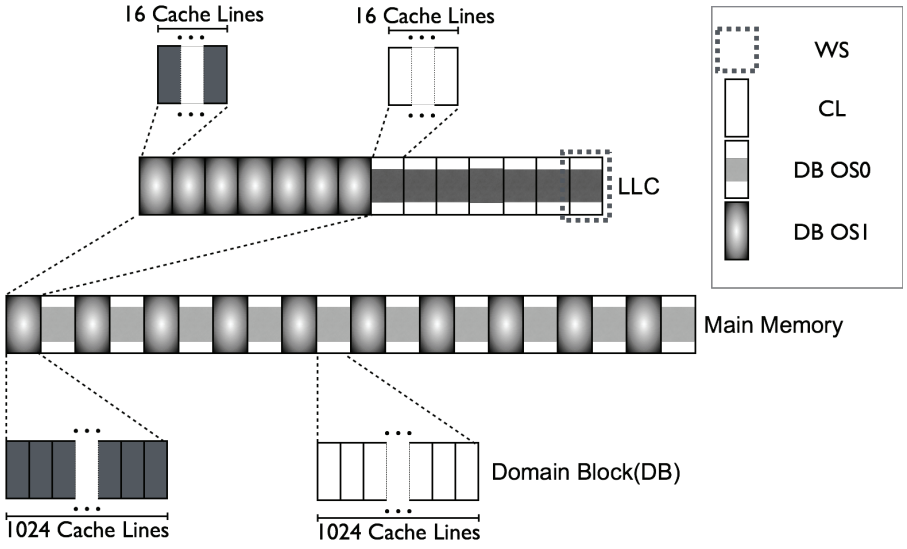

Figure 7: The principle of cache domain blocks

would be supported.

The aforementioned properties are needed to meet the NEAT attributes of the MILS security model.

- **Non-bypassable**: Once activated, the address translation will be enforced by the MPU or Second Stage MMU. Since this is implemented in hardware, it cannot be bypassed by the software running on the CPU. However, this implies that i/o devices which are capable of accessing the main memory directly need to be aware of the mapping method.
- **Always invoked**: According to the non-bypassable attribute, the translation of memory access is done for every read or write attempt of the processor.
- **Tamper proof**: The mechanism is implemented into hardware, which makes it tamper proof against external manipulation. However, the MMU translation tables reside within the main memory. The location needs to be in a restricted area that is not accessible for the processors. As a result, each DMA capable subsystem needs to be incorporated by a MMU/MPU.
- **Evaluateable**: The herein proposed method to separate shared caches, does not introduce any further complexity to the implementation of hard or software. Our method shows an alternative approach to building a second-level address translation.

6. EXPERIMENTAL SETUP

This section will give an overview of the system setup that was used to produce the results given in Section 7.. To produce the results a SoC platform was chosen which is available to the public domain. Therefore, a Pandaboard [23] that incorporates the Texas Instruments OMAP5 SoC and a set of peripherals necessary for ICM-applications is used. The OMAP5 implements a multi processing unit (MPU) subsystem with two ARM Cortex-A15 processors. The MPUs have a direct connection to the main memory or an external memory interface (EMIF). Each Cortex-A15 core has a private L1 64KiB (32KiB each for instructions and data) cache and a shared 2MiB L2 (unified) cache. The cache line size is 64 Byte and has 2048 WS.
Two OSs run on the platform, the adversary and the victim OS-domain. Both OS-domains consists of an upstream Linux Kernel (Version 3.8.13) and a suitable root file system. The methods to measure the proposed effects are implemented on OS-level, using Linux kernel modules.

In order to obtain the results the following functions have been implemented.

- **measure-loop():** The Loop simply executes iterations over a set of data. During each iteration it loads from a ML into a CPU register using the ARM *LDR* instruction.
- **DoS-loop():** Compares to the *measure-loop* without time measurements.
- **get_time():** Timestamps at start and end of each iteration to determine the CPU cycles consumed.
- **prepare_cachelines():** Determines the CLs to the targeted WS.
- **get_next_CL():** Iterates through the CLs.

The loops iterate over a set of ML/CLs determined by the equations given in Section 4..

```
measure_loop(){
  for (k = 0; k < TEST_ITERATIONS; k++) {
    reset_timer();
    prepare_cachelines();
    for (l = 0; l < value; l++) {
      start = get_time();
      get_next_CL();
      end = get_time();
      cycles += end - start; }}}
```

Listing 1: Pseudo code to measure the latency

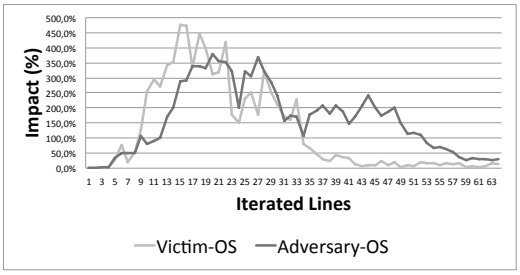

Figure 9: Analysis of WS hit

7. RESULTS

According to Section 4. and 5., the methods have been evaluated. The results are obtained by measuring latencies of memory accesses.
The measurements are performed as follows: The victim OS-domain logs the latency of its memory accesses by producing a characteristic workload through the access of a specific domain block. The adversary-OS works on the same amount of data within the same domain-block set. The following characteristics are considered:

- **CPU cycle count:** Number of cycles consumed by an operation. This value quantifies the duration of a measurement. The cycle count was taken from the timer subsystem of the experimental platform.
- **CL count:** Number of CLs allocated and iterated throughout the measurement.
- **DoS impact:** This value describes the increased percentage of the CPU cycle count of an operation being interfered with.

In order to produce the results the arithmetic mean value of the measurements has been computed. During the measurements some factors have been observed which produce particular outliers. Those factors are caused by functionalities such as cache prediction [24] or bus usage. Technically, it would be possible to turn off those processor features, but this would not produce real world results, because an adversary is not able to do so. The general aim was to prove that the concept fits the predictions rather than to build a polished output.

7.1 Maximum Impact

The first measurement shows how the thrashing impact compares to the number of CLs iterated in a single WS. Both systems run the *DoS-loop* and measure the latency concurrently. The results to prove the DoS method are depicted in Figure 9. The most significant impact to the execution performance of the victim OS peaks at about 457 percent. This means that by using the attack vector, it is possible to delay the execution of an operation up to this value. Another value observed is the number of CLs that have been allocated to cause the interference. The highest peak of interference appears when the victim and the attacker are using 16 CLs, meaning a full WS. If both systems use the same full WS the possibility that a single CL must be replaced by the cache is highest. As a result, the prediction of the attack model to overcommit a common WS is proven to be true.

One of the most important results obtained is the maximum thrashing impact. Here, the victim OS executes the *measure-loop* with latency measurement and the adversary OS only the *DoS-loop*. The results are shown in Figure 10. For these measurements both systems iterate over 16 CL, which produces the highest impact (compare Figure 10). Table 2 gives an overview of all observed values. In the test runs, the mean cycle count of a memory access is about 3,602. By activating the *DoS-loop* the cycle count increases up to 132,803. This implies a DoS impact of about 3686%. Compared to the value of the previous graph, the significantly higher result is justified by the frequency of the *DoS-loop*. If a single iteration of the *DoS-loop* runs without the latency marks, then the CPU cycle count per step is lower.

7.2 Domain Block Mapping

To evaluate the value of the DB memory mapping, the measurements made after establishing the mappings. The results in Figure 10 compare the CPU cycles consumed during a data fetch. According to the measurements with the identity mapping both systems iterate through the same full WS. The normal execution of the *measure-loop* reveals a mean cycle count of 3,599. By applying the DoS-loop to the measurement, the cycle count rises to 8,911. Consequently, this results in a DoS impact of about 247,55%. By comparing the mean cycle counts of DB and identity mapping the DoS impact subsides significantly.

7.3 Overhead Evaluation

Despite the advantages of the separability of LLC, the herein proposed method will introduce some overhead evaluation of certain attributes of the system execution. Therefore we compare the approach according to the memory access latency, to determine the cost for single memory access. Furthermore, the latency for memory accesses in consecutive memory areas in copy operations will be discussed.

The introduction of the proposed memory mapping will introduce overhead by retrieving the mapping entries from

Table 2: Domain Block Mapping impact

Mapping	Impact (%)	Mean CPU Cycles	Std. Derivation
identity	-	3,602	0,022
DB	-	3,599	0,021
identity DoS	3686,90%	132,803	2,265
DB DoS	247,55%	8,911	2,062

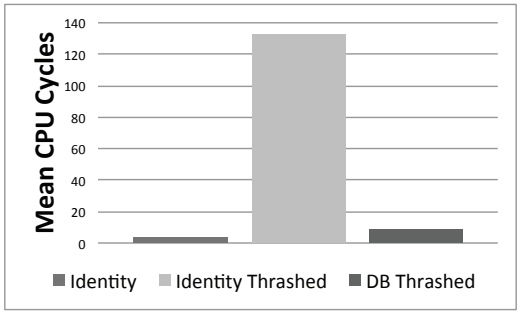

Figure 10: Comparision of indentity and DB mapping

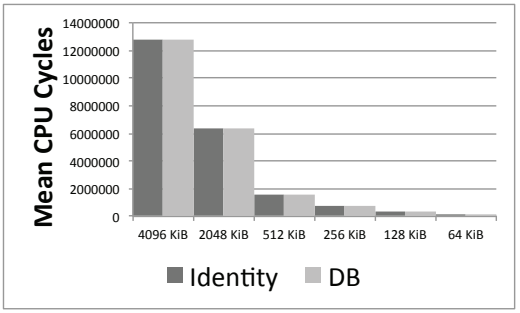

Figure 11: Memory bandwidth comparison

the MMU translation tables. This might be caused by additional table walks. A full translation table lookup is called a translation table walk. As mentioned previously, the MMU translation tables are concatenated tables, where each level describes a finer-grained amount of data, from the higher to the lower level. Using an identity memory mapping the granularity of Level 2 block descriptors are sufficient to produce a proper system mapping. According to the domain block mapping, Level 3 descriptors which map to the 4KiB pages need to be used rather than the 2MiB blocks in Level 2. This implies, that a lookup for a physical output address needs to walk through the descriptor levels 1 to 3, whereas Level 3 has architectural dependent number of entries.

As mentioned above, the mapping method in the second stage MMU level implies three table walks to convert the IPA input address into the PA output address. In order to quantify the cost of the additional table walk the latency of the *measure-loop* has been observed with the original 2 MiB *Level 2* mapping and the domain block mapping which resides in the *Level 3* translation table. The results show that the DB mapping method does not add any performance overhead to the memory access if a single WS is considered. The mean CPU cycles remain at 3,602 in identity mapping and 3,599 using the DB mapping.

In addition to the table walks, the MMU implements a cache for its recently used translation entries. This cache is commonly referred to as a translation look aside buffer (TLB). Depending on the particular architectural specification, the TLB caches up to 512 translation descriptors. Assuming a normal identity system mapping and our experimental platform, utilizing Level 2 descriptors for 2MiB blocks, there are approximately 2048 descriptors to be stored. In contrast to the DB mapping method which uses fine grained Level 3 descriptors, there are about 1048576 (size of main memory divided by the size of page mapped by the Level 3 descriptor) translations. The probability of refreshing the TLB is much higher than the DB mapping compared to with the normal method.

Particularly in this architecture the TLBs stores up to 512 translations. Since, in the previous scenario only 32 different CLs are accessed, these translations reside inside the TLB. Therefore, we increased the number of CL in our access scenario to exceed the capacity of the TLB. The results quantify the impact to access latency caused by refreshing the TLB. In this case the mean access latency of the *measure_loop* is about 2.52 percent.

Practically, in infotainment systems sometimes large amounts of data are going to be transferred. This could be graphics data or media files which are handled by the CPU. Therefore, we determine the behaviour of copy operations of larger data blocks compared to your previous measurements. Figure 11 shows a comparison of the mean CPU cycles consumed by copying a particular amount of data. It analyses the overhead for large data operations. The data block is contiguous allocated in the intermediate address space. The results show that the bandwidth of the identity and the DB mapping is barely at the same level. In fact, the domain block approach has no influence on the transfer of contiguous memory areas.

The evaluation shows the negative performance impact of the deliberate interference of shared caches. By utilizing the DB mapping, it is possible to substantially mitigate these effects. Furthermore, the performance overhead for domain blocking is negligible. Nevertheless, we still

observe a performance impact using the DB mapping. This might be caused by the bus interconnect which transports the data from the main memory through the caches to the CPUs.

8. CONCLUSION

In Mixed Criticality Systems the isolation of the independent system components is important. By implementing an AMP based Multi-OS this goal can be achieved by design. However, the separation of common resources like caches must be proven to achieve a reliable system compilation. In this article, a method has been introduced to avoid the surface for interference and DoS attacks on shared caches. In order to prove the separability of concurrent OS-domains in a mixed critical system, the herein proposed approach has been designed to meet the requirements of established security models. The method is based on the partitioning of memory lines within the main memory. According to the way-set associativity, these partitions, so-called domain blocks, can be assigned to OS-domains. The mapping is enforced using a MMU within the stage-two memory address translation. Additionally, the method fits to the AMP system design paradigm.
Furthermore, this article shows the significance and the need for a solution to mitigate or prohibit DoS attacks in shared last level caches. In addition to that, the surface for interference is quantified to evaluate our solution. This proposed domain block mapping breaks the AMP-paradigm down to the shared caches of CPU-subsystems in SoCs. Since the approach uses the second-stage MMU which is used by the system anyway, the complexity of the system design and the additional engineering cost of the method is kept to a minimum level. In addition, the measurement results show the negligible performance degradation. The method enables a more reliable implementation of AMP-based multi-OSs on MP-SoCs using shared caches, without the need to modify the pre-qualified hardware layout. The method and the results also have an impact on other disciplines related to SoCs. In the real-time research field, it can be used to make memory access more predictable, regardless of whether a multi-OS is implemented or not.

The consideration focuses on a dual core CPU subsystem. In the future, the technique can be applied and evaluated in SoCs that incorporate quad (or more) core processors that share a last level cache. The proposed domain blocks make it necessary to break the mapping down to the fully-addressable space on the SoC. Therefore, solutions to establish these domain blocks for devices which access the main memory directly must be considered. Furthermore, the cache interconnection bus provides a further surface for interference. In future this particular aspect has to be considered so it fits the AMP system design more adequately.

ACKNOWLEDGMENTS

The authors would like to thank Alexander Minor and Thorsten Schnarz for their excellent practical efforts and comments. Furthermore, we appreciate the detailed and helpful comments of the reviewers.

REFERENCES

[1] C. Hammerschmidt, “Harman brings virtualisation and scalability to automotive infotainment,” Jan. 2014.

[2] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann, “Engineering automotive software,” *Proceedings of the IEEE*, vol. 95, no. 2, pp. 356–373, 2007.

[3] M. Broy, “Automotive software engineering,” in *Software Engineering, 2003. Proceedings. 25th International Conference on*, 2003, pp. 719–720.

[4] ARM, “ARM Architecture Reference Manual - ARMv7-A and ARMv7-R edition,” 2012.

[5] H. Inoue, A. Ikeno, M. Kondo, J. Sakai, and M. Edahiro, “VIRTUS: a new processor virtualization architecture for security-oriented next-generation mobile terminals,” in *Design Automation Conference, 2006 43rd ACM/IEEE*. IEEE, 2006, pp. 484–489.

[6] J. Porquet, C. Schwarz, and A. Greiner, “Multi-compartment: A new architecture for secure co-hosting on soc,” in *System-on-Chip, 2009. SOC 2009. International Symposium on*, 2009, pp. 124–127.

[7] W. Kanda, Y. Murata, and T. Nakajima, “SIGMA System: A Multi-OS Environment for Embedded Systems,” *Journal of Signal Processing Systems*, vol. 59, no. 1, pp. 33–43, Sep. 2008.

[8] Y. Kinebuchi, T. Morita, K. Makijima, M. Sugaya, and T. Nakajima, “Constructing a multi-os platform with minimal engineering cost,” pp. 195–206, 2009.

[9] A. Crespo, I. Ripoll, and M. Masmano, “Partitioned Embedded Architecture Based on Hypervisor: The XtratuM Approach,” in *Dependable Computing Conference (EDCC), 2010 European*. IEEE, 2010, pp. 67–72.

[10] S. Cho and L. Jin, “Managing distributed, shared l2 caches through os-level page allocation,” in *IEEE/ACM INTERNATIONAL SYMPOSIUM ON MICROARCHITECTURE*. IEEE Computer Society, 2006, pp. 455–468.

[11] D. Tam, R. Azimi, L. Soares, and M. Stumm, “Managing shared l2 caches on multicore systems in software,” in *In Proc. of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA*, 2007.

[12] S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 111–122. [Online]. Available: http://dx.doi.org/10.1109/PACT.2004.15

[13] S. Park, C. T. Chou, and A. Kumar, "Fairness mechanism for starvation prevention in directory-based cache coherence protocols," Tech. Rep., 2012.

[14] M. Khare and A. Kumar, "Method and apparatus for preventing starvation in a multi-node architecture," Tech. Rep., 2002.

[15] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, April 2010, pp. 13–22.

[16] F. R. Wagner, W. O. Cesário, L. Carro, and A. A. Jerraya, "Strategies for the integration of hardware and software ip components in embedded systems-on-chip," *Integration, the VLSI journal*, vol. 37, no. 4, pp. 223–252, 2004.

[17] J. Alves-Foss, P. W. Oman, and C. Taylor, "The MILS architecture for high-assurance embedded systems," *... of embedded systems*, 2006.

[18] G. M. Uchenick and W. M. Vanfleet, "Multiple independent levels of safety and security: high assurance architecture for MSLS/MLS," in *Military Communications Conference, 2005. MILCOM 2005. IEEE*. IEEE, 2005, pp. 610–614.

[19] W. M. Vanfleet, R. W. Beckwith, B. Calloni, J. A. Luke, C. Taylor, and G. Uchenick, "MILS:Architecture for High-Assurance Embedded Computing," pp. 1–5, Jul. 2005.

[20] J. Alves-Foss, C. Taylor, and P. Oman, "A multi-layered approach to security in high assurance systems," in *System Sciences, 2004. Proceedings of the 37th Annual Hawaii International Conference on*. IEEE, 2004, p. 10 pp.

[21] J. M. Rushby, "Proof of separability A verification technique for a class of security kernels," in *International Symposium on Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, Jan. 1982, pp. 352–367.

[22] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[23] Texas Instruments, "OMAP5432 Multimedia Device - Silicon Revision 2.0 Evaluation Module," 2012.

[24] Texas Instruments, Incorporated, "OMAP 5 Specifications," Tech. Rep., 2013.