Titre: Title:	Cache Predictability and Performance Improvement in ARINC-653 Compliant Systems
Auteur: Author:	Alexy Torres Aurora Dugo
Date:	2019
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Torres Aurora Dugo, A. (2019). Cache Predictability and Performance Improvement in ARINC-653 Compliant Systems [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/3941/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/3941/
Directeurs de recherche: Advisors:	Gabriela Nicolescu
Programme: Program:	Génie informatique

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cache predictability and performance improvement in ARINC-653 compliant systems

ALEXY TORRES AURORA DUGO
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Juillet 2019

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Cache predictability and performance improvement in ARINC-653 compliant systems

présenté par **Alexy TORRES AURORA DUGO**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Giuliano ANTONIOL, président
Gabriela NICOLESCU, membre et directrice de recherche
Michel DAGENAIS, membre

DEDICATION

I dedicate my work to my family and friends,
thank you for supporting me

ACKNOWLEDGEMENTS

I would like to thank my research director Prof. Gabriela Nicolescu for her support and help during my master years. I am thankful toward Mannarino System and Softwares, especially Mr. Dahman Assal for all the support and advice they provided me. I would also like to thank all my laboratory colleagues for their help and the good times we had together. I am grateful for the help that Prof. Frédéric Pétrot provided me to find my way in the domain of computer science that is embedded systems. I would like to thank my family and friends that never ceased to support me during these past years. Finally, I would like to acknowledge Mitacs for the financial support they provided for this project and made it possible for me to conduct this research.

RÉSUMÉ

Depuis les années 2000, les processeurs multi-coeurs sont développés afin de répondre à une demande croissante en performances et miniaturisation. Ces nouvelles architectures viennent remplacer les processeurs mono-coeurs, moins rentables sur le plan des performances et de la consommation énergétique. De par cette transition, les systèmes avioniques actuels se retrouvent face à un défi de taille. Ces systèmes critiques n'utilisent que des processeurs mono-coeurs, éprouvés et validés depuis des années afin de garantir la fiabilité du système. Cependant, les fabriquants de processeurs et autres microcontrôleur délaissent peu à peu ces architectures pour ne produire que des processeurs multi-coeurs. Afin de maintenir les systèmes avioniques critiques, les intégrateurs doivent alors se tourner vers ces nouveaux processeurs. Cependant, cette transition n'est pas sans défi. Outre le fait de devoir assurer la portabilité des applications mono-coeur dans un environnement multi-coeurs, l'utilisation de plusieurs coeurs permet leur exécution concurrente. Ce nouveau paradigme apporte aux systèmes des comportements qui peuvent entrainer, dans certains cas, un dysfonctionnement complet du système. De tels comportements ne sont pas acceptables dans ces systèmes où la moindre faute peut provoquer des pertes humaines. Les systèmes critiques suivent certaines règles afin de garantir leur intégrité. Le standard ARINC-653 définit un ensemble de règles et de recommandations afin de développer ce genre de systèmes. Le standard introduit le concept de système partitionné où chaque partition s'exécute indépendamment des autres et ne peut pas influer sur le comportement du système ou des autres partitions. Ainsi, si une partition vient à fonctionner anormalement, son exécution ne peut compromettre le bon fonctionnement des autres partitions. Le problème émergeant dans les architectures multi-coeurs vient du fait que plusieurs partitions peuvent s'exécuter de manière parallèle. Cette nouvelle possibilité introduit de la concurrence sur les ressources du système, ce qui génère des comportements non prévisibles. Ces comportements, appelés interférences apparaissent lorsque plusieurs coeurs partagent les mêmes ressources. Lors d'un accès à ces ressources (mémoire, périphériques, etc.), un arbitrage doit être fait afin d'assurer l'intégrité des données. Cet arbitrage peut causer des délais dans l'accès à une ressource. De plus si plusieurs partitions accèdent à une même ressource, le concept d'isolation n'est plus respecté. Dans le cas des mémoires caches partagées, une partition peut évincer des données utilisées par une autre partition. Dans ce mémoire, nous étudions la possibilité d'empêcher l'évincement de données des caches privés d'un processeur. Cette méthode, appelée cache locking, permet de réduire le nombre de fautes de cache dans les caches privés et ainsi limiter les accès aux caches partagés. Cela permet de réduire les interférences liées aux caches partagés, non seulement en termes de concurrence d'accès, mais aussi d'évincement non voulus de données dans ces caches. Ainsi nous introduisons un outil de profilage d'utilisation de la mémoire dans les systèmes partitionnés. Nous présentons aussi un algorithme associé à cet outil permettant de sélectionner le contenu des mémoires caches devant être empêché d'être évincé. Cet outil propose un processus complet de traitement des traces d'accès mémoire jusqu'à la création des fichiers de configuration. Nous avons validé notre approche par le biais de simulation et d'expérimentation sur matériel réel. Un système d'exploitation temps réel respectant la norme ARINC-653 a été utilisé afin de conduire nos expérimentations. Les résultats obtenus sont encourageants et permettent de comprendre l'impact des méthodes de caches locking pour les systèmes embarqués multi-coeurs.

ABSTRACT

Due to their energy efficiency and their capability to be miniaturized, multi-core processors have been developed to replace the less cost-effective single-core architectures. Since around year 2000, processor manufacturers slowly stopped producing single-core processors. This raised an issue for avionic system designers. In these critical systems, designers use processors that have proven their reliability through time. However, none of such processors are multi-core. To be able to keep their system up to date, aerospace system designers will have to make the transition to multi-core architectures. This transition brings a lot of challenges to system designers and integrator. Current single-core applications may not be fully portable to multi-core systems. Thus developers will have to make sure the transition is possible for such applications. Multi-core CPUs offer the ability to execute multiple tasks in parallel. From this ability, new behaviors may induce delays, bugs and undefined behaviors that may result in general system failure. In critical systems, where safety is crucial, such comportment is unacceptable. Safety critical systems have to comply with multiple standards and guidance to ensure their reliability. One of the standard Real Time Operating Systems developers may rely on is the ARINC-653. This standard introduces the concept of partitioned systems. In such systems, each partition runs independently and should never be able to modify or impact the behavior of another partition. This concept ensures that if one partition comes to misbehave, the system's integrity is not impacted. In multi-core systems, multiple applications can run in parallel and access hardware and software resources at the same time. This concurrence, if not correctly managed, will introduce delays in execution time, loss of performances and unwanted behaviors. We call interferences any behavior introduced by this concurrence on the resources shared by different cores or partitions. When concurrent accesses to shared components occur, arbitration has to be done to ensure the data integrity. In most cases, this arbitration is the cause of interferences. However, other sources of interference exist. For instance, if two partitions share the same cache, one partition may evict other partition data from the cache. This leads to unpredictable delays when the next partitions will need to access the evicted data. In this thesis, we explore methods to prevent cache line evictions in private processor caches. This work allows to reduce the number of cache misses occurring at the private level, which reduces the amount of access done to the lower memory levels and reduces interferences related to them. We call this method cache locking. We introduce a framework capable of profiling memory accesses done by applications and propose a cache content selection algorithm that selects cache lines to be locked to reduce cache misses in private caches. We present the toll and the associated processing pipeline, from the memory profiling, to the cache locking configuration table generation. We validated our approach on simulated and actual hardware and used a proprietary ARINC-653 compliant system to conduct our experiments. The results obtained are encouraging and allow to understand the impact of private caches and cache locking methods to reduce multi-core interferences in safety-critical systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ACRONYMS

AER Acquisition Execution Restitution

APEX APplication EXecutive

API Application Programming Interface

ARINC Avionics Application Standard Software Interface

CAST Certification Authorities Software Team

CPU Central Processing Unit

DMA Direct Memory Access

EUROCAE European Organization for Civil Aviation Equipment

FPU Floating Point Unit

HM Health Monitor

IMA Integrated Modular Avionic

IO Input/Output

MMU Memory Management Unit

MPU Memory Protection Unit

OS Operating System

PMC Performance Monitoring Counters

RTCA Radio Technical Commission for Aeronautics

RTOS Real Time Operating System

SWaP Size Weight and Power

TDMA Time-Division Multiple Access

WCET Worst Case Execution Time

XML Extensible Markup Language

CHAPTER 1 INTRODUCTION

This chapter presents the main challenges that avionics program manufacturers face when new multi-core architectures are integrated in avionic systems. We then introduce the objectives and contributions of this thesis.

1.1 Context

Nowadays multi-core architectures are more and more present in embedded systems. Compared to single-core processors, these processors often have better performances and appear to be more energy efficient [1]. Therefore, processor manufactures are reducing the development of single-core processing units.

Multi-core processors allow different tasks to be extended in parallel. It is particularly appealing in systems where a lot of tasks have to execute in a limited amount of time. More and more computing intensive tasks appear in critical systems. Image processing, artificial intelligence and communications are some of these tasks. Enabling parallelism for processes will provide faster response time and, thus, improve safety of the system.

Actors in the aerospace domain only rely on proven architectures that have been used for years. PowerPC architectures, for instance, are often used for their reliability. However, looking at the current market of processors in critical embedded systems, ARM processors slowly replace PowerPC ones due to their lower prices and power consumption.

Avionics actors, being a small part of the processor market, will soon have to make the transition from single-core to multi-core processors if they want to keep their architectures up to date.

The criticality of aerospace applications enforces developers to design programs that comply to major standard such as ARINC-653 [2] standards or DO-178C [3] guidance documents. However such documents do not address multi-core architectures due to their lack of determinism and the multiple issues raised by their design. Recently, the Certification Authorities Software Team (CAST) released different papers about multi-core processors use in avionic systems. Most of the issues brought by multi-core processor are clearly stated but no actual mean of avoidance of such issues is defined.

To catch up with the multi-core tendency, one solution is to develop new Real Time Operating System (RTOS) capable of supporting such architectures while providing the same services and level of safety than single-core RTOS. The new RTOS will also have to take advantages of the increase of performance brought by the multi-core architecture (parallel execution, load balancing). RTOS developers started to release new versions of their product to support multi-core platforms[1]. However these new RTOS do not fully address the challenges brought my the new architectures.

One of the strongest constraints in the avionics domain is task isolation. The isolation principle defines that one application cannot impact the behavior of another application in the system [2]. From a single-core point of view, this principle is ensured by spacial isolation. Peripherals and memory areas can be isolated to restrict their access to one or more applications. Timing isolation is inherently achieved executing one application at a time.

Multi-core architectures allows faster processing. Indeed, the gain of performances comes from the multiple processing units, executing software code in a parallel manner. Multiple paradigms can be used to take advantages of multi-core processors. One may take one application and modify its code to run multiple parts of the application in parallel. In aerospace systems, this is not the targeted option. Current avionic applications are not designed to run in parallel and the effort to update the code of each application is not worth the cost. However, future applications may take into account the possibility to achieve multiple tasks in parallel. The other way of updating the system to multi-core processors is to have multiple applications running at the same time. This option is more realistic in the current context of aerospace systems.

In multi-core processors, temporal isolation needs to be explicitly addressed as multiple applications execute in parallel at a certain moment. Thus, applications no longer have exclusive access to the resources but instead share them with other independent applications. The principle of isolating applications is called partitioning. Application partitioning is an execution model that ensures the exclusive use of internal resources [4](caches, memories, hardware accelerators, etc.) and external resources [5] (Input/Output (IO), peripherals, etc.) at a given moment. Concurrent accesses to these shared resources create interferences. These interferences can bring the system in an unstable state [6] (bug, crashes, etc.) when multiple applications modify the memory state or access a peripheral. In most current systems, memories, peripherals and other resources cannot satisfy multiple requests at the same time. Thus, an arbitration must be made between the different requests they receive. This arbitration may delay the access to the resource for the processors (or component) which sent the request. These delays will impact the responsiveness and the execution time of independent applications. From the multicore designer's viewpoint, interference occurrence

[1] eg. DeOS (DDCI), VxWorks 653 (WindRiver)

is not a dysfunctional behavior, it is considered a performance bottleneck. However, for the avionics designers, interference occurrences are considered dysfunctional behaviors. In the avionics domain, where such non-deterministic behavior cannot be tolerated, as they can entail casualties, research has to be conducted to reduce the impact of interferences [7].

1.2 Research Challenges

One of the most important challenges to face in the real-time embedded systems domain is the ability to compute the Worst Case Execution Time (WCET) of each application or tasks running on the system. This metric allows bounding the task's execution. Bounding tasks execution time allow the system designer to ensure that the system can schedule the tasks correctly [8]. Interferences entailed by the presence of multiple cores executing in parallel will prevent the developers from computing or even measuring the WCET correctly. X. Jean et al. [9] proposed methods to allow WCET computations in non-deterministic systems. However, the current literature does not always address interferences and prefers to propose solutions to take them into account instead of mitigating them[2]. P. Huyck [10] and H. Agrou et al. [11] highlight the importance of the different issues raised by the use of multi-core architectures in current avionic systems.

1.3 Objectives

This section presents the global objective of this thesis as well as the detailed objectives defined during the conducted research.

Research Question In this master thesis, we answer the following question:
How to reduce interferences and improve performance of private and shared caches in ARINC-653 compliant environments?

Objectives The global objective of the thesis is to develop a solution to mitigate interferences with focus on interferences caused by caches in multi-core processors. We propose new methods to optimize the resources (caches) usage while ensuring isolation between applications in the system. The solution is integrated in a proprietary RTOS certified for avionic use. The following specific objectives have been defined for the project:

[2] For instance, one may compute WCET assuming that caches are disabled, thus, taking into account any cache-related interferences possible (as the worst case happens when all memory accesses produce a miss in the cache).

- to define, analyze and characterize interferences present in multi-core systems.
- to develop cache analysis tools to profile cache usage in the system.
- to design software solutions (algorithms) to mitigate the detected interferences.
- to integrate the developed method in the proprietary RTOS.
- to validate the approach and methods based on results gathered with the RTOS.

1.4 Contribution

To solve cache interferences, the proposed contributions are:

1. Propose a memory tracing framework, capable of analyzing memory accesses of real-time tasks. The framework embeds a memory tracing module integrated in the Qemu emulator. An ARINC-653 scheduler enables the framework to know the state of the system at any moment. We also integrated different memory trace based algorithms in the framework.

2. Define new approaches to select the cache lines to lock (prevent the lines from eviction)[3]. This selection can be based on multiple criteria;

3. Integrate cache locking mechanism in an ARINC-653 compliant RTOS;

[3] Cache locking concepts are defined later in this thesis.

CHAPTER 2 LITERATURE REVIEW

In this chapter, we present the related work conducted around interferences in multi-core architectures. We did not limit ourselves to cache related interferences and preferred to keep a large vision of what interferences are and where they occur in the system.

2.1 Interference in Multi-core Systems

Since multi-core architectures have been introduced, extensive research has been conducted to use them in the avionic domain. However, the parallelism brought by such processors comes at the cost of interferences.

In [12], caches are said to be the most important and critical interference channel in the system. Multiple sources of interferences have been observed such as caches, memory, shared busses, etc.

In [13, 14] interrupts are also shown to create interferences. Indeed, when multiple interrupts are routed to the same core, but not handled by this specific core, the core has to transfer the interrupts to the handling core(s). This leads to delay in execution and non-deterministic behavior if interrupts are not delivered with a fixed period. To avoid this, authors propose to disable interrupts usage from partitions and rely on the RTOS to enforce determinism on interrupts.

In [13, 15, 16], interference channels are classified and studied to quantify the impact they have on the system execution time. It is reported that even inside the CPU, interferences occur [15]. The Floating Point Unit (FPU) can be shared between multiple cores, which may lead to contention on this component. Other units such as the branch prediction unit will introduce non-deterministic behavior in the system.

Hyperthreading and cache coherence protocols have also been shown to introduce competition to resources that were supposedly private to a CPU core [17]. Other shared components of the system are categorized as interference channels. IOs are often considered as a part of the interference channels as they can modify the behavior of a partition. In [14] authors explain how a partition can steal execution time from another partition by using Direct Memory Access (DMA) transfers. Authors also show how a partition can lock the bus, stalling other cores of the processor and potentially leading to deadlines misses of other partitions. They propose to disable DMA interrupts for partitions and rely on polling to notify the partition when the transfer is complete. Interference mitigation has also been studied under multiple forms. In [18], a framework is developed to deploy IMA applications. An offline analysis is done by the tool to compute the WCET of each application of the system. This analysis is done thanks to a single-core approach. The authors present a model that divides system resources in private areas that are only used by one core. Using this method, they expect the partitions to behave in the same way as if they executed in single-core environment. A scheduling method is also presented to allow the correct use and repartition of the resources in the system. However, even if multiple interferences are solved by this approach, contention on shared resources such as busses or memory still exists. Other methods lean toward detection of interference from application profiling [19]. A solution to avoid interferences is to deactivate all core of the CPU except one [20]. This solution would, however, remove all the advantages brought by the multi-core processors.

2.2 Memory Interference in Multi-core Systems

We refer as memory interferences all interferences that occur in the memory hierarchy. This spans from the CPU's private caches to the central memory. Mass storage devices are considered as IOs and are excluded from this category. In this section we present different studies and mitigation efforts made to reduce memory interferences.

In [21], a suite of benchmarks is proposed. This suite allows to study the impact of shared cache interferences in the system. The presented methods consist in running the applications to test on N cores and to run the benchmark applications on the other cores. Benchmark applications will stress the shared resources and one can use probes such as Performance Monitoring Counters (PMC) to study the influence of interferences on the critical applications. Other works considering the same approach have been published such as the benchmarks proposed in [22].

As said earlier, shared caches are the component where most of the interferences occur. In [17], three cache-related interferences are defined.

The first interference type, also present in single-core systems, occurs when a partition evict data it does not own from the cache. In this case data that are being evicted are not owned by any partition currently executing in the system. Such interferences can be solved by invalidating the cache at each partition switch. This ensures that the state of the cache is always the same when the partition begins its execution.

The second type of interference is the same as the first but this time in multi-core systems. As many partitions can execute in parallel, a partition can evict data that are owned by another partition that currently executes in the system.

Finally, the third type of interference is cache contention. When multiple cores concurrently access the cache, the cache controller serializes the accesses. This has the effect to stall the cores waiting to access the cache and introduces delays in execution time. In [12] cache coherence protocols also induce interferences as a core can indirectly modify the content of another core's cache through the protocols. To mitigate such interferences, cache partitioning is one of the most effective solutions [23, 24]. In [25] a framework relies on cache coloring to divide the cache into multiple partitions. This method prevents partitions to evict from cache data they do not own. Authors rely on cache partitioning and task scheduling. Cache partitioning avoids cache-related interferences while task scheduling allows multiple partitions to use the same cache area: when two partitions use the same cache area, they will never be scheduled at the same time. Two mitigation methods are then possible:

- Partitions cannot evict data they do not own from the cache at any time. Each partition needs its private cache area.
- Two partitions can share the same cache area. They can evict each other's data, but will never be executed at the same time. If the two partitions flush and invalidate their cache area before executing, then no interference is possible.

In [26] an offline profiling framework is proposed to manage caches in multi-core systems. The goal of the tool is to determine the memory pages which are the most frequently accessed. The profiling data are compiled to generate a memory map which is used to generate an execution independent memory profile of the application. This profile is used to partition caches and thus avoid a partition from evicting another partition's data. Other partitioning approaches have been studied as in [27]. Authors use the memory manager of the Xen hypervisor to allocate pages in the system. This allows having multiple pools of addresses. Each pool refers to one or more partitions in the cache. Other sources of interferences are present in the cache such as cache writes buffers [28].

Locking caches has also been studied to reduce interferences in two ways. The first improvement brought by cache locking is the reduction of cache misses. Thus, locking data in private cache will reduce the amount of memory accesses in shared areas. In [29] static code analysis is explored to select which instruction data should be locked. However, such a method cannot be used for data as most data accesses are implicit and addresses are only known at run time. Using static analysis could not allow to determine which addresses are accessed. More methods for cache locking are studied in the literature review provided in Chapter 5 of this thesis.

Memory interferences also happen in DRAM memories. In [30] memory banks are allocated to different applications. As for cache partitioning, this avoids having multiple applications or partitions using the same memory bank and reduce the number of interferences. Such methods could be applied by tweaking the virtual memory addresses as done for cache partitioning. Application scheduling is also used to optimize memory use and reduce performance impact when multiple applications try to access the memory at the same time [31]. DRAM caches known as row buffers are also considered as interference channels. In [32], interbank interferences and the intrabank interferences are studied. The authors propose to model generated interferences and apply memory bank partitioning and budgeted memory accesses to mitigate them. In [33] and [22] memory bus contention is shown to create interference when multiple cores access the main memory. In [33] performance monitoring counters are used to set bandwidth limitations to partitions to prevent them from accessing the memory and adding contention on the memory bus. This method does not remove interference but allows bounding them. Budget servers have also been studied to bound interferences generated by memory accesses [34]. One may associate a budget to a partition, a partition's process or a core of the processor. The recurrent method is to use performance monitoring counters to track budget. These counters are present in most architectures and allow monitoring multiple events such as cache misses, CPU clock cycles, page fault counter, etc.

2.3 Communication Interference in Multi-core Systems

Contrary to memory interferences that occur in shared memory regions, communication interferences occur in shared communication channels. Multiple cores share busses, interconnects, IOs and other communications means. The most important part of communication interferences is due to contention on such components [35]. This issue is inevitable in multi-core systems; however, it is possible bound the effect of interference to ease the WCET measurement.

One type of communication interferences that the literature tries to tackle is IO interference. When multiple partitions try to access an IO or a peripheral, contention on the bus, communication ports and the IO itself occurs. In [36] a method to serialize accesses to IOs is proposed. In this approach a core is dedicated to IO management and all other cores are not allowed to provide any access to them. When a partition needs to access an IO, the access is serialized with other cores requests. The dedicated core can run one IO partition or more (for instance, one may choose to have one IO partition per regular partition). This method prevents the system from having multiple IO accesses occurring at the same time. However, IO partitions must be run with a high rate since many IO accesses can be done by a partition and should be quickly serviced.

To overcome the potentially high response time induced by the approach proposed in [36], research introduced hardware components to manage IO interferences. In [37] a framework is proposed to restrict partitions bandwidth usage. A virtualization system abstracts each peripheral in the systems. These virtual IOs are scheduled in a manner that tricks the partition that uses it into thinking it is executing in a single-core environment. Those virtual peripherals are scheduled thanks to an IO scheduler. This method relies on hardware components developed on FPGAs. These additional interfaces make a bridge between the system and the peripherals. The bridges allow monitoring peripheral traffic (income and outcome) and block any partition's traffic for which the budget has been exceeded.

Communication interferences are also introduced by hardware resources arbitration. It is important to consider different arbitration methods and study their impact on the system's behavior. In [35] a comparison between different resource sharing arbitration such as Time-Division Multiple Access (TDMA) is proposed. Associated with this contribution, the author defines a generalization of the WCET computing methods considering TDMA and Priority Division arbitration methods. Fair arbitration methods (Round Robin) and more complex schemes are studied (TDMA, Priority Division). The later methods associate priorities to requests on the components. The conducted experiments show that TMDA is the worst algorithm to use with at most 20% of total utilization while priority based and round robin schemes allow up to 65% of total utilization. Priority division, however, is in the middle and allows up to 45% of total utilization with 8 cores. Concerning the jitter introduced by arbitration schemes, results show that TDMA performs poorly, introducing up to 80% of jitter on 8 cores while the Priority based scheme introduces 40% of jitter. To optimize bus access arbitration, scheduling tables can be used to enforce the predictable bus access scheme for each partition or core [38].

2.4 Scheduling and WCET Analysis

Execution models have also piked interest of researchers to mitigate interferences. In [39] a two-step execution model is introduced. The first step consists in loading the data in private memory components. By serializing these accesses, interferences are avoided. Each shared memory access time slice is defined by the system integrator in charge of configuring the system scheduling. Once the first step is finished, the partition can proceed to execute its tasks and process the data loaded in its private memory. The resulting system consists of two execution modes. The first is a sequential mode when all data are loaded in private partitions' memory component. The second is a parallel mode when all partitions executed their tasks. Once the partition finished its processing, it waits for its next loading slice to push the data to shared memory and load new data. Both modes are not mutually exclusive as multiple partitions can process the data while a partition loads its own data. However, it is impossible to have more than one partition in the loading phase. Such models have also been studied by [40].

In [18] a method to deploy interference-free applications on multi-core platforms is proposed. The toolchain allows the developers to compute applications WCET thanks to single-core methods. Each application must be mapped on cores before the use of this tool. Another execution model is presented to separate shared data handling and private data processing. The Acquisition Execution Restitution (AER) execution model, proposed in [41], distinguishes three fundamental steps.

- The first step is data acquisition where the partition loads all the data needed for its execution from the shared memory. Only one partition at a time can enter the acquisition phase.
- Once the acquisition step finished, the execution step starts and the partition processes the data previously loaded. Multiple partitions can enter this phase at the same time. Since the execution phase does not require shared component accesses, no interference can happen.
- The final phase is the restitution phase. During this step, the partition writes back all the data to the shared memory. As for the acquisition phase, only one partition at a time executes the restitution step.

During acquisition phases, all partitions (excepted the one doing the acquisition) must be in the execution phase or waiting. Same goes for the restitution phase when only one partition can access the shared memory. However, multiple partitions can be in the execution phase.

Most of the previous work describes scheduling methods to overcome interference occurrence when hardware support is not present. The AER (acquisition, execution and restitution) model is proposed in [41]. Thanks to this model, the data are brought and written back to private memory sequentially (acquisition and restitution). Then, the execution phase can use the private memory to compute while other cores can access the shared memory to execute their acquisition or restitution phases.

***

In this chapter, we presented previous work conducted to mitigate interferences in avionic systems. In this thesis we focus on private cache interferences. We choose this type of interference because it is not widely represented in the literature and their mitigation profits both single-core and multi-core platforms. We also expect this work to take part of a more general research, which purpose is to treat the complete memory hierarchy.

CHAPTER 3 THEORETICAL BACKGROUND AND METHODOLOGY

In this chapter, we will introduce the basic concepts and definitions relative to our research topic. The notion of hard real-time systems and critical systems will be explained as well as the cache and memory aspects in multi-core systems.

3.1 Definitions

RTOS (Real Time Operating System) A real Time operating systems (usually abbreviated RTOS) is a specific type of highly time constrained Operating System. Each tasks of a system must be executed in a certain amount of time, making the system fail if this requirement is not met.

Periodic tasks A task that has an execution period, thus starting its execution each time this period is elapsed. This type of task also has an execution deadline. The deadline is inferior or equal to the period.

Aperiodic tasks A task that has no particular period and can be triggered anytime in the system. Such tasks also have deadlines, relative to the start time of the task.

Deadline miss A deadline miss occurs when a task does not finish its execution before reaching its deadline.

Cache memory A fast but usually small memory, temporarily storing data to serve them faster the next time they are accessed.

Cache miss / cache hit A cache hit occurs when a data accessed by a program is present in the cache. A cache miss occurs when the data is not present in the cache and needs to be loaded from slower memory.

Cache replacement mechanism When the cache memory is full and a cache miss occurs, the replacement algorithm will select a “victim” in the cache to evict and be replaced by the newly loaded data.

Interference (in multi-core context) A hazardous system’s behavior introducing non-determinism in tasks execution time (jitters, delays, etc.).

3.2 Methodology

Interferences analysis in multi-core systems

The first phase of the project is based on a literature review on the subject of interferences and, more generally, on the subject of multi-core architecture in aeronautics. This phase will allow to familiarize with the very special subject of critical RTOS in multi-core environments. The study of published works will give a global point of view of the current state of the art in the domain. Gathering the information allows us to find issues to address in our research. We also expect to learn about ARINC-653, CAST-32 [42] and other standards used in aeronautics. The literature review will span into four steps:

- Global analysis of the problems encountered in multi-core architecture with RTOS;
- Categorize multi-core interferences present in multi-core architectures;
- Gather experimental metrics and measures to be taken into account during the research;
- List existing solutions and possible improvements to be made;

The project being in direct partnership with an industrial company[1], a particular attention will be taken to report already existing patents.

Analysis of existing mitigation solutions

Multiple ARINC-653 compliant RTOS already exist on the market and some are DO-178 certified. These products embed different technologies to isolate resources and partitions (time and space partitioning). Some manufacturers also provide new methods to mitigate the presence of interferences in multi-core systems. However none of these RTOS are certified for multi-core use. The current solution to run on such systems is to deactivate the additional cores to limit the RTOS only to run on one core. A review of the state-of-the-art solutions for interference mitigation will be conducted. Where the first step of the methodology was to study interferences in multi-core systems, the second phase is to gather all the existing solutions to mitigate these interferences. This phase consists in providing an overview of the different mitigation means to the industrial partner. We will also gather the different metrics and results used to compare the solutions. Once this step completed, a suite of benchmarks will be developed to achieve this comparison. This suite will be reused to validate and compare our mitigation approaches. In this thesis, we focus on private and shared cache interferences.

[1] MANNARINO Systems and Software

Cache Interference Mitigation

The third phase of the proposed work is the development of cache interference mitigation means. Based on the preliminary literature review, we expect to develop a new approach to limit or mitigate interferences in private and shared caches. Several studies can be conducted to achieve that goal:

- Mitigation of isolation-related interference (e.g., modification of partition content by another partition);
- Mitigation of delays induced by contention of shared resources(e.g., cache contention);
- Interference aware memory management in multi-core systems (e.g., deterministic memory allocation);

For each developed method, experimentation will be done to compare our approach to other solutions. We may also produce different mitigation means. However we expect all these solutions to be independent. To validate and characterize the improvement brought by our solutions, we plan to use multiple ARINC-653 compliant operating systems such as VxWorks-653 [43], or the open source POK [44].

Integration of the prototype solutions in an existing RTOS

Once the solution validated, we expect to implement our work in a proprietary RTOS developed by the industrial. During the development of such mitigation means, we will also support the company team to design the RTOS to support multi-core systems.

3.3 Basic Concepts

In this section we present the basic concept related to our research. The notion of cache memory is introduced as well as cache locking and partitioning. We also explain the concepts of isolation defined by the ARINC-653 standard.

3.3.1 Cache Memories

Caches were introduced to address the growing gap between memory speed and CPU speed [1]. They are small and fast memories attached to the CPU. Caches are organized by level, creating together with the main memory the *memory hierarchy*. The faster the cache, the more expensive it is. This leads to a cache of different sizes depending on the cache level. Figure 3.1 shows a generic cache memory hierarchy. The first level in the hierarchy, L1, is usually segregated into two different caches: one cache stores the instructions and the other the data. This is called the *Harvard* model. The next levels are usually unified caches, they store both instructions and data. Caches can be classified in two categories: private caches, which are only used by one core and shared caches that are used by multiple cores. A cache is in general modeled as a matrix where rows are called *cache sets*, columns are called *cache ways* and cells are called *cache lines*.

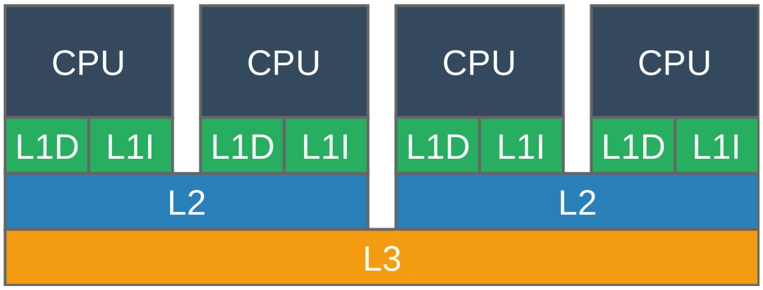


Figure 3.1 Example of cache hierarchy

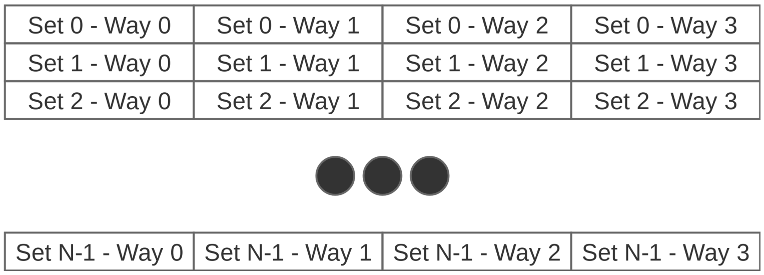
Cache associativity The mapping between memory addresses and cache lines can be achieved in different ways. The mean of mapping memory addresses to lines is called the cache associativity. Each address is mapped to a certain set. The way that will accept the data is determined by the cache replacement policy. The most used mapping policies are [45]:

- Direct mapping is the simplest mapping. The set of a line is computed by applying a modulo operation to the address. For a cache that contains n sets, the mapping operation is $address \mod n$ where $\mod$ is the modulo operation. Direct mapped caches contain one unique way. If data from another address is contained in the cache line, it is evicted and replaced by the new data.

- N-Way associative caches are composed of N ways. As for direct-mapped caches, the set of an address is computed with the operation $address \mod n$, where n is the number of sets in the cache. The way that will accept the new data is determined by the cache replacement policy, explained later in this section. Figure 3.2 shows a 4-way set associative cache.

- Fully associative caches are the most expensive caches to produce. Memory addresses can be mapped in any sets of the cache. These caches are actually composed of a unique set with multiple ways. Once again, the way that will be populated by the access depends on the cache replacement policy.

Other less popular policies exist such as N-Way skewed associative cache, pseudo-associative caches, etc [46].



Set 0 - Way 0	Set 0 - Way 1	Set 0 - Way 2	Set 0 - Way 3
Set 1 - Way 0	Set 1 - Way 1	Set 1 - Way 2	Set 1 - Way 3
Set 2 - Way 0	Set 2 - Way 1	Set 2 - Way 2	Set 2 - Way 3
Set N-1 - Way 0	Set N-1 - Way 1	Set N-1 - Way 2	Set N-1 - Way 3

Figure 3.2 Example of cache organization

Coherency In single-core CPUs cache lines can have two states.

- The valid state is used when the data has been loaded from the main memory and can be used by the CPU;

- The invalid state describes a line for which data needs to be loaded or reloaded;

At system startup, all lines of the cache are set to invalid. In multi-core systems coherency needs to be kept between the different private caches. Imagine the following scenario:

- Data A is loaded by CPU C1 and CPU C2;

- Data A is now contained in C1 private cache and C2 private cache;

- C1 modifies the value of data A, the modification occurs in its private cache;
- C2 now has an incorrect value of data A;

To avoid the case where a CPU's private cache contains incorrect data, cache coherence protocols were introduced. These protocols ensure a coherent state of the private caches. Shared caches must be kept coherent when not all CPUs of the system share the caches. This is the case for the level 2 (L2) caches in Figure 3.1. Multiple cache coherence protocols exist: MSI, MESI, MOESI, etc [47]. The description of such protocols is out of the scope of this thesis and will not be given.

Replacement Policy When all the ways of a set are used and a new data must be stored in the cache, the system needs to determine which way is to be replaced. This is the task that the replacement policy has to achieve. Multiple policies have been designed and each of them has a different goal [45]. Some of them may be used to increase memory reuse, others to decrease the impact of the replacement policy on determinism. In our approach, two replacement policies are used: Pseudo-Round-Robin and Pseudo-LRU. We choose to focus on these policies as they are the most commonly used [45].

Pseudo-Round-Robin is a simple and cost-effective replacement policy implemented in the PowerPC e200z7 processor. Thanks to a counter, the cache controller keeps track of the way to replace. When a cache miss occurs, the cache controller checks if the way pointed by the counter can be replaced (the way is eligible for replacement, i.e., it is not locked or disabled). The counter is increased until a replaceable way is encountered. If the counter reaches the maximal number of ways, it is set to 0. When the counter stops, it points to the way that will receive the new data. If no way to the set is replaceable, the cache is bypassed and the data retrieved from the main memory.

LRU (Least Recently Used) is a replacement policy that keeps records on the previously accessed ways. When a data needs to be loaded in the cache, the way that will be selected for replacement is the way that has been accessed the least recently. When ways are not eligible for replacement, they are not taken into account in the process of selecting the least recently accessed way.

Pseudo LRU is a variant of the LRU replacement policy. It aims to overcome the resource overhead that are needed by the LRU policy (counters to keep track of the least recently used way). This is an heuristic of the LRU algorithm based on binary search trees to lower the resource overhead. Multiple other policies exist that are aimed to increase different metrics such as data throughput, power efficiency, resource usage, etc.

Write policy are different ways to allocate and manage the data in the cache and the memory. We can distinguish two categories of write policies. The first category is data write back to the memory. When a data needs to be written back to the main memory or lower level caches, the cache controller needs to know when to write back this data. Two choices are possible:

- Write through policy defines that in case of cache hit or miss, the data is modified in the cache (in certain cases, described later in this section) and directly written back in the memory.
- Copy-back policy tries to improve data throughput by only writing data back when needed. Data is copied to the main memory or lower cache levels when the line containing the data is evicted or when the cache coherence protocol needs it. When a data must be evicted, it is only written back when modified (also called dirty).

Associated with write policies, write allocation policies are different and can be used with any write policy. Again, two choices are available:

- No write-allocate policy do not allocate a cache line on write miss. When a write miss occurs, the data is directly written in the main memory or the lower cache levels.
- Allocate on write policy allocates a new line on write miss. The data is loaded in the cache and modified directly in it. The allocation policy is the same as data read or instruction fetches.

3.3.2 Address representation in the cache

Caches have different representations of memory addresses compared to the CPU of the main memory. By separating an address in different blocks, one can compute in which set of the cache an address will be stored. Figure 3.3 shows how to separate an address in different parts. The first block, starting from bit 0 represents the offset of the address in a cache line. The size of this address block depends on the cache line size. In our example, 5 bits are used to represent the offset in the cache line since we have 32 bytes cache lines. The next part of the address is the set ID. This section starts at the end of the offset section and its size depends on the number of sets available in the cache. In the example we provide, the cache has 512 sets, which makes the set ID section of the address of the size of 9 bits. The rest of the address is called the tag. The tag is used to decide if the address requested by the CPU is present in the cache or not. The cache hit detection mechanism is out of the scope of this thesis and will not be further explained.

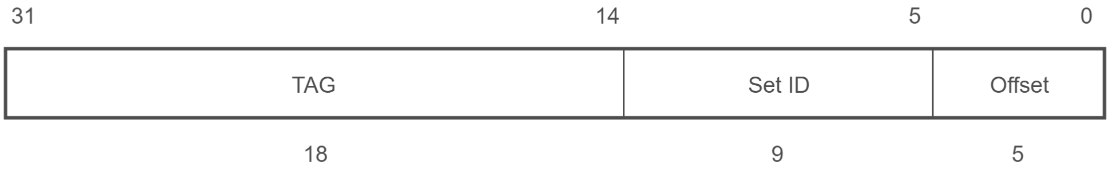


Figure 3.3 Address representation as seen by the cache

3.3.3 Cache Locking and Partitioning

Cache management methods give the user the possibility to modify how caches behave. This could be done to improve the system's performances, reduce non-deterministic behaviors or avoid data to be stored for different reasons. In this thesis we discuss two of them: cache locking and cache partitioning.

Cache locking consists in locking instructions or data in the cache. When a new data has to be loaded in the caches, locked lines are not taken into account in the set of candidates for eviction. This means a locked line will never be evicted from the cache until it is unlocked. Locking lines could be used to improve kernel services delivery time for instance. It has to be noted that a locked line can be invalidated. In that case the line will have to be reloaded from lower memory levels. Such a case can happen when two or more cores use the same data.

Cache coherency protocols are able to change lines coherency state without taking into account the line's lock state. Cache partitioning has a different impact on caches. Partitioning the cache is a more abstract concept and is usually managed by software means, contrary to cache locking that is hardware reliant. Partitioning the cache means allocating an area of it to one or multiple applications such that other applications cannot access this area. This is done by allocating specific virtual addresses to the different applications. Depending on the page size, a portion of the page number overlaps with the set index. The Operating System (OS) memory manager can tweak that portion of address to select in which sets the data contained in the page will be loaded. An example of the method is given in Figure 3.4. For this example, the cache has 512 sets and is 8-way associative. Cache lines contain 32 bytes of data. We are using 4 kilobytes pages. With this setting, bits 12 and 13 are contained in both the cache set ID portion of the address and in the page number portion. This allows the memory manager to split the address space into four partitions. Partition 0 will store data of pages for which the page number ends with bits 00, partition 1 will receive data for which the page number ends with bits 01 and so on. Reducing the page size will allow the user to segregate the cache in more than four partitions as the offset in the page portion of the address will be smaller. The same effect will be observed if the number of sets in the cache increases.

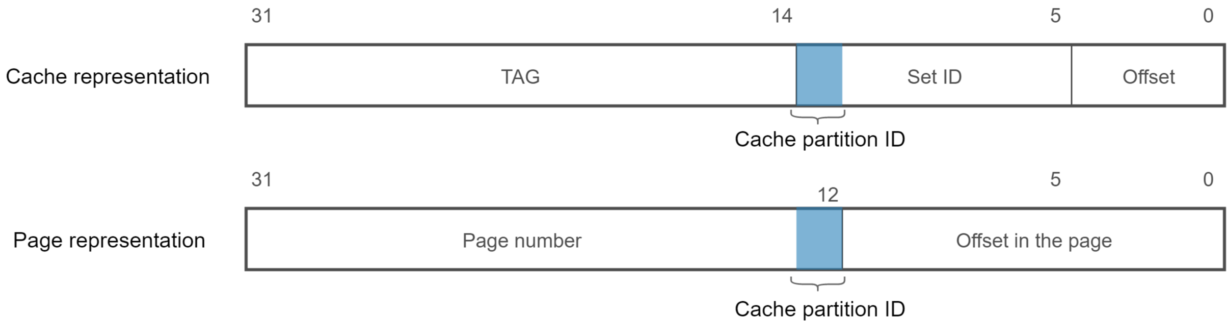


Figure 3.4 Virtual memory address compared to cache representation address

3.3.4 Integrated Modular Avionic

Integrated Modular Avionic (IMA) architectures have been developed since 1990. They were designed to solve multiple issues of the previous architectures called federated architectures. Federated architectures were designed such that each computer assisted features of a plane were deployed in separate units. These units were able to communicate through multiple networks of avionic buses. However this architecture is difficult to maintain and communications between the units became less efficient as the number of functionalities increased [48]. With the constant increase of computer-assisted features, federated architectures were not able to fulfill the needs of avionic system designers [48]. Figure 3.5 shows a simple federated architecture with five distinct tasks that execute on their own hardware. IMA is designed to regroup multiple functionalities of the plane in one single Line Replaceable Unit. Each functionality runs in a partitioned environment. This ensures that all the tasks executed by the system are independent and cannot interfere with each other. This architecture has the advantage of reducing the amount of equipment required in the plane, their weight and their power consumption [48][2]. Communication between the different partitions is made through

[2] This concept is known as Size Weight and Power (SWaP).

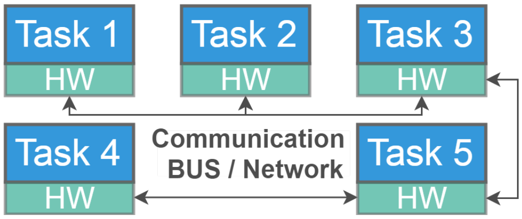


Figure 3.5 Example of federated architecture

shared memory, which increases the communication speed and reduce the workload on the avionic buses and networks. Figure 3.6 is an example of IMA architecture designed to make the transition from the federated architecture presented in Figure 3.5.

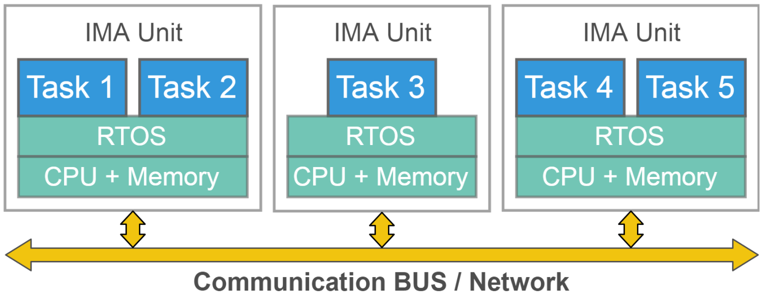


Figure 3.6 Example of IMA architecture

3.3.5 Design Assurance Levels

Due to their criticality, avionic systems must be certified by competent authorities. These organizations base their work on multiple standard such as the DO-178 guideline document. This guidance is developed by the Radio Technical Commission for Aeronautics (RTCA) and the European Organization for Civil Aviation Equipment (EUROCAE) since 1992 and is regularly updated. The last revision of the document is DO-178C. DO-178 guidance describes a set of objectives in order to certify a product. Five criticality levels are defined to classify software by their level of criticality:

- Level A applications are defined as applications for which failure may lead to catastrophic events (possible crash of the aircraft, flight safety compromised, potential casualties, etc.);

- Level B applications failure may induce serious problems, leading to few casualties;

- Level C failure drastically drops in terms of severity compared to level B as failure of such application may provoke serious dysfunction to fundamental airplane functions, causing passenger discomfort or increase the crew workload;
- Level D failure is considered as minor;
- Level E failure does not impact safety;

When an RTOS is aimed to host mixed criticality software, the RTOS should be certified to the highest level of application the RTOS will host. Failures as described in the guidance are undefined behaviors, erroneous output, deadline misses, etc. For each DAL, a number of objectives are defined [49]:

- **DAL A**: 71 objectives defined;
- **DAL B**: 69 objectives defined;
- **DAL C**: 62 objectives defined;
- **DAL D**: 25 objectives defined;
- **DAL E**: 0 objectives defined;

The highest DO-178 levels enforce traceability in the software. This means that each line of code (and for higher DAL, each assembly instruction) must be traceable to a requirement of the system. The object code must be traceable to the source code, source code must be traceable to lower-level requirements which are traceable to higher-level requirements that are refined from specification documents.

3.3.6 ARINC-653 and Isolation

The Avionics Application Standard Software Interface (ARINC)-653 standard is a guideline document to design partitioned systems. This document is used in avionics to isolate multiple applications that run on the same system. This has the advantage to avoid side effect when one application crashes. Each independent environment of the system is called a partition. A partition contains processes that execute as if they were the only processes in the system. Each process shares the same address space and are scheduled with real-time schedulers. An analogy can be made between POSIX systems and ARINC-653 systems: the ARINC partitions are POSIX processes and ARINC processes are POSIX threads. The standard defines how to achieve this partitioning [2] using the concepts of time partitioning, space partitioning, application executive and health monitor.

Time partitioning enforces the system's partitions to run during a limited amount of time at strictly defined periods. This principle allows the processes running in the partition to "think" they are the only ones running in the systems. Time partitioning defines the notion of time window and major time frame.

- A time window is a span of time during which an application runs. A partition can have multiple time windows of different lengths. During its execution, a partition has exclusive access to the CPU.
- The major time frame is a succession of time windows that repeats itself through time. It defines how partitions will execute and ensures no overlapping is possible.

Figure 3.7 shows an example of major time frame. In this setting, partition 1 has two time windows of 5 ms and 10 ms. The major time frame can contain gaps, during which no partition runs. This is the case from 15 ms to 25 ms. When the end of the major time frame is reached, the system repeats the pattern.

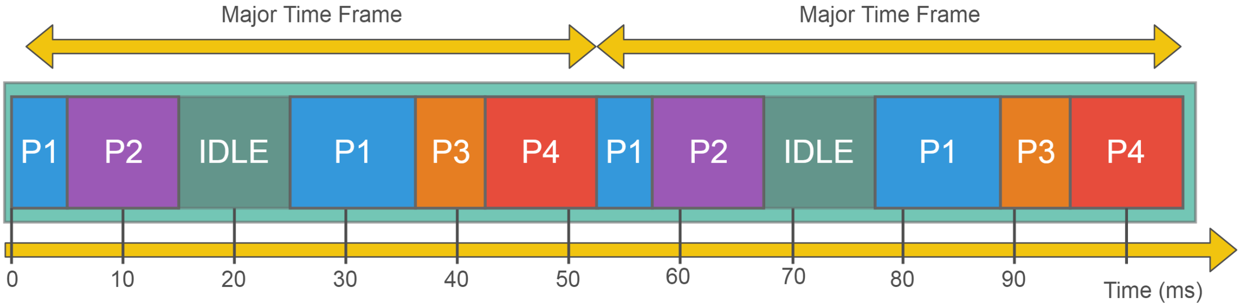


Figure 3.7 Example of a major time frame

Space partitioning or resource partitioning is the second aspect of partitioned systems. It allows reserving resources for each partition and isolates these resources from partitions that do not use them. This ensures that a partition can't access resources of another partition and modify its behavior. Resources can also be shared by different partitions when needed (e.g., screen, shared memory used to communicate, etc.). Resource partitioning applies to memory in the system. Memory isolation can be done by the use of Memory Management Unit (MMU), Memory Protection Unit (MPU) or other memory protection means. Other resources such as IOs or peripherals can be reserved for one or more partitions. To this end, some systems propose IO-MMU. One may consider bandwidth limitations as resource partitioning. In single-core processors, only one partition can use any resource at a time. This ensures exclusive use of a resource during the partition's time window. However, in multi-core systems, this rule does not apply. Multiple partitions can execute in parallel, which means that two or more partitions can access a resource at the same time. This is obvious for the memory, but IOs and peripherals also suffer from this.

APplication EXecutive (APEX) also known as APEX is a set of standard interfaces with the RTOS. The Application Programming Interface (API) increases the portability of ARINC-653 compliant applications. Each function of the system can be used with the APEX such as process creation, IO and peripheral accesses, time management, synchronization primitives, inter and intra-partition communications, etc.

Health Monitor (HM) or HM is a background task that ensure the correct behavior of the system. The HM is in charge of handling errors and exceptions that occur when a partition is executed. It allows the user to associate a handler to each error that can occur. For instance, when a deadline miss is detected in a partition, the HM can restart the partition or trigger a routine to recover from the error.

3.3.7 Interference in Multi-core Architectures

Interferences are defined as unwanted behavior caused by resource sharing in multi-core systems. Examples of such behavior are: time delays, execution time jitters or data corruption. This is indeed not acceptable in critical systems where timing must be ensured and execution time strictly bounded.

Interferences are present in both single-core and multi-core systems. An example of single-core interference is when two partitions evict each other's data in private caches. To avoid this interference, private caches can be flushed at each partition switch [50]. That way, the state of the cache when a partition is scheduled is always the same.

Multi-core platforms can execute multiple tasks in parallel. Each processing unit (or core) has its own execution environment and does not share its private resources such as CPU registers, private caches, etc. However multiple resources are shared between these cores. Shared caches are used by multiple cores at the same time. This allows faster inter-core communication and reduces the number of different components in the system. Buses are used by multiple cores to access the memory of the IOs, and the memory itself is shared among them. Figure 3.8 shows an example of interference-prone architecture where interferences are marked in red. Shared resources where interferences occur are called interference channels.

Most resources that are accessed concurrently cannot serve all the requests at the same time. Controllers are in charge of selecting the request to serve by sequencing all the requests. Selecting the next core to serve can be done in a fair way (e.g., first come first served) or unfair way (e.g., Core 0 always has the priority on other cores). Such strategies will enforce stalls on cores waiting for their request to be served. This is where one type of interference occurs. Core execution's gets delayed because of the contention on shared resources. A principle of partitioned systems is broken when this condition occurs. Interferences are evidences that a partition executing on a core can modify the behavior of a partition executing on another core.

We classify contention on shared resources as direct interference. This category regroups all interferences that can be observed and modify the behavior as they occur. Even when concurrent accesses on shared resources are avoided, interferences might be present. In [7] caches are described as the most interference-prone resources. When a core modifies the content of a cache that is shared with another core, it might evict data that were used by this second core. In that case, the second core will have to reload the data from the memory. This will result in execution time delays for the partition that executed on this core.

The behavior when partitions evict other partition's data in a shared cache is classified as an indirect interference. In that case, the delay induced by the interference is only observable when the partition tries to access the evicted data. It is even possible that the data will never be accessed again. In that case the interference will not induce any delay in execution time.

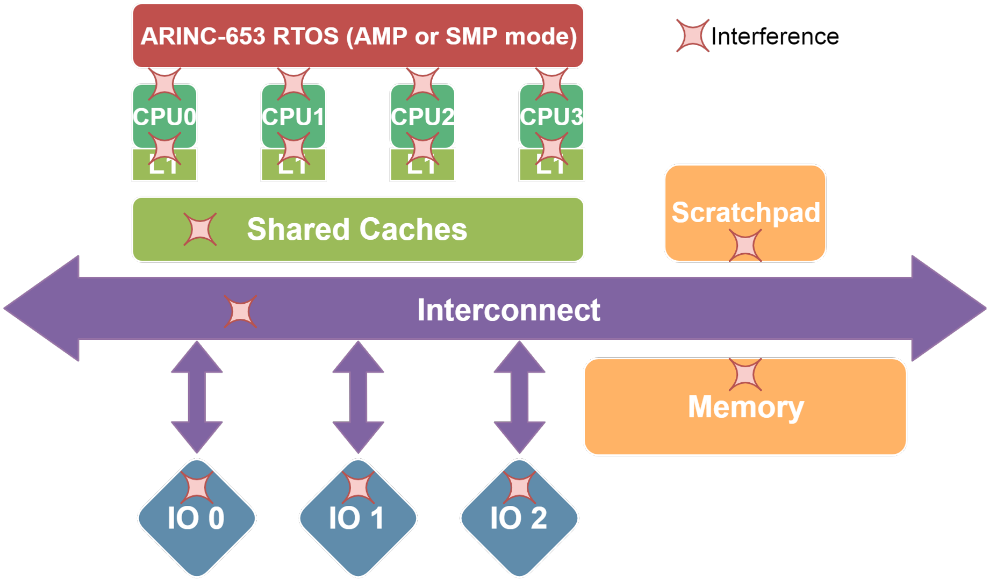


Figure 3.8 Interference-prone architecture

***

In this chapter, we presented the different types of interferences occurring in single-core and multi-core architectures. We also defined the basic concepts used in this thesis. In the next chapter, we discuss the state of the art solution proposed to mitigate those interferences.

CHAPTER 4 CACHE LOCKING FRAMEWORK

In this section, we present the memory tracing framework. This is a solution to mitigate interferences in multi-core environments. We can also use it in single-core systems to provide better performances.

4.1 Framework Goal

The framework is designed to be modular. This decision allows removing, adding or replacing different features of the framework. It also eases the communication between the different modules. Each of these modules will have a specific task to execute in the framework. The next section presents the different modules in detail. Modularity allows parallel execution of different tasks and group heterogeneous tools in only one entity. This increases the ease of use of our method and reduces the possibility of misuse of the different components. While the presented work only relies on cache locking algorithms (as discussed in Chapter 5), the algorithm using traces processed by the framework can be exchanged with any other algorithm (e.g., memory profiler). Communication between the components is done through shared memory to limit the performance loss. Each component is developed to be hot swappable[1] libraries running in their dedicated thread.

Figure 4.1 shows the overview architecture of the framework, and its different modules are presented in the next section. Communications between the modules are represented by links between them. We integrated an ARINC-653 partition scheduler simulator in the framework. This module is useful when processing multiple files. When switching from a trace file to the next one, the ARINC-653 scheduler also switches to associate the current trace file to the currently running partition. To manage multi-core architectures, we associate one ARINC-653 scheduler per core. An effort has been done to allow parallelism of the different tasks of the framework. As we rely on ARINC-653 concepts, each partition memory trace can be processed independently when they are not running in parallel. This notion of independence between partitions is very important. Indeed, when running the memory analysis for two distinct partitions, if one partition is modified, the analysis does not need to be run again for the unmodified partition. However, in multi-core systems, the user will have to take particular care of this assumption. In an environment where multiple tasks run in parallel, the traces used by the framework will have to be contained in the same file for all partitions running in parallel. All the settings used in the framework (ARINC-653 scheduler, algorithms

[1] Components can be exchanged during execution

used to process traces, cache simulator models, etc.) are loaded from configuration files. This allows the framework to be set to fit any architecture the RTOS runs on. Figure 4.1 gives an overview of the framework. The data flow starts with trace files from the RTOS. Such files can be generated with a modified version of Qemu or a probe that can trace access from the actual hardware. These traces are processed by a cache simulator and injected in the algorithm embedded in the framework. This algorithm outputs the solution and two performance analyzers (one before processing and one after) are in charge of evaluating the performances gain or any metrics used to quantify the improvement brought by the solution. Each module of the Figure 4.1 is defined in the next section.

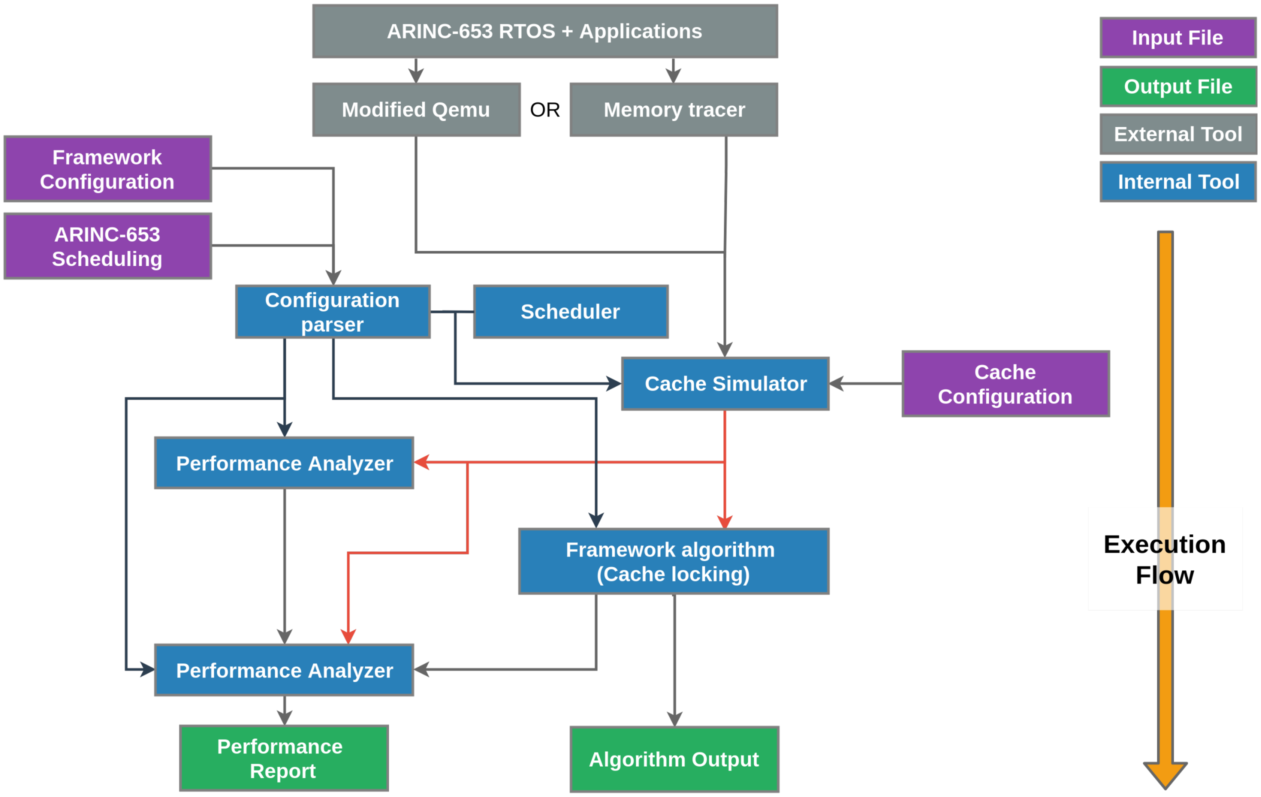


Figure 4.1 Cache Locking Framework

4.2 Modules Definition

Each module of the framework can be seen as a library. The core of the framework instantiates and links the libraries together. The communications and execution flow is controlled by the core itself. In order to know the current stage of the process, each module must communicate its state to the core. This section describes each module presented in Figure 4.1.

Memory tracer The memory tracer registers all accesses made by the CPU to the memory. When tracing the accesses, caches are not considered. This choice allows the framework to implement its own flexible cache simulation tool. We defined two methods to gather the traces. The first option is to use a modified version of Qemu [51]. However this approach raises issues concerning the notion of time with the emulator. These issues are discussed later in this thesis. We also use external probes to gather the traces from the actual hardware on which the RTOS is aimed to run. Multiple tracing probes exist such as the Trace32 from Lauterbach[2]. Later in this section, we define the trace format as well as how we instrumented Qemu to output such traces. However we will not focus on traces gathered from the probe as each probe has its specifications. To stay compatible with the framework, for each tracing probe we use, we convert the probe's traces to the framework's trace format.

Configuration Parser To manage the different settings used in the framework we defined a configuration parser. This module takes multiple Extensible Markup Language (XML) files that contain the different settings available for the framework. The following options can be set to fit the user's needs:

- Type of algorithm used in the framework (e.g., *CacheLockingGreedy* to instantiate the greedy version of the cache locking algorithm presented in Chapter 5 of this thesis).
- Number of major time frame to take into account during the process. Each trace used by the framework represents the execution of an application during one major time frame.
- Scheduler configuration file path. This parameter points to the file that contains the ARINC-653 scheduling schema. This file is later used by the ARINC-653 Scheduler module.
- Cache simulator configuration file path. This option points to the file that contains the cache simulator configuration file.
- The trace gathering method. This option is used to choose between Qemu and a probe for the execution traces provider.
- The directory containing the trace files.
- The directory where you write the files output by the algorithm.

The configuration manager also takes an ARINC-653 scheduler configuration file. This type of file is defined in the next paragraph.

[2] https://www.lauterbach.com/frames.html?home.html

ARINC-653 Scheduler As previously stated, each trace file corresponds to a certain time window, during which at most one partition executes on each core of the system. To associate each trace file with the correct partitions, we developed an ARINC-653 scheduler simulator. The modules take an XML configuration file describing the scheduling schema. Then, for each trace file to process, the simulator computes to which partition the trace file is associated with. This method is useful when the algorithm used in the framework needs to know which partition has produced the currently processed traces. The simulator focuses on ARINC-653 partitions scheduling. However it does not simulate the intra-partition scheduling of ARINC-653 processes.

Cache Simulator To analyze the transactions going through the memory bus, we integrated a cache simulator to the framework. For each access, the simulator is in charge of identifying the cache where the access missed and hit. Each level of cache is represented and for each of these levels, the access result (hit or miss) is reported. The simulator is modular and accesses various models of caches. This allows us to represent any cache architecture we need for our tests. The following parameters can be set:

- Cache hierarchy (number of levels, caches per level, type of cache);
- Cache size;
- Cache organization (number of ways and number of sets);
- Line replacement policy (FIFO, Pseudo-LRU, Pseudo Round Robin, etc.);
- Cache access latency (hit latency, miss latency);
- Address width (32 bits or 64 bits);
- For multi-core architectures, the cache coherence protocol can be set (None, MESI, MOESI, etc.);

In this thesis we only use the cache simulator as a tool capable of computing the access hit and miss in the system. This simulator was developed in our research group by J.B. Lefoul.

Performance Analyzer A performance analyzer module has been developed in order to compute the performance before and after running the algorithm. Various metrics can be used in the analyzer. The following list gives the metrics we used in this thesis.

- Number of hit/miss per cache;
- Total cache miss penalty (number of cycles the CPU was stalled waiting for data);

- Number of read/write that produced a hit/miss per cache;
- Number of miss entailed by the eviction of a data by another partition;

This list can be modified to add new metrics to fit the user's needs.

Figure 4.2 shows the analyzer's execution flow. The performance analyzer first runs the simulator with the traces provided by the user. For this first run, no additional configuration is given to the simulator. The framework's algorithm is then run and once the algorithm output is generated, the performance analyzer launches the cache simulation again. This time the algorithm's output is provided to the cache simulator. In our case, cache locking configuration files are provided to the cache simulator. This has the effect to lock the lines in the simulator and gather the performance improvement provided by the current locking scheme.

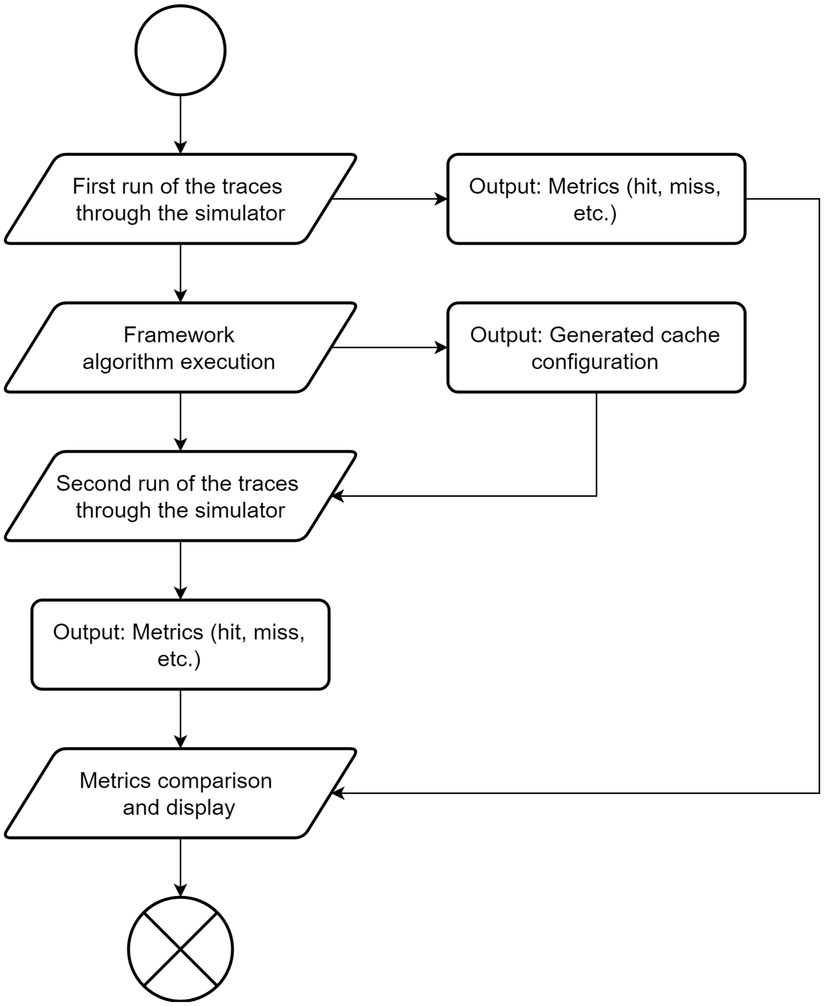

First run of the traces through the simulator
Output: Metrics (hit, miss, etc.)
Framework algorithm execution
Output: Generated cache configuration
Second run of the traces through the simulator
Output: Metrics (hit, miss, etc.)
Metrics comparison and display

Figure 4.2 Performance Analyzer execution flow

Framework Algorithm Any algorithm can be developed to fit in the framework. Such algorithms can use the cache simulator to gather memory access information or use plain traces to perform its own profiling or analysis. In our approach we use cache content selection generators. A selector will choose multiple cache lines to be locked in the cache to improve performances, predictability and reduce shared cache contention in multi-core environments. The algorithms used in this thesis are presented in Chapter 5. Algorithms such as cache partition generation algorithms, memory profilers, memory bus analyzer or other trace related profiling algorithms can be used in our framework. The algorithms' output can be defined by the developer to fit its needs. In the framework, all the tools linked to the algorithm (performance analyzer, cache simulator) can be easily modified to fit the needs of the algorithm. All modules of the framework are provided as libraries and the algorithm designer only has to include these libraries in his project to use the framework.

4.3 Input and Output Definitions

The framework has multiple sources of information defined as **inputs**. The type of data and their quantity may vary depending on the needs of the user. We can divide these input data in two categories:

- **The configuration data** files used to configure the framework itself. These files are represented in Figure 4.1 as FRAMEWORK CONFIGURATION and ARINC-653 SCHEDULING CONFIGURATION FILE (this file gives the scheduling scheme used by the system). The internal configuration manager parses and stores the different information present in these files. The framework configuration can be divided in multiple files. Cache configuration files are also defined to be read and processed by the cache simulator itself. These files allow the user to describe the cache model with the different settings detailed in section 4.2. To ease the process of configuration, all the configuration files use the XML format and are easily modifiable by the user.
- **The trace data** files are used to profile the memory in the system. Such files contain the different traces produced by the memory tracers. An effort was made to standardize the trace format. This enforces compatibility with the cache simulator. The compliance to this standard is left to the user. In our approach, we defined two trace sources details in the next section of this chapter. To reduce the memory footprint and the trace generation time, we choose to use plain binary files to describe the memory traces.

The **output** files in Figure 4.1 are the result of the algorithm processing. The first generated file is the performance analyzer report. This report contains information on the performance improvement concerning the metric selected by the user. Our report contains the cache misses and hits for each cache before and after the content is locked in the cache. We also provide a comparison between the two measurements and compute the improvement brought by the method. The second group of files generated by the framework is the direct output of the algorithm. The format of these files is left to be defined by the user.

4.4 Memory Tracing

The framework relies mainly on memory tracing. The trace information will give the cache simulator and the different algorithms the information they need to profile the system's memory accesses. An effort has been made to reduce the memory footprint of the data as well as the processing time needed to generate and parse the traces. In this section, we present our memory trace format, we define the means used to generate these traces and the limitation of our solution.

4.4.1 Trace Definition

The memory traces are stored in a file that comprises a header and the traces definitions. Figure 4.3 shows the organization of a trace file. The header occupies the first 16 bytes of the file and gives information about it. Then follows the memory access traces stored in order of occurrence in the system.

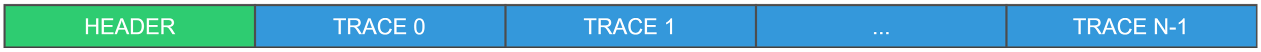


Figure 4.3 Trace file representation

The header shown in 4.4 is 16 bytes long and is organized as follows:

- Bytes 0 to 7 give the trace file size in bytes;
- Bytes 8 to 9 give the size of one trace in bytes;
- Byte 10 gives the trace files version used by the parser to verify the compatibility with the traces;
- Bytes 11 to 16 are reserved for future use;

Each memory trace is defined as a structure of 20 bytes for 64 bits memory traces and 16 bytes for 32 bits memory traces. A memory trace, shown in Figure 4.5, is composed of the

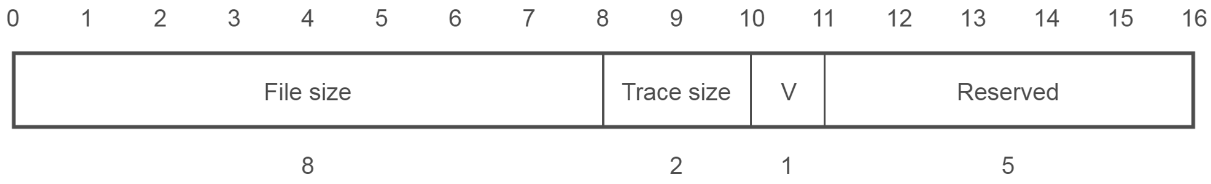


Figure 4.4 Trace format header

access address, the time of the access in the system and other meta data used to process the trace. The different fields of the memory trace structure are:

- Bytes 0 to 3 give the access address (for the64-bit version, this field extends from byte 0 to byte 7);
- Bytes 4 to 11 give the access time stamp (for the 64-bit version, this field extends from byte 8 to byte 15);
- Bytes 12 to 16 give the access flags (for the 64-bit version, this field extends from byte 16 to byte 19);

To allow the cache simulator or any other algorithm to fully understand the environment in which the access is made, we added a flag field to the structure. This field is a group of bits, defining different environment specific data. The following flags are available:

- Code ID is the core index on which the access was initiated. This is particularly useful in multi-core environments to determine the first cache that will be impacted by the access.
- W is the read/write flag. When set to 1, the access is a write operation; otherwise the access is a read operation.
- I is the instruction/data flag. When set to 1, the access is an instruction fetch/write; otherwise the access is a data read/write.
- U is the protection level flag. When set to 1, the access was generated while the CPU was in user mode; otherwise the access was generated while the CPU was in kernel mode.
- Type is the access type flag. On specific architectures, some instructions generate special access such as flush operations. The Type field allows to define the following access types: 0 = access, 1 = flush, 2 = invalidate, 3 = flush/invalidate, 4 = lock, 5 = unlock, 6 = prefetch type, 7 = zeroize.

- G is the granularity flag. For some access such as the invalidate operation, the granularity can designate: 0 = cache line, 1 = cache set, 2 = cache way, 3 = global cache.
- S is the access size flag. The size of an access can be: 0 = one byte, 1 = two bytes, 2 = four bytes, 3 = eight bytes.
- C is the coherency required flag. For some accesses, cache coherency is not required. When coherency is required, C is set to 1; otherwise its value is set to 0.
- L is the level flag. For some operations such as the flush operation, the cache level is needed to determine which cache will be impacted by the operation. The values can be: 0 = all levels, 1 = level 1, 2 = level 2, 3 = last level cache.
- CI is the cache inhibited flag. When set to 1, the access should bypass the cache without affecting it; otherwise CI is set to 0.
- WT is the cache policy flag. When set to 1, the cache should behave as if write-through policy was enabled, if set to 0, the cache should behave as if it was in copy-back mode.
- E is the exclusive flag. When set to 1, the access should modify it corresponding cache line coherency value to EXCLUSIVE.

Bits 24 to 31 of the flags field are not defined and can be used to output more information in the future.

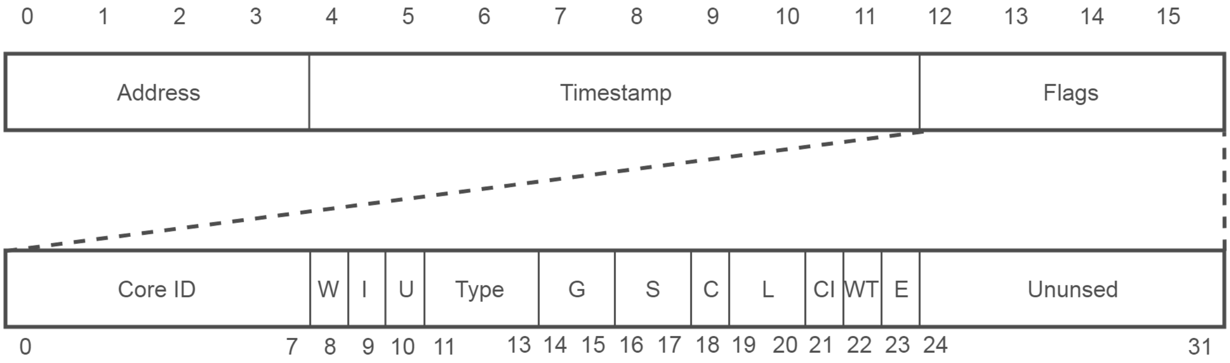


Figure 4.5 Trace format definition

4.4.2 The proposed approach for QEMU as a memory tracer

To be able to trace memory accesses of a partitionm we instrumented Qemu [51]. Qemu is a fast emulator capable of emulating many guest architectures such as PowerPC, X86, ARM, Sparc, etc. Qemu was created by Fabrice Bellard and presented in 2005. It does not only emulate the guest CPU but also the hardware such as serial interfaces, chipsets, etc. The emulator offers the possibility to only emulate user mode, to run applications designed to run in this environment. More works are conducted to make it faster and a Linux kernel module called KVM allows Qemu to use virtualization to improve its performances [52]. To perform fast emulation, Qemu relies on code generation on the fly to translate the guest program instructions to the host instruction types. Qemu parses each instruction executed by the guest system. During the parsing process, Qemu generates emulation code that can be executed on the host system. This means that the code is not executed directly when parsed by Qemu, but later, during the execution of the emulator. This method allows Qemu to generate translation tables that will be used to speedup the next instruction translation. This table works like a cache that stores the last translated instructions. When Qemu encounters an instruction that has already been translated, it reuses the translation generated in the translation table. We used this translation to our advantage to add tracing code during the execution of the guest.

Figure 4.6 shows the execution flow of our method. During the code generation process, we inject additionally generated code to perform a call to our trace engine. This way, when the generated code is executed, a specific call is made to our trace engine that gathers the machine state (CPU registers values, MMU state, etc.) and generates the trace defined in the previous section. At execution time, the trace engine accesses the MMU to gather the translation between virtual and physical addresses. Other information is gathered from the MMU such as the cache inhibition state of the access, the coherency flags or the cache policy flags. The CPU registers are read to collect other data such as the accesses in kernel or user mode, the cache state (enabled/disabled) etc. Data access tracing is done such that when a data access instruction (eg. lwz, stw, lwz[3] is being processed by Qemu, we have our specific call to the code generated by Qemu. This call is issued when the generated code is executed. For instruction fetch tracing, the process is the same, but instead of only tracing a subset of instructions, we trace all the instructions of the ISA. With our approach, when the processed instruction performs a memory access, one trace is generated for the instruction fetch and another is generated to trace the data memory access itself. We instrumented Qemu to trace memory accesses on X86 and PowerPC architectures. To validate our approach we developed a minimalist kernel that we called MiniKernel. This kernel was ported on PowerPC e500 and X86 CPUs and allowed us to tightly configure the hardware it ran on. Minikernel is a kernel-mode only executing OS. It allows paging, memory partitioning through privilege management and mono-tasking execution mode. During the execution of MiniKernel we were able to check if the memory accesses were correctly traced

[3] lwz: load word, stw: store word on PowerPC architecture

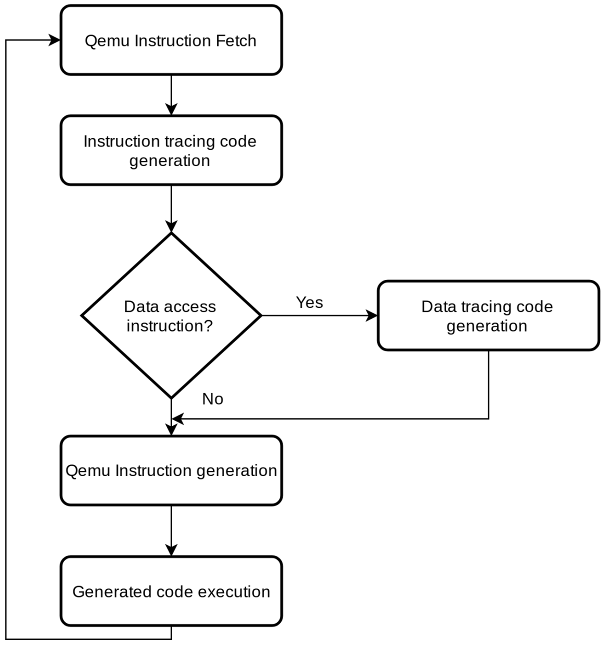

Qemu Instruction Fetch
Instruction tracing code generation
Data access instruction?
Yes
Data tracing code generation
No
Qemu Instruction generation
Generated code execution

Figure 4.6 Qemu translation process

in multiple conditions (MMU enabled/disabled, kernel/user mode).

4.4.3 QEMU Limitation and Proposed Solutions

Qemu is a fast emulator but is not appropriate for real-time systems, where faster[4] executions are required. Since time is not simulated, most real-time systems will detect timing anomalies. This is what happens with ARINC-653 compliant systems. When a deadline miss is detected, the partition will restart. This leads to impossible tracing of ARINC-653 executions on Qemu. Moreover, tracing added an overhead of 200% to the emulation time, which increased the rate of deadline misses. To solve this issue, we made the choice to prototype our framework and algorithm on traces that we gather on hard real-time systems and validate our approach on actual hardware with tracing tools plugged to the system (e.g., JTAG tracing probe).

4.5 Design choices

Prior to the implementation of the framework, we made the decision to keep the software as extensible as possible. The use of design patterns allows us to replace each module by any other module respecting the API defined by the framework.

[4] Read “real time”

4.5.1 Programing Language

We selected C++ as the programming language to develop the framework. C++ is an object-oriented programming language. This goes with our philosophy of extensibility as object-oriented programming facilitates the use of well-known design patterns (strategies, factories, etc.). The use of such design patterns permits a better modularity of our framework. The second reason for choosing C++ is its community. Many developers use C++ to develop software nowadays. Moreover, most real time operating system developers will use C or C++ to develop their OS. While using the framework does not require any programming knowledge, choosing C++ enlarges the public that will be able to extend the framework. Finally, we needed a language capable of producing optimized and fast code. Interpreted languages such as Python would not meet our performance requirements. The same remark applies to Java, since its virtual machine would slow our processing flow.

4.5.2 Performance Improvements

Due to the overhead added by tracing on Qemu, deadline misses make tracing impossible. To solve this issue we linked Qemu directly to the cache simulator. For this purpose, we used a shared memory section between the cache simulator and Qemu. This method is often used in co-simulation to increase the performances of the system simulation. Instead of storing the data to the hard drive, we create multiple shared buffers between the two applications to allow faster communication. When this method is used, the overhead drops from 200% to 55%. However, this solution reveals to be slower when the traces are meant to be parsed more than once in the framework. When we store data, the trace generation time is greater than the trace generation time when using shared memory. However when using shared memory, the emulation must be executed every time the traces have to be parsed. Since the generation time is longer (by a factor of at least 10) than the trace parsing time, it is better for the user to use the trace storage feature when in need of parsing the data more than once.

***

In this chapter, we presented the framework we used to mitigate interferences in single-core and multi-core architectures, we defined the set of modules that the framework embeds and their use. In the next chapter, we present a cache content selection algorithm that has been developed to be used in the framework.

CHAPTER 5 ARTICLE 1: CACHE LOCKING CONTENT SELECTION ALGORITHMS FOR ARINC-653 COMPLIANT RTOS

In this chapter, we present the cache locking algorithms we designed. Both algorithms are used to select the lines to lock in the cache to reduce the number of cache misses. This chapter is a paper submitted at the CODE+ISSS 2019 conference.

Cache locking content selection algorithms for ARINC-653 compliant RTOS

Alexy Torres Aurora Dugo
alexy.torres-aurora-dugo@polymtl.ca
École Polytechnique de Montréal
Montréal, QC, Canada

Jean-Baptiste Lefoul
jean-baptiste.lefoul@polymtl.ca
École Polytechnique de Montréal
Montréal, QC, Canada

Felipe Gohring de Magalhaes
felipe.gohring-de-magalhaes@polymtl.ca
École Polytechnique de Montréal
Montréal, QC, Canada

Gabriela Nicolescu
gabriela.nicolescu@polymtl.ca
École Polytechnique de Montréal
Montréal, QC, Canada

Dahman Assal
dahman.assal@mss.ca
Mannarino Systems & Software Inc.
Montréal, QC, Canada

5.1 Abstract

Avionic software is the subject of stringent real time, determinism and safety constraints. Software designers face several challenges, one of them being the interferences that appear in common situations, such as resource sharing. The interferences introduce non-determinism and delays in execution time. One of the main interference prone resources are cache memories. In single-core processors, caches comprise multiple private levels. This breaks the isolation principle imposed by avionic standards, such as the ARINC-653. This standard defines partitioned architectures where one partition should never directly interfere with another one. In cache-based architectures, one partition can modify the cache content of another partition. In this paper, we propose a method based on cache locking to reduce the non-determinism and the contention on lower level memories while improving the time performances.

5.2 Introduction

Critical embedded systems, more specifically airborne systems, rely on processors with demonstrated reliability through time. These processors usually implement less complex technologies than our everyday computer processors. It is difficult for critical system designers to rely on new Commercial Off-The-shelf (COTS) processors. Their complexity and optimization techniques come at the cost of potential uncovered hardware issues or bugs. In the case of safety-critical systems, such behavior is not acceptable as the presence of unknown or unpredictable events can lead to casualties [20].

Nowadays, multiple applications run on the same Integrated Modular Avionic (IMA) system [53]. Specialized Real Time Operating Systems (RTOS) have been designed to prevent these applications to interfere between each other. The result is a *partitioned RTOS* that provides complete isolation between applications and allows them to run as if they were the only program on the system. Standards such as the ARINC-653 [2] have been defined to help the design of RTOS, providing safety and compliance with state-of-the-art recommendations.

Avionics systems rely on the concept of Design Assurance Level (DAL). DAL establishes different levels (from level A to E) of criticality to which applications in IMA systems can conform. Level A applications are for which a failure results in fatalities (and loss of the aircraft in most cases) while a level E application failure is considered as minor and does not impact the safety [49]. Most Real Time Operating Systems allow mixed criticality applications to run in the same system and share time and hardware resources.

Aerospace domains are a small part of the CPU market and processor manufacturers slowly stop the production of safety-critical specific CPUs. If avionics system designers want to keep their system up to date, the transition to COTS processors is inevitable. COTS provide better Average Case Execution Time (ACET) in exchange of a degraded Worst-Case Execution Time (WCET). This trade-off is explained by optimization techniques used in modern processors such as out-of-order execution, access reordering and optimized caches [54].

COTS processors are more complex and leaned toward efficiency instead of reliability and time correctness. The performance increase comes at the cost of deteriorated determinism, which generates interferences [55]. Interferences are non-deterministic behaviors that introduce delays in memory access, jitters in peripheral usage and so on. Such behavior breaks most real-time system constraints and, in the worst cases, leads to casualties. Execution time is computed in a way that the program should never run longer than its WCET. However, due to interferences, it is impossible to precisely compute this WCET. One can use large margins to ensure that the WCET will never be reached but this solution leads to underused CPUs [7]. To reduce these margins and certify the RTOS to the highest DAL, it is imperative to mitigate or bound interferences in critical systems. Critical systems, such as ARINC-653, have tight constraints. Ideally, no interference should be found in the system. Many research work has been done in order to reduce the sources of interferences on these systems.

Cache memories are the most prominent source of interferences in single-core and multi-core processors [7]. In single-core environments, intra-partition interference can be seen as one process may evict another process data which generates the interferences. One straightforward solution to avoid interferences is to deactivate caches. However the performances would be unacceptable. Contrary to previous work [56–59] that only rely on one type of data to lock in the cache -either instruction or data— we present an algorithm that manages both data types. We allow the user to choose if only data, instructions or both must be locked. This is particularly important as the application characteristics influence the cache usage. Thus, analyzing only one data type could lead to non-optimal solutions.

In this paper, we propose a method to mitigate interferences that occur in the memory hierarchy levels positioned close to the processor: private caches. Our approach is to rely on cache content selection algorithms for cache locking to avoid cache line eviction and improve the hit rate in the system. To do so, we selectively choose data and/or instructions to be locked in the cache. This solution improves the performance and predictability of private caches while it decreases the workload on lower memory levels, where most interferences are present. We keep in mind that predictability is one key concept of ARINC-653-compliant systems, this is why we rely on static cache locking methods. Two approaches are explored, one based on a greedy algorithm and one based on a genetic algorithm. The proposed algorithms are studied and compared to point out their advantages and usage.

5.3 Background and Context

5.3.1 From Federated To IMA Architecture

Before the late 1990s, aircraft systems design followed a federated approach to develop architectures for airplanes [48]. This type of architecture consists of multiple Line Replaceable Units (LRU). A unit (or module) can be seen as an independent box or computer and performs one specific function in the plane. All the modules are connected through different networks to communicate. Federated architectures became obsolete with the increase of computer-assisted functions present in aircraft [60]. Federated architectures suffer from network contention, which results in a decrease of performance when different modules need to communicate [48]. Weight of the equipment is also an issue when considering fuel efficiency of the plane.

Integrated Modular Avionics (IMA) architectures address issues related to federated systems. Instead of scattered modules, IMA groups multiple applications in one unit. This reduces the size, weight and power consumption (SWaP) of the equipment [48]. Moreover, it facilitates the design and maintenance of avionic systems. Figure 5.1 shows a generic IMA based architecture. Each IMA unit is a different computer that schedules the tasks in the system. IMA units are linked through communication buses or networks to allow the tasks to transmit data.

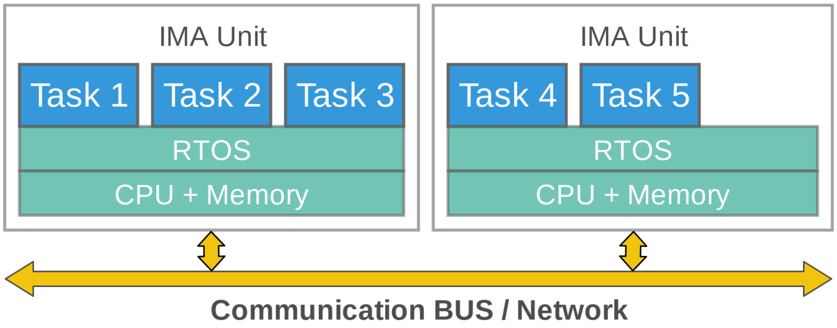


Figure 5.1 IMA architecture example

5.3.2 ARINC-653 And Partitioned Systems

ARINC-653 (Avionics Application Standard Software Interface) is a standard in the avionic domain. It specifies multiple design rules and implementation details for which any avionic RTOS should comply. The ARINC-653 specification mainly defines the following concepts [2]:

ARINC-653 partition An ARINC-653 partition consists of different tasks, called processes that share the same resources. As in regular real-time systems, each process of a partition has a period and deadline. Aperiodic processes also exist and have a deadline but no defined period. The partition's processes can be seen as POSIX threads while partitions themselves are similar to POSIX processes.

Time partitioning The concept of time partitioning defines time windows for which an application executes with exclusive access to all the resources it is allowed to use. A major time frame defines how partitions are scheduled. Figure 5.2 shows an example of time partitioning where the scheme P1 - P2 - P3 - P2 repeats itself.

Space partitioning Space partitioning isolates system's resources among the different partitions. Such isolation can for instance be achieved thanks to the memory management unit (MMU) or memory protection unit (MPU) for the memory, IO-MMUs for Input/Outputs (IO) and peripherals. Bandwidth limitation is seen as a form of space partitioning as it bounds the use of buses by the partitions in the system.

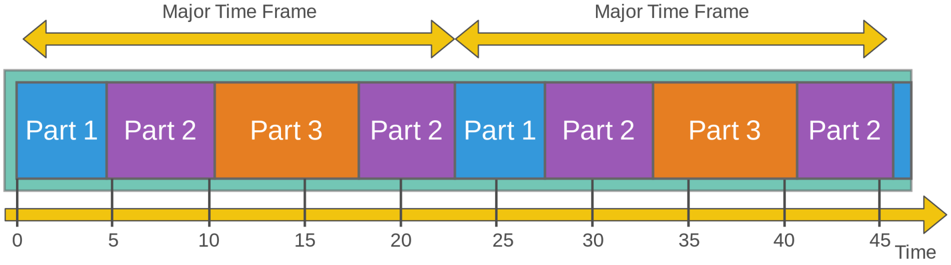


Figure 5.2 Partitioned RTOS major time frame example

Health monitoring The Health monitor (HM) is a specific process of the system used to handle errors and restore the system to a stable state. Each error (exception, deadline miss, etc.) is associated with a handler routine.

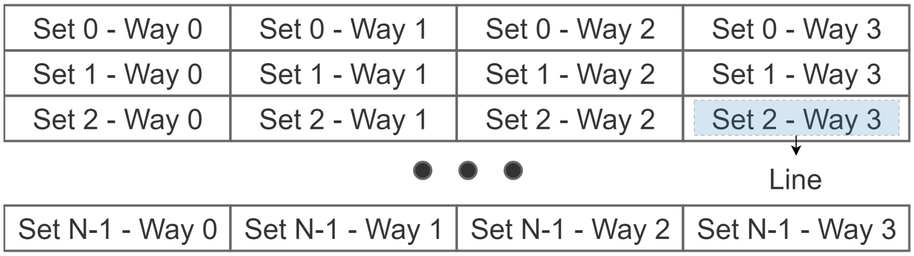
APEX The Application Executive (APEX) provides a standard interface between the applications, RTOS services, system IOs and peripherals. The standard defines multiple function calls for process creation, time and synchronization primitives, inter and intra-partition communications and resource management.

5.3.3 Caches Architecture

Caches are used to overcome the gap between memory speed and processor speed [46]. These memories are often organized in levels, which form the memory hierarchy from the processor to the main memory. The first level of cache is the smallest but the fastest. Follows the second cache level, bigger but slower, then the third level, etc. It is usual to separate the first level of the cache to segregate programs data from instructions. Then the next levels of caches are unified to store both data and instructions.

Figure 5.3 presents the basic cache architecture. Caches are organized in ways and sets and can be represented as a matrix: the sets are the rows and the ways the columns. A cache block (or cache line) is the smallest part of the cache. It contains multiple bytes and is shown as a cell of the matrix representation. If all the cache ways contain loaded data, one of the ways is selected for replacement. This principle is called cache line eviction. Multiple replacement policies exist such as Least Recently Used (LRU) or Round Robin [61].

When multiple partitions are executing concurrently, one partition may evict another's data in the cache. This leads to unpredictable scenarios, as the amount of data to be loaded for every partition is unknown. Further, Interrupt Service Routines (ISR) are shown to pollute the private caches with RTOS data and code. Cache coherence protocols also have an impact on the performance of the cache. All of these contribute to increasing the uncertainty of the



Set 0 - Way 0	Set 0 - Way 1	Set 0 - Way 2	Set 0 - Way 3
Set 1 - Way 0	Set 1 - Way 1	Set 1 - Way 2	Set 1 - Way 3
Set 2 - Way 0	Set 2 - Way 1	Set 2 - Way 2	Set 2 - Way 3
Set N-1 - Way 0	Set N-1 - Way 1	Set N-1 - Way 2	Set N-1 - Way 3

Figure 5.3 4-Way associative cache architecture

system, making caches the source of multiple interferences [62].

Cache locking is a method that enables the selection of memory content to load in the cache and prevents this content to be evicted from the cache. This principle could potentially increase the cache hit/miss ratio [63]. In this paper, we rely on cache locking to reduce execution time unpredictability while we improve private cache usage. This leads to better performances and global predictability.

5.4 Related Work

Multiple works have been conducted to analyze and mitigate interferences in aerospace systems. In this section, we present approaches on cache locking methods for private caches.

Cache locking has been studied in multiple contexts. In [57] cache, locking methods are used to lower the system's energy consumption. In [67], the authors discuss the impact of cache locking on the prediction and improvement of programs' execution time. In [68] a cache locking based approach for improving cache performances is presented. Compared to the above-mentioned contributions, we use cache locking to reduce interferences generated by private caches. Our approach aims to reduce cache workload in lower cache levels with the improvement of cache usage in higher private cache levels.

In [56, 57, 59, 66, 67, 70], *dynamic locking* methods are presented, where the cache lines to be locked are selected at run-time. This allows more flexibility but also introduces non-deterministic and unpredictable locking schemes. Therefore, we cannot rely on dynamic locking in safety-critical systems. We neither can rely on compilation time to analyze accessed memory [65]. This is why we propose a trace based offline algorithm that provides the system integrator[1] with the set of lines to lock at the partition switch.

Static locking allows defining an offline cache locking scheme. This enables the user to select

[1] As per DO-297: The system integrator integrates the system and the applications to create the IMA system.

Table 5.1 Methods for cache locking

References	[56]	[57]	[58]	[59]	[64]	[65]	[66]	[67]	[68]	[69]	Our approach
Static	✗	✗	✓	✗	✓	✓	✗	✓	✓	✓	✓
Dynamic	✓	✓	✗	✓	✗	✗	✓	✓	✗	✗	✗
Data	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓
Instruction	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Kernel	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✓
User	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓
Full	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Partial	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	✓

which lines are to be locked at run-time. However, this choice will not be modified at any point during the execution of the application. In [58, 64, 65, 67–69] static cache locking is preferred to avoid unpredictability brought by dynamic locking.

Instruction locking allows locking program instructions in caches. Work conducted in [56] is aimed to lock instruction cache only and rely on different assumptions that are only valid in the contexts of the presented scenarios of the paper. To lock instruction cache, one can rely on code analysis or introduce locking at compilation time. In [57–59, 64–69] instruction locking is studied to either improve cache performances, increase execution predictability or reduce energy consumption.

While instruction locking can be achieved thanks to static code analysis, it is harder to rely on this method to select data cache lines to lock. Indeed, data accesses may not be explicitly written in application source code (compiled or not) [65]. In [58] and [59] *data cache locking* is presented to reduce data access latency. Our proposition does not segregate instruction or data type accesses: although the user can choose to lock only instruction type lines or data type lines (or both), the algorithm is robust to resolve both.

When selecting the content to lock in the cache, it is interesting to define if user-space or kernel-space content will be selected. *Locking kernel content* allows the reservation of cache space for particular data or instruction of the OS. In [57] kernel space locking is considered to reduce the RTOS service response time. However, the literature does not study the effect of locking both kernel and user space data. All other contributions presented in this paper rely on user privilege locking.

Our method allows the user to select kernel content to lock for all partitions and leave free the rest of the cache. This approach can be used to select part of the data or instructions of the RTOS to lock. The use of a global kernel locking scheme also reduces the overhead of the solution as it is possible to lock cache lines at boot time and never change it during the system's execution.

Finally, cache locking can be separated in two main categories. In [64], *full cache locking* is studied. This method defines a complete set of addresses to lock such that no line of the cache is left unlocked. Most works presented in the literature allow full cache locking with the exception of [67]. Preventing all lines from the cache to be evicted leads to obtain the same predictability as if the cache where disabled while taking advantage the performance increase brought by caches. However, our experiments showed that fully locked caches may lead to worst performances, as demonstrated in the results section of this paper.

In [68] and [69], *partial cache locking* is studied to allow more flexibility in terms of content to lock. This approach aims to lock a portion of the cache only and keep multiple cache lines unlocked. Our method selects the best approach and can detect when it is not advantageous to apply a full-cache locking.

Various methods can be used to select content to be locked in caches. In [56], phase-based classification of content is explored to lock caches. In [58] the authors present an instruction profiling method to detect which memory region should not be evicted. Optimization methods are also studied, in [64] an integer linear programming approach is defined to select the optimal set of instructions to lock.

Table 5.1 sums up the classification of the previously mentioned methods compared to our approach. In the literature, dynamic locking and static locking are equally represented. To the best of our knowledge, we have the first solution considering instruction and data locking for both kernel and user space. Contrary to most work, we describe a method to take into account the complete cache hierarchy. The proposed approach is also applicable on large-scale applications while some previous studies only allow profiting from cache locking on a small portion of programs such as loops or repetitive functions.

5.5 Cache-Locking Algorithms

Some applications have better performance when the entire cache is locked, while others perform better when only a small part of the cache is locked [67]. Our approach distinguishes these two types of applications and select the memory content to lock accordingly. Content selection for cache locking is an NP-Hard problem [63]. Thus, we rely on meta heuristics to solve the problem as the quantity of data is too big to compute the optimal solution.

In this paper, we propose algorithms to select content to lock for all cache levels. This means the use of our solution can be extended to shared caches in multi-core architectures. Our approach considers both data and instructions simultaneously. Instructions can be done by static analysis but not data selection. Moreover, using separate methods to select instructions and data might lead to less precise and effective allocation as both would need to be aware of the already locked lines. This way, the usage of our method enables a better solution. In this work, the granularity of the locking is the line. We select which way to lock for each set of the cache, which corresponds to cache lines.

5.5.1 Overview

We developed and compared two algorithms to generate the set of addresses to lock in the cache. The first is based on a greedy algorithm. We later introduce a genetic algorithm to approach the optimal solution in an acceptable time.

These algorithms were tested in a framework which enforces different processing steps. Memory access traces are gathered during the program's execution and stored for offline processing. The offline processing tool takes the traces as input and runs the algorithms. For each memory access, we get the information about produced hit or miss in the cache. It is important to highlight that we introduce the first approach that considers both the numbers of hits and misses to select the addresses to lock. At the same time, we prioritize highly used cache lines that will produce more misses if not locked in the cache. Further, we also consider kernel and user spaces, enlarging the solution exploration space. All these settings can be defined atomically, so the locked lines reflect the optimal solution for each application. Data privilege is embedded in the traces, allowing the algorithm to apply locking on either kernel, user or both privilege levels. When the processing step is done, the algorithms output the addresses to lock for each cache.

One of the main goals of the two algorithms is to allow multiple levels locking. The greedy algorithm achieves this by maintaining a table containing the lines locked from the first level to the last level. This ensures the algorithm not to lock the same memory block twice. The genetic algorithm achieves this by allowing a memory block to appear only once in the solution representation. This way, a memory block can only be locked once.

5.5.2 Greedy Selection Algorithm

The first algorithm relies on the greedy approach. This paradigm takes the optimal local solution to a problem for each step. However, the global solution might not be optimal.

Figure 5.4 shows the execution flow of our algorithm. Our approach is to rely on the number of hits and misses per memory block. After processing the memory access traces of a partition, we decide the number of ways to lock per set for each cache of the system. Three iterations loops are used. The first level iterate on cache levels. The second loops on each cache of the current cache level. The last nested loop will iterate over all ways of the current cache.

Cache map generation The first step of the algorithm is the cache map generation: the access traces are parsed and, for each access done by the CPU, we register in which caches the access produced a miss or a hit. This structure contains all the registered accesses and is referred as cache map. This cache map is defined as follows:
$AccessedBlocks : \{(id_c, id_b), (N_m, N_h)\}$ the map that associates the pair (id_c, id_b), a cache ID and block ID, to the corresponding pair (N_m, N_h) with:

- N_m The number of misses generated by accesses to the block in the cache.
- N_h The number of hits generated by accesses to the block in the cache.

For simplicity $AccessedBlocks$ can be accessed as a matrix:

- $AccessedBlocks[id_c]$ provides a list of memory blocks $\{id_b, (N_m, N_h)\}$ in $\mathcal{O}(1)$
- $AccessedBlocks[id_c][id_b]$ provides the pair (N_m, N_h) in $\mathcal{O}(1)$

The initial cache map is generated without any locked cache lines. We base our analysis on this map to produce a set of lockable lines. During the execution, the cache map is edited to take into account modifications done to the configuration such as newly locked lines. We also store the cache hit and miss latency for each cache. This information enables the algorithm to compute the weight of an access.

Main processing loop A main processing loop is repeated for each cache level. As we want the algorithm to take into account already locked cache lines to avoid duplicates, we start to process the first level of the cache. From that point we iterate to the last level of the cache. We process each cache of the currently selected cache level independently. For each cache, we select the lines to lock and save the selection. When all caches of the current level are processed and the lines to lock saved, we create a new cache map. This time, the process takes into account all the lines selected for all already processed caches. It prevents the algorithm to select multiple times a line to lock when not needed. We update the weights for each set as locking lines in upper levels will impact hit/miss ratio for lower cache levels.

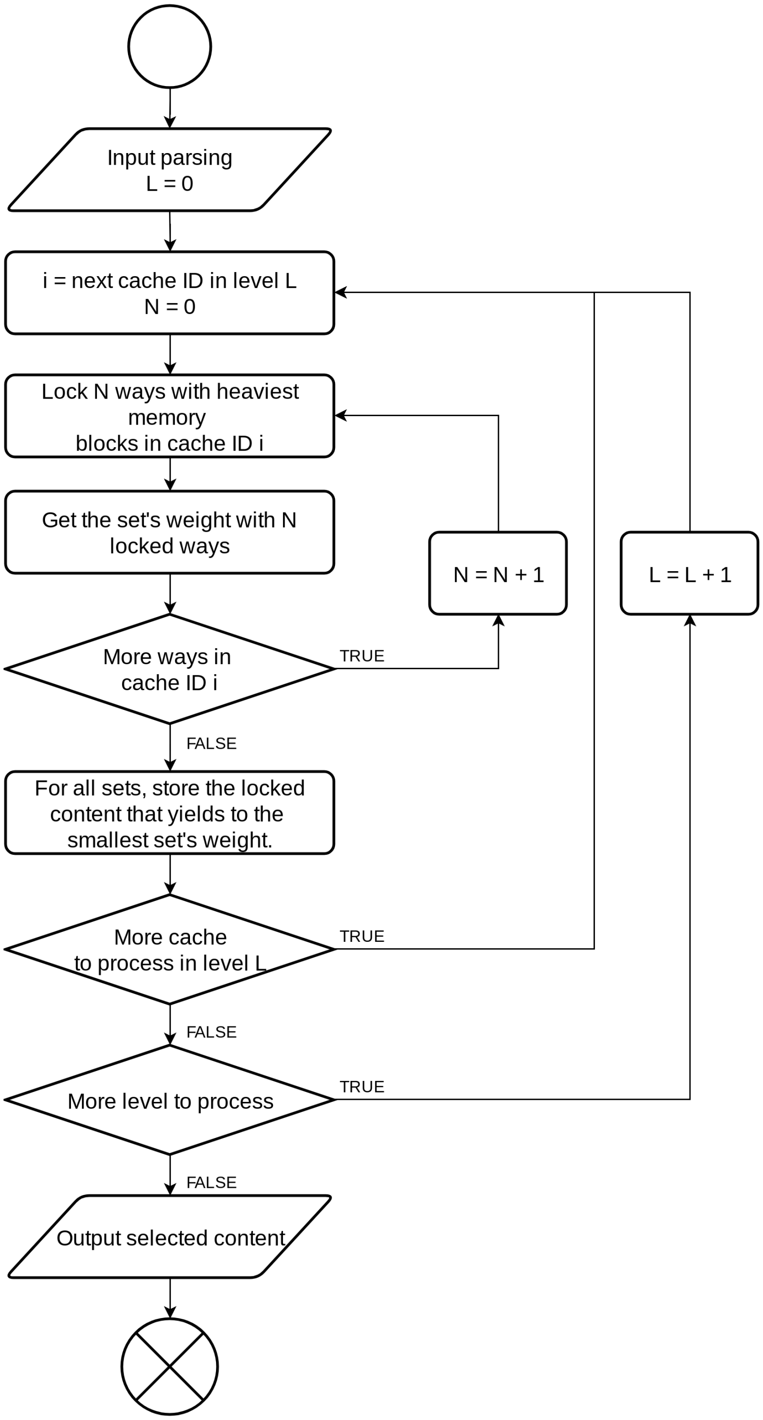

Input parsing
L = 0
i = next cache ID in level L
N = 0
Lock N ways with heaviest memory
blocks in cache ID i
Get the set's weight with N locked ways
More ways in cache ID i
TRUE
N = N + 1
FALSE
For all sets, store the locked content that yields to the smallest set's weight.
More cache to process in level L
TRUE
FALSE
More level to process
TRUE
L = L + 1
FALSE
Output selected content

Figure 5.4 Greedy algorithm process flow

In order to avoid the algorithm to reselect lines for level 2 that are already locked in level 1 cache, before we start the process for the level 2 cache, we remove from the candidates to lock all lines that are already locked in other caches. When all levels have been processed, the analysis terminates and outputs the lines to lock for each caches in the system.

Per cache lines selection To select the cache lines to lock for a specific cache, the algorithm first computes the "weight" for each set of the cache with no lines locked. The weight computation for a set is given in equations 5.1, 5.2 and 5.3. The weight is computed starting with one way locked until the maximal number of ways to lock for this cache is reached. All the weights are stored in the $SetWeights$ map defined as follows:

$SetWeights : \{(id_c, id_{set}, n), (b[], \omega)\}$ the map that associates the pair $(b[], \omega)$. $b[]$ is a list of blocks locked in the set id_{set} of the cache id_c when n ways are locked. ω is the weight of id_{set} of the cache id_c when n ways are locked.

- $W_m : id_c \longmapsto x$ is the function that associates a cache ID (id_c) to the cost of a cache miss x. In our approach we use the cache miss latency provided by the CPU manufacturer.
- $W_h : id_c \longmapsto x$ is the function that associates a cache ID (id_c) to the cost of a cache hit x. In our approach we use the cache hit latency provided by the CPU manufacturer.

$$weight(id_c, id_{set}) = \sum_{n=1}^{N} W_m(id_c) \times \alpha + W_h(id_c) \times \beta \tag{5.1}$$

$$\alpha = \begin{cases} 1, & \text{if access n miss in set } id_{set} \text{ of cache } id_c \\ 0, & \text{otherwise} \end{cases} \tag{5.2}$$

$$\beta = \begin{cases} 1, & \text{if access n hit in set } id_{set} \text{ of cache } id_c \\ 0, & \text{otherwise} \end{cases} \tag{5.3}$$

Algorithm 1 shows how, for each number of ways to lock, we select the lines to be locked. The intuitive idea is to lock blocks that generated the highest number of misses while we also take into account hits to prevent the eviction of a highly used block from the candidates.

The function $GetSetId$ returns the cache set id of a memory block in time $\mathcal{O}(1)$. The variable $current$ used in $GetBlocks$ is a list of pairs (ω, id_b) that contain the lines to lock associated

```
function GetBlocks($id_c$, $nbBlocks$)
    $blocks[setId] \leftarrow \varnothing$
    for all $id_b \mid (id_c, id_b) \in AccessedBlocks$ do
        $setId \leftarrow GetSetId(id_b)$
        $current \leftarrow blocks[setId, nbBlocks].blocks$
        $(N_m, N_h) \leftarrow AccessedBlock[id_c][id_b]$
        $\omega \leftarrow N_m \times W_m(id_c) + N_h \times W_h(id_c)$
        if $Size(current) < nbBlocks$ then
            $current \leftarrow current \cup (\omega, id_b)$
            $ReplaceSmallest(current, id_b)$
        end if
        $blocks[setId, nbBlocks].blocks \leftarrow current$
    end for
    return $blocks$
end function
```

Algorithm 1 Heaviest blocks selection

to the weight of the lines. *current* is sorted by weight and can be implemented as a binary heap, which makes the function *SmallestOf* of complexity $\mathcal{O}(1)$ and *ReplaceSmallest* of complexity $\mathcal{O}(log(n))$.

For each way, we select the line to lock. This step calculates the weight of all sets in the cache. Algorithm 2 explains how we compute the weight for each set.

The last operation is to associate a list to each set. This list contains the weight of the set when no way is locked, when one way is locked, etc. We iterate on the list of sets of the cache and for each set, select the best weight of its associated list. From this choice, we know which lines to lock for this particular set.

5.5.3 Genetic Selection Algorithm

In parallel to the greedy algorithm, we developed a genetic algorithm (GA) to solve the cache content selection problem. GA can produce a solution to a NP-hard problem in a reasonable time[2]. By nature, GA are problem agnostic, which means that the algorithm does not need to be developed to solve a particular problem and have a generic approach [71].

The objective of the GA is to minimize the number of misses in caches. However, one could add a new constraint in order to, for instance, lower energy consumption by modifying the

[2] We use GA for a single objective problem, as opposed to its usual application, in order to reduce the execution time required by optimal algorithms (ex. ILP).

```
function PopulateSetWeight($l$, $lockedBlocks$, $selection$)
    $GenerateCacheMap(selection \cup lockedBlocks)$
    for all $(id_c, l) \in Caches \mid l = i$ do
        for all $id_b \mid (id_c, id_b) \in AccessedBlocks$ do
            $setId \leftarrow GetSetId(id_b)$
            $setWeight \leftarrow 0$
            $(N_m, N_h) \leftarrow AccessedBlock[id_c][id_b]$
            $setWeight.\omega \leftarrow setWeight.\omega + N_m \times W_m(id_c)$
            $setWeight.\omega \leftarrow setWeight.\omega + N_h \times W_h(id_c)$
            $SetWeights[id_c][setId][n] \leftarrow setWeight$
        end for
    end for
end function
```

Algorithm 2 Set weight computation

fitness function proposed in Equation 5.4. GA relies on a set of individuals (population) that are solutions of the problem. Multiple iterations of the algorithm apply transformations to the population:

- Combination of two individuals to produce two new individuals (crossover)
- Modification of an individual that already exists (mutation)
- Injection of new individuals

The initial population can be randomly generated or computed with other algorithms. Our approach is to generate 40% of the initial population with outputs from the greedy algorithm and randomly generate the rest of the population.

The number of individuals in a population must remain constant. To achieve this, only a subset of the generation is selected to be part of the next generation. The selection is made between all individuals: already present individuals, individuals created by crossover or injected in the population. Equation 5.4 represents the fitness function or out algorithm. It relies on the weight of a solution to quantify the quality of a solution. The weight of a solution is calculated thanks to the cache simulation. To calculate the weight, the algorithm sums the weights of each access made by the partition. As previously defines, the weight of one access is defined by the sum of the miss and hit it generated in the caches.

In Equation 5.4, $W_m(j)$ and $W_h(j)$ are the weight of a miss or a hit for the cache j. A_{im} equals 1 if the access A produced a miss in cache j, 0 otherwise. A_{ih} equals 1 if the access A produced a hit in the cache j, 0 otherwise.

$$f(solution) = \sum_{i=0}^{A} \sum_{j=0}^{C} A_{im}(C_j) \times W_m(C_j) + A_{ih}(c_j) \times W_h(C_j) \tag{5.4}$$

Problem representation

We chose a list based representation to express our model. The head of the list is the Last-Level Cache (LLC). It is the point that all caches have in common in most architectures. Each cache level is a node of the list that contains meta data such as the cache IDs in the level, the pointer to the next level of the cache, etc. For each cache of the level, a matrix is added to the node. Figure 5.5 shows how we represent the data in the algorithm.

Crossover operator

The crossover operator used in the algorithm is based on the uniform crossover operator [72]. For two given individuals in the population, we take caches of same ID in their caches representation. We create two new individuals (children) from these. For each set *i* of the first child, we randomly select between the lines locked in the set *i* of the first parent and the second parent. Once the first child is created, the second child takes all the non-selected lines. Figure 5.6 shows how the crossover operation is done.

From our experiments, this crossover operator is the one that yields the best results. We implemented basic operators such as one point and two points crossover but the convergence time provided by these operators were slower. We also created children by random cache lines selection instead of random cache sets selection. This solution reduced the algorithm to a random search and provided poor results.

Mutation operator

Mutation is a key part of genetic algorithms. While crossover operators tend to converge to a good solution (local minima), mutation allows to maintain diversity in the population. Mutation allows the solution to jump from a local minimum to another point in the search space. The mutation operator enable the algorithm to explore the search space to find better solutions.

The operator we rely on proceeds to randomly select sets of the cache matrix to be mutated. The selection probability of a set in the matrix is chosen by the user. This probability will depend on the number of accesses made by the program. We lock the lines of and its memory access pattern. Once the sets to mutate have been selected, we apply the mutation:

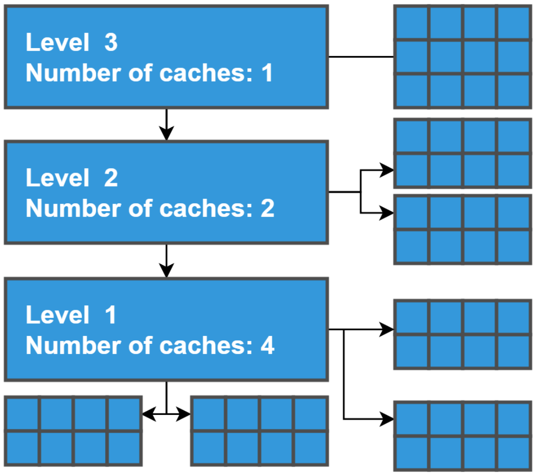
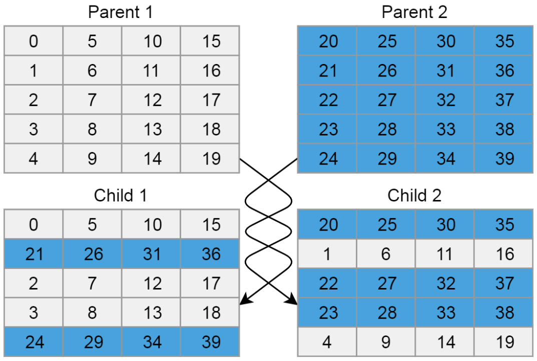


Figure 5.5 GA cache representation



Parent 1

0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

Parent 2

20	25	30	35
21	26	31	36
22	27	32	37
23	28	33	38
24	29	34	39

Child 1

0	5	10	15
21	26	31	36
2	7	12	17
3	8	13	18
24	29	34	39

Child 2

20	25	30	35
1	6	11	16
22	27	32	37
23	28	33	38
4	9	14	19

Figure 5.6 Crossover operator used in the genetic algorithm

- For each way of the set and a probability α;
- Randomly select a memory block in the list of accessed blocks for the current set;
- If no memory block was selected $(1 - \alpha)$, define the cache line as not locked;

The probability α is computed as follows:

$$\alpha = \begin{cases} 2\,\dfrac{x \bmod y}{y}, & \text{if } x \bmod y < \dfrac{y}{2} \\ -2\,\dfrac{x \bmod y}{y}, & \text{otherwise} \end{cases} \tag{5.5}$$

x is the current generation number and y is the period in number of generations. This oscillates the probability to lock a line between 0 and 1. The proposed mutation operator permits to create solutions where no lines are locked or where all lines are locked depending on the generation. This way, the mutation operator explores all lock patterns of the search space.

The population selection is made so 60% of the fittest individuals make it to the next generation and 40% are randomly selected. The same applies for the mating pool selection.

5.6 Results

In this section we present the results observed with our approach. First, we compare the performances between the greedy and the genetic algorithms. Then, we discuss the quality of the solutions generated by these two algorithms. In a second time, we present the performance and predictability improvement brought by the cache locking method when applied to ARINC-653 systems. We run all tests on the MPC5777C board manufacture by NXP. In our test bench, we run a proprietary ARINC-653 compliant RTOS, provided by our industrial partner, on a single core. The RTOS was modified to implement the cache locking mechanism at run time. The cache is defined as follows: 16KB L1 Data cache 16KB L1 instruction cache, Harvard architecture. 256KB L2 cache.

5.6.1 Benchmarks

The results presented in this section were obtained using benchmarks and instruction intensive applications. Mälardalen benchmarks [73] and SNU-RT benchmarks [74] were ported to the target system. We use data intensive benchmarks to show the improvement of our method on private L1 data caches. In addition to these benchmarks, we developed RArray, an application producing a miss for each data access. This application allows us to verify our approach. We added five instruction-intensive applications presented in Table 5.2. We use these applications to stress the L1 instruction cache as most benchmarks provided only stress the data caches. The five applications are presented in Table 5.2. Each application is considered as a partition in the target system and is composed or multiple threads to generate intra-partition interferences.

5.6.2 Algorithm Comparison

We compared the execution time for the two proposed algorithms. To select the used parameters for the genetic algorithm, we performed multiple tuning operations to find the values yielding to the best results. This leads to the values: *Crossover Probability* = 0.96, *Mating Pool Size* = 30, *Mutation Probability* = 0.35, *Population Size* = 100 and *Iteration Count* = 100.

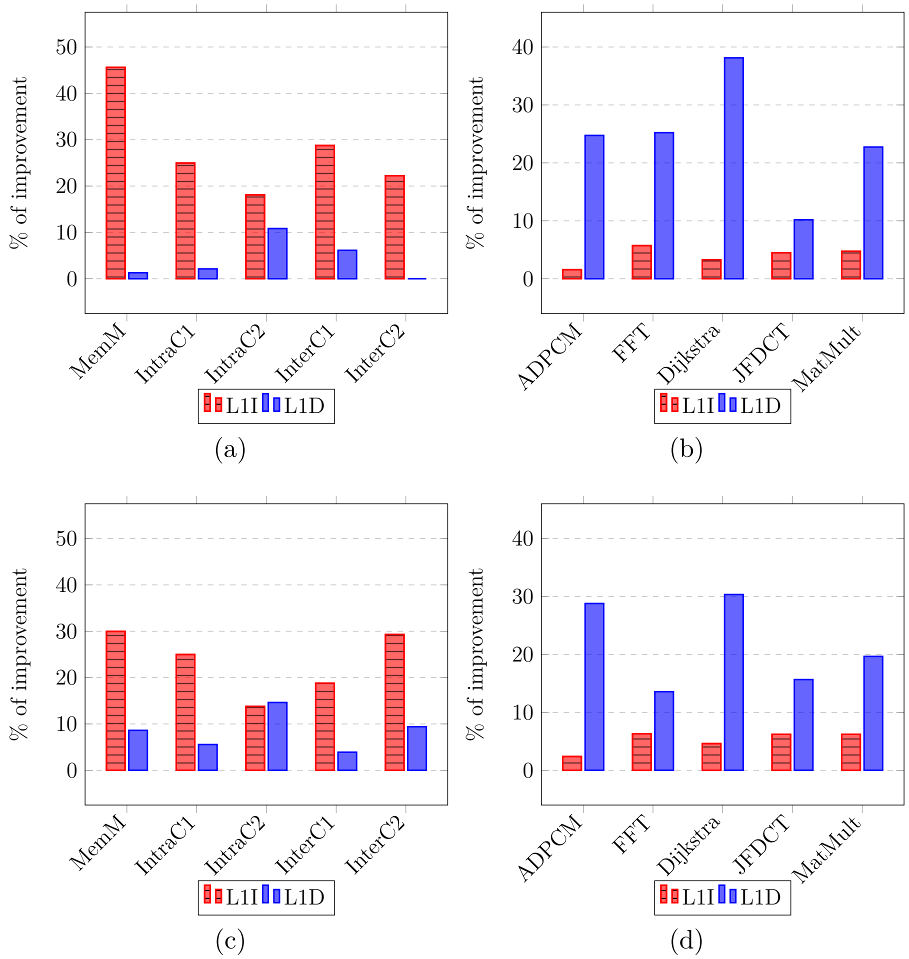
Table 5.2 Instruction intensive applications

Application	Domain	Description
MemM	Instruction intensive	System calls through RTOS API to allocate and deallocate memory
IntraC1, IntraC2	Instruction intensive	APEX calls used for intra partition communication (semaphores, mutex, etc.)
InterC1, InterC2	Instruction intensive	APEX calls used for inter partition communication (sampling ports, queuing ports, communication pipes)
JFDCT	Data intensive	JPEG integer implementation of forward DCT
Dijkstra	Data intensive	Find the shortest path between non-negative nodes. Used for GPS, for instance.
FFT	Data intensive	Fast Fourrier Transform
ADPCM	Data intensive	Pulse Code Modulation for audio conversion from and to digital signal
MatMult	Data intensive	Basic implementation of matrix multiplication

To compare the time to produce a solution, we provided the algorithms with trace files of different sizes, out of 20 executions for each trace file. The results show that both the greedy and genetic algorithm execution's time grow linearly compared to the size of the trace. However the genetic algorithm is 95 times slower than the greedy algorithm. This result is due to the trace file parsing time. Indeed, in our genetic approach we set the number of iterations to 100. The trace parsing is executed 100 times more compared to the greedy approach. The obtained results are given in Table 5.3.

Table 5.3 Execution time for different trace sizes

File size (MB)	Greedy (s)	Genetic (s)
7	0.2	15.1
15	0.4	35.7
29	0.8	70.3
57	1.45	147.2
113	2.85	292.6
225	5.7	540.1
449	11.3	1200.0
987	22.6	2243.5
1974	45.3	4719.3
3948	92.1	9516.72



(a) (b)

(c) (d)

Figure 5.7 Cache miss improvement using PLRU and PRR policies. Results are shown for instruction intensive applications (a) with a reduction of miss up to 45% and data intensive applications with a reduction of miss up to 38% (b). Instruction intensive applications (c) with a miss reduction up to 30% and data intensive applications with a miss reduction up to 31% (d).

5.6.3 Performance Improvement

Table 5.4 describes the overall improvement brought by the algorithms. We provide the percentage of cache misses avoided with cache locking when using the greedy and the genetic approaches. In the case of JFDCT, the greedy algorithm avoids 10% of the misses while the genetic algorithm will only reduce the number of misses by 1%. For the RRArray benchmarks, both algorithm found the optimal solution and for Djiktra, the greedy algorithm yields better results. This is due to the size of the research space. We use cache locking for highly constrained embedded systems (with a RAM size of 512KB and a flash size of 16MB), the search space of the genetic algorithm is considerably reduced. This prevents the algorithm to find a global minimum. However we expect our genetic approach to yield better results than the greedy approach in the case of general purpose systems. To leverage this issue, if one still wishes to use the genetic algorithm, we inject the solution output by the greedy algorithm in the initial population. With the use of elitism in the genetic algorithm, this ensures the result will be at least as good as the one generated by the greedy algorithm.

Table 5.4 Greedy algorithm based approach against genetic algorithm based approach. Results are expressed in the percentage of avoided misses. The greedy approach performs better in most cases.

Algorithm	**JFDCT**	**RRarray**	**Dijkstra**	**FFT**
Greedy	10.18%	75.98%	38.13%	25.22%
Genetic	1.11%	75.98%	15.28%	25.53%

Table 5.5 shows the performance degradation that occurs when the cache is fully locked. Both Djikstra and JFDCT benchmarks were tested. The performance degradation is given in percentage of misses compared to the execution when no line is locked. A positive number means an increase of cache misses. One of the goals of our approach is to avoid such cases. An overlock situation occurs when locked lines prevent other data from populating the cache. This results in an increase of cache misses. As our results show, such cases not only happen when the cache is fully locked but appear when only 60% of it is locked.

Table 5.5 Cache miss increase on overlocked caches

% of cache locked	20%	40%	60%	80%	100%
Dijkstra	-19%	-12%	38%	+172%	+385%
JFDCT	-1%	-10%	+175%	+856%	+2232%

We conducted our study on two different cache line replacement policies. The best results were obtained using the Pseudo-LRU algorithm. Figures 5.7(a) and 5.7(b) show the improvement provided by our approach for instruction intensive applications. Results show an improvement of 27.93% on average with peaks to 45.63%. The second test bench is targeted to stress the data cache. The quantity of data used by these applications spans from 20Kb to 360Kb. By applying cache locking methods at run-time, results show an improvement of 21.62% on average with peaks to 38.13%. This means that we avoided up to 38.13% of the cache misses that were present when the cache was not locked. Figures 5.7(c) and 5.7(d) show results with the PRR replacement policy. For instruction intensive tests, we obtained better results for the data caches than with the PLRU replacement policy. As for the data intensive tests, the results were better than the PLRU policy for the instruction caches. This behavior shows that cache locking works better on the PLRU replacement policy when less data is manipulated by the cache.

The improvement is limited for applications where cache locality has been taken into account by the developers. For applications where memory accesses are not optimized for caches use (e.g., Dijkstra), our algorithm provides an improvement of at least 20%.

Figure 5.8 shows the percentage of accesses avoided after locking the L1. The results take into account the reduced number of hits and misses in the shared cache. This reduction is directly related to the decrease of miss rate in L1 caches. As 31.5% of accesses in the L2 on average can be avoided, we expect our method not only to reduce the contention on shared caches, but also to reduce the interferences related to cache eviction by other cores.

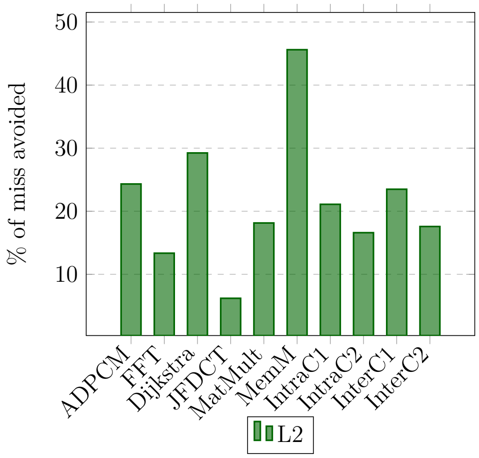


Figure 5.8 L2 accesses avoided for PRR policy. Our algorithm allows to reduce the workload on L2 cache by 45% in some cases and of 31.5% on average.

Figure 5.9 demonstrates the execution time improvement on the PowerPC e200z7 processor. This processor uses the PRR replacement policy. We reduce the number of cycles where the CPU is stalled waiting for data from the L2 caches or the main memory. In our architecture, locking the instruction cache is more profitable as instructions are stored in the flash of the MCU whereas the data are stored in the RAM, which is faster than the flash.

Table 5.6 illustrates the number of locked lines for each tested application. PLRU and PRR are considered. The table shows that regardless the reallocation policy, our approach can be used. Furthermore, the average number of locked lines, for each application is similar. This shows the generality of our algorithms, as opposed to the solutions presented in Section 5.4.

5.6.4 Predictability Improvement

Not only we reduce the execution time, but we increase the predictability on the system by reducing the size of the cache that is not locked. This reduces the non-determinism

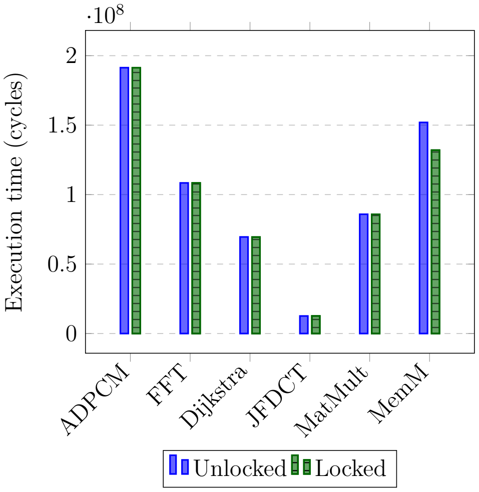


Figure 5.9 Execution time comparison in CPU cycles. The overhead is comprised in the measure, meaning that our approach does not add execution time overhead when used.

Table 5.6 Comparison between PLRU and PRR algorithms regarding the cache locked lines.

	Pseudo-LRU		**PRR**	
Application	**Instruction**	**Data**	**Instruction**	**Data**
MemM	211 (41.2%)	54 (10.5%)	198 (38.7%)	67 (13.1%)
IntraC1	134 (26.2%)	76 (14.8%)	154 (30.1%)	85 (16.6%)
IntraC2	127 (24.8%)	45 (8.8%)	111 (21.7%)	76 (14.8%)
InterC1	196 (38.3%)	54 (10.5%)	234 (45.7%)	76 (14.8%)
InterC2	183 (35.7%)	15 (2.9%)	165 (32.2%)	13 (2.5%)
ADPCM	77 (15%)	101 (19.7%)	65 (12.7%)	121 (23.6%)
Dijkstra	25 (4.9%)	342 (66.8%)	25 (4.9%)	342 (66.8%)
JFDCT	56 (10.9%)	98 (19.1%)	43 (8.4%)	103 (20.1%)
MatMult	51 (10%)	312 (60.9%)	61 (11.9%)	326 (63.7%)

introduced by cache replacement policies. Table 5.7 represents the standard deviation of the execution time over 20 executions for each application.

Table 5.7 Standard deviation of execution time (CPU cycles). Results show that we significantly reduce the execution time standard deviation by 97.29% on average.

Application	**Not locked**	**Locked**	**Improvement (%)**
FFT	868.09	0.55	99.93
ADPCM	1123.80	9.78	99.13
Dijkstra	664.39	10.55	98.49
MatMult	342.13	0.54	99.84
MemM	4031.65	442.01	89.04

5.6.5 Overhead of the Solution

The overhead added by the algorithm only takes place in the RTOS at partition switch. The content selection is done offline and is not considered as an overhead. At partition switch, the RTOS executes two operations: cache flushing, which unlocks the lines and then it proceeds to lock the lines for the newly scheduled partition.

Cache invalidation/flush is done by hardware, this process is called flash invalidation and only takes 134 CPU cycles on the PowerPC e200z7 [75]. It should be noted that cache flushing is a mandatory process to execute at partition switch in ARINC-653-compliant systems, even when the cache locking technique is not used [50]. We can then neglect the cache unlocking process as it does not add any overhead.

On the PowerPC e200z7 it takes 30,000 CPU cycles to entirely lock the cache. This is three times greater than the RTOS initial context switch time. However, at run-time, this overhead is reduced by the gain in time that comes from the cache miss reduction. This is shown by the execution times measurement that show no increase in execution time for any application in Figure 5.9.

5.6.6 Comparison with previous work

Many works have studied the impacts of caches in the system. Different solutions emerged from such studies: static or dynamic locking; to lock data or instructions; to consider user or kernel spaces, amongst others. Many works focus on different aspects of the system (WCET, power, etc), and yet the developed techniques may be employed for different ends. With that in mind, we have compiled state-of-the-art works in the context of cache locking and analyzed their impact on miss-rates of the cache. We use these values to compare their solutions with ours. In [58], using static locking, authors state an improvement of miss rate up to 24%. The obtained results in [68], when using the same cache size as our setup, point out to an improvement of about 30%. A technique combining cache locking and instruction pre-fetching, presented in [69], achieves an average of 20% improvement. Regarding fully dynamic techniques, the results are more relevant. In [76], using a WCET-aware dynamic locking, authors state results showing up to 70% improvement. The same is true for [70], where dynamic locking is employed and results are in average 60% better. As discussed through the paper, dynamic locking is not compatible with the principle of ARINC-based systems. This type of technique adds timing and spatial uncertainties to the system, which are not tolerated. Thus, although good numbers could be obtained using these techniques, we cannot rely on it. Regarding static techniques, the obtained results of our approach can rank up to 45% of cache misses avoided, proving that our solution is more efficient when compared to works listed here. Figure 5.10 sums up the comparison between our approach and previous work.

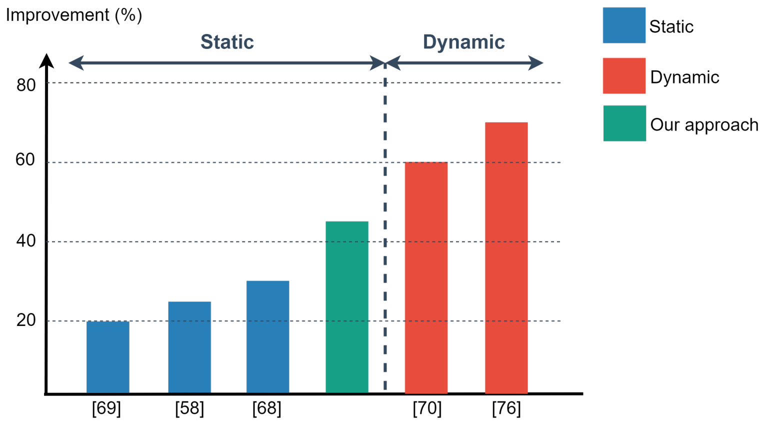


Figure 5.10 Comparison of our solution with previous work.

5.7 Conclusion

In this paper, we presented two algorithms used for cache locking content selection. We showed that the greedy approach is better in our context due to the high constraints on memories in embedded critical systems. The proposed method allows the user to select between kernel and/or user data which is interesting in terms of safety and can ease the certification process.

Our approach relies on partitioned RTOS concepts to ensure cache locking at partition switch and provides performance improvements for the newly scheduled partition. We demonstrated an average improvement of 27.93% on average with peaks up to 45.63% for the instruction cache and an improvement of 21.62% on average with peaks up to 38.13% for the data cache. Such improvements also come with an increase in terms of predictability in the system. Thanks to the reduction of the cache misses in private caches, shared cache contention is reduced as well as shared cache workload. This result is particularly interesting as cache partitioning can prevent indirect shared cache interferences but cannot prevent cache contention.

Finally, the use of the proposed solution for static locking makes the certification process easier and the integration with ARINC-653 compliant systems more robust. Indeed, the algorithm output can be verified and validated by the system integrator before being used in the system at run-time. This way, qualification of the offline tool and the algorithm is not needed.

5.8 Acknowledgments

This research was conducted in partnership with MANNARINO System & Software. The authors would like to thank MANNARINO Systems & Software for their support and their help. We also would like to thank MITACS and CRIAQ for supporting this research.

CHAPTER 6 GENERAL DISCUSSION

In this chapter, we present the experimental results showing the efficiency of our approach. We decided to highlight the performance improvement brought by the method and the increase of predictability in execution time for partitioned applications.

6.1 Considered Architectures

Our approach applies to any architecture that allows locking their caches. To show the portability of our solution, we based our tests on two widely spread architectures used in embedded systems: the PowerPC e200/e500 and the x86 architecture.

6.1.1 PowerPC e200/e500

The PowerPC e200 is a 32-bit RISC processor used in automotive and aerospace microcontrollers [77]. It is developed and maintained by Freescale and was released under seven versions: from z0 to z7 excluding the z2 version. The e200z7 used in our experiment has separated 16Kb instruction and data caches based on the Harvard architecture. Table 6.1 gives the specification of the e200z7 caches used in this chapter. The PowerPC e500v1 core is a more recent implementation of the Power ISA v2.03 (binary compatible with the e200). Freescale designed the e500 for high-end microcontrollers. The e500v1 embeds a separate instruction and data cache architecture (Harvard) for which each L1 cache has 32 KB of storage space. A unified L2 cache is also present and differs in size depending on the version of the core (from 256 KB to 1 MB). The PowerPC e200z7 was used to get the results on actual hardware while the e500 processor was emulated on Qemu.

6.1.2 x86 architecture

To apply our process to a second architecture, we used the Intel Atom N270 [78] emulation provided by Qemu alongside the cache simulator used in the framework to provide cache results. The Intel Atom N270 is used in embedded systems for its small energy requirements. From our experiment on the e200z7, the simulator developed in our research group gives the same results as the actual cache with an error deviance of 0.1%. Thus we have a high degree of confidence about our result concerning the potential use on actual hardware. The caches of the Atom N270 are separated in two caches for the first level, a 32 KB instruction cache and a 24 KB data cache. We also simulate the L2 unified cache of 512 KB.

Table 6.1 Cache architectures of the e200z7, e500v1 and Intel Atom N270

Processor	Type	Size	Associativity	Line size
E200Z7	Instruction Data	16 KB 16 KB	4-way 4-way	32B 32B
E500V1	Instruction Data	32 KB 32 KB	8-way 8-way	32B 32B
Intel Atom N270	Instruction Data	32 KB 24 KB	8-way 6-way	64B 64B

6.2 Benchmarks

We conducted our test on two categories of tests, one aimed to stress the data cache and the other aimed to stress the instruction cache. All the tests were gathered from the SNU-RT benchmark suite [74] and the Mälardalen benchmarks [79]:

- ADPCM: Adaptive Differential Pulse Code Modulation is a compression algorithm with loss based on signal prediction.
- FFT is a fast Fourier transform algorithm.
- Dijkstra is the well-known shortest path computation algorithm.
- JFDCT is a discrete cosine transform application using a 8×8 kernel on variable size matrices.
- MatMult is a basic matrix multiplication algorithm.
- SHA1 is a SHA hashing algorithm.
- QSORT is a non-recursive version of the Quick Sort algorithm.

Each benchmark is executed on its own partition in a single-core environment. We rely on a proprietary ARINC-653 compliant RTOS to execute the benchmarks.

6.3 Results Gathering and Methodology

We separate our result gathering methodology in two axes. The first results are obtained on emulated architectures (Atom N270 and PowerPC e500v1). For each benchmark we use the following process:

1) Trace the execution of the benchmark's partition. This step gathers all the instructions data accesses for one execution.

2) We process the traces in the cache simulator. This allows us to detect the number of hits and misses for each cache of the simulated architecture.

3) The cache locking algorithm is used in the framework to select which lines are to be locked for a given benchmark.

4) We collect another execution trace for the current benchmark.

5) With the cache locking configuration file generated in step 3), we process the trace gathered in step 4). This gives us the number of hits and misses when locking the cache.

During step 3), we generate three configuration files:

- Scheme with both kernel and user data locked;
- Scheme with kernel data locked only;
- Scheme with user data locked only;

This helps to study the impact of allowing users to select which type of data should be locked.

To validate our simulations, we use the MPC5777C MCU from Freescale to execute our test on actual hardware. The MCU embeds a PowerPC e200z7 processor. Our benchmarking process is slightly different from the one used in the simulation:

1)
    a. Trace the execution of the benchmark's partition. This step gathers all the instructions data accesses for one execution.

    b. We run each benchmark ten times to gather the execution time of the partition when no cache locking is applied.

2) We process the traces in the cache simulator. This allows us to compute the number of hits and misses for each caches of the simulated architecture.

3) The cache locking algorithm is used to select which lines are to be locked for a given benchmark.

4) We apply the cache locking through the cache locking module embedded in the RTOS and run the benchmark ten times to measure the execution time of the partition when cache locking is applied.

5) For each execution, we detect the number of cache hits and misses to quantify the improvement brought by our method.

To quantify the overhead of the solution, we rely on the execution time of the complete partition, from the context switch, before the locking process is initiated, to the end of the partition (when all processes of the partition finished their execution and the partition's idle process is scheduled).

6.4 Results synthesis

In this section, we present the obtained results using our method to lock the lines.

6.4.1 Simulated results

For single-core results, we collected the data on Qemu using the e500v1 processor. All results are presented as a part of the paper we submitted and can be found in Chapter 5 of this thesis.

Multi-core Improvements

Multicore results were obtained using the Intel Atom N270 emulation of Qemu. This architecture was emulated on two cores to study the impact of our approach on multicore systems. The cache configuration is the following:

- L1 Instruction cache: 32 KB, 8-way associative (1 per core);
- L1 Data cache: cache: 24 KB, 6-way associative (1 per code);
- L2 Unified cache: 512 KB, 8-way associative (shared by all cores);

We used a x86 kernel that we modified to be non-preemptive. We place our tracing points to ensure the execution of each application is similar to an ARINC-653 partition. We added five benchmark applications to our test bench on a x86 platform:

- MemMng is a memory management benchmark used to allocate and release various sizes memory chunks.
- Intra1 uses intra-partition communication mechanisms (shared memory, mutex, semaphores) to allow communication between tasks in the partition.

- Intra2 uses intra-partition communication mechanisms (shared memory, mutex, semaphores) to allow communication between tasks in the partition.
- Inter1 uses inter-partition communication mechanisms (queues, mailboxes) to allow communication between two partitions.
- Inter2 uses inter-partition communication mechanisms (queues, mailboxes) to allow communication between two partitions.

To gather the results, we selected two different applications to run in parallel on two distinct cores. Figures 6.1 and 6.2 show the improvement brought by the framework on the Intel Atom N270 architecture. In the multicore environment, the algorithm allows to reduce the number of misses in L1 caches. This results in fewer accesses in the L2 cache. The reduction allows to decrease the number of data that are evicted in the L2 caches, which decreases the number of misses occurring at this level. A benefit of cache locking is the workload decrease brought to shared cache levels. Indeed, reducing the number of misses that occur in the first level will reduce the number of accesses done to lower cache levels. Figure 6.3 shows the reduction of accesses in the shared L2 cache. All accesses made by both cores are potential

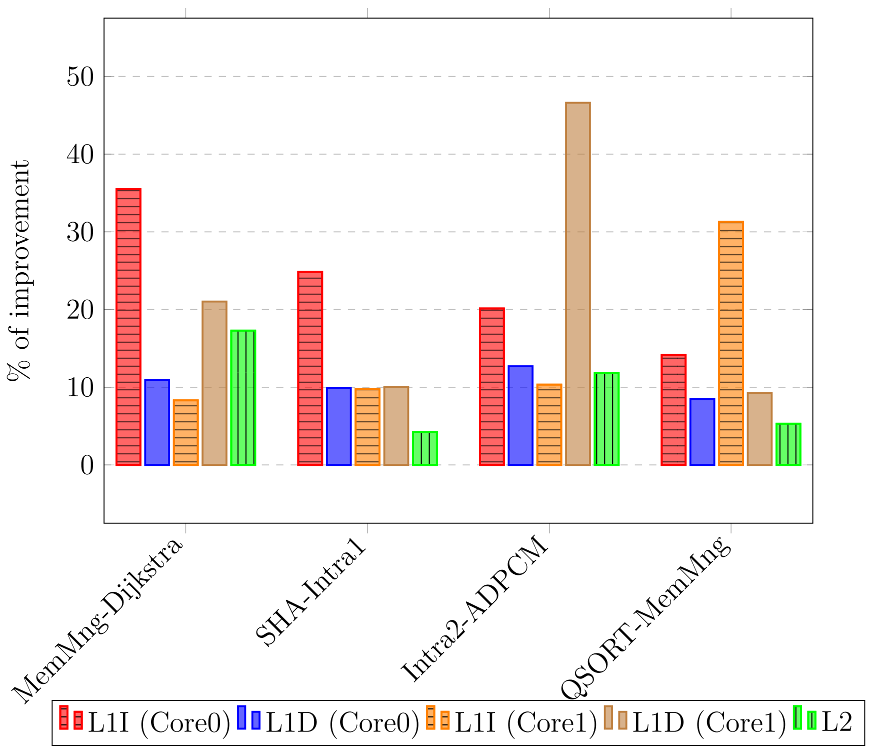
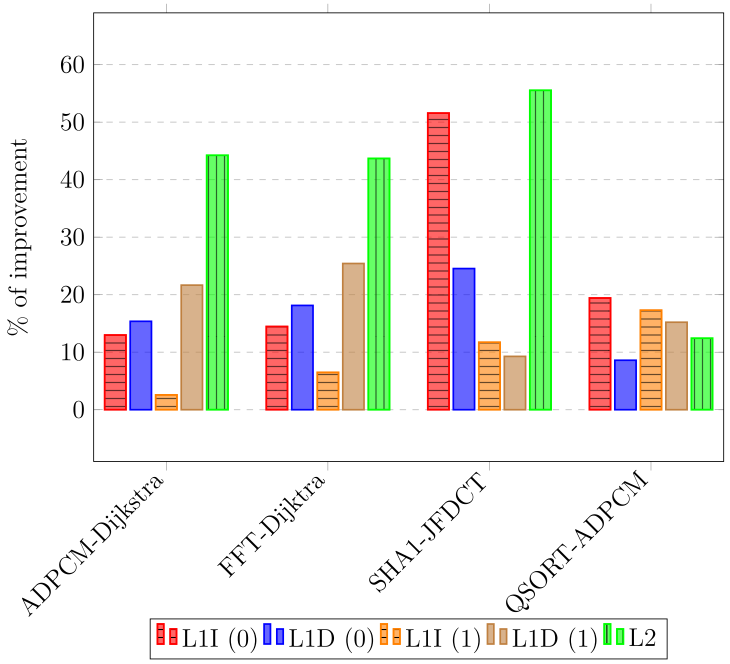
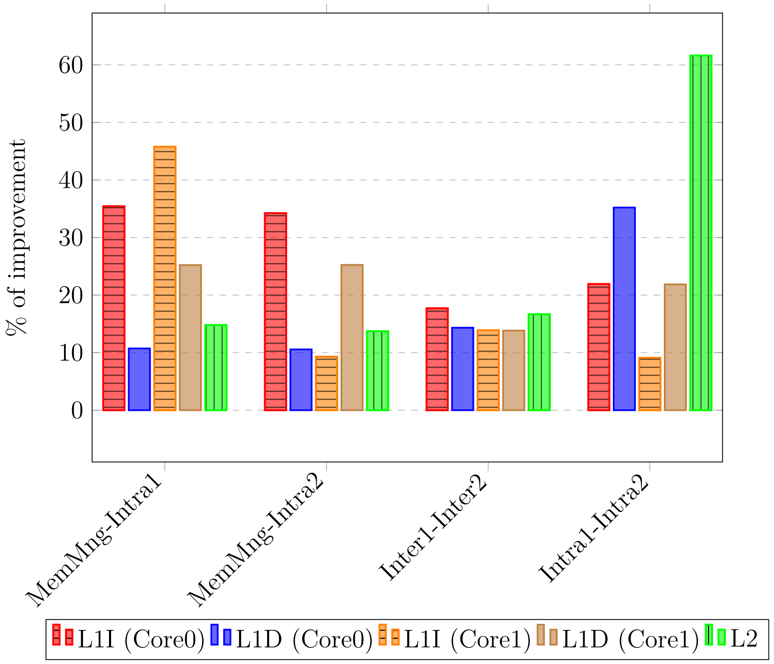


Figure 6.1 Cache miss improvement on Intel Atom N270. Results are shown for co-running instruction and data-intensive applications with a miss reduction up to 47% in private L1 and 17.5% in shared L2.



(a)



(b)

Figure 6.2 Cache miss improvement on Intel Atom N270. (a) Results are shown for co-running data intensive applications with a miss reduction up to 51% in private L1 and 55% in shared L2. (b) Results are shown for co-running instruction intensive applications with a miss reduction up to 46% in private L1 and 62% in shared L2.

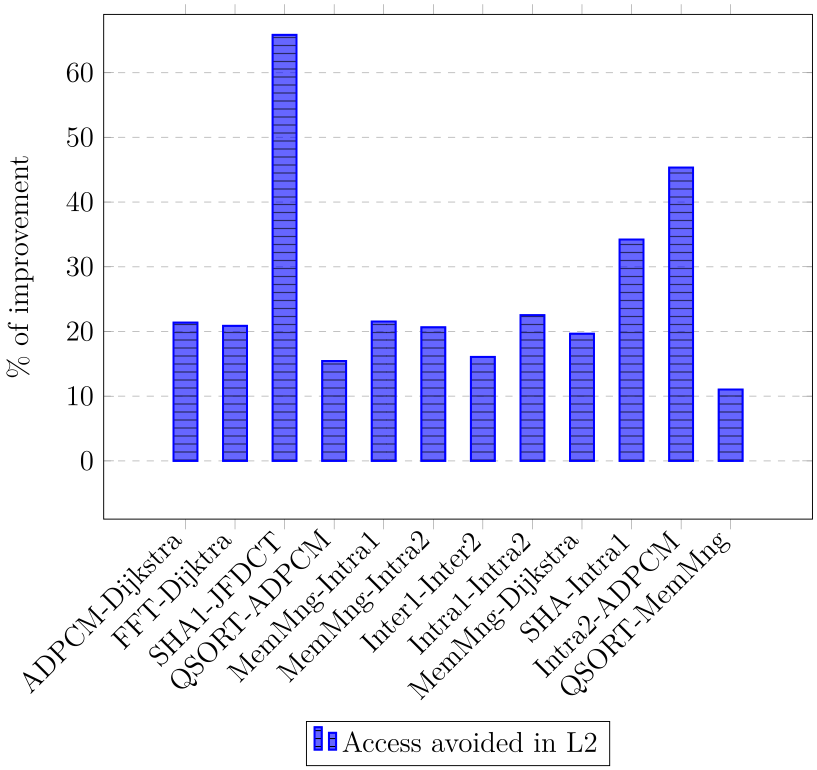


Figure 6.3 L2 accesses avoided on Intel Atom N270. Results are shown for co-running applications. All accesses to the L2 are potential contention factors. Results show a reduction of 26.21% on average with peaks up to 65.84%.

cache contention sources. As previously explained, shared cache contention is an interference that cannot be mitigated by partitioning the cache. The results we provide show that cache locking reduces the contention on shared caches, which leads to better performances and a reduction of the delays induced by cache contention interferences.

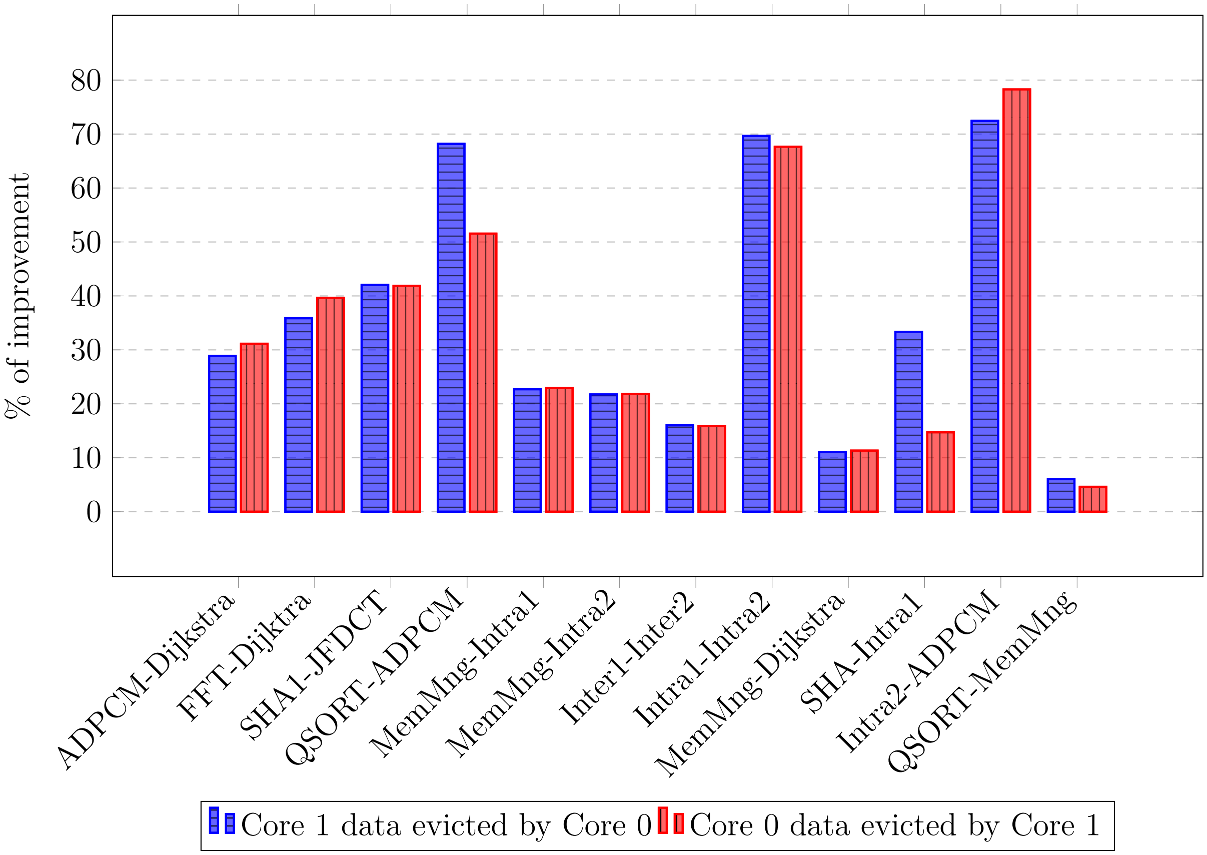


Figure 6.4 Percentage of indirect interferences avoided (cache line eviction). Our approach allows to reduce the amount of indirect interference of 33.22% on average with peaks up to 78.28%.

Finally, we measured the amount of indirect interferences generated when co-running applications. Indirect interferences appear when an application of the partition evicts data that it does not own. For instance, partition 1 runs on core 0 and partition 2 runs on core 1. Partition 1 may evict data from the shared cache that were loaded by partition 2. This kind of interference is usually fully mitigated thanks to cache partitioning. However, cache locking may be an interesting alternative when partitioning is not possible. Figure 6.4 shows the impact of locking mechanisms on the number of indirect interferences in shared caches. It represents the percentage of undesired evictions that were avoided thanks to cache locking. Indeed, cache locking cannot remove shared cache indirect interferences, but can improve performances.

6.4.2 Results on MPC5777C architecture

Results were gathered on the MPC5777C using the e200z7 processor. All results are presented as a part of the paper we submitted and can be found in Chapter 5 of this thesis.

***

In this chapter, we presented the results obtained applying our method to different architectures. This study shows that our approach is widely portable and can be used in multiple cases.

CHAPTER 7 CONCLUSION

7.1 Summary of the proposed contribution

Multi-core architectures raise multiple challenges for safety critical system designers. Such architectures introduce timing hazards known as interferences. Multiple studies have been conducted to mitigate or bound such behavior. However, few of these works are currently applied to state of the art commercial ARINC-653 compliant RTOS. The existing literature relies on shared resource optimization, partitioning and segregation to mitigate interferences. In this thesis, we presented two contributions to mitigate interferences.

We introduced a framework capable of profiling memory usage of an ARINC-653 application. Such a framework can be used to characterize the memory usage of an application, understand its performance issues and its access patterns.

We also proposed two cache locking algorithms that we integrated to the framework. The algorithm allows the system integrator to select content to lock in private caches. This locking scheme allows a better use of the cache at private levels (usually L1 caches).

In our experiments, we show that our work profits single-core and multi-cores.

In the first case, cache locking provides better performance and predictability to the systems. In single-core environments, we determined that our approach can reduce the number of cache misses up to 45% (Chapter 5) with an average of 28%. The experiments also show that the predictability of execution time can be greatly increased. In our case, we observed a reduction of the execution time standard deviation of 97.29%. In domains where determinism is critical, such improvements cannot be neglected.

In multi-core environments, cache locking reduces the contention on shared caches but also reduces indirect interferences when cache partitioning is not available in the system. By co-running applications, we observed interference behaviors. We were able to quantify the reduction of such interferences when cache locking is used. Our method allows reducing the contention on shared caches by reducing accesses to these shared components. We observed a reduction of accesses to shared caches of 26.21% on average with peaks up to 65.84%. This led to lower indirect interferences too as locking the cache allowed to reduce indirect interference of 33.22% on average with peaks up to 78.28%.

7.2 Limitations

Even though our approach is generic and can be applied in different manners, some limitations can be found:

- Application profiling relies on the repetitive nature of memory access patterns. When an application has a random way of accessing the memory, the quality of the solution provided can decrease. Consequently, sporadic tasks are not able to profit from cache locking as we defined it.
- Our approach relies on cache simulation which slows down the processing. The bigger the trace is, the slower the process is even if it relies on linear or polynomial time algorithms. This process could not be used for general purpose applications where the profiling time would generate large traces.
- Cache locking relies on hardware components to manage the cache. Few architectures allow locking and general purpose processors usually don't implement this mechanism. However, architectures in avionics systems tend to be specific processors. Such processors are more likely to provide cache locking mechanisms.

7.3 Future Research

The framework we presented in this thesis can be extended to solve multiple problems. Virtually any trace based algorithm can be integrated in the framework. Future research could lead to develop cache partitioning algorithms based on traces of memory profile. As the framework allows the profiling of caches, we can also extend its use to memory bus profiling or, more generally, memory use that is not in the scope of CPU caches. Only little work has been conducted to study cache locking and partitioning together in critical systems. We suggest that studying the interaction between the two concepts could lead to predictable and performant caches in ARINC-653 compliant multi-core systems.

The cache locking algorithm proposed in this work may also be revisited to solve different problems. In this thesis we aimed to reduce the amount of cache misses to increase performances. However, one may change this objective and propose new ways to compute the weight used to quantify the hit and miss ratio. Consumption-related models, interferences model or other models could be used to select which type of data should be locked in the cache. As we presented a genetic algorithm, multi-objective problem could be proposed such as reducing the energy consumption while providing RTOS services faster.

Our approach allows the framework to help system designers and integrators to mitigate other types of interferences. Shared caches partitioning is often used to mitigate shared caches interferences. However, the size of such partitions has to be determined. The framework, with specifically developed algorithms can help the integrator to make such decision. Bus contention can also be studied and avoided with our tool. The framework can be used to give bus access budget to each partition of a system. This can be achieved by adding to the framework a trace analyzing algorithm capable of detecting the need of bus accesses for each partition. Finally, IO or memory partitioning could be addressed in the same way. We only provide some example of interferences that can be mitigated thanks to our tool. However, many more issues can be addressed thanks to our tool (such as partition profiling, memory usage profiling, etc.).

In the aerospace domain, a number of issues rises from the advent of multi-core processors. However, our research demonstrated that architectures allowing cache locking are good candidates for multi-core adoption in this domain. Such techniques combined with other interference mitigation methods will lead to fully certifiable multi-core ARINC-653 RTOS.

REFERENCES

[1] D. Geer, “Chip makers turn to multicore processors,” *Computer*, vol. 38, no. 5, pp. 11–13, May 2005.

[2] ARINC, *ARINC Specification 653: Avionics Application Software Standard Interface*, Aeronautical Radio INC Std., 2015.

[3] *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*, RTCA SC-205; EUROCAE WG-12 Std., January 2012.

[4] L. M. Kinnan, “Use of multicore processors in avionics systems and its potential impact on implementation and certification,” Published in 2009 IEEE/AIAA 28th Digital Avionics Systems Conference, Orlando, FL, USA, October 2009, pp. 1.E.4–1–1.E.4–6.

[5] J. E. Kim *et al.*, “Integrated modular avionics (ima) partition scheduling with conflict-free i/o for multicore avionics systems,” Published in 2014 IEEE 38th Annual Computer Software and Applications Conference, Vasteras, Sweden, July 2014, pp. 321–331.

[6] J. Littlefield-Lawwill and L. Kinnan, “System considerations for robust time and space partitioning in integrated modular avionics,” Published in 2008 IEEE/AIAA 27th Digital Avionics Systems Conference, St. Paul, MN, USA, Oct 2008, pp. 1.B.1–1–1.B.1–11.

[7] R. Fuchsen, “How to address certification for multi-core based ima platforms: Current status and potential solutions,” Published in 29th Digital Avionics Systems Conference, Salt Lake City, UT, USA, Oct 2010, pp. 5.E.3–1–5.E.3–11.

[8] P. A. Laplante and S. J. Ovaska, *Real-Time Systems Design and Analysis: Tools for the Practitioner*, 4th ed. Wiley-IEEE Press, 2011.

[9] X. Jean *et al.*, “A software approach for managing shared resources in multicore ima systems,” Published in 2013 IEEE/AIAA 32nd Digital Avionics Systems Conference (DASC), East Syracuse, NY, USA, Oct 2013, pp. 7D1–1–7D1–15.

[10] P. Huyck, “Arinc 653 and multi-core microprocessors — considerations and potential impacts,” Published in 2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC), Williamsburg, VA, USA, Oct 2012, pp. 6B4–1–6B4–7.

[11] H. Agrou *et al.*, “A design approach for predictable and efficient multi-core processor for avionics,” Published in 2011 IEEE/AIAA 30th Digital Avionics Systems Conference, Seattle, WA, USA, Oct 2011, pp. 7D3–1–7D3–11.

[12] R. Fuchsen, “How to address certification for multi-core based ima platforms: Current status and potential solutions,” in *29th Digital Avionics Systems Conference*, Oct 2010, pp. 5.E.3–1–5.E.3–11.

[13] A. Löfwenmark and S. Nadjm-Tehrani, “Challenges in future avionic systems on multi-core platforms,” in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, Nov 2014, pp. 115–119.

[14] J. Littlefield-Lawwill and L. Kinnan, “System considerations for robust time and space partitioning in integrated modular avionics,” in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, Oct 2008, pp. 1.B.1–1–1.B.1–11.

[15] V. Paun, B. Monsuez, and P. Baufreton, “On the Determinism of Multi-core Processors,” in *French Singaporean Workshop on Formal Methods and Applications*, Singapour, Singapore, Jul. 2013. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01214947

[16] J. Nowotsch and M. Paulitsch, “Leveraging multi-core computing architectures in avionics,” in *2012 Ninth European Dependable Computing Conference*, May 2012, pp. 132–143.

[17] O. Kotaba *et al.*, “Multicore in real-time systems–temporal isolation challenges due to shared resources,” in *Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems*, 2013.

[18] S. Girbal *et al.*, “A complete toolchain for an interference-free deployment of avionic applications on multi-core systems,” in *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, Sept 2015, pp. 7A2–1–7A2–14.

[19] I. Bate and B. Lesage, “Exploring and understanding multicore interference from observable factors,” 2017.

[20] P. Parkinson, “Update on using multicore processors with a commercial arinc 653 implementation,” 04 2017.

[21] P. Radojkovic *et al.*, “On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments,” *TACO*, vol. 8, pp. 34:1–34:25, 2012.

[22] A. Löfwenmark and S. Nadjm-Tehrani, "Experience report: Memory accesses for avionic applications and operating systems on a multi-core platform," in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, Nov 2015, pp. 153–160.

[23] T. King, "Managing cache partitioning in multicore processors for certifiable, safety-critical avionics software applications," in *2014 IEEE/AIAA 33rd Digital Avionics Systems Conference (DASC)*, Oct 2014, pp. 8C3–1–8C3–7.

[24] H. Yun *et al.*, "Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013, pp. 55–64.

[25] B. C. Ward *et al.*, "Outstanding paper award: Making shared caches more predictable on multicore platforms," in *2013 25th Euromicro Conference on Real-Time Systems*, July 2013, pp. 157–167.

[26] R. Mancuso *et al.*, "Real-time cache management framework for multi-core architectures," in *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2013, pp. 45–54.

[27] X. Jin *et al.*, "A simple cache partitioning approach in a virtualized environment," *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pp. 519–524, 2009.

[28] H. Yun and P. K. Valsan, "Evaluating the isolation effect of cache partitioning on cots multicore platforms," in *Untitled*, 2015.

[29] X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 272–282, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/885651.781062

[30] and Z. Cui *et al.*, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *2012 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 367–375.

[31] Y. Kim *et al.*, "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2010, pp. 65–76.

[32] H. Kim *et al.*, “Bounding memory interference delay in cots-based multi-core systems,” in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014, pp. 145–154.

[33] A. Agrawal *et al.*, “Contention-Aware Dynamic Memory Bandwidth Isolation with Predictability in COTS Multicores: An Avionics Case Study,” in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Bertogna, Ed., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 2:1–2:22. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7174

[34] R. Inam *et al.*, “The multi-resource server for predictable execution on multi-core platforms,” in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014, pp. 1–12.

[35] T. Kelter *et al.*, “Evaluation of resource arbitration methods for multi-core real-time systems,” in *13th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), C. Maiza, Ed., vol. 30. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2013, pp. 1–10. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2013/4117

[36] J. E. Kim *et al.*, “Integrated modular avionics (ima) partition scheduling with conflict-free i/o for multicore avionics systems,” in *2014 IEEE 38th Annual Computer Software and Applications Conference*, July 2014, pp. 321–331.

[37] E. Betti *et al.*, “Real-time i/o management system with cots peripherals,” *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 45–58, Jan 2013.

[38] J. Rosen *et al.*, “Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip,” in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, Dec 2007, pp. 49–60.

[39] F. Boniol *et al.*, “Deterministic execution model on cots hardware,” in *Architecture of Computing Systems – ARCS 2012*, A. Herkersdorf, K. Römer, and U. Brinkschulte, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 98–110.

[40] R. Pellizzoni *et al.*, “A predictable execution model for cots-based embedded systems,” in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, April 2011, pp. 269–279.

[41] G. Durrieu *et al.*, "Predictable Flight Management System Implementation on a Multicore Processor," in *Embedded Real Time Software (ERTS'14)*, TOULOUSE, France, Feb. 2014. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01121700

[42] C. A. S. Team, "Cast-32, multi-core processors," USA, November 2016.

[43] P. Parkinson, "Update on using multicore processors with a commercial arinc 653 implementation," 04 2017.

[44] J. Delange and L. Lec, "Pok, an arinc653-compliant operating system released under the bsd license," 02 2019.

[45] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite," in *Proceedings of the 42Nd Annual Southeast Regional Conference*, ser. ACM-SE 42. New York, NY, USA: ACM, 2004, pp. 267–272. [Online]. Available: http://doi.acm.org/10.1145/986537.986601

[46] A. J. Smith, "Cache memories," *ACM Comput. Surv.*, vol. 14, no. 3, pp. 473–530, Sep. 1982. [Online]. Available: http://doi.acm.org/10.1145/356887.356892

[47] M. Lis *et al.*, "Memory coherence in the age of multicores," in *2011 IEEE 29th International Conference on Computer Design (ICCD)*, Oct 2011, pp. 1–8.

[48] C. B. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," in *2007 IEEE/AIAA 26th Digital Avionics Systems Conference*, Oct 2007, pp. 2.A.1–1–2.A.1–10.

[49] R. SC-205, "Do-178c, software considerations in airborne systems and equipment certification," 2011.

[50] C. A. S. Team, "Cast-20," USA, 2003.

[51] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247401

[52] Otto-Hahn-Ring, "Towards linux as a real-time hypervisor," in *Untitled*, 2009.

[53] R. SC-205, "Do-297, integrated modular avionics (ima) development guidance and certification considerations," 2005.

[54] I. Bate *et al.*, "Use of modern processors in safety-critical applications," *The Computer Journal*, vol. 44, no. 6, pp. 531–543, 2001.

[55] B. D. B. et al., "Impact of cache partitioning on multi-tasking real time embedded systems," in *2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2008, pp. 101–110.

[56] T. Adegbija and A. Gordon-Ross, "Phase-based cache locking for embedded systems," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, ser. GLSVLSI '15. New York, NY, USA: ACM, 2015, pp. 115–120. [Online]. Available: http://doi.acm.org/10.1145/2742060.2742076

[57] K. Kang, K. Park, and H. Kim, "Functional-level energy characterization of uc/os-ii and cache locking for energy saving," *Bell Labs Technical Journal*, vol. 17, no. 1, pp. 219–227, June 2012.

[58] Y. Liang and T. Mitra, "Instruction cache locking using temporal reuse profile," in *Design Automation Conference*, June 2010, pp. 344–349.

[59] W. Zheng and H. Wu, "Wcet: Aware dynamic instruction cache locking," *SIGPLAN Not.*, vol. 49, no. 5, pp. 53–62, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2666357.2597820

[60] F. Boniol, *New Challenges for Future Avionic Architectures.* Cham: Springer International Publishing, 2013, pp. 1–1. [Online]. Available: https://doi.org/10.1007/978-3-319-00560-7_1

[61] J. Reineke *et al.*, "Timing predictability of cache replacement policies," *Real-Time Systems*, vol. 37, no. 2, pp. 99–122, Nov 2007. [Online]. Available: https://doi.org/10.1007/s11241-007-9032-3

[62] A. Löfwenmark and S. Nadjm-Tehrani, "Challenges in future avionic systems on multi-core platforms," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, Nov 2014, pp. 115–119.

[63] S. Mittal, "A survey of techniques for cache locking," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 21, no. 3, pp. 49:1–49:24, May 2016. [Online]. Available: http://doi.acm.org/10.1145/2858792

[64] S. Plazar *et al.*, "Wcet-aware static locking of instruction caches," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser.

CGO '12. New York, NY, USA: ACM, 2012, pp. 44–52. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259023

[65] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, ser. RTSS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 114–. [Online]. Available: http://dl.acm.org/citation.cfm?id=827272.829141

[66] B. Cilku, D. Prokesch, and P. Puschner, "A time-predictable instruction-cache architecture that uses prefetching and cache locking," in *2015 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, April 2015, pp. 74–79.

[67] M. Loach and W. Zhang, "Exploring hybrid cache locking to balance performance and time predictability," in *SoutheastCon 2015*, April 2015, pp. 1–4.

[68] H. Ding, Y. Liang, and T. Mitra, "Wcet-centric partial instruction cache locking," in *DAC Design Automation Conference 2012*, June 2012, pp. 412–420.

[69] F. Ni *et al.*, "Combining instruction prefetching with partial cache locking to improve wcet in real-time systems," *PloS one*, vol. 8, p. e82975, 12 2013.

[70] T. Adegbija and A. Gordon-Ross, "PhLock: A Cache Energy Saving Technique Using Phase-Based Cache Locking," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 1, pp. 110–121, Jan 2018.

[71] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," *Machine Learning*, vol. 3, no. 2, pp. 95–99, Oct 1988. [Online]. Available: https://doi.org/10.1023/A:1022602019183

[72] W. M. Spears and K. D. De Jong, "On the virtues of parameterized uniform crossover," NAVAL RESEARCH LAB WASHINGTON DC, Tech. Rep., 1995.

[73] J. Gustafsson *et al.*, "The Mälardalen WCET Benchmarks: Past, Present And Future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, ser. OpenAccess Series in Informatics (OASIcs), B. Lisper, Ed., vol. 15. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010, pp. 136–146, the printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2010/2833

[74] "Snu-rt real-time benchmarks," http://archi.snu.ac.kr/realtime/benchmark.

[75] F. Semiconductor, *e200z760n3 Power Architecture® Core Reference Manual*, 2nd ed. Freescale Semiconductor Literature Distribution Center: Freescale, July 2012.

[76] W. Zheng and H. Wu, "Wcet: Aware dynamic instruction cache locking," *SIGPLAN Not.*, vol. 49, no. 5, pp. 53–62, Jun. 2014. [Online]. Available: http://doi.acm.org/10.1145/2666357.2597820

[77] R. Wilson, "What microprocessors need for military grade aerospace systems." [Online]. Available: https://www.electronicsweekly.com/news/what-microprocessors-need-for-military-grade-aerospace-systems-2015-11/

[78] Intel, "Intel atom processor n270." [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/36331/intel-atom-processor-n270-512k-cache-1-60-ghz-533-mhz-fsb.html

[79] J. Gustafsson *et al.*, "The mälardalen wcet benchmarks: Past, present and future," in *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.