Improving the reliability of heterogeneous multicore architecture for intelligent transportation systems

Fabien Bouquillon

École Doctorale - Mathématiques, sciences du numérique et de leurs interactions (EDMADIS)

Improving the Reliability of Heterogeneous Multicore Architecture for Intelligent Transportation Systems

Améliorer la fiabilité des architectures multicore hétérogènes pour les systèmes de transport intelligents

Thèse de doctorat
Informatique

Fabien Bouquillon

Soutenue le 18 Octobre 2022

Supervisors

Directeur de thèse
Giuseppe Lipari
Université de Lille

Directeur de thèse
Smail Niar
INSA Hauts-de-France
LAMIH UMR CNRS
Université Polytecnique Hauts-de-France

Jury

Rapporteur
Christine Rochange
Université Toulouse III-Paul Sabatier

Rapporteur
Olivier Sentieys
Université de Rennes 1

Examinateur
Florian Brandner
Télécom Paris

Président
Alberto Bosio
École Centrale de Lyon

Abstract

Real-time systems must provide functionalities that need to produce their results within predefined time windows. Some of these functionalities may be critical: if they produce wrong results or produce good results too late, failures occur which, in some extreme case, may cause the loss of human lives.

Intelligent transportation systems are a good example of real-time systems with critical functionalities. These vehicles embed complex features to enhance the driving experience, like the so-called Advanced Driver-Assistance Systems (ADAS). Such complex functionalities possess non-functional requirements, as the fact that they must produce results within precise time windows, or that must be robust to transient hardware faults. For example, the *automatic obstacle detection* feature assists the driver by alerting on obstacles on the vehicle path, so preventing accidents dues to inattention or fatigue: however the recognition of an obstacle must be performed in due time, otherwise the vehicle will harm someone or destroy itself.

Thus, a precise analysis of the temporal behavior of these systems is required to guarantee that all the timing constraints are respected. For modern vehicles, the electronic boards chosen for these systems need to provide high performance, since there is a pressure to integrate all the critical and non-critical functionalities on the same board and reduce cost. Moreover, with the miniaturization of electronic components and with the reduction in voltage, systems may be subject to transient faults during their lifetime, provoking unexpected failures. Thus, there is a need to have analysis taking into account both reliability and the respect of the timing constraints. In this thesis, we have proposed a set of solutions that are positioned at two levels:

1. We have developed models to analyze the execution time of real-time systems that integrate caches. The major scientific contribution of the thesis at this level is an improved analysis of the effect of preemptions on memory access times in a system scheduled by Earliest Deadline First.

2. We have also designed techniques to increase the reliability of real-time systems integrating caches. Our approach is novel in the sense that we propose a method to protect tasks code from transient faults in the cache by adding protection mechanisms to the tasks code while respecting the timing constraints.

Our work is original in that it lies at the intersection of several areas:

1. The domain of real-time critical systems;

2. The field of processor system architectures in general and that of programmable embedded systems integrating caches in particular;

3. The field of reliability and robustness of real-time critical systems.

Résumé en Français

Les systèmes temps réel implémentent des fonctionnalités qui doivent produire leurs résultats dans une fenêtre de temps donnée. Certaines de ces fonctionnalités peuvent être critiques. Si une fonctionnalité critique produit un résultat erroné ou produit un bon résultat au-delà d'une certaine limite temporelle, une défaillance se produit. Lorsqu'une défaillance se produit, des événements catastrophiques peuvent s'ensuivre, comme la perte d'une vie humaine. Les systèmes de transport intelligents sont un bon exemple de systèmes temps réel dotés de fonctionnalités critiques. Ces véhicules intègrent des fonctionnalités qui améliorent la conduite en aidant le conducteur, comme les systèmes avancés d'assistance au conducteur c(ADAS). Ces fonctionnalités doivent produire de bons résultats dans une fenêtre de temps précise comme pour la détection d'objets pour éviter de blesser des usagers de la route. Une analyse précise du comportement de ces systèmes est par conséquent nécessaire pour garantir que les contraintes de temps sont respectées. Pour les véhicules modernes, les cartes électroniques choisies pour ces systèmes doivent être très performantes, et doivent intégrer toutes les fonctionnalités critiques et non critiques sur la même carte afin de réduire les coûts. Avec la miniaturisation des circuits électronique et la réduction de la tension électrique, les systèmes peuvent subir des fautes transitoires pendant leur durée de vie, provoquant des erreurs dans leurs comportements. Il est donc nécessaire d'avoir une analyse prenant en compte à la fois la fiabilité et le respect des contraintes temporelles. Dans cette thèse, nous avons proposé un ensemble de solutions qui se positionnent à deux niveaux :

1. Nous avons mis au point des modèles d'analyse de temps d'exécution des systèmes temps-réels intégrants des mémoires caches. La contribution scientifique majeure de la thèse à ce niveau est une meilleure analyse de l'impact des préemptions entre les tâches sur les temps d'accès à la mémoire dans un système ordonnancé par Earliest Deadline First.

2. Nous avons aussi conçu des techniques pour augmenter la fiabilité des systèmes temps-réel intégrant des mémoires caches. Notre approche est nouvelle dans le sens où nous proposons une méthode qui permet de protéger le code des tâches des fautes transitoires dans la mémoire cache en ajoutant des mécanismes de protection au code des tâches tout en respectant les contraintes temporelles.

Comme on peut le voir, notre travail est original du fait qu'il se trouve à l'intersection de plusieurs domaines :

1. Le domaine des systèmes temps-réel critiques;
2. Le domaine des architectures de systèmes de processeurs en général et celui des systèmes embarqués programmables intégrant des mémoires caches en particulier;
3. Le domaine de la fiabilité et de la robustesse des systèmes temps-réels critiques.

Acknowledgement

I would like to express my gratitude to Giuseppe Lipari and Smail Niar for introducing me to the world of research and guiding me through this thesis. I can't thank them enough for their feedback and support over the last four years. I would also like to express my gratitude to the permanent members of the two teams I joined for their advice, particularly Clément Ballabriga for assisting me with OTAWA and Julien Forget for his precious reviewing. Of course, I'd like to express my gratitude to all of the PhD students, former PhD students, and post-docs who shared my office with me. I would also like to express my sincere thanks to Filip Marković, who assisted me in producing a counter-example of his method.

I would thank Fédération de Recherche Transports Terrestres & Mobilité FR TTM (FR3733), supported by CNRS, Université Polytechnique Hauts-de-France (UPHF), Université de Lille (UL) and Centrale Lille Institut (CLI), for the financial support provided during the 3 years of my Ph.D. This Ph.D. corresponds to a cooperation between two laboratories : LAMIH at UPHF and CRISTAL at UL.

I'd also like to show my thankfulness to the members of my thesis jury for their remarks and advices. Finally, I want to express my sincere thanks to my relatives and friends for their support over the past four years.

Contents

List of Figures

List of Tables

Part I

Motivation and Background

Chapter 1

Problem & Motivation

Contents

This chapter will introduce real-time systems and Intelligent Transportation Systems (ITS), using modern vehicles as an illustration. Then, I will describe the unresolved problems and challenge handle by these systems. Finally, this chapter will conclude with a summary of the thesis' contents.

1.1 Real-Time Systems

Real-time systems are computing systems that must react within precise time constraints to events in the environment. As a consequence, the correct behavior of these systems depends not only on the value of the computation but also on the time at which the results are produced [28, 68].

A real-time system interacts with its environment through the use of sensors and/or actuators. Given the dynamic nature of the environment, the system's response to a change in the environment cannot be postponed indefinitely. The state of the system at any point in time must be correct in relation to the state of the environment at the same point in time. Any state of the system that causes an accident in the environment or in the system itself, such as the destruction of system components, is regarded as a failure.

Aircraft autopilots, nuclear power plant system control, robotics and virtual reality applications are just a few examples of real-time systems in use today. Other examples include autonomous vehicles or missile defense applications that are used to detect, track, and intercept missiles in order to prevent the destruction of an area or ship. Indeed, we can find real-time systems in a wide variety of fields, each of these systems must adhere to the temporal constraints defined by their mission in order to perform effectively. This does not imply that all of these systems must respond immediately; rather, they must respond in a timely and appropriate manner. For example virtual reality applications must react to the user's movement at a rate such that the user does not perceive any delay between its actions in the real world and those in the virtual reality. It is also not necessary to select the most performant electronic board if two different electronic boards with varying levels of performance can integrate virtual reality applications while ensuring that their respective timing constraints are respected throughout the system's life. Indeed, a system may seem slow to us, but if we are sure that its temporal constraints will be respected, there is no reason to increase its speed. Also, it is essential to recognize that the average reaction time of a system is not a guarantee of its ability to respect its temporal constraints. For example, a system that reacts very quickly on average but too slowly in some rare situations is not desirable. Indeed, without a guarantee that the system will always react in time, one can assume that accidents will occur, with potentially catastrophic consequences.

1.2 Intelligent Transportation Systems

A real-time system is exemplified by intelligent transportation systems. They evolve in an ever-changing environment and must make decisions based on their perception

within certain time constraints imposed by the environment. Intelligent Transportation Systems are transportation management systems; they integrate not only vehicles, roads, communication, and control infrastructures, but also the people who are present in the surrounding area. They seek to alleviate traffic congestion and pollution in the city, while also increasing vehicle and pedestrian safety.

Furthermore, we can consider these systems as cognitive systems according to [34], i.e. systems that learn to respond to a particular situation based on data collected during previous interactions with the environment. Three steps can be used to model the functioning of these systems.

- The collection of data defining the environmental context;
- The analysis of data and the planning of actions;
- Action execution.

Today, ITS are critical, as existing transportation infrastructures are insufficient to solve existing transportation problems [8]. They rely on technologies such as the Global Positioning System (GPS) to estimate traffic congestion, the communication technologies between vehicles and the network to share or get data with vehicles or with the intelligent transportation system's control center. Additionally, they are based on functionalities embedded in modern vehicles that aid the driver in her driving by utilizing Advanced Driver Assistance Systems (ADAS) [76]. Each of these technologies can be classified into two distinct categories

- Critical functionalities;
- Functionalities for comfort purpose.

1.2.1 Critical functionalities

All functions that have a significant control over the vehicle or that have the purpose of preventing an immediate accident are classified as critical features. These features must adhere to strict time constraints in order to avoid collisions. A delayed response can have severe consequences, including the loss of a human life. For instance, the following functionalities can be considered critical:

Cruise control. With the Cooperative Adaptive Cruise Control (CACC) [63], cruise control features for automobiles have become a reality. This technology aims to control a vehicle on a road in conjunction with other vehicles, forming what is known as a distributed hybrid system [56].

Collision Avoidance Systems (CAS). These systems are based on active sensors such as radar, laser and lidar or passive sensor as optical camera or acoustic sensors to detect information on the environment and taking decision to avoid collision [65].

Antilock Braking Systems (ABS). These systems are active safety systems and are designed to maintain steering control during heavy braking by preventing the wheels from locking [38].

Autonomous Emergency Braking (AEB). This feature is intended to detect situations in which emergency braking is required and to bring the vehicle to a complete stop in these circumstances [40].

1.2.2 Functionalities for comfort purpose

These features are intended to enhance the driver's overall comfort. They entail automating non-mission critical tasks, as well as advising and providing additional information to the driver. These include the following features:

Automatic Climate Control: which automatically regulates the air humidity and temperature inside the vehicle.

Automatic high beam control: which controls the vehicle's headlights in order to avoid blinding an oncoming vehicle detected by a front-facing camera.

Biometric seat: utilizes a camera to determine the driver's level of stress, distraction, and fatigue. When the driver's fatigue level is too high, for example, this feature alerts him and suggests a break. According to Mukhtar et al. [65], the primary causes of accidents are driver inattention, fatigue, and immature behavior [69]. Additionally, more than half of accidents involving inattention are caused by driver distractions [52] such as eating, drinking, or using a phone or other multimedia device in the vehicle [78]. Fatigue is also a significant factor, accounting for between 25% and 30% of road accidents [36].

GPS Tracking: enables the vehicle's position in the environment to be tracked in real time, assisting the driver on his route and alerting him when he exceeds the route's maximum allowed speed.

1.3 Autonomous Vehicles

Additionally, modern vehicles are increasingly becoming self-driving. However, various classifications propose several levels of autonomy, with the final level representing a vehicle with perfect autonomy [31, 1, 30, 12]. We present in Table 1.1 the SAE international classification.

Level	Name	Description

0	No Driving Automation	The driver must perform all the driving tasks.
1	Driver Assistance	The driver must perform all tasks that the assistant cannot, as well as supervise and intervene as necessary to ensure that the vehicle behaves appropriately. The driver also has the responsibility to choose when the assistance should intervene and when it should stop, moreover he must also be able to regain control of the vehicle at any time, even if he does not want to. When activated, the driving automation system performs either the longitudinal **or** lateral vehicle motion control subtask and disengages immediately upon driver request.
2	Partial Driving Automation	The driver must perform all tasks that the assistant cannot, as well as supervise and intervene as necessary to ensure that the vehicle behaves appropriately. The driver also has the responsibility to choose when the assistance should intervene and when it should stop, moreover he must also be able to regain control of the vehicle at any time, even if he does not want to. When activated, the driving automation system performs both the longitudinal **and** lateral vehicle motion control subtask and disengages immediately upon driver request.
3	Conditional Driving Automation	The driver decides when to activate the automated driving system. When it is activated the driver must maintain vigilance and be prepared to regain vehicle control. This may occur if the system requests driver intervention or if the driver detects a system error. The automated driving system performs the tasks that are possible for it to do. If the automated driving system detects a situation that it cannot handle, it requests that the driver regain control of the vehicle. In addition, if the driver requests to regain control of the vehicle, the vehicle must give it back immediately.
4	High Driving Automation	When the automated driving system is engaged, the driver becomes a passenger. He may request to regain control of the vehicle, but there is no assurance he will receive it immediately. In some instances, the automated driving system may delay handing over control to the driver. When the automated driving system's operating range is exceeded, the system requests the intervention of the driver.

5	Full Driving Automation	When the automated driving system is engaged, the driver becomes a passenger. He may request to regain control of the vehicle, but there is no assurance he will receive it immediately. In some instances, the automated driving system may delay handing over control to the driver.

Table 1.1: SAE Autonomous vehicle classifications [30]

While fully autonomous vehicles are not yet commercially available, some vehicles already possess advanced levels of autonomy. Indeed, Tesla vehicles already meet the SAE's level 2 autonomy standard. We can conclude that, as innovation for autonomous vehicles advances, correct embedded computing systems will be required to integrate additional functionalities and to handle the increased number of tasks. Additionally, these new functionalities will become increasingly critical, necessitating adherence to extremely stringent time guarantees.

1.4 Problems

The new technologies presented so far are an excellent way to address the lack of safety for vehicles and pedestrians caused by city traffic congestion.

However, at the same time there is a pressure for these vehicles to embed more functionalities to enhance driving, and a budgetary pressure regarding electronic components and design and development costs. In short, there is a need for high performance computing platforms with affordable price. Component-Off-The-Shelf (COTS) microprocessors are good candidates for these platforms. An example of architecture that meets the demand of such systems is the ARM Big Little, which allows a tradeoff between performance and energy consumption [27].

However, system analysis is extremely expensive. In avionics, the verification phase accounts for an average 60% of the project budget [18]. However, when we consider the financial loss caused by an aircraft accident, this cost becomes reasonable. In the automobile industry, budgets are more constrained, but financial losses during an accident can be substantial; therefore, the verification phase is just as crucial. A variety of issues contribute to the cost of the analyses.

The need for a precise model of the hardware for analysis tools: having an accurate model of complex systems becomes increasingly difficult. In addition, the exact specification for certain components, such as GPUs, is not publicily available, it is known only to the manufacturer.

The need for a precise model of the software: obtaining an accurate software model is difficult for the same reason as hardware models, namely the increasing complexity of

systems. Tasks can be dependent on one another, requiring the completion of another task before their execution can begin. There is, also, a risk of contention between tasks over shared resources that can be utilized by only one task at a time. In addition to temporal constraints, task scheduling should also take into account system temperature and energy consumption, among other factors.

The need for having the least pessimistic analysis: the majority of system analyses solutions in the literature are static, where an execution time is obtained without executing the code. These analyses extract properties from the binary code as well as the hardware and software models that are valid for the lifetime of the system. However, it is hard to extract certain properties, necessitating a trade-off between accuracy and analysis complexity [81]. The pessimistic nature of these analyses is a result of their consideration of the worst-case scenario.

The COTS microprocessors' high performance is a result of their complex features, which include a shared bus, direct memory access (DMA), pipelines, branch prediction and cache memories. They were designed to boost the average performance of commercially available processors. However, their design does not account for the need to predict their behavior, which makes their analysis extremely complex: for instance, to compute precisely the worst-case execution time of a program, it may be necessary to model domino effects or timing anomalies for pipelines and branch prediction [14]. To circumvent these issues, the models used to represent their behavior contain a large number of assumptions, which add significant amounts of pessimism to the execution time of a task.

1.4.1 Models of cache memories

An important set of simplifying assumptions is done in the analysis of cache memories. Caches are used to bridge the speed gap between the processor and the main memory. These memories store a copy of the data accessed from memory closer to the processor, thereby speeding up subsequent accesses. The speed with which data is accessed is determined by the cache memory's contents. However, its content is contingent upon the memory access history, which is complex to predict. Additionally, these memories are susceptible to transient faults as a result of the miniaturization of electronic chips and the reduction of electric voltage. These faults can result in bit-flipping in the cache memories, resulting in data corruption. The cache memories thus present a *lack of predictability* and a *lack of reliability*.

Additionally, cores may request access to memory via the shared bus at the same time in multicore systems, resulting in bus contention [71]. However, these accesses are caused by cache misses. As the classification of a memory access as a cache hit or a cache miss depends on the cache's content, it is difficult to predict bus contention. Thus, enhancing cache prediction can consequently enhance bus-level contention prediction.

1.5 Contributions and organization of the thesis

This thesis proposes a more accurate representation of software in the analysis and a reduction in the analysis' pessimism, focusing on the cache memory utilization by real-time tasks.

First in Chapter 2, we describe the model used to represent tasks, as well as the various scheduling algorithms and schedulability analyses proposed in the literature. The cache memories and their vulnerability are then presented in greater detail in Chapter 3. This chapter focuses on the improvements made to the task model and scheduling analysis in order to account for cache memories in these analyses, as well as the state of the art regarding these analyses and models. In addition, we will describe how to measure the vulnerability of a cache memory and the various mechanisms proposed to safeguard these memories from the literature. In Chapter 4 we present our contribution to improve the predictability of the cache memories while in Chapter 5 we address the problem of reliability of the caches. Finally in Chapter 5.5, we conclude our contributions, we sumarize the most important results and we discuss some possible extensions of our work.

Chapter 2

Real-Time Systems Model

Contents

In this chapter, I will discuss how real-time systems are modeled in the literature and in this thesis, as well as the analyses that allow the worst-case execution time of a task to be computed. Also, we will see the various algorithms used to schedule the execution of tasks, as well as the analyses that ensure that this scheduling adheres to temporal constraints.

2.1 Task Model

A real-time system can be model as a set of concurrent software tasks. A task is a *software thread*, that is a sequential piece of code that executes onto a multi-threaded operating system. In Listing 2.1 we see the pseudo-code of a typical real-time task in a POSIX-like environment: after a first phase that initializes the local and global variables of the task, it enters a (finite or infinite loop) where, after executing its functionality the task suspends itself by invoking a function (in the pseudo-code of Listing 2.1 this function is denoted as `wait_for_activation()`, but it could be different, depending on the operating system). While the thread is suspended, other concurrent threads can execute. To restore execution, the task must be activated either by a timer (e.g. a *periodic* task) or by an external event (for example, an interrupt from a device driver, or a signal coming from another software task), in which case we have a *sporadic* or *aperiodic* task.

To analyze a system consisting of a set of concurrent tasks, we will now provide a mathematical model of the temporal behaviour of the tasks. In this thesis, we consider a set of N independent real-time tasks, denoted as $\mathcal{T} = \{\tau_1, \cdots, \tau_n\}$. A real-time task τ_i is described as a (possibly infinite) succession of jobs. In Listing 2.1, a job is an instance of the execution of the piece of code inside the while loop, between two consecutive activations. We denote with $j_{i,k}$ the k^{th} jobs of τ_i. In our model, a job $j_{i,k}$ is characterized by three parameters. The first parameter is the *arrival time*, noted $a_{i,k}$, that correspond to the moment when the job is ready to start its execution; notice that this instant may not be the start of the execution of the job since other jobs from other task may have to

```
void *mytask(void *args)
{
  // Initialization

  while(TRUE) {

    // code of the task's instance
    // (job)

    wait_for_activation();
  }
}
```

Listing 2.1: Code of a real-time task

finish their execution before. The second parameter is the job's absolute deadline, noted $d_{i,k}$, that corresponds to the moment before which $j_{i,k}$ need to have finished its execution. Finally, the latest parameter is the execution time of the job, noted $c_{i,k}$.

We called *periodic task*, a task that has its jobs arriving in a periodic scheme. Periodic tasks are the most predictable tasks since we know exactly the arrival of each of their jobs. In Listing 2.1, the function `wait_for_activation()` suspends the task waiting for activation from the software timer provided by the OS. For example, in the POSIX standard, the function `wait_for_activation()` is replaced by function `clock_nanosleep()`.

Sporadic tasks are tasks that have their jobs arriving at or after a minimum interarrival time since the arrival time of the previous job. These tasks are less predictable than periodic tasks since we do not know exactly the arrival time of the jobs. They are used to model systems where tasks are activated by external events which are not triggered regularly. For example, the reception of a packet from an Ethernet network is often modelled by a sporadic task. However, since we can establish a minimum interarrival time, we can upper bound the load caused by a sporadic task on the system by considering its jobs to arrive at their earliest time.

Finally, there exists another category of tasks, the *aperiodic tasks* for which we do not have any guarantees concerning the arrival time of their jobs. They are the least predictable type of tasks because we cannot upper bound their workload.

The classic model by Liu and Layland [55] characterizes a task τ_i with the tuple (C_i, D_i, T_i). With C_i the worst case execution time of its jobs $C_i = \max_{\forall k}(c_{i,k})$, D_i the relative deadline, i.e. the length of any interval $[a_{i,k}, d_{i,k}]$, and T_i the minimum interarrival time between two jobs (if we consider sporadic tasks) or the period of the arrival times (if we consider periodic tasks). Thus, in both cases $T_i = \min_{\forall k}(a_{i,k+1} - a_{i,k})$.

We denote by $U_i = C_i/T_i$ the utilization of τ_i.

An example of a sporadic task model is shown in Figure 2.1. It shows 3 jobs of task τ_i, with their execution represented by the blue boxes.

Also, there are two different models of tasks concerning the deadlines, the constrained deadline tasks and the implicit deadlines tasks. In the first model the relative deadline is inferior or equal to the period $D_i \leq T_i$; in a case of implicit deadline tasks, the deadline is equal to the period $D_i = T_i$.

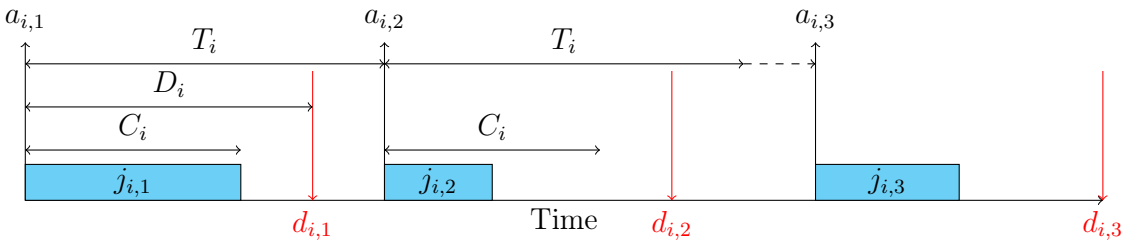


Figure 2.1: Sporadic task model

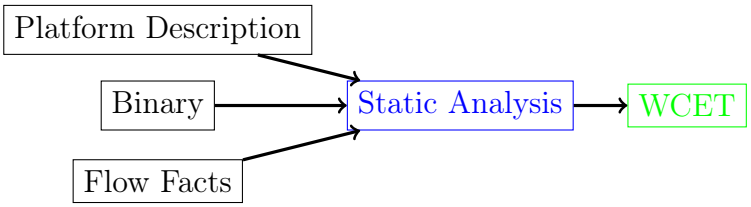


Figure 2.2: Static analysis

2.2 WCET estimation

The execution time of a task depends on several parameters such as the platform characteristics, its status or the data that are handled by the task. To estimate the worst-case execution time C_i of a task, measuring the execution time of numerous instances of the task might be seen as a good idea. However, it is not the case, since the instance that produce the worst case execution may not be observed during the measurements. In this case the value obtained may be too optimistic. To provide a safe upper bound of worst-case execution time C_i, we must statically analyze the code of the task.

Static analyses extract properties from the code that are applicable to all instances (jobs) of a program. The analysis can be done on either the source code or the compiled binary. On the other hand, it is impossible to know a large portion of the properties about a code because doing so would require solving undecidable problems. In these instances, static analyses treat those properties as unknown, and if these properties have an impact on the WCET, the worst-case scenario is always considered [81]. The WCET of a program obtained from a static analysis is therefore an upper bound on the actual value.

Additionally, a precise model of the platform on which the program will run is required for static analysis. The more precise the model, the more precise the analysis, and the closer the WCET upper bound is to the actual value. Figure 2.2 shows a generic flow to obtain an estimation of the WCET. In the diagram, the program's binary code and the platform's description serve as inputs for the static analysis. The Flow Facts of the program, are already-known properties of the program's execution, such as the maximum number of loop iterations. In a static analysis, every task is analyzed in isolation. The impact of the other tasks on the execution time of the analyzed task is not considered. This simplifies considerably the analysis, however it represent a source of imprecision: additional analysis will be required when integrating all tasks in the system.

Several estimation strategies have been proposed for the WCET [85]. According to Bai et al. [15], the most prevalent technique for performing a static analysis is the Implicit Path Enumeration Technique (IPET) [54]. This procedure is based on the basic blocks (BBs), those are blocks of sequential instructions that do not contain any branch or call except at their ends. The IPET analysis consists of four steps:

- Analysis of the various execution paths to construct the BBs, the Control Flow Graph (CFG), and to identify the unfeasible paths;
- Analysis considering hardware features based on execution history such as cache memory or branch prediction;

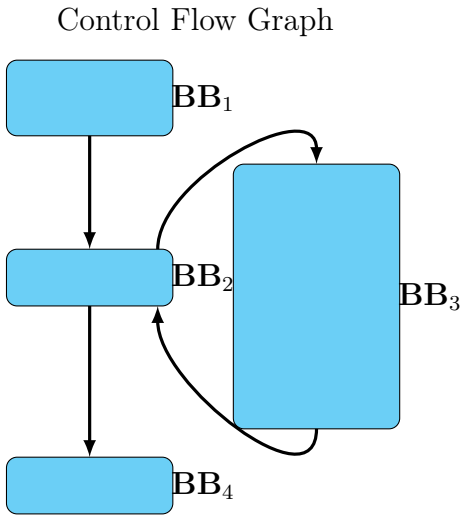


Figure 2.3: Task CFG containing a loop

- Compute the WCET of the BBs;
- Compute the WCET of the task with an Integer Linear Program (ILP).

The CFG referred to in the first step of the IPET method corresponds to a graph depicting the various possible execution paths of the analyzed task. In order to construct this graph, the analysis connects the BBs with edges representing the destination of the calls or the branches.

An example of task CFG is shown in Figure 2.3. In this figure we can see a task with 4 BBs, BB_1, BB_2, BB_3, and, BB_4. Also, this task includes a loop represented by BB_2 and BB_3. The entry and the exit of the task are respectively BB_1 and BB_2 in this example.

In this thesis we use OTAWA [17] to perform the static analysis of a task, which is based in the IPET method.

2.3 Real-Time Tasks Scheduling

According to Section 2, a real-time system comprises numerous tasks. Therefore, these tasks must share physical resources like CPUs and cache memories. Certain resources, such as the CPU, can only be utilized by one task at a time. Therefore, resource utilization must be scheduled between tasks.

The WCET analysis is insufficient to ensure that the time constraints will be met, as this step assumes that each task will run independently on its own platform, without sharing any resource with the others.

On the Figure 2.4 is a an example of a possible schedule of two tasks τ_i and τ_j that are sharing a platform with a single core. The task τ_i has WCET $C_i = 5$ and deadline $D_i = 6$, τ_j has WCET $C_j = 3$ and deadline $D_j = 7$. If we consider the two tasks independently, both of them will meet their deadlines. Nonetheless, as demonstrated by this example, τ_j misses a deadline due to the fact that a single core can only execute one task at a time.

Figure 2.4: Scheduling of 2 tasks.

• *Single core* • Multicore – Global – Partitioned	• Online scheduling – Task level fixed priority: Deadline Monotonic (DM), Rate Monotonic (RM), Fixed Priority in general. – *Job level fixed priority: Earliest Deadline First (EDF).* – Dynamic priority: P-Fair, Least Laxity First (LLF). • Offline scheduling – Time Triggered	• *Preemptive* • Limited Preemption • *Non-Preemptive*

Table 2.1: Scheduling Taxonomy

Therefore, it is necessary to ensure that all time constraints are met when scheduling tasks.

Table 2.1 presents a taxonomy of scheduling algorithms. The left column displays the various methods of task allocation among system cores. In a system with a single core, all tasks will utilize the same core. There are two options for multicore systems: allowing tasks to be scheduled on any core (*Global*) or assigning a specific core to each task (*Partitioned*). In the latter case, each instance of the task must run exclusively on this core. This is equivalent to considering multiple single-core systems, although you must take into account that other system features, such as the bus and cache memory, are still shared.

In the middle column, the various scheduling algorithms are represented, including

online schedulers that will schedule the tasks during the execution of the system and offline schedulers that will determine which task must be executed based on an offline schedule. *Time-triggered* is an example of an offline scheduler.

For online schedulers, there are three distinct categories of algorithms. The first category corresponds to algorithms that assign a fixed priority to each task, and each instance of a task will have that priority. When several tasks are ready for execution, the one with the highest priority will be executed first. This category contains all variations of Fixed Priority (FP) [37, 55], such as Rate Monotonic (RM), which assigns a priority inversely proportional to the period of the task, and Deadline Monotonic (DM), which assigns a priority inversely proportional to the relative deadline of the task, such that the task with the shortest relative deadline will have the highest priority.

The second category corresponds to algorithms that assign a fixed priority to each instance (job), so that two instances of the same task may not have the same priority. This category includes Earliest Deadline First (EDF) [55], an algorithm that executes the job with the earliest absolute deadline that is ready to be executed.

In the last category, algorithms that assign dynamic priority to each instance are presented. In these algorithms, we find Least Laxity First (LLF) [64], which executes the instance with the smallest laxity (the remaining time between the expected end of execution and its absolute deadline).

The right column represents the various preemptions modes permitted for online schedulers. In preemptive systems, all preemptions are permitted, meaning that a task can be interrupted at any point during its execution. In systems with limited preemption, a task can only be preempted at certain specific points of its execution. And finally, there are systems that prohibit preemption.

In this thesis, we focus on single-core systems with EDF scheduling so that we can later adapt the solutions to partitioned multicore systems. In addition, both fully preemptive and non-preemptive systems are addressed.

We will now present the FP and EDF schedulers for single-core systems and their scheduling analysis in greater detail.

2.4 Fixed Priority

Fixed Priority (FP) is a family of schedulers rather than a single scheduler. FP consists of assigning each task a fixed priority. Consequently, each job within a task has the same priority. The scheduler executes the ready task with the highest priority at any instant t. This priority can be assigned manually by developers, or calculated by heuristics such as *Deadline Monotonic* (DM), which assigns a priority P_i inversely proportional to the relative deadline of the task τ_i ($P_i \cong 1/D_i$). In the case of *Rate Monotonic* (RM), the priority of a task τ_i is inversely proportional to its period that is $P_i = 1/T_i$. Furthermore, DM is optimal in the class of fixed priority preemptive scheduling of periodic and sporadic tasks with implicit and constrained deadlines, while RM is optimal only if all the deadlines are implicit [53]. In other words, if it exists a schedule with any fixed priority assignment

that respects time constraints, then scheduling with RM or DM will also respect time constraints.

2.4.1 Preemptive tasks schedulability analysis

To ensure that time constraints are respected under RM scheduling, we can verify that the utilization of the system does not exceed U_{lub} [55],

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq U_{lub} \qquad U_{lub} = N(2^{1/N} - 1) \tag{2.1}$$

However, this is a sufficient test, as there are schedulable systems that do not satisfy this condition. Conversely, if the system utilization is greater than 1, the system cannot be scheduled.

To ensure that time constraints are met with FP in general when each task has a different priority, we can conduct the exact test below. If the maximum response time R_i for each task τ_i (the maximum time between the activation of a task's job and the completion of its execution) is not greater than D_i [11], then the system is schedulable.

$$\forall \tau_i \in \mathcal{T} | R_i \leq D_i \tag{2.2}$$

A task's response time is calculated iteratively by Equation (2.4). The initial step is to assume that $R_i = C_i$. Then, the following iterations consist of adding to the WCET of the task the execution time of tasks that can preempt it based on the response time calculated in the previous iterations. When the response time exceeds the relative deadline or when the response time does not change from one iteration to the next one, we terminate the procedure.

$$R_i^0 = C_i \tag{2.3}$$

$$R_i^{k+1} = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i^k}{T_j} \right\rceil \cdot C_j \tag{2.4}$$

2.4.2 Non-preemptive tasks schedulability analysis

The schedulability analysis for Fixed Priority Non Preemptie (FP-NP) is identical to that for FP with shared critical resources. We assume that the processor is a critical shared resource protected by a semaphore, hence a task that has begun execution cannot be interrupted. Therefore, the response time for FP-NP with a different priority per task can be calculated as follows [32, 33]:

$$R_i = S_i + C_i \tag{2.5}$$

Where S_i corresponds to the worst start time of the task τ_i,

$$S_i^0 = B_i \tag{2.6}$$

$$S_i^{k+1} = B_i + \sum_{\forall j \in hp(i)} \left(\left\lfloor \frac{S_i^k}{T_j} \right\rfloor + 1 \right) \cdot C_j \tag{2.7}$$

with B_i the maximum blocking time a job of task τ_i can experience [73]. B_i can be calculated as follows:

$$B_i = \underset{\forall j \in lp(i)}{max} (C_j) \tag{2.8}$$

In the previous equation $lp(i)$ represents the set of tasks with a lower priority than τ_i. We denote by $hp(i)$ the set of tasks with a higher priority than τ_i.

2.5 Earliest Deadline First

Earliest Deadline First (EDF) is a scheduler that assign a priority for each job of a tasks inversely proportional to its absolute deadline. With EDF, the ready job with the earliest deadline is executed first. The *preemption-level* is defined for a task τ_i as $\pi_i = 1/D_i$. Baker [16] proved that, in EDF scheduling, a task τ_j may preempt a task τ_i only if $\pi_j > \pi_i$. We assume that tasks are sorted in non-ascending order of preemption-level: for any two tasks τ_j and τ_i, $\pi_j > \pi_i$ implies $j < i$. EDF is also optimal, which mean that if there exists an online scheduler that respects the timing constraint for a set of tasks, EDF can also schedule these tasks while respecting their timing constraints. In this chapter, the scheduling analyses for EDF are presented for systems with and without preemptions.

2.5.1 Preemptive tasks

To analyze the schedulability of a preemptive system under EDF scheduling, we can use the processor demand bound analysis first proposed by Baruah et al. [19]. This analysis works for tasks with constrained and implicit deadlines. It consists in computing the processor demand of the task set at each deadline in the interval $[0, L]$, where L is an estimated upper bound on the first idle time. The demand bound function [19] of a task τ_i in the interval $[0, t]$ can be computed as:

$$\mathsf{dbf}_i(t) = \eta(i, t) \cdot C_i \tag{2.9}$$

With $\eta(i, t)$ an upper bound to the number of instances of the sporadic task τ_i that have arrival and deadline in interval $[0, t]$:

$$\eta(i, t) = \max \left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1 \right) \tag{2.10}$$

The demand bound of the task consists of the sum of the demand bound for each task in the task set:

$$\mathsf{dbf}(t) = \sum_{i=1}^{N} \mathsf{dbf}_i(t) \tag{2.11}$$

If for each deadline the demand bound of the task set is less than the deadline, then the system is schedulable. Otherwise, it is not. Thus, the system is schedulable if and only if:

$$\forall t \leq L, \quad \mathsf{dbf}(t) \leq t \tag{2.12}$$

2.5.2 Non-preemptive tasks

In the case of non-preemptive tasks, we employ the following test, proposed by Jeffay et al. [46], to ensure that time constraints are guaranteed. However, this test is valid only for systems with tasks that have implicit deadlines.

It consists of two conditions, the first of which verifies that the processor's global usage does not exceed its computing capacity:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1 \tag{2.13}$$

The second condition verifies that blocking between tasks does not jeopardize conformance to time constraints:

$$\begin{aligned} &\forall i, 1 < i \leq n;\ \forall t, T_1 < t < T_i; \\ &t \geq C_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t-1}{T_j} \right\rfloor \cdot C_j \end{aligned} \tag{2.14}$$

Similarly to FP, we can consider the processor to be a critical shared resource. The blocking time is also calculated in the same manner, with the exception that $lp(i)$ represents the set of tasks with a lower preemption-level than τ_i. Baker et al. [16] proposed another test to verify the schedulability of system with implicit deadlines based on the utilization of the system:

$$\underset{k=1,\cdots,N}{\forall k} \left(\sum_{i=1}^{k} \frac{C_i}{D_i} \right) + \frac{B_k}{D_k} \leq 1 \tag{2.15}$$

When dealing with tasks that have constrained deadlines, the demand bound function can be modified to consider the blocking time as follows [20]:

$$\mathsf{dbf}(t) = B(t) + \sum_{i=1}^{N} \mathsf{dbf}_i(t) \tag{2.16}$$

$$B(t) = max(C_j | D_j > t) \tag{2.17}$$

Chapter 3

Cache memories

Contents

The high performance, measured by the number of instructions executed per second, of a microprocessors is limited by the speed of data transfer between the main memory (DRAM) and the processor [43], which necessitates the use of cache memories. Cache memories maintain a copy of the accessed data in order to speed up the data transfer for future accesses. Cache memory access latency is significantly lower than DRAM access latency. However, cost per byte of a cache memory is much higher than DRAM, so only small sizes are possible. Consequently, cache memory cannot replace main memory.

3.1 Cache Memories Hierarchy

There are multiple cache memories in multicore systems, which can be shared by multiple cores or reserved for a specific core. In addition, some of them may contain only instructions, others only data, or both instructions and data simultaneously. The majority of multicore systems consists of three levels of cache memory. The closest level to the cores consists of an instruction cache (IL1) and a data cache (DL1) for each core. These caches are not shared with other cores and are directly connected to the core. The second level consists of one cache memory (L2) per core; these cache memories are linked to the IL1 and DL1 of their respective core; these caches may contain both instructions and data simultaneously. The third and final level contains a unique cache memory (L3) for all multicore systems; this cache is connected to the L2 caches. An example of multicore architecture with 2 cores and 3 level of caches is presented in Figure 3.1.

3.2 Structure

A *K-way set-associative* cache memory is a matrix of **cache lines**, the basic memory block unit of a cache memory. Cache lines are regrouped into **cache sets** of K elements, also

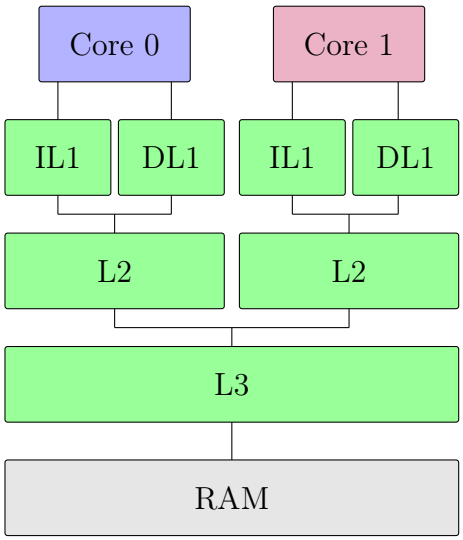


Figure 3.1: Example of multicore architecture

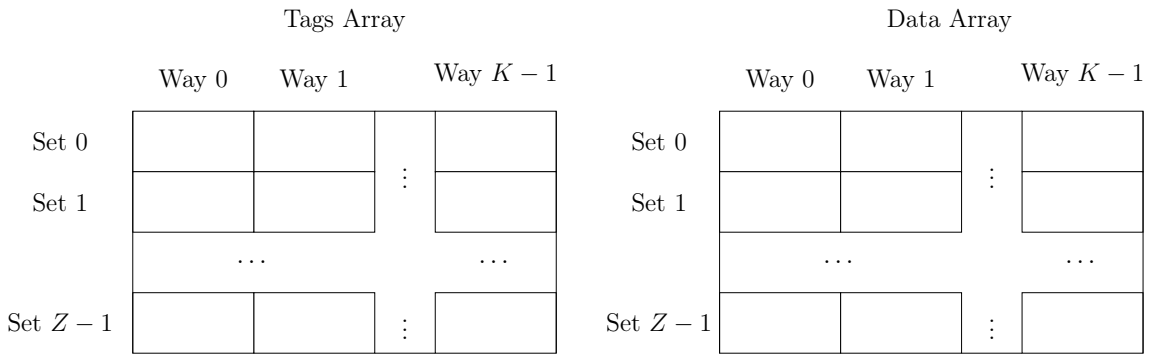


Figure 3.2: Example of a K-way set-associative with Z cache sets

known as **ways**. A cache memory with a single way per set is known as a *direct-mapped cache*, and a cache with a single set is known as a *full-associative cache*.

The memory is divided in **cache blocks** (CB), which are memory blocks of the size of a cache line. Each of these CBs is assigned an *index* computed as the modulo between the address of the CB and the number of cache sets, which corresponds to the cache set where the CB may be stored. Two CBs may have the same index: to differentiate them, each one is assigned a *tag* computed from their address. Figure 3.2 represents an exemple of a *K-way set-associative* with Z cache sets. It is composed of two matrixes, the data array that contains the CBs and the tags array that contains the tags of the CBs stored in the cache memory. The columns of these matrixes correspond to the *cache ways* and the lines, the *cache sets*.

As it exists two types of memories in a computer, physical and virtual, a cache memory is in one of the followings categories:

- Physically indexed, physically tagged cache memories (PIPT);
- Virtually indexed, virtually tagged cache memories (VIVT);
- Virtually indexed, physically tagged cache memories (VIPT).

The operating system uses virtual addressing for the memory of processes and, by extension, tasks. Each process has a dedicated address space. Consequently, two data coming from separate processes can have the same virtual address. To determine the location of data in physical memory, the Memory Management Unit (MMU) must convert virtual addresses to physical addresses.

Therefore, PIPT caches must convert the virtual address to a physical address in order to obtain the index and tag of a CB. Figure 3.3 represents this type of cache memory. The virtual address is translated with the MMU that contains the Translation Lookaside Buffer (TLB). The TLB contains the recent translations to speed up the MMU executiontime. VIVT caches use only the virtual address for the calculation of the index and the tag, removing the need to translate the physical address into the virtual address, it is shown with Figure 3.4. This accelerates the access to CBs in comparison to PIPT caches. However, these caches must be invalidated entirely whenever a process is changed.

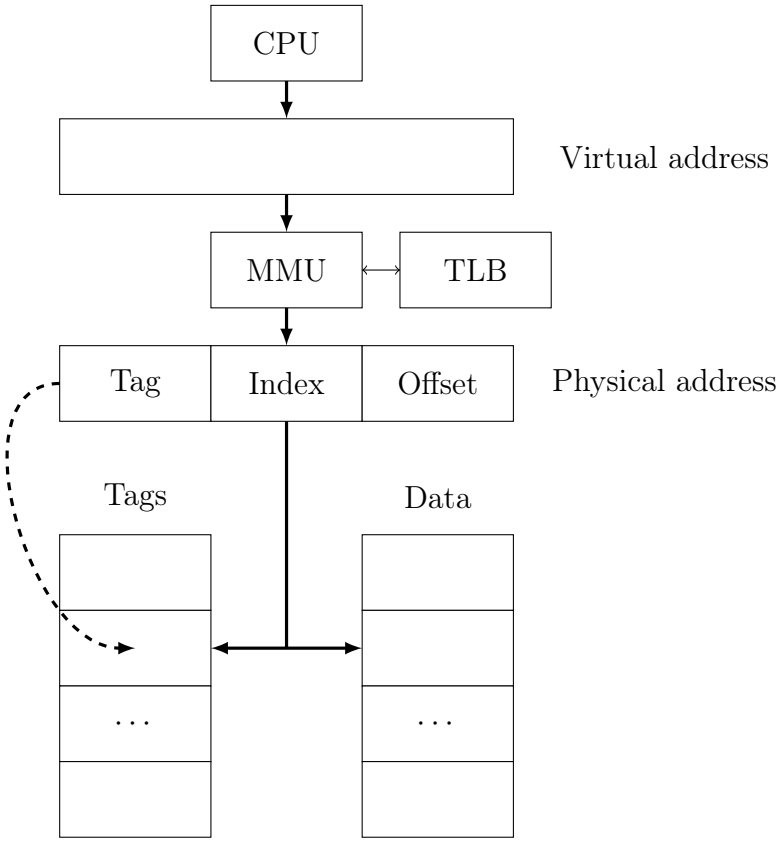


Figure 3.3: PIPT cache

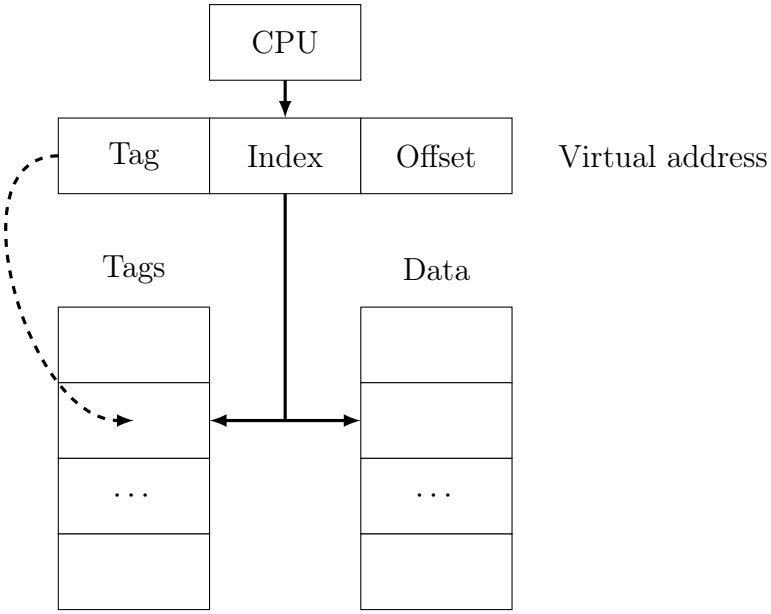


Figure 3.4: VIVT cache

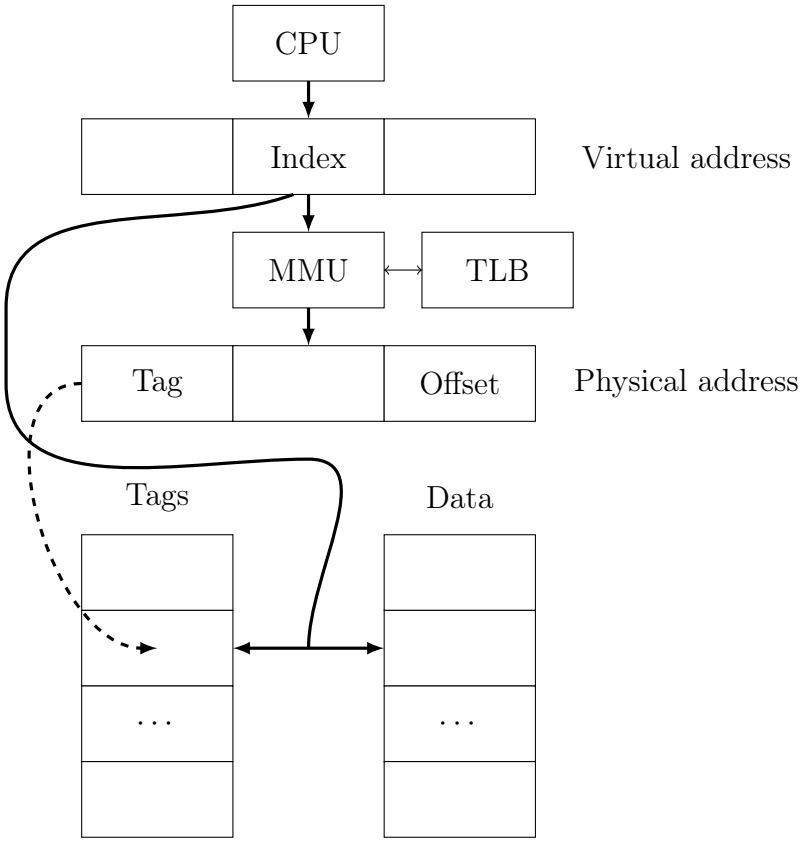


Figure 3.5: VIPT cache

VIPT cachees, describe with Figure 3.3 are a compromise between the two previously mentioned cache types. The translation of the virtual address into a physical address and the calculation of the tag are performed while the index is being calculated from the virtual address and the cache set is being accessed. This enables the same performance as the VIVT caches without invalidating the cache memory when a process is changed. However , a process can have two virtual addresses that point to the same data and, therefore, the same physical address [72]. Consequently, two instances of the same CB can coexist in the cache memory.

It is necessary to ensure that all instances of a CB are updated when writing. Because of this, data or hybrid caches (data and instructions) are typically of the PIPT type, whereas instruction caches that do not support writes are typically of the VIPT type.

When the processor needs to access an instruction, the index and the tag are computed from the virtual address. The tag is compared with all the CBs contained in the corresponding cache set: if a match is found (*hit*), then its *age* in the cache set is updated according to the replacement policy; if the tag is not found (*miss*), the CB is loaded from the main memory and stored in the cache set according to the replacement policy.

In this thesis we only consider a system composed of a single core with an instruction cache virtually indexed and physically tagged. The contribution of this thesis targets **direct mapped** and **set-associative** as they are the most frequently used caches. Also,

Fully-associative caches which are caches with only one index are not suited for large cache memory due to the large number of comparators [41].

3.2.1 Replacement policy

When a CB must be stored in a cache set, an algorithm chooses in which cache way the block has to be stored. Such algorithm is called a replacement policy algorithm. Several replacement policies exist [50]:

- *First In First Out* (FIFO). The oldest block present in the cache memory is the one which is replaced by another one if there are no free ways available.
- *Least Recently Used* (LRU). Each cache way in a cache set is assigned an *age*: at each access, ages are updated so that a CB with a more recent access than another has a lower age. When a CB has to be loaded in a cache set that is full, the least recently used CB is evicted and all the ages are updated accordingly. This replacement policy is the most predictable.
- *Pseudo Least Recently Used* (PLRU). LRU is difficult to implement as it needs to track the age of all the blocks in the memory cache. Thus, a family of replacement algorithms with a behavior close to LRU was designed: the PLRU family. This family of algorithms includes tree-PLRU and bit-PLRU.
- *Random* when a block must be stored, a random way is selected. It is the most unpredictable replacement policy.

In this thesis, we only consider the **LRU replacement policy**.

3.2.2 Write Policy

As a data can have multiple copies stored in multiple cache memory, when data is written all copies must be updated. However updates may be done at different times. They can be done directly after the write, we call this policy "write-trough"; or it can be done when the data is evicted, in this case it is called the "write-back" policy.

In this thesis only **instruction cache memories** are targeted, there is no write policy for this memories since instructions are read only.

3.3 Cache Analysis for WCET estimation

To take cache memory into account when estimating the WCET, an analysis classifying each memory access as a cache hit or a cache miss must be provided. There are two types of cache analysis: *must analysis* and *may analysis*. Instead of classifying each memory access, these analyses assign a category to each line block (LB), a block corresponding to the intersection of a CB and a BB, to determine whether accessing the latter will result in a hit or a miss. The *may analysis* categorizes LBs as either *always miss* or *unknown*.

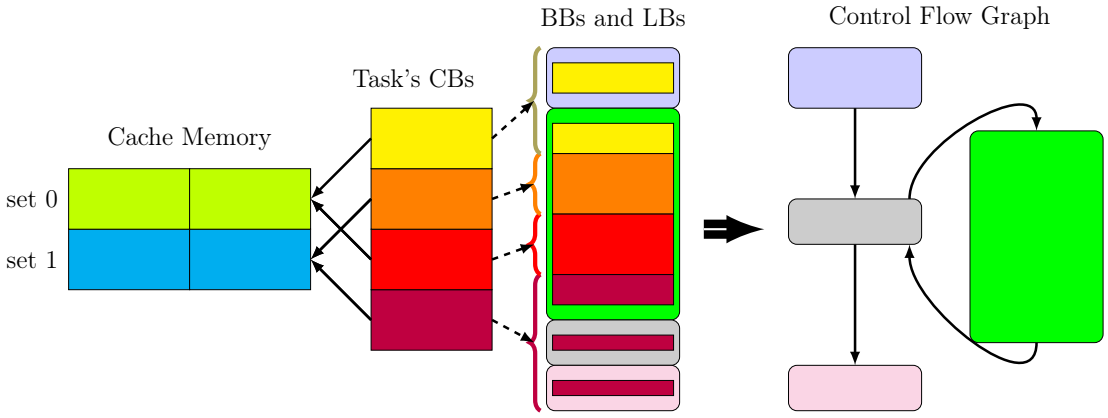


Figure 3.6: Relation between CBs and BBs

However, this analysis is overly optimistic if every access to a LB classified as unknown is considered a cache hit. Alternatively, if we consider unknown accesses as accesses that always cause cache misses, we would be considering a system without cache memory, which would be a very pessimistic scenario.

The *must analysis* categorizes each LB as *always hit* or *Unknown*. This analysis can be used safely for calculating the WCET if each access to a LB classified as unknown is considered as always miss.

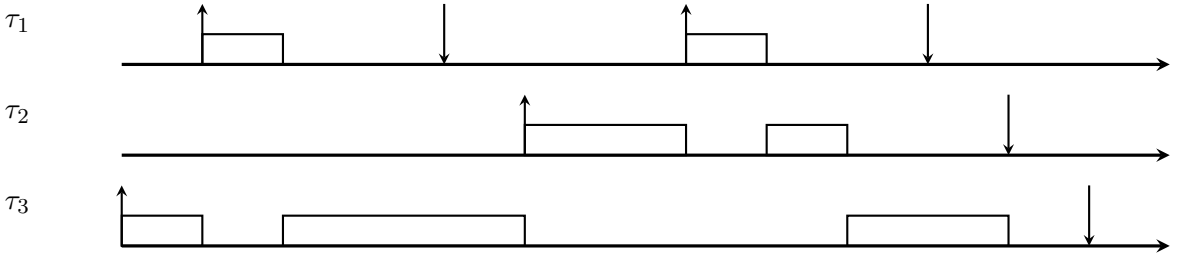
Healy et al. [42] propose an improvement in the cache analysis by providing more categories for classifying the LBs:

- Always Hit (AH);
- Always Miss (AM);
- Unknown (U);
- First HIT (FH);
- First Miss (FM).

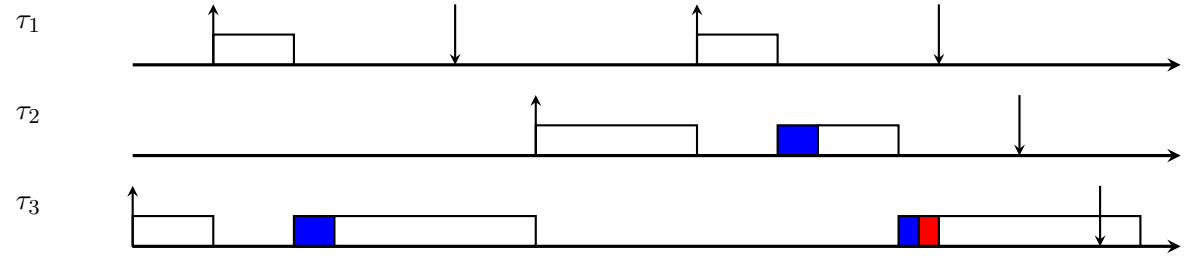
The categories FH and FM are used to classify the accesses that are hits or misses during the first execution of the line block, and the opposite for subsequent executions. This analysis is used to calculate the WCET for the tasks in this thesis.

3.3.1 Example

Figure 3.6 represents the relation between the CBs and the BBs of an example task. The cache memory is composed of two cache sets and it is depicted on the left. At its right, the binary code of the task is composed of 4 CBs, their arrows in direction of the cache memory point towards their respective cache set. Continuing to the right, we show the BBs and LBs as computed by the WCET analysis: for example, the yellow CB is composed of two LBs, one in the purple BB and the other in the green BB.



(a) Example of a scheduling without CRPD



(b) Example of a scheduling with CRPD

On the right, we depict the CFG built from the binary during the WCET analysis. At the beginning of the execution of the task the yellow CB will be stored in one of the green cache lines. As the yellow CB is not present in the cache before the execution of the purple BB, the yellow LB will be classified as *always miss*. During execution, the program will first move to the grey BB, and the dark red CB will be loaded into the blue cache set. However, if the task jumps into the green BB and returns to the grey one later, the dark red CB will still be present in the cache. Therefore, it is classified as *first miss*.

3.3.2 Cache Related Preemption Delay

The classification of cache accesses with WCET analysis cannot be guaranteed in a system that supports preemption. When a preemption occurs, the CBs of the preempted task may be evicted by the CBs of the preempting task. The subsequent access to these CBs will result in a cache miss. Thus, when a CB of a preempted task is evicted and its subsequent access is considered as a cache hit by the WCET analysis, an additional delay occurs to access the CB that was not accounted for by the WCET analysis. This additional delay is known as "Cache Related Preemption Delay" (CRPD), and corresponds to the reloading of the CB. By definition, there is no CRPD for non-preemptive scheduling algorithms; however, if the scheduling algorithm used permits preemptions, the CRPD must be taken into account during the scheduling analysis.

Figures 3.7a and 3.7b depict a scheduling example with three tasks τ_1, τ_2 and τ_3, with τ_1 having the highest preemption-level, τ_2 having an intermediate preemption-level, and τ_3 having the lowest preemption-level. In this example, we assume that each task's execution time is equal to its WCET. The white boxes indicate the execution time taking into account the WCET. The blue boxes indicate the CRPD caused by task τ_1 on the other tasks. The red box represents the CRPD resulting from the task τ_2.

If we disregard the CRPD, we can observe that τ_3 completes its execution on time in Figure 3.7a. In Figure 3.7b, however, as the CRPD is considered, we can see that task τ_3 completes its execution after the deadline.

Therefore, the CRPD must be considered during the scheduling analysis to ensure that all time constraints are met, or the scheduling must be adjusted to avoid this phenomenon.

3.4 Related Works on CRPD

Many methods have been proposed in the literature to address the problem of the inter-task interference due to cache memory. We can classify them into two different approaches.

3.4.1 Avoiding CRPD

The first method involves avoiding CRPDs between tasks. The various strategies in this approach are designed to compartmentalize tasks in the cache and restrict preemptions. Non-preemptive systems, on the other hand, are sensitive to long blocking times: a lengthy task with a low preemption-level may block an urgent task with a high preemption-level for the duration of its execution, resulting in unintended timeouts. Regarding task isolation in cache memory, this method necessitates a substantial amount of memory space in the cache and underutilizes it.

Luniss et al.[59] used simulated annealing to find a code layout in the memory that minimizes the CRPD. However, tasks are not isolated on cache memory, so inter-task and inter-core interference are still present. They used the linker to configure the code layout. Mancuso et al. [60] propose a complete framework which defines, isolates and locks most important memory areas in memory cache. These techniques are based on cache coloring partitioning and cache locking, its purpose is to reduce conflicts and enhance predictability but the cache is not optimally used because only the most important memories area are in the cache and to access other areas require costly RAM access.

Kim et al. [48] propose a practical OS-level cache management scheme using page coloring. They work on partitioned fixed priority preemptive scheduling system where they partition cache memory between cores with page coloring. In their works tasks may share the same cache area, thus intra-core interference is still present.

Ward et al. [84] consider colors as shared resources protected by critical sections, thus priority inversion may occur during execution. To reduce this problem they propose to slice tasks' periods, but their method may force the preempted task to reload its data (the set of data pages that a task may access in one job).

3.4.2 Improving CRPD estimation

A second approach is to reduce the pessimism of the CRPD analysis. Lee et al. [51] introduced the notion of *Useful Cache Block* (UCB) as a block that is used by the task at some point in the code, and that will be reused by the same task later in the code. They proposed a static analysis of the binary code of the task to compute a set of UCBs

for each preemption point. They also propose to compute the cost of n preemptions for a task as the size of the union of the n largest UCB lists. However, this technique suffers from a large pessimism because it analyzes each task in isolation. In particular, it does not consider the set of CBs of the preempting tasks.

Tan et al. [80] take into account both the preempting and the preempted tasks. In order to simplify the analysis and avoid combinatorial explosion, Tan et al. use a single set of UCB per task, obtained as the union of all UCB set for every preemption point.

In Lunniss et al. [57] and Altmeyer et al. [6], the authors propose two approaches called *UCB-union multiset* and *ECB-union multiset*. The first one consists in computing a global cost in an interval of length t by combining the UCBs of all instances of lower preemption-level tasks in the interval, and intersecting the result with the *Evicting Cache Blocks* (ECBs) of the higher preemption-level task which causes the preemption. The number of elements in the resulting set is an upper bound to the number of evicted CBs. In the *ECB-union multiset* approach, all ECBs of higher preemption-level tasks are merged together and intersected with the UCB of the preempted task to obtain a cost of preemption for the lower preemption-level task. Since none of two methods dominates the other, the authors propose a *combined approach* to improve the precision. Their framework has been initially proposed for Fixed priority scheduling systems. It has been extended later to Earliest Deadlines First scheduling systems [57]. We will extensively discuss such approaches in Section 3.5.

Shah et al. [74] reproduced CRPD analysis methods from Altmeyer et al. [6]. They show that the generation of synthetic task sets used in [6] is unrealistic, and propose a different way to generate task sets based on low-level analysis with LLVMTA. They also conclude that block reload time has a low impact on the schedulability. Something that we could not confirm in our experiments. We will discuss this discrepancy in Section 4.6.2.

Previous papers mostly consider direct-mapped caches. Concerning the cache model and the LRU replacement policy, Burguière et al. [26] present how to adapt existing CRPD analysis from direct mapped caches to set-associative caches. They show that only adaptation for LRU replacement policy is possible, since PLRU and FIFO cannot be bounded using the number of ways.

Altmeyer et al. [7] propose to use the ages of the UCBs and the number of reloading of ECBs to eliminate the UCBs that cannot be evicted from the analysis. They demonstrate their algorithm on the Papabench benchmark suite [66] where they compare against the UCB-union approach with a cache of 8 ways and 32 cache rows. However, the number of tasks which can be involved in the same preemption is smaller than the number of cache ways, therefore the results cannot be generalized to an arbitrary cache setting.

Marković et al. [61] recently proposed two novel methodologies for Fixed priority fully preemptive scheduling which improve the CRPD cost estimation of [6]. Nonetheless, a counterexample to these methodologies is provided in Chapter 4.1.

3.5 Reminder on CRPD Analysis

3.5.1 Useful CBs and Evicting CBs

To compute the Cache Related Preemption Delay, the notions of *Useful Cache Block* (UCB) and *Evicting Cache Block* (ECB) have been defined in the literature. We recall the definitions here and introduce the notation that will be useful in the rest of the thesis.

3.5.1.1 ECBs

The evicting CBs of a task consist of all the CBs of this task that can evict CBs of another task during a preemption. Most papers in the literature assume direct-mapped caches (only one cache way in the cache memory), therefore the ECBs of a task are modeled as a single set of cache set indexes. Indeed, in the case of a direct-mapped cache memory only one CB per set can be evicted.

In the case of a set-associative cache memory with the LRU replacement policy, a preempting task that accesses a given CB can evict all the CBs present in the corresponding set in a *chain reaction* [26].

Consider, as an illustration taken from [26], a full associative cache (single index) with four ways. This cache contains the following CBs of task τ_i, ordered from most frequently used to least frequently used, at a point p during the execution of task τ_i : $[1, 2, 3, 4]$. After this point p, τ_i accesses the CBs in this order : $4 \rightarrow 3 \rightarrow 2 \rightarrow 1$. If τ_i execution does not undergo any preemption, then these four accesses are cache hits. However, suppose that at this point p, task τ_j preempts τ_i. The set of ECBs of τ_j consists of a single element: $\{e\}$. Following this preemption, the cache will contain blocks $[e, 1, 2, 3]$. When τ_i attempts to access block 4, a cache miss will occur; the cache will contain the values $[4, e, 1, 2]$; consequently, block 3 will also be considered a cache miss. Likewise, block 2 and block 1 accesses will result in cache misses.

In summary, regarding no preemption:

$$[1, 2, 3, 4] \xrightarrow{4} [4, 1, 2, 3] \xrightarrow{3} [3, 4, 1, 2] \xrightarrow{2} [2, 3, 4, 1] \xrightarrow{1} [1, 2, 3, 4] \text{ 0 cache misses}$$

and, regarding a preemption:

$$[1, 2, 3, 4] \xrightarrow{e} [e, 1, 2, 3] \xrightarrow{4} [4, e, 1, 2] \xrightarrow{3} [3, 4, e, 1] \xrightarrow{2} [2, 3, 4, e] \xrightarrow{1} [1, 2, 3, 4] \text{ 4 cache misses}$$

Therefore, in this paper we represent the ECBs of a task as a *multiset*: for each ECB index, the multiset contains K instances of the index, where K is the number of ways in the cache set. From now on, we will use the symbol $\mathcal{E}_i$ for describing the multiset representing the *ECB*s of a task τ_i.

For example, consider a task τ_i that accesses the CBs with index $3, 6, 7, 10$ in a cache set with 2 ways. Then $\mathcal{E}_i = \{3, 3, 6, 6, 7, 7, 10, 10\}$.

For other replacement policies, such as FIFO, Burguière et al. [26] demonstrated that the maximum number of cache misses that the eviction of a block by another task would cause cannot be determined and cannot be limited by the maximum number of ways. They

utilized a full associative FIFO cache with two ways and the following non-preemption scenario corresponding to the execution of a task τ_i as an illustration.

$$[2,1] \xrightarrow{1} [2,1] \xrightarrow{3} [3,2] \xrightarrow{2} [3,2] \xrightarrow{4} [4,3] \xrightarrow{3} [4,3] \text{ 2 cache misses}$$

In this example, if another task preempt τ_i with a single ECB e just before the first access to block 1, the following scenario occurs:

$$[2,1] \xrightarrow{e} [e,2] \xrightarrow{1} [1,e] \xrightarrow{3} [3,1] \xrightarrow{2} [2,3] \xrightarrow{4} [4,2] \xrightarrow{3} [3,4] \text{ 5 cache misses}$$

3.5.1.2 UCBs

Given a preemption point p in the code of a task, the set of UCBs represents all CBs that are present in the cache *and* may be reused sometime later in the code. In fact, in case of preemption at point p, in the worst case all the evicted CBs that are UCBs will be reloaded later on, increasing the WCET of the task. Thus, only the CBs that can be classified as *first miss*, *always hit* and *first hit* will be considered as UCBs. In our model, we suppose that the LB that is preempted by another task is always evicted, thus after each preemption we have to consider the reload time of the CB of this LB. Therefore, each preemption point in the task is associated with a (possibly different) set of UCBs. In the case of a fully preemptive system, each BB can be considered as a preemption point [51]. To reduce the complexity of the analysis, recent approaches in the literature [6, 57, 5, 61, 58, 25, 26] use a single set of UCB cache set index for the entire task: the set is computed as the *fusion* of all UCBs for the different preemption points (see Section 3.5.2 for a definition of the fusion operation). Moreover, because we consider a set-associative cache memory in contrast to the current state of the art, which primarily considers direct-mapped caches, we use multisets instead of simple sets to represent the UCBs. This allows us to represent the fact that multiple UCBs can exist in the same cache set at the same preemption point, which would not be possible with a simple set.

In this thesis, we use the more complete model of one UCB multiset per preemption point. Therefore, in our model each task is characterized by a *set of multisets* (SOM) of UCBs. We mitigate the combinatorial explosion of the analysis by using an approximation function that trades off complexity against precision (described in Algorithm 1).

The multiset representing the *UCBs* of a task for a given preemption point p is denoted as $\mathcal{U}^p$ and the *SOM* representing the set of $\mathcal{U}$ for the task is denoted by $\overline{\mathcal{U}}$. For example, consider a task τ_i that has two preemption points, where for the first preemption points 4 UCBs are present in the cache with index $3, 6, 6, 7$ and 2 CBs with index $7, 7$ for the second preemption point. Then $\overline{\mathcal{U}}_i = \{\{3, 6, 6, 7\}, \{7, 7\}\}$. As several useful CBs may be stored at the same time in the same cache set, a $\mathcal{U}^p$ may have multiple instance of the same index.

3.5.2 Operations on multisets

In this section we formally define multisets and *SOM*, and we define their operations.

Definition 3.5.1. *A* multiset *is a set that may contain more than one instance of the same element. Given an element $a \in \mathsf{A}$, we denote as $\mathsf{rep}_{\mathsf{A}}(a)$ the number of instances of a in A: for all $x \notin \mathsf{A}$, $\mathsf{rep}_{\mathsf{A}}(x) = 0$.*

We define the following operations on multisets:

- The size of a multiset, denoted by $|\mathsf{A}|$ is the number of elements in the multiset, including repetitions.
- The *multiset union* between multisets A and B is a multiset, it is denoted by $\mathsf{A} \uplus \mathsf{B}$, and it is defined as:

$$\forall x \quad \mathsf{rep}_{\mathsf{A} \uplus \mathsf{B}}(x) = \mathsf{rep}_{\mathsf{A}}(x) + \mathsf{rep}_{\mathsf{B}}(x)$$

- The *multiset fusion* between multisets A and B is a multiset, it is denoted by $\mathsf{A} \bigtriangledown \mathsf{B}$, and it is defined as:

$$\forall x \quad \mathsf{rep}_{\mathsf{A} \bigtriangledown \mathsf{B}}(x) = \max(\mathsf{rep}_{\mathsf{A}}(x), \mathsf{rep}_{\mathsf{B}}(x))$$

- The *multiset intersection* between multisets A and B is a multiset, it is denoted by $\mathsf{A} \sqcap \mathsf{B}$, and it is defined as:

$$\forall x \quad \mathsf{rep}_{A \sqcap B}(x) = \min(\mathsf{rep}_{\mathsf{A}}(x), \mathsf{rep}_{\mathsf{B}}(x))$$

Definition 3.5.2. *$\overline{\mathsf{A}}$ denotes a* SOM*, that is a set whose elements are multisets. The largest size of any multiset in a* SOM *$\overline{\mathsf{A}}$ is denoted as:*

$$\mathsf{MS}(\overline{\mathsf{A}}) = \max_{\mathsf{A} \in \overline{\mathsf{A}}}(|\mathsf{A}|)$$

Since $\overline{\mathsf{A}}$ is a set, the usual operations of set union, set intersection and set difference apply, respectively, with the usual symbols.

We extend the operations of *multiset union* and *multiset intersection* to *SOM* in the most natural way: the multiset union (resp. intersection) between $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$ is the *SOM* obtained by applying the multiset union (resp. intersection) to every pair of elements of $\overline{\mathsf{A}}$ and $\overline{\mathsf{B}}$. We define the intersection between a multiset $\mathcal{E}$ and *SOM* $\overline{\mathsf{A}}$ as the *SOM* obtained by applying the multiset intersection of every element of $\overline{\mathsf{A}}$ and the multiset $\mathcal{E}$.

3.5.3 EDF analysis with CRPD

In order to include the CRPD in the analysis, Lunniss et al. [57] redefined Condition (2.12) as:

$$\forall t \leq \mathrm{hp}(\mathcal{T}), \quad \mathsf{dbf}(t) = \sum_{i}^{N} \mathsf{dbf}_i(t) + \gamma(t) \leq t \tag{3.1}$$

where $\text{hp}(\mathcal{T})$ is the hyperperiod of task set $\mathcal{T}$ and $\gamma(t)$ is an upper bound to the total CRPD in interval $[0, t]$.

To compute $\gamma(t)$, we use the *Combined* approach of Lunniss et al. [57]. This approach is the combination of two methods, the *UCB-union multiset* and the *ECB-union multiset*.

As our task model differs from the state of the art (we use a *SOM* to represent the UCBs of the task instead of a simple multiset), we denote by *ECB-union* SOM the modified version of the second approach based on our task model, and *Combined* SOM will be the combination of *UCB-union multiset* and *ECB-union* SOM. In order to present the following equations in short form, we use $(\mathcal{U}_k)^x$ as a short form for the multiset union of $\mathcal{U}_\mathsf{k}$ with itself x times.

Given a *SOM* of UCBs $\overline{\mathcal{U}}_k$ for all the possible preemption points of task τ_k, a single multiset UCB can be computed as:

$$\mathcal{U}_k = \bigtriangledown_{\forall \mathcal{U}_k^P \in \overline{\mathcal{U}}_k} \mathcal{U}_k^P$$

3.5.4 UCB-union multiset

First, the maximum number of preemptions by task τ_j in interval $[0, D_i]$ is computed as:

$$\mathbf{Pr}_j(D_i) = max\left(0, \left\lceil \frac{D_i - D_j}{T_j} \right\rceil\right) \tag{3.2}$$

Then, the method constructs the multisets of UCBs and ECBs for the interval $[0, t]$:

$$M_{t,j}^{\mathbf{ucb}} = \biguplus_{\forall k, t \geq D_k > D_j} (\mathcal{U}_k)^{\mathbf{Pr}_j(D_k) \cdot \eta(k,t)} \tag{3.3}$$

$$M_{t,j}^{\mathbf{ecb}} = (\mathcal{E}_j)^{\eta(j,t)} \tag{3.4}$$

where $(\mathcal{E}_j)^x$ is a short form for the multiset union of $\mathcal{E}_\mathsf{j}$ with itself x times. Finally, the overall preemption cost is computed as the intersection of the ECB and UCB multisets obtained above:

$$\gamma_j^{\mathbf{ucbm}}(t) = \mathsf{BRT} \cdot (|M_{t,j}^{\mathbf{ucb}} \sqcap M_{t,j}^{\mathbf{ecb}}| + Y(j,t)) \tag{3.5}$$

$$Y(j,t) = min\left(\sum_{\forall k, t \geq D_k > D_j} \Big(\mathbf{Pr}_j(D_k) \cdot \eta(k,t)\Big), \eta(j,t)\right) \tag{3.6}$$

with BRT the block reload time of a CB from the main memory. As we consider the CB of the preempted LB evicted for each preemption, we add $Y(j,t)$ to the cost to consider this assumption.

3.5.5 ECB-union *SOM*

This approach is the version of ECB-union multiset modified by using a *SOM* to represent the UCBs of a task for the different preemption points. To obtain the same result as ECB-union multiset, $\overline{\mathcal{U}}'_k$ can be used instead of $\overline{\mathcal{U}}_k$ with $\overline{\mathcal{U}}'_k = \{\mathcal{U}_k\}$.

The first step consists in taking the *SOM* of UCBs for the preempted task, and compute the size of the worst-case intersection with higher preemption-level tasks' ECBs:

$$Q_j^{\mathbf{max}}(\overline{\mathcal{U}}_k) = \max_{\forall \mathcal{U} \in \overline{\mathcal{U}}_k} \left(\left| \mathcal{U} \sqcap \mathcal{E}'_j \right| \right) + 1 \tag{3.7}$$

With $\mathcal{E}'_j$ defined as:

$$\mathcal{E}'_j = \left(\biguplus_{\forall h, D_h < D_j} \mathcal{E}_h \right) \uplus \mathcal{E}_j$$

Notice that we add 1 to the cost to consider the CB of the preempted LB as evicted. Then, the algorithm computes a multiset as the union of all costs of the multisets computed for the preemptions that occur in interval $[0, t]$:

$$Q_{t,j} = \biguplus_{\forall k, t \geq D_k > D_j} \left((\{Q_j^{\mathbf{max}}(\overline{\mathcal{U}}_k)\})^{\mathbf{Pr}_j(D_k) \cdot \eta(k,t)} \right) \tag{3.8}$$

Finally:

$$\gamma_j^{\mathbf{ecb}\boldsymbol{SOM}}(t) = \mathsf{BRT} \cdot \mathsf{sum\_max\_elems}(Q_{t,j}, \eta(j,t)) \tag{3.9}$$

where function $\mathsf{sum\_max\_elems}(\mathsf{A}, n)$ computes the sum of the n greatest elements in multiset A.

Notice that we slightly changed the algorithm of [57], in particular we modified Equation (3.7): instead of using a multiset which represents the UCBs of all the tasks, we changed the Equation to use only the multiset of UCBs at the point p of the task which is involved in the worst CRPD cost. This modification actually improves the performance of *ECB-union multiset* as it reduces the pessimism of considering additional CBs.

The combined *SOM* approach is the minimum between the demand bound of the task set computed with the previous approaches.

$$dbf^{combinedSOM}(t) = min(dbf^{\mathbf{ucbm}}(t), dbf^{\mathbf{ecb}\boldsymbol{SOM}}(t)). \tag{3.10}$$

3.6 Weak robustness in computing systems

A well-known source of problems is the presence of faults on L1 cache memories. There are two types of faults, permanent faults and transient faults [29, 4, 70].

Permanent faults, when they appear, remain for the whole life of the system. They are due to short or open circuit provoked by error in the manufacturin process, the aging of the system and process variations [29, 70, 13, 67].

Transient faults are logical faults in the operation of a circuit that occur at random and are primarily caused by charged particle emissions [29, 47]. These faults can provoke

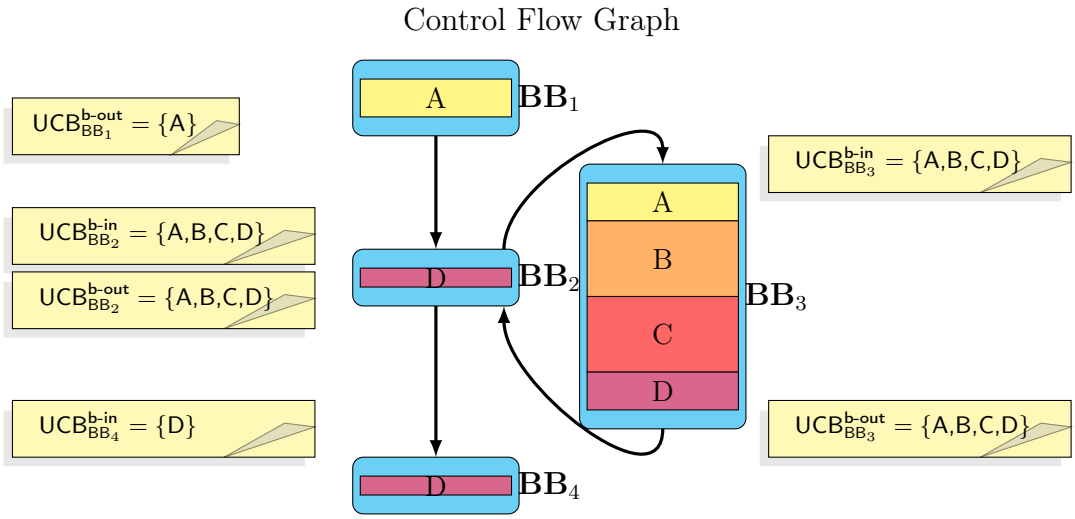


Figure 3.8: Example of task CFG

bit-flipping in the cache memory. With the progress of the technology, which tends to integrate more transistors at a smaller scale, the presence of bit-flipping errors in cache memories becomes a challenge [82].

Transient faults are temporary unpredictable faults due to environment factors, such as temperature and radiations. They corrupts data in memory units, in particular SRAM-based cache memories. Transient faults are different from permanent faults because the hardware is not damaged, and the electronics components affected by these faults will work correctly after the errors have been corrected.

Definition 3.6.1 (Vulnerability interval)**.** *Given a CB x used by task τ_i, a* vulnerability interval *of x is an interval of time between the moment x is loaded into the cache and a subsequent read operation on x, without any reload or write operation within the interval. During one of its vulnerability interval, x might suffer from transient fault in the cache memory.*

Definition 3.6.2 (Vulnerable path)**.** *Given a CB x used by task τ_i, a* vulnerable path *of x is an execution path between two BBs in the task's CFG corresponding to a* vulnerability interval *of x. During the execution of this path, x might suffer from a transient fault in the cache memory.*

3.6.1 Example of vulnerability

Figure 3.8 represents the CFG of a task and the correspondence between BBs and CBs: the nodes of the graph (blue rectangles) represent the BBs; the colored rectangles inside the nodes represent the LBs; rectangles with the same color represent LBs belonging to the same CB. For example, the two yellow rectangles are two LBs belonging to the same CB: A, and to two different BBs: BB_1 and BB_3 (nodes of the tasks).

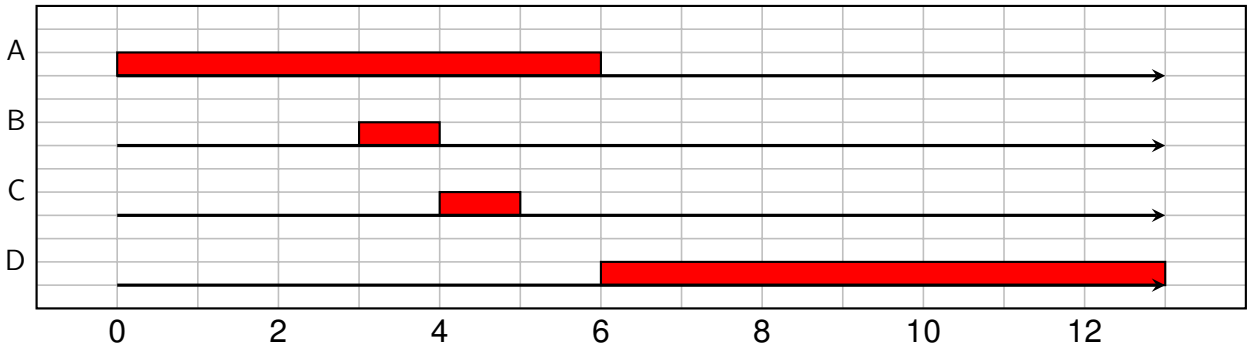


Figure 3.9: Example vulnerability intervals

In the example, we consider a set associative cache with $K = 2$ which contains only 2 cache sets. We assume that the addresses of the CBs are such that A and C fit into cache set 1, and B and D into cache set 2.

The graph contains a loop between BB_2 and BB_3, therefore their execution is repeated a certain number of times.

After the execution of BB_1, A is considered as UCB because there exists a path without eviction between BB_1 and BB_3. Thus, we can state that A is vulnerable on path $BB_1 \rightarrow BB_2 \rightarrow BB_3$. Following the same reasoning, A, B, C and D are both vulnerable along path $BB_3 \rightarrow BB_2 \rightarrow BB_3$. On path $BB_2 \rightarrow BB_4$, neither A nor B and C need to be considered as vulnerable, because they are not accessed after BB_4.

3.6.2 Estimation of the cache memory vulnerability

Temporal Vulnerability Factor(TVF) is a metric proposed by Wang et al. [83] to measure the vulnerability of a cache. This metric measures a vulnerability factor between 0 (not vulnerable) and 1 (completely vulnerable) for a given execution trace of time t for the cache. During this execution trace, the system will use k data items. Depending on the accuracy of the analysis, these data items may be bits, LBs or CBs, for example. Each data item is distinct from the others, and they do not share any data. The cache's TVF can be calculated as follows:

$$TVF_{cache} = \frac{\sum_i^k (data\_item_i \cdot \sum_j vul\_inter_{i,j})}{\sum_i^k (data\_item_i \cdot t)} \tag{3.11}$$

where $vul\_inter_{i,j}$ corresponds to the size of the j^{th} vulnerable interval of the i^{th} data item. $data\_item_i$ is the size in bytes of the i^{th} data item.

Figure 3.9 shows the vulnerability intervals of CBs A, B, C and D in red. In this example, we consider an execution trace of size $t = 13$ and CB's size of 64 bytes. The TVF for this example can be computed as follows:

$$TVF_{cache} = \frac{64 \cdot 6 + 64 \cdot 1 + 64 \cdot 1 + 64 \cdot 7}{4 \cdot 64 \cdot 13}$$

$$TVF_{cache} = \frac{960}{3328}$$

$$TVF_{cache} = 0.288$$

3.6.3 Protection mechanisms

To reduce the impact of these transient faults, hardware protection mechanisms are usually proposed. We list here three types of protection mechanisms: the first and most naive is to disable cache units; the second type is the use of Error Detection mechanisms, such as a parity code or double-modular redundancy. These mechanisms detect the corrupted instructions or data and trigger a reload of the memory block from a higher memory layer. The last type consists of Error Correcting Codes (ECC) or Triple-Modular Redundancy (TMR) which can also automatically correct the data.

While protecting the cache, these mechanisms decrease the system performance in terms of executed instructions per second. This loss of performance comes from the reduction of the useful cache memory size, the error correction logic [62] or by delays added by the protection mechanism to detect and correct faults. Hijaz et al. [44] proposed a study to quantify the impact of a constant delay on cache hit access. They show in their experiments that increasing the L1 cache hit latency from 1 to 2 cycles incurs a performance loss of 10%, and when the latency climbs to 3 cycles, the loss of performance is more than 30%.

We assume in this thesis that higher layers of memory (L2, L3 or DRAM memory) are protected by fault-tolerance mechanisms. In fact, the impact of error correction mechanisms for DRAM in terms of performance loss is less important, and it is usually estimated around 1-2%. Also, the cost of an ECC DRAM memory is proportionally lower than the cost of the L1 cache memory [45].

3.7 Related Works on Cache Memory Vulnerability

Faults in computing systems can be classified in two categories: *permanent faults* when it affects the hardware in an irreversible way, and *transient faults* when it does not affect the hardware in an irreversible way.

In Siddiqua et al.[75] the authors realized several experiments to collect data concerning memory robustness and faults. They gathered data memory reliability from the Cielo supercomputer at Los Alamos National Laboratory over a five-year period. The investigation is centered on DRAM (main memory) and SRAM (cache memory and registers). Each fault is classified as either permanent or transient. Their experimental results show that 99.36% of the faults in L3 cache (SRAM) are transient faults, and 99.98% of the errors are single bit errors. The fault does not alter the entire cache index and cache way, as they mostly affect only one cache line at a time, while the other types of faults are permanent faults and affect cache ways entirely.

3.7.1 Permanent Faults

When a permanent fault occurs, Agnola et al. [2] propose a method for reorganizing memory locations called *Self Adaptive Cache Memories*. Their method consists in replacing a faulty cache line location by a healthy cache line location from another set to avoid an entirely faulty cache set. They limit the number of faulty cache line per set to one. When all the cache sets contain a faulty cache line, the mechanism reduces the associativity, thus not considering the faulty cells anymore. Wilkerson et al. [86] present a scheme to improve the reliability of cache memory when a low-voltage is used. Cache memories consume an important quantity of energy. To reduce this consumption, it has been proposed to use a near-threshold voltage for caches at the cost of higher fault probability. The authors provide a precise characterization of errors that may occur during systems execution at low-voltage. They derive the relation between the voltage and the probability of errors in the cache memories. They assume that a probability of failure lower than 0.001 respects the manufacturing yields and suggest minimal voltage value according this assumption.

Yan et al. [87] consider voltage scaling for delay sensitive L1 cache memory. Reducing voltage of a chip is an efficient way to reduce power consumption, but has some drawbacks, such as the reduction in reliability. They propose specific methods for data and instruction caches. The method proposed for instruction caches is a remapping of instructions from defective blocks to free-fault blocks in cache memory. They use Built-in Self Test (BIST) to identify defective words and remap them by using static compilation, just-in-time compilation or binary translation. They achieve an energy reduction of 64% per instruction.

The most important differences between these works and ours is the type of considered faults. Indeed, this thesis focuses only on transient faults. Furthermore, they do not consider real-time guarantees.

3.7.2 Transient Faults

Sugihara et al. [79] propose a MIP formulation that produces a scheduling of non-preemptive tasks on multicore systems allowing to minimize the vulnerability of the systems while respecting real-time constraints. They use a reliable cache architectures instead of our cache misses insertion. Reliable cache architectures reduce vulnerability by deactivating cache ways or by merging them. While our method is similar to their approach, we deactivate cache lines at a finer granularity and at determined points in the program. Also, their methods do not work with direct mapped caches and necessitates specific hardware modifications.

Wang et al. [83] develop a *lifetime* model for L1 caches. They propose to classify intervals of time during which cache elements (cache lines, words, ...) are used as *vulnerable* or *non-vulnerable*. If a fault (a bit-flip) happens between two read instructions, the fault becomes an error. The variable accessed by the instructions is considered as *vulnerable* and the interval between the two reads is declared as a *vulnerable interval*. On the other hand, if the fault happens between a read and a write instructions, the faulty data is overwritten and does not cause any errors in the program execution. For instruction

Ref	PF	TF	ECCs	RT	HM
[86]	✓				
[87]	✓				
[79]		✓	✓	✓	
[83]		✓			
[82]		✓			✓
[77]		✓		✓	
[21]		✓		✓	
Our proposition		✓	✓/ ×	✓	✓/ ×

Table 3.1: State of the art methods against our contributions on improving reliability

caches, the vulnerability interval corresponds to the time the instruction remains in the cache before being evicted by the cache replacement algorithm. In this respect, Wang et al. [83] propose the TVF (Temporal Vulnerability Factor), a reliability score for the cache: a higher score indicates a less reliable cache.

Driven by the TVF metric, they propose a Clean Cache line Invalidation (CCI) technique to refresh the data in the cache after a given amount of time without any activity. For the instruction cache, they propose to insert cache misses (called *cache scrubbing*) instead of CCI to refresh the instructions of the L1 cache from the L2 cache. To reduce the overhead, they propose to insert cache misses during the idle cycles of cache memory. In this thesis, we propose to insert cache misses at strategic points during the task's execution while considering timing constraints.

Later, the same authors focused on instruction cache memories for embedded systems [82]. A new metric has been proposed, the System-level Instruction Cache Vulnerability Factor (SICVF). This metric aims to have a more accurate reliability measurement by considering some specific properties of the 32 bit ARM ISA. Then, they suggest ways to improve the ARM ISA in order to reduce the SICVF. In comparison, our contribution neither requires new instructions in the ISA nor code re-compilation.

Song et al. [77] propose a predictable system-level fault tolerance system implementation on the COMPOSITE component-based OS. Their method does not require hardware redundancy. It performs recovery of components by tracking their state during their execution. The authors provide also timing analysis of their system. Their method does not focus specifically on cache memories but on the reliability of OS components. However, they consider timing constraints.

Hardware redundancy based methods such as DMR, TMR or CRC, are expensive solutions for embedded systems [21]. However, Bhat et al. observe a trend around software based approaches to increase reliability. In their framework, the authors propose a novel task allocation heuristic to respect fault-tolerance requirements and minimize the number of cores in multi-core architecture. They also provide a schedulability analysis and testing of their AUTOSAR-based framework. While their method takes into account timing constraints, our work differs from theirs in that we do not require any redundancy. In addition, we use cache miss insertion to improve reliability.

A summary of the comparison between the state-of-the-art methods presented in this chapter and our method is given in Table 3.1. In this table, PF and TF represents methods that target respectively, permanents faults and transient faults. ECCs symbolizes the methods using ECCS, RT, the methods considering real-time constraints and finally, HM, the methods requiring hardware modifications. The mark ✓/ × for ECCs and HM means that the methods may use ECCs or need harware modifications but it is not mandatory.

Part II

Contributions

Chapter 4

Improving CRPD analysis under EDF scheduling

Contents

The scheduling analysis of a real-time system requires an estimation of the *Worst-Case Execution Times* (WCETs) of all the tasks, and an analysis of their interactions. The WCET is usually calculated by analyzing the task as if it executed *in isolation* on the target platform. However, even when tasks are considered to be functionally independent, they can indirectly interfere with each other due to shared hardware resources, like cache memory, bus, DMA, etc. Concerning the interference due to shared caches, a higher preemption-level task that preempts a lower preemption-level task can evict some of the CBs of the latter; when the lower preemption-level task resumes execution, it may need to reload the CBs, thus increasing its execution time.

Several solutions to this problem have been proposed in the literature, from non-preemptive or limited-preemptive scheduling strategies, to fully preemptive systems with *Cache Related Preemption Delay* (CRPD) analysis. In this chapter we focus on improving the CRPD analysis of single-processor fully preemptive real-time systems with one level of cache. In addition, we provide a counterexample to the *Partitioning-ver1* and *Partitioning-ver2* analyses [61].

CRPD analysis works as follows: first, a WCET is computed for each task, assuming the task is executed alone on the processor. The computed WCET includes the effect of cache misses caused by the task itself (*intra-task* cache misses). A static analysis also produces one or more lists of CBs used by the task.

The CRPD analysis computes an upper bound to the number of cache misses caused by preemptions to consider in the scheduling analysis.

One limitation of existing CRPD analyses with the Earliest Deadline First scheduling policy is the computation of the number of preemptions on a given task. Since most of the existing analyses were originally designed for Fixed Priority, they consider a *preemption interval* (that is the interval of time where a task may be preempted) of the size of the task's worst-case response time. However, computing the task's response time for EDF is computationally expensive (and very complex). For this reason, existing CRPD analysis consider a preemption interval size that is equal to the relative deadline of the task, thus introducing extra pessimism.

Another source of pessimism is the computation of the set of UCBs that may be evicted by a preemption. Existing analyses use a single set for every possible preemption point, thus simplifying the analysis by introducing extra pessimism. We propose to consider a number $M > 1$ of UCB sets per task, thus increasing precision without exploding the complexity.

Summarizing, in this chapter we present a counter example of CRPD analyses from the literature and two improvements over the state of the art CRPD analysis:

- a simple algorithm for computing a more precise *preemption interval* of a task scheduled by EDF on a single processor; by reducing the preemption interval, we tighten the upper bound on the number of preemptions a task can suffer;

- A simple algorithm to reduce the number of UCB sets per tasks to a given constant $M > 1$.

This chapter is organized as follows. First we present a counter example for two analyses from the state of the art in Section 4.1. Then, in Section 4.2, we introduce the model used in in the rest of this chapter. In Sections 4.3 and 4.4, we present our original contributions. We discuss the complexity of the preemption interval length computation in Section 4.5. We evaluate our methods against the state of the art [57, 6], and we present the results of the evaluation in Section 4.6.

4.1 Partitioning-ver1 and ver2

4.1.1 Reminder of the approaches

Marković et al. [61] proposed two methods for computing the CRPD, called *Partitioning-ver1* and *Partitioning-ver2*. The algorithms assume Fixed Priority scheduling with preemptive tasks. In order to calculate the response time of a task using their methods, they proposed the following equation:

$$R_i^{l+1} = C_i + \gamma(i, R_i^l) + \sum_{\forall h \in hp(i)} \left\lceil \frac{R_i^l}{T_h} \right\rceil \cdot C_h \tag{4.1}$$

with $\gamma(i,t)$, the CRPD between the first i tasks by considering a set of tasks ordered from highest to lowest priority, during the interval of time $[0,t]$.

Their method for computing $\gamma(i,t)$ consists of four steps.

1. Count the number of preemptions of τ_h on τ_j during interval $[0,t]$. To do so, they propose the following equation:

$$E_j^h(t) = \begin{cases} \left\lceil \frac{t}{T_h} \right\rceil, & \text{if } \left\lceil \frac{t}{T_h} \right\rceil \leq \left\lceil \frac{t}{T_j} \right\rceil \\ \left\lceil \frac{t}{T_j} \right\rceil \cdot \left\lceil \frac{R_j}{T_h} \right\rceil, & \text{otherwise} \end{cases} \tag{4.2}$$

   It is the maximum number of times a job of task τ_h starts in the interval $[0,t]$, if this number is less than the maximum number of jobs of task τ_j starting in this interval. Otherwise, it is the maximum number of jobs of task τ_j beginning within the interval $[0,t]$ multiplied by the maximum number of times a job of task τ_j can be preempted by τ_h.

2. Build the *preemption partitions* during interval $[0,t]$. It consists in computing a multiset $\Lambda_{i,t}$ of *preemption partitions* between the first i tasks, with each partition not considering twice a preemption between two tasks.

$$\Lambda_{i,t} = \{\lambda_1, \lambda_2, \cdots, \lambda_Z\} | \lambda_r = \{(\tau_h, \tau_j) | r \leq E_j^h(t) | \forall h, j \leq i\} \tag{4.3}$$

   The rationale is that, by partitioning the preemptions into disjoint sets, they can avoid a complete enumeration of all possible preemptions while still obtaining an upper bound.

3. Compute CRPD cost for each preemption partition.

   At this stage, they propose two divergent approches. In fact, Partitioning-ver1 proposes a method based on the combined approach by Altmeyer et al. [6], whereas Partitioning-ver2 proposes a completely new method.

   Partitioning-ver1. To evaluate $\gamma_i(\lambda_r)$, the CRPD of a partition λ_r, Marković et al. modified the equations used by the combined approach as follows:

$$\gamma_i(\lambda_r) = min(\gamma_i^{ecbp}(\lambda_r), \gamma_i^{ucbp}(\lambda_r)) \cdot \mathsf{BRT} \tag{4.4}$$

$$\gamma_i^{ecbp}(\lambda_r) = \sum_{h=1}^{i-1} \gamma_{i,h}^{ecbp}(\lambda_r) \tag{4.5}$$

$$\gamma_{i,h}^{ecbp}(\lambda_r) = \max_{\forall \tau_k \in aff(i,h,\lambda_r)} \left\{ min \left(\left| \left(\bigcup_{\tau_{h'} \in (hp(h,\lambda_r) \cup \{\tau_h\})} \mathcal{E}_{h'} \right) \cap \mathcal{U}_k \right|, \mathsf{MS}(\overline{\mathcal{U}}_k) \right) \right\} \tag{4.6}$$

$$\gamma_i^{ucbp}(\lambda_r) = \sum_{h=1}^{i-1} \gamma_{i,h}^{ucbp}(\lambda_r) \tag{4.7}$$

$$\gamma_{i,h}^{ucbp}(\lambda_r) = min \left(\left| \left(\bigcup_{\tau_k \in aff(i,h,\lambda_r)} \mathcal{U}_k \right) \cap \mathcal{E}_h \right|, \sum_{\forall \tau_k \in aff(i,h,\lambda_r)} \mathsf{MS}(\overline{\mathcal{U}}_k) \right) \tag{4.8}$$

   with $aff(i, h, \lambda_r)$, the set of tasks with priorities higher than or equal to P_i which can be preempted by τ_h according to λ_r; and $hp(h, \lambda_r)$ denotes the set of tasks with higher priorities than task τ_h that, according to λ_r, can preempt τ_h.

   Partitioning-ver2. As an alternative to the combined approach, Marković et al. propose an equation to compute the preemption cost for a preempted task τ_i by a list of preempting tasks PT such as:

$$\gamma(\tau_i, PT) = |\mathcal{U}_i \cap (\bigcup_{\forall \tau_h \in PT} \mathcal{E}_h)| \cdot \mathsf{BRT} \tag{4.9}$$

   In partitioning-ver2 the CRPD cost of a partition $\gamma_i(\lambda_r)$ consist in the CRPD cost of the worst preemptions scenario that can be built with λ_r. To build the list of the preemption scenarios possible for a partition λ_r, Marković et al. propose an Algorithm in [6]. The preemption costs of the preemptions scenarios are computed with Equation (4.9). An example is given in Section 4.1.2.3.

4. Compute the overall CRPD cost.

   This final step is simply the total CRPD costs for each partition.

$$\gamma(i,t) = \sum_{\forall \lambda_r \in \Lambda_{i,t}} \gamma_i(\lambda_r) \tag{4.10}$$

   with $\gamma_i(\lambda_r)$ the CRPD cost of preemption partition λ_r.

Unfortunately, both methods are incorrect, in the sense that they cannot correctly upper bound the CRPD, as shown by the following counterexample.

4.1.2 Counter Example

Consider 4 sporadic tasks τ_1, τ_2, τ_3 and τ_4, with $P_1 > P_2 > P_3 > P_4$, whose parameters are described in Table 4.1, while their UCBs and ECBs sets are described in Table 4.3.

Name	WCET	Deadline	Period
τ_1	20	300	440
τ_2	50	700	1000
τ_3	100	800	1000
τ_4	300	900	1000

Table 4.1: Counter Example

Consider the preemption matrix $A_{4,1000}$, reported in Table 4.2, which has been computed using Equation (4.2). This matrix represents the possible preemptions during any interval of time of size 1000. Thus, this matrix should also represent the scenario from Figure 4.1 where the execution of tasks considered by the WCET analysis tool is represented by the white boxes, while the CRPD is represented by the red boxes. The CRPD cost for this scenario can be computed as: $\{(\tau_4, \{\tau_2, \tau_1\}), (\tau_2, \{\tau_1\}), (\tau_4, \{\tau_3, \tau_1\}), (\tau_3, \{\tau_1\})\}$ with,

$$(\tau_4, \{\tau_2, \tau_1\}) = |(\mathcal{E}_1 \cup \mathcal{E}_2) \cap \mathcal{U}_4| = 11$$
$$(\tau_2, \{\tau_1\}) = |\mathcal{E}_1 \cap \mathcal{U}_2| = 8$$
$$(\tau_4, \{\tau_3, \tau_1\}) = |(\mathcal{E}_1 \cup \mathcal{E}_3) \cap \mathcal{U}_4| = 11$$
$$(\tau_3, \{\tau_1\}) = |\mathcal{E}_1 \cap \mathcal{U}_3| = 8$$

Which gives a CRPD cost of: $38 \times \mathsf{BRT}$.

4.1.2.1 Partitioning

For both methods, *Partitioning-ver1* and *Partitioning-ver2*, two partitions are built: $\lambda_1 = \{(\tau_1, \tau_2), (\tau_1, \tau_3), (\tau_1, \tau_4), (\tau_2, \tau_3), (\tau_2, \tau_4), (\tau_3, \tau_4)\}$ and $\lambda_2 = \{(\tau_1, \tau_4)\}$.

The CRPD cost for the matrix $A_{4,1000}$ is computed as $(\gamma(4, \lambda_1) + \gamma(4, \lambda_2)) \cdot \mathsf{BRT}$. The cost of λ_2 in both approaches is:

$$\gamma(4, \lambda_2) = |ECB_1 \cap UCB_4| \times \mathsf{BRT} = 5 \cdot \mathsf{BRT}$$

We now compute the cost of λ_1 with both approaches starting with Partitioning-ver1.

	τ_2	τ_3	τ_4
τ_1	1	1	2
τ_2	-	1	1
τ_3	-	-	1

Table 4.2: Preemptions matrix

4.1.2.2 Cost of λ_1 with Partitioning-ver1.

We only compute the cost of $\gamma_4^{ecbp}(\lambda_1)$, since $\gamma(4,\lambda_1) = min(\gamma_4^{ecbp}(\lambda_1), \gamma_4^{ucbp}(\lambda_1))$. Then,

$$\gamma_4^{ecbp}(\lambda_1) = \gamma_{4,1}^{ecbp}(\lambda_1) + \gamma_{4,2}^{ecbp}(\lambda_1) + \gamma_{4,3}^{ecbp}(\lambda_1)$$

$$\gamma_{4,1}^{ecbp}(\lambda_1) = max(|\mathcal{E}_1 \cap \mathcal{U}_2|, |\mathcal{E}_1 \cap \mathcal{U}_3|, |\mathcal{E}_1 \cap \mathcal{U}_4|) = 8$$

$$\gamma_{4,2}^{ecbp}(\lambda_1) = max(|(\mathcal{E}_1 \cup \mathcal{E}_2) \cap \mathcal{U}_3|, |(\mathcal{E}_1 \cup \mathcal{E}_2) \cap \mathcal{U}_4|) = 11$$

$$\gamma_{4,3}^{ecbp}(\lambda_1) = max(|(\mathcal{E}_1 \cup \mathcal{E}_2 \cup \mathcal{E}_3) \cap \mathcal{U}_4|) = 13$$

$$\gamma_4(\lambda_1) \leq 32 \cdot \mathsf{BRT}$$

We can say now that $\gamma(4,t) \leq 37 \cdot \mathsf{BRT}$ with Partitioning-ver1.

4.1.2.3 Cost of λ_1 with Partitioning-ver2.

The cost of $\gamma_4(\lambda_1)$ with Partitioning-ver2 can be described as:

$$\gamma_4(\lambda_1) = \mathsf{BRT} \cdot max \begin{cases} \gamma_4(\{(\tau_4,\{\tau_1\}),(\tau_4,\{\tau_2\}),(\tau_4,\{\tau_3\})\}) & = 25 \\ \gamma_4(\{(\tau_4,\{\tau_1,\tau_2\}),(\tau_4,\{\tau_3\}),(\tau_2,\{\tau_1\})\}) & = 29 \\ \gamma_4(\{(\tau_4,\{\tau_1,\tau_3\}),(\tau_4,\{\tau_2\}),(\tau_3,\{\tau_1\})\}) & = 29 \\ \gamma_4(\{(\tau_4,\{\tau_2,\tau_3\}),(\tau_4,\{\tau_1\}),(\tau_3,\{\tau_2\})\}) & = 25 \\ \gamma_4(\{(\tau_4,\{\tau_1,\tau_2,\tau_3\}),(\tau_3,\{\tau_1,\tau_2\}),(\tau_2,\{\tau_1\})\}) & = 31 \\ \gamma_4(\{(\tau_4,\{\tau_1,\tau_2,\tau_3\}),(\tau_3,\{\tau_1\}),(\tau_3,\{\tau_2\})\}) & = 28 \end{cases}$$

$$\gamma_4(\lambda_1) = 31 \cdot \mathsf{BRT}$$

We can say now that $\gamma(4,t) = 36 \cdot \mathsf{BRT}$ with Partitioning-ver2.

4.1.2.4 Conclusions

The CRPD cost of the matrix $A_{4,1000}$ with *Partitioning-ver1* is less than or equal to $(32+5) \cdot \mathsf{BRT} = 37 \cdot \mathsf{BRT}$; with *Partitioning-ver2* the cost is $(31+5) \cdot \mathsf{BRT} = 36 \cdot \mathsf{BRT}$. In both methods the cost computed is less than the CRPD cost of the scenario ($38 \cdot \mathsf{BRT}$). This shows that *Partitioning-ver1* and *Partitioning-ver2* do not provide a correct upper bound for this specific scenario.

Tasks	ECBs	UCBs
τ_1	{1,2,3,4,5,6,7,8,9,10,11,20}	$\emptyset$
τ_2	{1,2,3,4,6,7,8,9,12,13,14,15,16,17,20}	{1,2,3,4,6,7,8,15,16,17,20}
τ_3	{2,3,4,5,7,8,9,10,11,13,14,16,17,18,19,20}	{3,4,5,8,9,10,11,14,17,19,20}
τ_4	$\emptyset$	{5,6,7,8,9,12,13,14,15,16,17,18,19}

Table 4.3: ECBs and UCBs sets

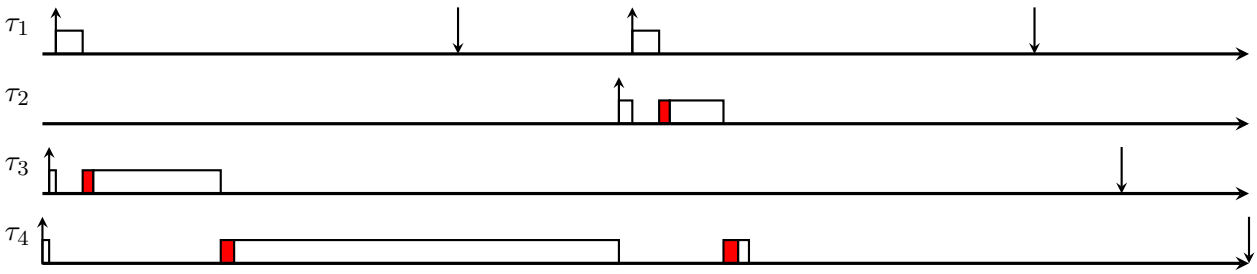


Figure 4.1: Counter example

4.2 System model

In this chapter we consider, a preemptive system of N independent real-time sporadic tasks with constrained deadlines. We assume that the tasks are scheduled by Earliest Deadline First (EDF) [19].

4.3 Preemption Interval

In [57], in the case of EDF scheduling the number of preemptions on a job of task τ_i is computed using the relative deadline D_i. In particular, all instances of jobs with arrival and deadline in $[0, D_i]$ may preempt τ_i. This is a pessimistic assumption because jobs that arrive after the completion of τ_i are counted as preempting jobs. A less pessimistic assumption would be to consider only jobs that arrive before the worst-case response time of τ_i. However, computing an upper bound to the response time of a task in EDF is complex.

In this chapter, we propose to use the concept of *preemption interval*: an upper bound I_i to the length of the interval of time where a job of task τ_i may be preempted. Therefore I_i is a bound on the length of the interval between the start of the execution of any job of τ_i and its completion. Indeed, a task can be preempted only after it has started its execution, and before it completes.

Please notice that I_i is different from the worst-case response time R_i: the latter corresponds to the size of an interval starting from the job's arrival until its completion. As the start time of a job is always greater than or equal to its arrival time, it follows that $R_i \geq I_i$ by definition.

Let $\eta'_i(j,t)$ be the number of jobs of task τ_j that can preempt τ_i on an interval of length t, with $t \leq D_i$.

Lemma 4.3.1. *The number of preemptions by higher preemption-level task τ_j on task τ_i on any contiguous interval of size t inside $[0, D_i]$ (with $t \leq D_i$) is:*

$$\eta'_i(j,t) = min\left(\left\lceil \frac{t}{T_j} \right\rceil, \boldsymbol{Pr}_j(D_i)\right).$$

Proof. $\eta'_i(j,t)$ cannot be larger than $\mathbf{Pr}_j(D_i)$, that is the maximum number of preemptions of τ_j on τ_i. Also, since any job of τ_j must arrive inside the interval, $\eta'_i(j,t)$ cannot be larger than the number of jobs of τ_j arriving in any contiguous interval of size t. The latter can be computed as $\left\lceil \frac{t}{T_j} \right\rceil$. Hence the lemma is proved. □

Preempting jobs, that arrive during the preemption interval I_i of a task τ_i, contribute to increasing the length of the interval: in fact, we must account for their execution and for their cache impact.

Thus, to compute I_i we use an iterative formula starting with the WCET of the task. Since the preemption interval can only increase with the preempting jobs and cannot be greater than the relative deadline, we can use $\eta'_i(j,t)$ to count the number of instances of τ_j that can increase the preemption interval. The WCET of the preempting jobs and the CRPD provoked by them must be added to the WCET of the task to obtain the new preemption interval. The iterative procedure is guaranteed to stop, since $\eta'_i(j,t)$ is bounded by a constant $\mathbf{Pr}_j(D_i)$.

Theorem 4.3.2. *The following iterative equation gives an upper bound to the preemption interval of task τ_i:*

$$\begin{cases} I_i^{(0)} = C_i \\ I_i^{(k)} = C_i + \gamma'(I_i^{(k-1)}) + \displaystyle\sum_{j|D_j<D_i} C_j \cdot \eta'_i(j, I_i^{(k-1)}) \end{cases} \tag{4.11}$$

where $\gamma'(I_i)$ is the CRPD provoked by the preempting jobs during interval I_i. The iteration stops when $I_i^{(k)} > D_i$ (unschedulable) or $I_i^{(k-1)} = I_i^{(k)}$.

Proof. Only higher preemption-level tasks can interfere with the execution of a given task. In our model, we only consider two resources, the processor and the cache memory, thus the possible sources of interference are the suspension of the execution by a higher preemption-level task, and its CRPD. Lemma 4.3.1 provides an upper bound on the number of preemptions in interval I_i. Therefore, the value obtained in the last iteration represents an upper bound to the length of the preemption interval, or the fact that the task is not schedulable. □

Computing the CRPD during the Preemption Interval. Equations (3.3), (3.6) and (3.8) use $\mathbf{Pr}_j(D_i)$ to count the number of preemptions by task τ_j on a job of τ_i.

We assume to compute the preemption interval in the task order from τ_1 (the task with the highest preemption-level) to τ_n (the task with the lowest preemption-level). Of course, $I_1 = C_1$. We define $\mathbf{Pr}'_j(D_i)$ as:

$$\mathbf{Pr}'_j(D_i) = min\left(\left\lceil \frac{I_i}{T_j} \right\rceil, \left\lceil \frac{D_i - D_j}{T_j} \right\rceil\right) \tag{4.12}$$

and we use $\mathbf{Pr}'_j(D_i)$ instead of $\mathbf{Pr}_j(D_i)$ in the above equations. Also, to properly bound the number of preempting jobs from higher preemption-level tasks during the computation of a preemption interval I_i, we use $\eta'_i(j, I_i)$, which represents the number of jobs arriving in the preemption interval I_i and with relative deadline less than D_i, instead of $\eta(j, I_i)$ in Equations (3.3), (3.4), (3.6), (3.8) and (3.9). Therefore, we change the sum $\sum_{\forall k, t \geq D_k > D_j}$ to $\sum_{\forall k, D_k > D_j}$ in the Equation (3.6) and the multiset union $\biguplus_{\forall k, t \geq D_k > D_j}$ to $\biguplus_{\forall k, D_k > D_j}$ in Equation (3.3) and Equation (3.8).

Computing the CRPD with the Preemption Interval. To do so, we use $\mathbf{Pr}'_j(D_i)$ instead of $\mathbf{Pr}_j(D_i)$ in Equations (3.3), (3.6) and (3.8).

4.4 Reduce the number of UCBs

As we use a multiset to represent the UCBs for each preemption point in the task, the size of the obtained *SOM* may be too large. Indeed the complexity of the *Combined* SOM approach depends on the size of the *SOM* of each preempted task.

Algorithm 1 describes the *reduce* operation on a *SOM* $\overline{\mathsf{A}}$. The algorithm removes the multiset X with the minimum size from the copy of the input vector (Lines 3 and 4); then it performs a *multiset fusion* of $\overline{\mathsf{X}}$ with every other multiset of $\overline{\mathsf{R}}$, and selects the one that gives the smallest result (Lines 5-9); then, it removes this multiset and adds the result of the fusion (Line 10). It repeats the operation as long as the size of $\overline{\mathsf{R}}$ is greater than M_{reduce} (while loop at Line 2).

The resulting *SOM* contains at most M_{reduce} multisets. We highlight that the *reduce* operation may increase the size of the multisets, thus introducing some pessimism; however, by selecting the merged multiset with smaller sizes, the pessimism is kept in check.

4.5 Complexity

We denote by ω an upper bound to the maximum number of instances of any task contained in the hyperperiod $\mathrm{hp}(\mathcal{T})$:

$$\omega = \max_{\forall \tau_i \in \mathcal{T}} \left(\left\lfloor \frac{\mathrm{hp}(\mathcal{T}) - D_i}{T_i} \right\rfloor + 1 \right)$$

Algorithm 1 reduce()

Require: A vector of multisets $\overline{\mathsf{A}}$
Require: Max size M_{reduce}
Ensure: A vector of multisets $\overline{\mathsf{R}}$
1: $\overline{\mathsf{R}} \leftarrow \overline{\mathsf{A}}$
2: **while** $|\overline{\mathsf{R}}| > M_{reduce}$ **do**
3: $\mathsf{X} \leftarrow$ multiset with minimum size in $\overline{\mathsf{R}}$
4: $\overline{\mathsf{R}} \leftarrow \overline{\mathsf{R}} \setminus \{\mathsf{X}\}$
5: $\mathsf{L} \leftarrow \mathsf{X} \bigtriangledown \mathsf{Y}$, with any $\mathsf{Y} \in \overline{\mathsf{R}}$
6: **for** $\mathsf{Z} \in \overline{\mathsf{R}}$ **do**
7: **if** $|\mathsf{X} \bigtriangledown \mathsf{Z}| < |\mathsf{L}|$ **then**
8: $\mathsf{L} \leftarrow \mathsf{X} \bigtriangledown \mathsf{Z}$
9: $\mathsf{Y} \leftarrow \mathsf{Z}$
10: $\overline{\mathsf{R}} \leftarrow \overline{\mathsf{R}} \cup \{\mathsf{L}\} \setminus \{\mathsf{Y}\}$
return $\overline{\mathsf{R}}$

We denote by ψ an upper bound of the maximum number of preemptions by any task on any job from another task as follows:

$$\psi = \max_{\forall \tau_j, \tau_i \in \tau; j<i} (\mathbf{Pr}_j(D_i))$$

The complexity of any multiset operation (union, intersection, etc.) is bounded by $\mathcal{O}(Z^2)$ with Z the number of cache sets (we represent a multiset as a list of pairs (value, repetition factor)). The complexity of any operation between a multiset and a *SOM* is $\mathcal{O}(M_{reduce}Z^2)$, with M_{reduce} the maximum number of elements in a *SOM*.

4.5.1 Complexity of UCB multiset

In the UCB multiset union approach, we need to perform the union of the UCBs of lower preemption-level tasks. In the worst-case, the complexity is $\mathcal{O}(NZ^2)$. As a task τ_i can be preempted at most $\omega\psi$ times by another task τ_j, the complexity of the union between all the lower preemption-level tasks is $\mathcal{O}(\omega\psi NZ^2)$. The intersection with the ECB has complexity $\mathcal{O}(Z^2)$, thus the complexity for a preempting task is again $\mathcal{O}(\omega\psi NZ^2)$, and the total complexity is $\mathcal{O}(\omega\psi N^2Z^2)$.

4.5.2 Complexity of ECB-union *SOM*

The ECB-union *SOM* approach can be split in three steps: the computation for the preempting task, building the list of costs, and summing the largest costs. The first step is a union between all the preempting tasks ECBs, thus its complexity is $\mathcal{O}(NZ^2)$. The second step consists of listing all the preemption costs, its complexity is $\mathcal{O}(N\omega\psi M_{reduce}Z^2)$.

The last step consists of sorting and summing the ω greatest values. The number of element in the list is bounded by $N\omega\psi$, thus the complexity is $\mathcal{O}(\omega^2 N\psi \log(N\omega\psi))$. The

complexity for the three steps is $\mathcal{O}(N\omega\psi(\omega\log(N\omega\psi) + MZ^2))$. Since we do this for N tasks, the final complexity can be bound by $\mathcal{O}(N^2\omega\psi(\omega\log(N\omega\psi) + M_{reduce}Z^2))$.

Since ω and ψ are pseudo-polynomial in the size of the input (they depend on periods and relative deadlines), we can state that the overall complexity is pseudo-polynomial, hence in the same complexity class as the original demand bound function.

The iterative formula for computing the preemption interval has the same structure of the response time analysis in Fixed Priority analysis, hence it is also pseudo-polynomial in the input.

4.6 Evaluation

4.6.1 Experiments Setup

To evaluate the impact of the preemption interval on schedulability. We consider a single core ARM7 processor with one level of set-associative cache memory. We use OTAWA [17] to compute the WCET of the tasks and the list of UCBs and ECBs. Blocks are collected based on the method proposed by Healy et al. [42]. Tasks are chosen from the Malärdalen benchmark suite [39] and from TACLeBench [35].

Tasks	Bench	WCET	Nb ECB	Nb UCB
fibcall	Malärdalen	5235	14	7
lcdnum	Malärdalen	9132	16	8
duff	Malärdalen	12034	20	10
binarysearch	TACLeBench	21511	42	21
insertsort	TACLeBench	26086	58	29
iir	TACLeBench	31439	62	31
complex_updates	TACLeBench	62040	64	34
ns	Malärdalen	88058	32	16
cnt	Malärdalen	112574	54	27
ud	Malärdalen	132411	64	36
fir2dim	TACLeBench	228105	64	36
ludcmp	TACLeBench	340181	64	37
crc	Malärdalen	442317	64	32
expint	Malärdalen	554166	46	25
nsichneu	Malärdalen	569174	64	32

Table 4.4: List of tasks used in the experiments: Cache size of 2KB, 2 ways, $\mathsf{BRT} = 200$ cycles.

The analysis has been repeated multiple times with different cache configurations: 2KB, 4KB, with 2 and 4 ways. The size of a cache line has been fixed to 32 bytes, as it is a standard in ARM processors, like the MPCore A7.

Following of [24], we set the Block Reload Time $\mathsf{BRT} = 50, 100, 200$. In all configurations we considered the LRU replacement policy. In Table 4.4 we report the value of

the WCET (in processor cycles) and the size of the ECB multiset for the corresponding benchmarks. The number reported for the UCB is the size of the multiset obtained by merging all the UCB multisets for each preemption point using the *fusion* operation. Please notice that the WCET and the number of UCBs and ECBs depends on the cache configuration.

To adhere to a more realistic setting in a real compiler and linker, once the tasks have been selected to build a task set of 6,8,10 or 12 tasks, their memory locations are randomly assigned by adding a random offset to the cache sets index of their CB addresses such that:

$$\mathcal{U}_i'^k = \biguplus_{\forall e \in \mathcal{U}_i^k} \{(e + \phi) mod Z\} \tag{4.13}$$

with Z the number of cache sets in the cache, and ϕ a random number in $[0, Z-1]$.

We generate a set of utilizations using the UUnifast algorithm [22], and we compute a tentative period $T_i' = \frac{C_i}{U_i}$. In order to generate realistic workloads, and to avoid an excessive length of the hyperperiod of the task sets, we approximate the value of T_i' to a value T_i taken from a list of periods (expressed in thousands of "ticks"): $\{10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000\}$, taken from [49]. Then, we assign each task a relative deadline uniformly chosen in interval $[\max(C_i, 0.9T_i), T_i]$. Finally, all schedulability analyses were performed with the data computed in the previous step.

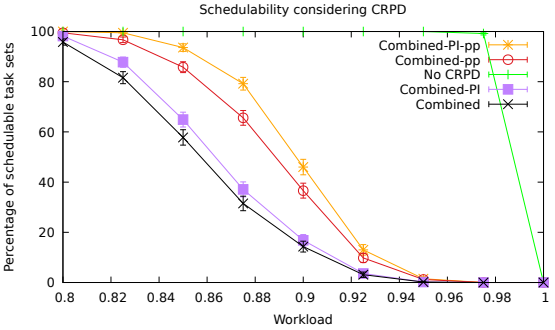
4.6.2 Results

All experiments have been conducted using the tasks of Table 4.4, taken from [39] and [35], and applying all algorithms on 1000 randomly generated task sets per utilization point.

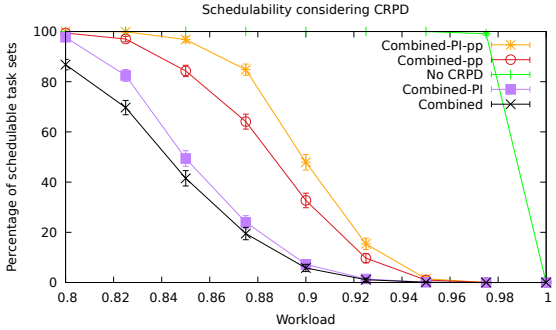
In all the figures the *No-CRPD* curves represent the results of the dbf analysis *without* considering the CRPD cost, and it is reported as a reference for the other algorithms. We decide to use the combined approach from Lunniss et al. [57] to see the impact of the preemption interval. Our approach is labeled as *Combined-PI* and *Combined-PI-pp* for *Combined preemption interval* and *Combined preemption points and preemption interval*. The original version from the literature are denoted as *Combined* and *Combined-pp*, however, the maximum number of preemption points per task is limited with Algorithm 1. To the best of our knowledge, the combined approach is the best approach so far for EDF scheduling. For both *Combined-PI* and *Combined* we use the reduce operation 1 to consider only one multiset of UCBs per task. For each point, we also report the 95% confidence interval, computed as:

$$CI = \hat{p} \pm Z^* \sqrt{\frac{\hat{p} \cdot (1 - \hat{p})}{n}}$$

where $\hat{p}$ is the ratio of schedulable task sets for a given workload, n is the sample size (here 1000), and 1.96 is the value for Z^*.

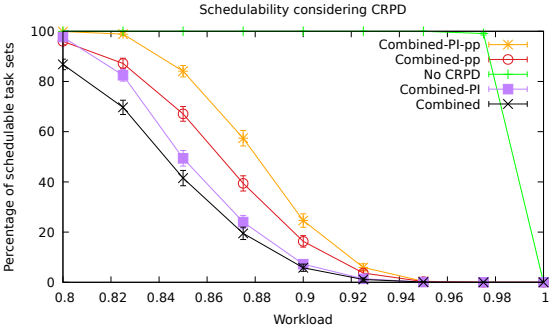


(a) Schedulability of $n_{tasks} = 12$ with cache of 2KB, 2 ways, BRT $= 200$ and $M_{reduce} = 4$

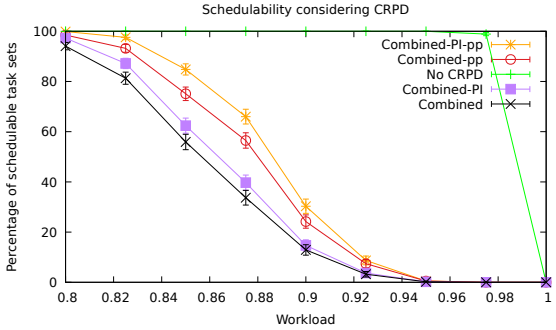


(b) Schedulability of $n_{tasks} = 12$ with cache of 4KB, 2 ways, BRT $= 200$ and $M_{reduce} = 4$

Figure 4.2: Impact of cache size.

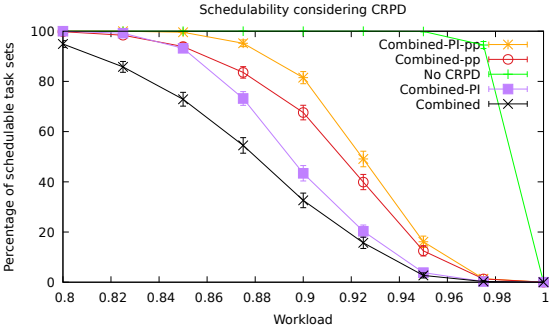


(a) Schedulability of $n_{tasks} = 12$ with cache memory of 4KB, 2 ways, BRT $= 200$ and $M_{reduce} = 2$

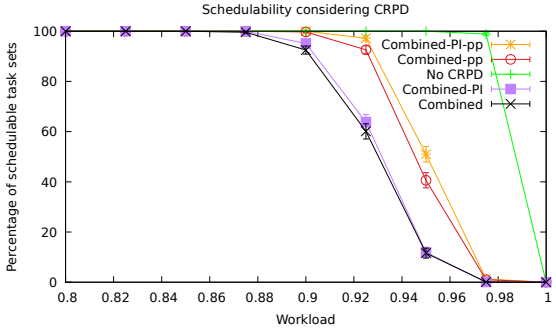


(b) Schedulability of $n_{tasks} = 12$ with cache memory of 4KB, 4 ways, BRT $= 200$ and $M_{reduce} = 2$

Figure 4.3: Impact of number of ways.



(a) Schedulability of $n_{tasks} = 6$ with cache memory of 4KB, 2 ways, BRT $= 200$ and $M_{reduce} = 4$



(b) Schedulability of $n_{tasks} = 12$ with cache memory of 4KB, 2 ways, BRT $= 50$ and $M_{reduce} = 4$

Figure 4.4: Impact of number of tasks and BRT.

Impact of the cache size The variation of the cache size has a strong impact on the Combined and Combined-PI-pp methods. Consider the experiments in Figure 4.2a and Figure 4.2b, whose only difference is the size of the cache (2KB and 4KB, respectively). The Combined analysis loses 10% of schedulable task sets already at $U = 0.8$ as we increase the size of the cache, whereas Combined-PI-pp increases its performances for all values of the workload. In particular, at $U = 0.875$ Combined-PI-pp increases its performance by more than 10%. We explain this difference with the fact that using only one UCB set per task produces larger UCB set as the cache size increases, thus increasing pessimism.

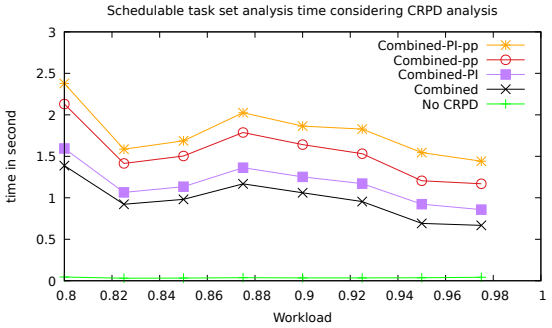
Impact of the number of cache ways and cache sets. Figures 4.3a and 4.3b show two different scenarios that use a 4KB cache, the first with 2 ways and the second with 4 ways. As you can see, the second scenario provides better performance than the first scenario while the performance of different versions of the combined approach are closer. We explain this by observing that our algorithms is more effective in reducing the pessimism as the scenario becomes more competitive (less number of ways).

Impact of the number of tasks. The number of tasks has a strong impact on the schedulability ratio. Comparing Figure 4.2b (12 tasks) with Figure 4.4a (6 tasks), we notice two things: first, by increasing the number of tasks, the performance of Combined-PI decreases to become closer to combined; Second, by increasing the number of tasks, the performance of Combined-PI-pp decreases slower than the performance of Combined-pp, increasing the gap between the two approaches. This means that the combination of the Preemption Interval and the use of several UCB multisets to describe preemption points is more useful with a high number of tasks.

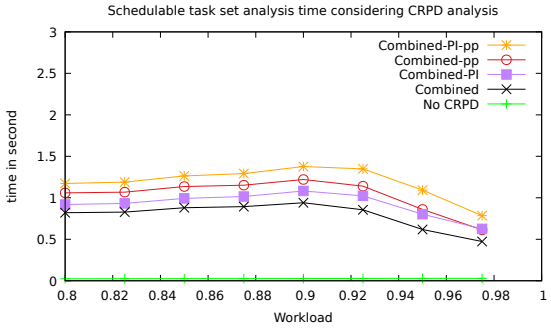
You may notice that overall the schedulability ratio is lower in the case of a task set with 6 tasks compared to a task set of 12 tasks at $U = 0.975$. This is due to the algorithm we use for generating the task set: in fact, in order to obtain the same workload, a set of 6 tasks must have a higher average workload per task $\frac{C_i}{T_i}$ and thus $\frac{C_i}{D_i}$ than a set of 12 tasks. Therefore, a set of 6 tasks is marginally less schedulable than a set of 12 tasks.

Impact of the block reload time. By comparing Figure 4.2b and Figure 4.4b, we conclude that the block reload time has a strong impact on schedulability, since 90% of task sets are schedulable with combined-PI-pp and a $\mathsf{BRT} = 50$ compared to 50% in the case of $\mathsf{BRT} = 200$ at $U = 0.9$. With a $\mathsf{BRT} = 50$ the gap between Combined-PI-pp and Combined-pp is smaller and we observe the same between Combined-PI and Combined. However, there is an augmentation of 40% in the schedulable task sets between Combined and Combined-PI-pp. This gap shows the gain in precision when using the reduce operation. As you will see later in the section, using the *reduce* heuristic allows us to find a compromise between the accuracy of the analysis and its complexity.

The impact of the BRT on the schedulability contradicts the conclusions of Shah et al. [74]. However, we observe that in the experiments of [74] the task sets are generated differently: the WCET of all tasks is almost a multiple of the BRT (probably due to the



(a) Analysis time for $n_{tasks} = 12$ with cache memory of 2KB, 4 ways, BRT = 50 and $M_{reduce} = 4$



(b) Analysis time for $n_{tasks} = 12$ with cache memory of 2KB, 4 ways, BRT = 50 and $M_{reduce} = 2$

Figure 4.5: Impact of their reduce operation on analysis time.

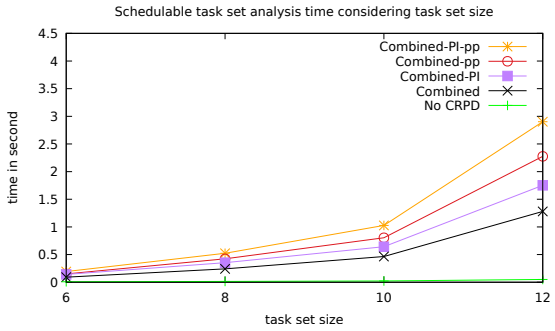


Figure 4.6: Analysis time for a cache memory of 4KB, 2 ways, $U = 0.85$, BRT = 200 and $M_{reduce} = 4$

different WCET analysis tool), and the periods are computed as $T_i = \frac{C_i}{U_i} \approx \frac{k_i \mathsf{BRT}}{U_i}$, with U_i randomly generated by UUnifast. Therefore, the dominant factor in their experiments is the number of preemptions, and not the value of BRT. In our case, 1) the WCETs and the UCBs and ECBs have been computed by OTAWA, and the WCETs are not proportional to the BRT; 2) periods are selected from a list. We believe that our experimental setting presents a better approximation of a real application.

Analysis time. We can see in Figure 4.6 the analysis time of schedulable task sets for the different approaches with 4 configurations of task set sizes: 6, 8, 10 and 12 tasks with a processor load of 0.85. The other parameters are a reduce operation parameters M_{reduce} equal to 4 with BRT = 200 and a cache memory of 4KB and 2 ways. Between the first and the last case (6 and 12 tasks, respectively), we observe a multiplying factor of 10 for the analysis time of the combined approach with *SOM* as model for the UCBs.

The reduce operation has a strong impact on the analysis time of schedulable task sets as you can see with Figure 4.5a and Figure 4.5b. Both experiments use the same cache configuration, a 2KB with 4 ways cache memory and a BRT = 50. The task set is of size

12 for both experiments and the reduce operation parameter M_{reduce} is set to 4 in the first configuration and 2 in the second. As you can see, multiplying by two M_{reduce} doubles the gap between combined-PI and combined-PI-pp time analysis. The same observation is also true between Combined and Combined-pp. We think that using all the UCB multisets obtained by the WCET analysis for each task is not scalable. It also shows the importance of heuristics such as the reduce operation to keep the complexity in check, to increase the number of schedulable task sets that the analysis can verify.

4.7 Conclusions and future work

In this chapter we improve the precision of CRPD analysis while proposing a compromise with the complexity of the analysis. We target fully preemptive real-time tasks scheduled by EDF on single processor systems with set-associative caches. In particular, we propose two original contributions:

- A method to precisely compute the Preemption Interval length of each task (Section 4.3), thus the number of preemptions that a task can undergo;
- A heuristic to keep the complexity of analysis in check (Algorithm 1).

In this chapter we worked on the basic techniques behind CRPD analysis. For simplicity, we decided to focus on single-processor architectures with one single level of cache and we only consider instruction caches. While this is a very limited setting compared to modern architectures, we believe that our propositions represent a building brick that can be reused to improve CRPD analysis in more complex settings like multilevel caches and multicore systems with private and shared caches.

Regarding possible extensions for data caches, we believe that the problem lies mostly in the correct computation of the UCBs and ECBs for data, especially in the presence of pointers in the code.

Chapter 5

Reducing Task Set Vulnerability

Contents

In this chapter, we propose a methodology to reduce the *vulnerability* of a hard real-time application to soft errors caused by transient faults in *Instruction L1* (IL1) cache memories. The vulnerability of an instruction is proportional to the time it spends in the cache, and it is therefore subject to transient faults. Our method is based on a static analysis tools to analyze a binary program and compute the overall vulnerability of its instructions. Estimating the vulnerability of data in the data cache is left for futur works. The tool we use to perform the static analysis can not yet extract the properties concerning the relation between the anlyzed program and the data cache.

We propose to reduce this vulnerability by *invalidating* some CBs at specific instants during the execution. In this way, we force vulnerable instruction blocks to be reloaded from higher layers in the memory hierarchy. Since adding invalidation points will likely increase the WCETs of the tasks, we perform a static analysis to guarantee that the system remains schedulable after modification. In this way, all the application deadlines are respected.

In other words, our proposal is to select the most vulnerable blocks and to choose the most suitable moments when to reload these blocks without impacting real-time constraints. We also present different possible practical implementations of the cache invalidation mechanism according to the type of the IL1 cache memory, i.e. direct mapped or set-associative. Finally, we analyze how our methodology can be combined with existing hardware protection mechanisms as ECC memories. The performance of our methodology in terms of vulnerability reduction is evaluated with a set of experiments on benchmarks.

First, in Section 5.1 we present the system model and our assumptions for this chapter.

Our contributions to reduce the vulnerability in the cache memory are detailed in Section 5.2 and Section 5.3. Then, we present a case study and experiments to demonstrate the improvements in reliability with our methods compared to systems without any protection in Section 5.4. Finally, Section 5.5 gives a conclusion and presents new research avenues for future works.

5.1 System model

In this chapter we consider, a non preemptive system of N independent real-time sporadic tasks with implicit deadlines. We assume that the tasks are scheduled by Non-Preemptive Earliest Deadline First (NP-EDF) [46]. With this scheduling, we do not have to consider CRPD between tasks.

In this chapter we also need to estimate the *worst-case vulnerability* of a cache line. Therefore, during the vulnerability analysis, all scenarios with a potential cache hit will be considered as always hit (*may analysis*). Therefore, when referring to UCBs in this chapter, we are referring to those obtained via the may analysis.

In addition, contrary to CRPD analysis, we consider two sets of UCBs per BB, one at the entry and the other at the exit. Since we do not have to consider the BBs as preemption points.

Finally, we consider that the contents of the IL1 cache are flushed just after the execution of a job. Therefore, the instructions of a task cannot be vulnerable during the

execution of another one.

5.2 Task Profile

To reduce the vulnerability of a task, we can artificially invalidate CBs at some point in the code. In this way, the CBs will be reloaded from the main memory at their next read accesses, thus reducing their vulnerability. This is equivalent to forcing artificial cache misses. However, inserting cache misses also increases the task's worst-case execution time: if we insert too many cache misses, the WCET will increase to the point that the task set becomes unschedulable. Therefore, we need to carefully select the locations where to invalidate the cache misses so that the task set remains schedulable.

The number λ of potential locations where to invalidate the cache for a set of tasks is linear in the size of the code. However, the number of configurations, is 2^{λ} which is exponential in the code size. Even the use of specialized solvers for exploring all possible configurations is too expensive.

This chapter takes a different approach. At first, we analyze one task at a time to build a *task profile*, which consists of a list of tuples containing the WCET, the task's vulnerability factor, and the combination of cache invalidation locations. Then, we use these profiles to determine the *best* configuration of artificial cache misses for each task, allowing the set of tasks to remain schedulable.

5.2.1 Computing the vulnerability factor of tasks

In this chapter, we use the *TAsk Vulnerability Factor* (TAVF) as a metric to evaluate the vulnerability of a task in the system, that is the probability that a fault occurring on one of the CBs used by the task will affect its behavior.

TAVF is a metric similar to the *Temporal Vulnerability Factor* (TVF) proposed by Wang et al. [83]. The TVF represents the probability that a fault occurring on a used part of cache memory will provoke a failure in the application. Therefore, the TVF is a global metric on the cache, whereas TAVF is specific to a given task.

To compute the TAVF, we first need to compute an upper limit to the vulnerability of the task. We consider the case where all instructions are vulnerable during its worst-case execution time, and we denote it as *task exposition*. Therefore, the task exposition can be computed simply by multiplying the WCET of the task times its size in bytes.

We use OTAWA [17], a static analysis tool that is mainly used to estimate the WCET of a task, and that we extended to compute its vulnerability.

The vulnerability of a task consists in summing the worst vulnerability of each LB. It can be decomposed into two components:

- The *baseline vulnerability* is the vulnerability of a LB during the execution of those BBs which contain instructions from the corresponding CB;
- The *path vulnerability* is the vulnerability of a LB during the execution of those BBs which contain no instructions from the corresponding CB.

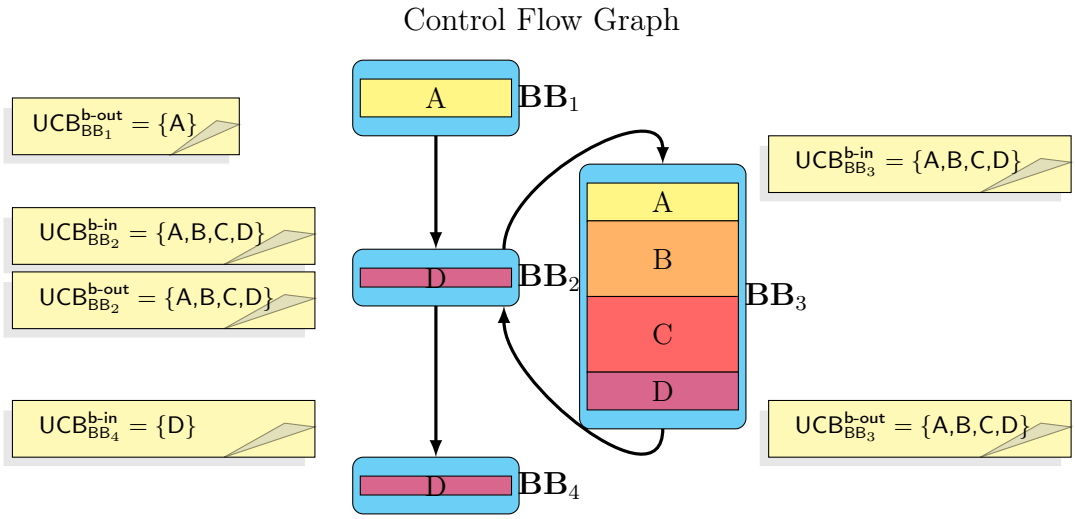


Figure 5.1: Example of task CFG (bis)

5.2.1.1 Computing the baseline vulnerability

Similarly to [83], we only consider as vulnerable the time between two readings of the same CB without the latter being evicted.

Since the WCET estimation tool has a time granularity of a BB, we consider that all BB instructions are vulnerable during the entire BB execution.

Equation (5.1) allows us to compute the baseline vulnerability for a LB lb.

$$\nu_{\mathsf{lb}}^{\text{bv}} = d_{\mathsf{lb}} \cdot I_{\mathsf{lb}} \cdot \mathsf{wcet}_{\mathsf{BB}_{\mathsf{lb}}} \tag{5.1}$$

where $\mathsf{wcet}_{\mathsf{BB}_{\mathsf{lb}}}$ corresponds to the WCET of $\mathsf{BB}_{\mathsf{lb}}$ which represents the BB that contains lb. I_{lb} denotes the maximum number of executions of lb for one instance of the task it belongs, and d_{lb} the size (in bytes) of the vulnerable instructions of the CB of lb during the execution of $\mathsf{BB}_{\mathsf{lb}}$.

Equation (5.2) shows how d_{lb} is computed: $\mathsf{UCB}_{\mathsf{lb}}^{\text{b-out}}$ is the list of UCBs at the output of $\mathsf{BB}_{\mathsf{lb}}$, in the same way, the list of UCBs at the input of $\mathsf{BB}_{\mathsf{lb}}$ is denoted by $\mathsf{UCB}_{\mathsf{lb}}^{\text{b-in}}$. Also, with $\mathsf{CB}_{\mathsf{lb}}$ we denote the CB that contains lb.

$$d_{\mathsf{lb}} = \begin{cases} |\mathsf{CB}_{\mathsf{lb}}|, & \text{if } \mathsf{CB}_{\mathsf{lb}} \in \mathsf{UCB}_{\mathsf{lb}}^{\text{b-out}} \\ |\mathsf{lb}|, & \text{otherwise} \end{cases} \tag{5.2}$$

During the execution of lb, its instructions are always vulnerable. However, if $\mathsf{CB}_{\mathsf{lb}}$ belongs to $\mathsf{UCB}_{\mathsf{lb}}^{\text{b-out}}$ then other instructions from $\mathsf{CB}_{\mathsf{lb}}$ may be used later by other BBs, and in this case we have to consider the entire CB as vulnerable. In BB_3 in Figure 5.1, the LB that belongs to CB D is vulnerable during the execution of BB_3. Furthermore, as D is present in $\mathsf{UCB}_{\mathsf{BB}_3}^{\text{b-out}}$ as it can be reused later without eviction in the meantime in BB_2 and BB_4, the other instructions of D are also vulnerable during the execution of BB_3.

To compute the baseline vulnerability for a task τ_i, we just sum the baseline vulnerability of its LBs:

$$\nu_i^{\text{bv}} = \sum_{\forall \mathsf{lb} \in \tau_i} \nu_{\mathsf{lb}}^{\text{bv}} \tag{5.3}$$

Theorem 5.2.1. *The baseline vulnerability of a task τ_i computed with Equation (5.3) is an upper bound to the sum of the vulnerabilities of the task's CBs during their execution.*

Proof. Suppose an instruction i is stored in the cache while another instruction j from LB lb belonging to the same CB of i is currently executed. Suppose that a path exists from j to a point where instruction i is executed without being evicted from the cache. Observe that instruction i is vulnerable on this path.

There are two cases:

- the instruction i is part of lb;
- the instruction i is not part of lb.

Equation (5.1) covers the first case, since it considers LBs vulnerable during the execution of their BB (it multiplies the LB size by the WCET of its BB).

Furthermore, Equation (5.1) considers that the entire CB is vulnerable, and not only the executed LB belonging to it, if it belongs to the set $\mathsf{UCB}_{\mathsf{lb}}^{\text{b-out}}$. Since i will be executed in the future without being evicted, its CB belongs to $\mathsf{UCB}_{\mathsf{lb}}^{\text{b-out}}$. This covers the second case.

Since Equation (5.3) is the sum of the baseline vulnerability for each LB of the task computed with Equation (5.1), we conclude that Equation (5.3) is an upper bound to the sum of the CBs vulnerability during their execution. □

5.2.1.2 Computing path vulnerability

The baseline vulnerability of a task is not sufficient to bound its vulnerability. During the execution of a basic block BB, CBs that have no instructions in BB can be vulnerable if they belong to the UCB set at the exit of BB. However, their vulnerability is not considered by Equation (5.3), because the actual value of their vulnerability depends on the *path* followed by the program. Therefore, we need to look at all paths in the graph to effectively compute this additional vulnerability. We denote this component of the vulnerability as *path vulnerability* since it affects only blocks in a path in which they are not used.

For example, in Figure 5.1 CB A consists of two LBs present in BBs BB_1 and BB_3. By the static analysis of the code, we know that A is present in $\mathsf{UCB}_{\mathsf{BB}_1}^{\text{b-out}}$, $\mathsf{UCB}_{\mathsf{BB}_2}^{\text{b-in}}$, $\mathsf{UCB}_{\mathsf{BB}_2}^{\text{b-out}}$ and $\mathsf{UCB}_{\mathsf{BB}_3}^{\text{b-in}}$. Thus, A is vulnerable on path $\mathsf{BB}_1 \rightarrow \mathsf{BB}_2 \rightarrow \mathsf{BB}_3$. As BB_1 and BB_3 contain a LB from A, the vulnerability of A during the execution of BB_1 and BB_3 is already considered by its baseline vulnerability. The path vulnerability therefore must just account for the vulnerability of A during the execution of BB_2.

We propose to compute for each LB the path vulnerability $\nu_{\mathsf{lb}}^{\text{path}}$ as follows:

$$\nu_{\mathsf{lb}}^{\text{path}} = \begin{cases} d_{\mathsf{lb}} \cdot I_{\mathsf{lb}} \cdot \mathcal{P}_{\mathsf{lb}}^{\max}, & \text{if } \mathsf{CB}_{\mathsf{lb}} \in \mathsf{UCB}_{\mathsf{lb}}^{\text{b-in}} \\ 0, & \text{otherwise} \end{cases} \tag{5.4}$$

where $\mathcal{P}_{\mathsf{lb}}^{\max}$ is the WCET of the longest path from the last access to $\mathsf{CB}_{\mathsf{lb}}$ until $\mathsf{BB}_{\mathsf{lb}}$. We consider only the paths where $\mathsf{CB}_{\mathsf{lb}}$ is in the cache along the entire path without being used (and it used at the end of the path). Similarly to the baseline vulnerability, the path vulnerability for a task τ_i can be computed as:

$$\nu_i^{\text{path}} = \sum_{\forall \mathsf{lb} \in \tau_i} \nu_{\mathsf{lb}}^{\text{path}} \tag{5.5}$$

Theorem 5.2.2. *Equation (5.5) provides an upper bound to the path vulnerability for task τ_i.*

Proof. To be considered as vulnerable at a point p, a LB lb must have its CB in the cache, and it must be executed later without any eviction in the meantime.

If lb is vulnerable during the execution of a basic block BB that does not contain any LBs from $\mathsf{CB}_{\mathsf{lb}}$, then BB must be at least in one path of $\mathcal{P}_{\mathsf{lb}'}^{\max}$, where lb' is a LB from $\mathsf{CB}_{\mathsf{lb}}$.

We are now in one of these two cases:

- $\mathsf{lb} = \mathsf{lb}'$, the vulnerability of lb between the start of the execution of BB and the start of the execution of the BB containing lb is bounded by Equation (5.4). Indeed, this equation consists in multiplying the length of the maximum path from $\mathcal{P}_{\mathsf{lb}}^{\max}$ by the size of lb or by the size of its CB.

- $\mathsf{lb} \neq \mathsf{lb}'$, the vulnerability of lb between the start of the execution of BB and the start of the execution of the BB containing lb' is also bounded by Equation (5.4). Indeed, as the BB of lb' is inside a path through a future execution of lb, $\mathsf{CB}_{\mathsf{lb}} \in \mathsf{UCB}_{\mathsf{lb}'}^{\text{b-out}}$. In this case, the equation consists in multiplying the length of the longest path from $\mathcal{P}_{\mathsf{lb}'}^{\max}$ by the size of $\mathsf{CB}_{\mathsf{lb}}$.

Thus we can say that the path vulnerability of $\mathsf{CB}_{\mathsf{lb}}$ can be bounded by $\sum_{\forall \mathsf{lb}' \in \mathsf{CB}_{\mathsf{lb}}} \nu_{\mathsf{lb}'}^{\text{path}}$.

As Equation (5.5) is the sum of ν_{lb}^{path} for all the LBs of task τ_i, it is also an upper bound of the path vulnerability for task τ_i. □

Finally, we compute the TAVF of τ_i as:

$$f_i^v = \frac{\nu_i^{\text{bv}} + \nu_i^{\text{path}}}{C_i \cdot |\tau_i|} \tag{5.6}$$

where $|\tau_i|$ is the size in bytes of the task's code.

LB	vulnerable data size	baseline vulnerability	path vulnerability
$\mathsf{A}_{\mathsf{BB}_1}$	$\lvert\mathsf{A}\rvert$	$\lvert\mathsf{A}\rvert \cdot \mathsf{wcet}_{\mathsf{BB}_1}$	0
$\mathsf{D}_{\mathsf{BB}_2}$	$\lvert\mathsf{D}\rvert$	$\lvert\mathsf{D}\rvert \cdot \mathsf{wcet}_{\mathsf{BB}_2}$	$\lvert\mathsf{D}\rvert \cdot 0$
$\mathsf{A}_{\mathsf{BB}_3}$	$\lvert\mathsf{A}\rvert$	$\lvert\mathsf{A}\rvert \cdot \mathsf{wcet}_{\mathsf{BB}_3}$	$\lvert\mathsf{A}\rvert \cdot \lvert\mathsf{BB}_3 \rightarrow \mathsf{BB}_2 \rightarrow \mathsf{BB}_3\rvert$
$\mathsf{B}_{\mathsf{BB}_3}$	$\lvert\mathsf{B}\rvert$	$\lvert\mathsf{B}\rvert \cdot \mathsf{wcet}_{\mathsf{BB}_3}$	$\lvert\mathsf{B}\rvert \cdot \lvert\mathsf{BB}_3 \rightarrow \mathsf{BB}_2 \rightarrow \mathsf{BB}_3\rvert$
$\mathsf{C}_{\mathsf{BB}_3}$	$\lvert\mathsf{C}\rvert$	$\lvert\mathsf{C}\rvert \cdot \mathsf{wcet}_{\mathsf{BB}_3}$	$\lvert\mathsf{C}\rvert \cdot \lvert\mathsf{BB}_3 \rightarrow \mathsf{BB}_2 \rightarrow \mathsf{BB}_3\rvert$
$\mathsf{D}_{\mathsf{BB}_3}$	$\lvert\mathsf{D}\rvert$	$\lvert\mathsf{D}\rvert \cdot \mathsf{wcet}_{\mathsf{BB}_3}$	$\lvert\mathsf{D}\rvert \cdot 0$
$\mathsf{D}_{\mathsf{BB}_4}$	$\lvert\mathsf{D}_{\mathsf{BB}_4}\rvert$	$\lvert\mathsf{D}_{\mathsf{BB}_4}\rvert \cdot \mathsf{wcet}_{\mathsf{BB}_4}$	$\lvert\mathsf{D}_{\mathsf{BB}_4}\rvert \cdot 0$

Table 5.1: Vulnerable instructions and path vulnerability for each LB in the example of Figure 5.1.

5.2.1.3 Example

In this example we consider task τ_i whose CFG is shown in Figure 5.1. To compute its vulnerability factor, we start by computing the size of the vulnerable instructions for each LB with Equation (5.2). The results are reported in the second column of Table 5.1.

The **baseline vulnerability** of a task is obtained by summing the LB baseline vulnerabilities with Equation (5.1). The results are reported in the third column of Table 5.1, and the baseline vulnerability of task τ_i is the sum of its elements.

Then, the **path vulnerability** is computed and the results are shown in the fourth column of Table 5.1. As you may notice, the value of $\nu^{\text{path}}_{\mathsf{D}_{\mathsf{BB}_4}}$ is 0 since its greatest vulnerable path is $\mathsf{BB}_2 \rightarrow \mathsf{BB}_4$ and BBs at both sides of the path are not considered in the execution time of the path.

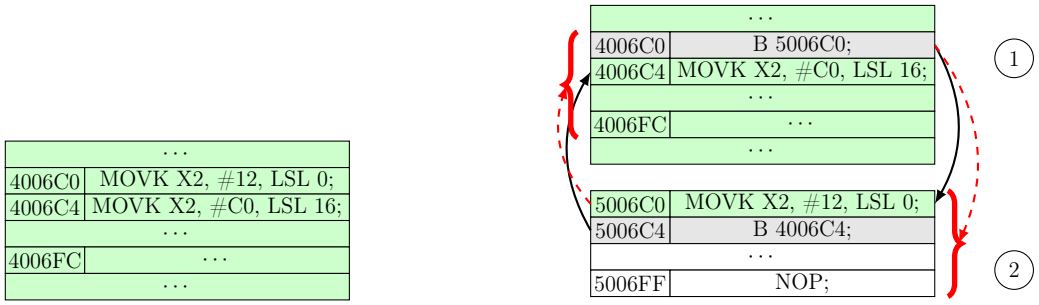
Finally, the vulnerability task factor of this example can be computed by summing all elements of the last two columns, and dividing them by the task's WCET multiplied by the size of the task.

5.2.2 CB invalidation

Without hardware support, we can invalidate cache lines by adding special sections of code that perform the invalidation before executing the target instructions. In this section, we present different methods to invalidate the cache depending on its architecture: direct mapped or set-associative.

To invalidate a cache line at a point p, we change the instruction at this point with a jump to an additional section of code that executes the replaced instruction after invalidating the cache line. These additional section of code will be invalidated immediately after their execution to avoid adding too much vulnerability to the task.

We now present some examples demonstrating how cache lines can be invalidated on processors based on the ARMv8 AArch64 ISA for direct mapped and set-associative caches. However, our method can be easily adapted to other architectures. In the considered architecture, instructions are coded on 32 bits, and a cache line contains 64 bytes, therefore a CB contains 16 instructions.



(a) Original task instructions. (b) Modifications to invalidate the cache line.

Figure 5.2: Invalidation mechanism for direct mapped cache memories.

Let us consider the code sample presented in Figure 5.2a. Here we have a CB going from address 4006C0 to 4006FC. In Figure 5.2b we present the code transformation to invalidate the CB in a direct-mapped cache of 16KB.

This mechanism can be viewed as a jump through a CB with the same cache set index that evicts the targeted CB and a return to this CB.

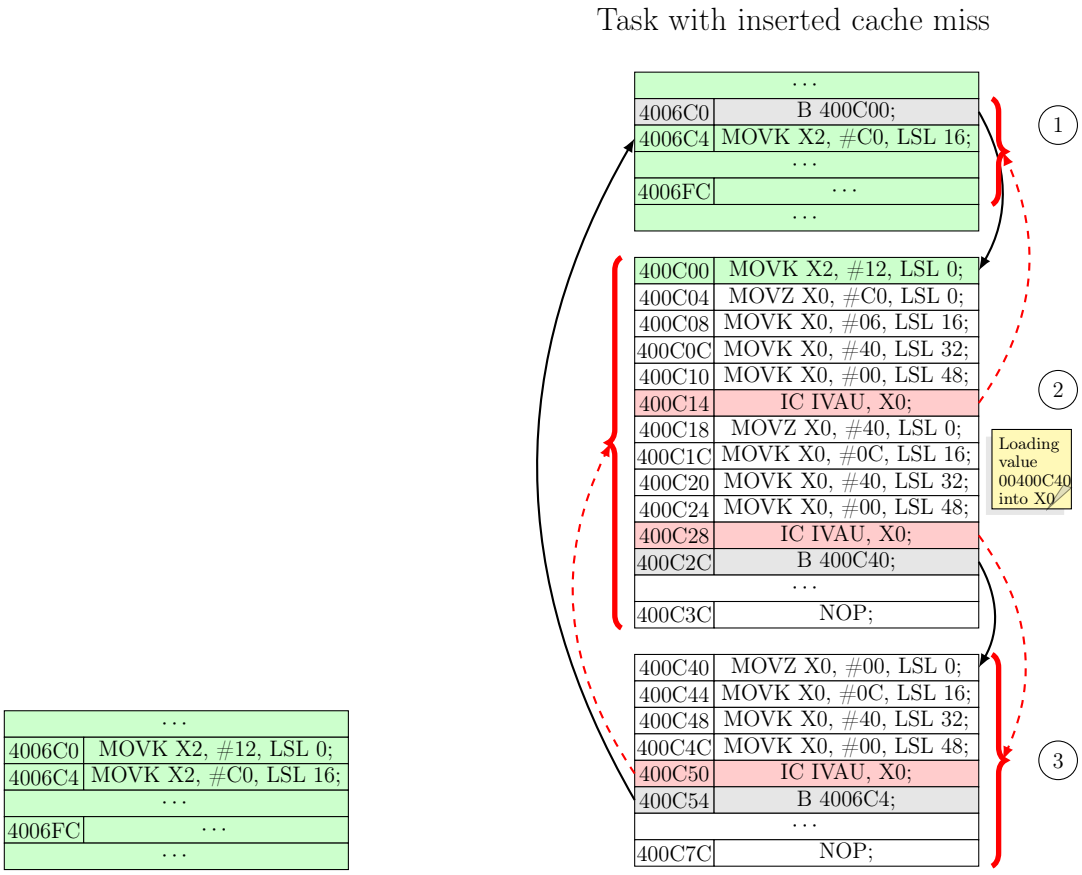
(1) presents the modified code of the task: the instruction at address 4006C0 is replaced by an unconditional branch instruction B 5006C0 which re-routes the execution flow to address 5006C0. As a consequence, instructions at addresses 5006C0 and 5006C4 are executed (see CB (2) in the figure). They are composed of the substituted instruction and the unconditional branch to the instruction at address 4006C4. Notice that this special CB (2) is placed at an address which shares the same cache set index as (1). Since this is a direct mapped cache, these CBs cannot be present at the same time in the cache memory. This mechanism adds two cache misses to the task (for recharging each block (1) and (2)) and only two jump instructions for each invalidation locations. We observe that in this approach the first jump instruction is still vulnerable.

However, this simple strategy cannot be used for set-associative caches, because multiple blocks with the same index can be present at the same time in the cache. Invalidating all the cache lines of a given cache set is not efficient.

Therefore, we designed a second and a more complex strategy for set-associative caches, which we describe in Figure 5.3b. Here, CB (1) represents the modified task instructions, while (2) and (3) are the newly inserted CBs in the code.

Again, we substitute the instruction at address 4006C0 in CB (1) with an unconditional branch instruction B 400C00 to CB (2). After jumping into (2), we execute the original instruction, then, we find the instructions to invalidate CB (1). Adress to be invalidated is loaded in register X0, and instruction IC IVAU, X0 is executed. This instruction is a special instruction that invalidates a CB in the instruction cache until the point of unification [9]. It is the point where instruction and data caches and translation tables are guaranteed to see the same copy of a memory location [10].

In the figure we highlight the invalidation mechanism with a red dashed arrow: the CB targeted by the arrow is invalidated by the IC IVAU instruction at the start of the arrow. Notice that, in this example, we assume that register X0 is reserved for the sole



(a) Original task instructions (b) Modified task code.

Figure 5.3: CB invalidation mechanism for set-associative cache memory.

purpose of cache invalidation. If X0 is used for another purpose, another register can be chosen, obviously.

At this point we have to remove the additional code from the cache memory to protect it from soft errors and continue the execution of the task. However, a jump instruction is needed to go back to the next task instruction. This jump must not be present in the cache memory too, otherwise it is also vulnerable. Therefore, we use a third block (3). We first invalidate it to ensure that it is not vulnerable. A second jump is performed to (3) where, after invalidating (2), we jump to the next instruction in the original location. As a result, all blocks are invalidated and 3 cache misses and 18 instructions to the task code are added.

We assume that, by configuring the compiler and the linker, all the additional CBs are reserved a single cache index, such that no other CB in the task uses the same cache index. In this way, the additional CBs will not interfere with the rest of the task execution. Notice that reserving more than one cache line for this additional code is useless since all the CBs put in this cache line will be invalidated just after being used.

The WCET of an *Inserted Cache Miss* (ICM) is defined by Equation (5.7)

$$C^{\text{ICM}} = \delta + \mathsf{BRT} \cdot \beta \tag{5.7}$$

where δ is the execution time of the code of the section and the jump instruction, and β corresponds to the number of CBs used by the additional section.

When a cache miss is added at LB lb the impact on the WCET of the task can be computed as follows:

$$C^{\text{impact}}_{\mathsf{lb}} = I_{\mathsf{lb}} \cdot (C^{\text{ICM}} + \mathsf{BRT}) \tag{5.8}$$

Let a list of LBs γ corresponding to the location of the inserted cache misses, the modified WCET of a task τ_i, C'_i can be computed as:

$$C'_i = C_i + \sum_{\forall \mathsf{lb} \in \gamma} C^{\text{impact}}_{\mathsf{lb}} \tag{5.9}$$

Even if the additional sections of code have a limited vulnerability, we still need to consider it with the following equation:

$$\nu^{\text{ICM}} = \sum_{\forall \mathsf{lb}' \in ICM} \mathsf{wcet}_{\mathsf{lb}'} \cdot |\mathsf{lb}'| \tag{5.10}$$

$$\nu^{\text{ICM}}_{\mathsf{lb}} = I_{\mathsf{lb}} \cdot \nu^{\text{ICM}} \tag{5.11}$$

where $\forall \mathsf{lb}' \in ICM$ iterates on the LBs of the inserted cache miss mechanism.

5.2.3 ICM and TAVF

Cache misses have an impact on the task vulnerability factor. Invalidating block $\mathsf{CB}_{\mathsf{lb}}$ at the start of lb reduces its vulnerability, but we increase the execution time of all paths that contain lb and therefore increase the vulnerability of other LBs that may be vulnerable along those paths. For this reason it is necessary to compute the overall vulnerability for a cache misses combination.

First, we notice that by invalidating a LB lb, d_{lb} will be equal instruction size, here the jump instruction.

$$d_{\mathsf{lb}} = \begin{cases} \sigma, & \text{if } \mathsf{lb} \text{ is invalidated} \\ |CB_{\mathsf{lb}}|, & \text{else if } \mathsf{lb} \in UCB^{\text{b-out}}_{\mathsf{lb}} \\ |\mathsf{lb}|, & \text{otherwise} \end{cases} \tag{5.12}$$

where σ is the size of the jump instruction (4 bytes in the previous example for ARMv8 architecture).

Second, the impact of the cache misses needs to be considered also along the vulnerable paths. For any LB lb, we consider all vulnerable paths ending on lb. Let Γ_{lb} be a list of

LBs different from lb that contain an invalidation. We compute the impact of the inserted cache misses on the path vulnerability of lb with the following equation:

$$\rho_{\mathsf{lb}} = \sum_{\forall m \in \Gamma_{\mathsf{lb}}} \begin{cases} C_m^{\text{impact}}, & \text{if } \exists p \in \mathcal{P}_{\mathsf{lb}} | m \in p \\ 0, & \text{otherwise} \end{cases} \tag{5.13}$$

where $\mathcal{P}_{\mathsf{lb}}$ is the list of vulnerable paths ending in lb.

Finally, Equation (5.4) is modified to account for the inserted cache misses:

$$\nu_{\mathsf{lb}}^{\text{path}} = \begin{cases} d_{\mathsf{lb}} \cdot (I_{\mathsf{lb}} \cdot \mathcal{P}_{\mathsf{lb}}^{\max} + \rho_{\mathsf{lb}}), & \text{if } CB_{\mathsf{lb}} \in UCB_{\mathsf{lb}}^{\text{b-in}} \\ 0, & \text{otherwise} \end{cases} \tag{5.14}$$

Example Consider again the example of Figure 5.1, and suppose we invalidate LB $\mathsf{A}_{\mathsf{BB}_3}$. The value of $d_{\mathsf{A}_{\mathsf{BB}_3}}$ is equal to σ. The vulnerability path of LBs $\mathsf{B}_{\mathsf{BB}_3}$, $\mathsf{C}_{\mathsf{BB}_3}$, $\mathsf{D}_{\mathsf{BB}_3}$ and $\mathsf{D}_{\mathsf{BB}_2}$ is changed according to Equation (5.14), since each of their vulnerable paths contain BB_3.

5.2.4 Transformation to a QP problem

We now show how the different cache misses combinations are explored to build the task profile.

As mentioned earlier, the number of ICM combinations is exponential in the number of potential cache misses locations. We propose to use Quadratic Programming (QP) to search a combination with the lowest vulnerability in a given interval while providing a WCET bounded by a given value C^{bound}. The idea is to run several instances of the QP problem, every time lowering the WCET bound, thus obtaining a pareto-front of vulnerability/WCET.

We denote as $\mathcal{X}$ a list of decision variables. Each element X_j of this list corresponds to the invalidation of LB lb_j, and is equal to 1 when lb_j is invalidated, and 0 if it is left unmodified. We denote as $\mathcal{V}$ a list of real values V_j, each one corresponds to the path vulnerability of a LB lb_j.

We first define the constraint on the WCET:

$$C_i + \sum_{\forall \mathsf{lb}_j \in \tau_i} C_{\mathsf{lb}_j}^{\text{impact}} \cdot X_j < C^{\text{bound}}. \tag{5.15}$$

Then, we build a constraint to compute V_j.

We start by computing the length of vulnerable path V_j^{path} according to the inserted cache misses with Equation (5.16).

$$V_j^{\text{path}} = I_{\mathsf{lb}_j} \cdot \mathcal{P}_{\mathsf{lb}_j}^{\max} + \sum_{\forall \mathsf{lb}_k \in p | \forall p \in \mathcal{P}_{\mathsf{lb}_j}} X_k \cdot C_{\mathsf{lb}_k}^{\text{impact}}. \tag{5.16}$$

Then, this length is used to compute the path vulnerability V_j:

$$V_j = V_j^{\text{path}} \cdot ((1 - X_j) \cdot d_{\mathsf{lb}_j} + X_j \cdot \sigma) + X_j \cdot \nu_{\mathsf{lb}_j}^{\text{ICM}} \tag{5.17}$$

The factor $X_j \cdot \sigma$ from Equation (5.17) corresponds to the case where a cache miss is inserted to LB lb and the factor $(1 - X_j) \cdot d_{\mathsf{lb}_j}$ corresponds to the case when no cache miss is inserted.

We now limit the values of the vulnerability with an upper bound. The vulnerability should never exceed the vulnerability $V^{\text{Task-max-vul}}$ of the unmodified program (Equation (5.18)).

$$\nu_i^{\text{bv}} + \sum_{\forall \mathsf{lb}_j \in \tau_i} V_j \leq V^{\text{Task-max-vul}} \tag{5.18}$$

Finally, the objective function is a minimization of the task vulnerability:

$$fct^{\text{obj}} = min\ \nu_i^{\text{bv}} + \sum_{\forall \mathsf{lb}_j \in \tau_i} V_j \tag{5.19}$$

We now present the Algorithm 2 that uses the QP problem to build the task profile.

Algorithm 2 Task profile builder

Require: a task τ_i
Ensure: a task profile $\mathcal{L}$
1: $\mathcal{L} \leftarrow \emptyset$
2: $V^{\text{Task-max-vul}} \leftarrow \nu_i^{\text{bv}} + \nu_i^{\text{path}}$
3: $C^{\text{bound}} \leftarrow \infty$
4: **while** *continue* **do**
5: $\quad \mathcal{S} \leftarrow QP(V^{\text{Task-max-vul}}, C^{\text{bound}})$
6: $\quad$ **if** $\mathcal{S} = \emptyset$ **then**
7: $\quad\quad$ break
8: $\quad$ **else**
9: $\quad\quad \mathcal{L} \leftarrow \mathcal{L} \cup \{(\mathcal{C}_i(\mathcal{S}), \frac{\mathcal{V}_i(\mathcal{S})}{C_i \cdot |\tau_i|}, \mathcal{S})\}$
10: $\quad\quad C^{\text{bound}} \leftarrow \mathcal{C}_i(\mathcal{S})$
return $\mathcal{L}$

It starts at Line 1 by the initialization of the combinations list $\mathcal{L}$ to an empty set since we have not yet computed any combination of cache misses. Then we set the vulnerability and WCET upper bounds respectively $V^{\text{Task-max-vul}}$ and C^{bound}. On Line 4 the algorithm builds a combination of inserted cache miss $\mathcal{S}$ with a QP generated as presented earlier in this section at Line 5. If no combination of inserted cache misses is found, we exit the loop at Line 7. Otherwise, at Line 8, we add to $\mathcal{L}$ the WCET and the vulnerability factor of the task considering cache misses combination $\mathcal{S}$ respectively $\mathcal{C}_i(\mathcal{S})$ and $\frac{\mathcal{V}_i(\mathcal{S})}{C_i \cdot |\tau_i|}$, and the combination $\mathcal{S}$ itself. The vulnerability and WCET bounds are also updated with the current combination values for the next iteration.

5.2.5 Using ECC SRAM memories

So far, we presented a methodology which reduces the vulnerability for COTS hardware architectures using cache invalidation. However, our static analysis method can be easily

extended to hardware that provides some protection mechanisms for cache memory. In this section we discuss the application of our method to hardware which features *Error Correcting Code* (ECC) memories, and we propose a minor modification in the ISA to exploit our methodology.

ECC protected memories can tolerate temporary faults by detecting and correcting soft errors. Each time the processor accesses one location in the memory, an algorithm is performed in the hardware circuitry that compares the stored code and the computed code. To do this, the ECC memory uses extra bits to code enough information to recover from faults. However, ECC memories need more die space compared to non-ECC memories to store the extra bits and the encoding and decoding algorithms, and need an additional delay to access the data, thus reducing the performance.

We can use our method to selectively enable the ECC correction mechanisms only on those memory instructions which have the highest vulnerability, as computed with our proposed methodology.

We suppose that the ARM ISA is modified to reserve a bit in the instruction encoding. This bit is denoted as *vulnerable mode*: when an instruction is fetched from the cache memory with its vulnerable mode bit equal to 1, the cache memory dedicated hardware performs the ECC decoding algorithm on the CB's instruction and produces the correct instruction even in the presence of a temporary fault. In addition, the other instructions of the CB are updated with their decoded versions. If the vulnerability mode bit is set to zero, the instruction is stored and later returned without ECC comparison. The ECC reduces the vulnerability to zero. However, access to instructions with the vulnerable mode bit set to 1 takes more time than the other instructions, therefore it increases the WCET of the task.

We observe that the mechanism described above has a much lower overhead than the cache invalidation technique proposed in the previous section, both in terms of added delay and in terms of additional code and vulnerability, however, it needs the support for special ISA and ECC cache memory.

To compute the task profile using this mechanism, we substitute Equation (5.8) with:

$$C_{\mathsf{lb}}^{\text{impact}} = I_{\mathsf{lb}} \cdot (C^{\text{ECC}}) \tag{5.20}$$

where C^{ECC} is the extra delay of the ECC mechanism expressed in cycles.

As the ECC mechanism does not use any additional instructions, thus removing also additional vulnerability sources, the vulnerability constraint modeled with Equation (5.17) is simplified as follows:

$$V_j = V_j^{\text{path}} \cdot (1 - X_j) \cdot d_{\mathsf{lb}_j} \tag{5.21}$$

5.3 Reducing Task Set Vulnerability Factor

Once the tasks' profiles have been computed, we can set up a QP problem to reduce the total *Task Set Vulnerability Factor* (TSVF) while ensuring the respect of the schedulability constraints.

For each task τ_i of task set τ we denote as $\mathcal{X}_i$ a list of decision variables. An element $X_{i,j} \in \mathcal{X}_i$ corresponds to the selection of the j^{th} configuration in the profile of τ_i: If the value of this variable is 1 then the j-th configuration is selected, otherwise it is not. As only one configuration must be chosen for each task profile, the first constraint of our QP problem is:

$$\forall \tau_i \in \mathcal{T}; \sum_{\forall j} X_{i,j} = 1 \tag{5.22}$$

Let $C_{i,j}$ and $V_{i,j}$ be respectively, the WCET of τ_i and its TAVF when the j^{th} configuration of the task profile is selected.

We assume that the tasks are scheduled according to the non-preemptive EDF policy. We use the following equations, proposed by Jeffay et al. [46], for the schedulability constraints:

$$\sum_{\forall \tau_i \in \mathcal{T}} \sum_{\forall j} \frac{C_{i,j} X_{i,j}}{T_i} \leq 1 \tag{5.23}$$

$$\forall \tau_i \in \mathcal{T}; i < 1; \forall L; T_1 < L < T_i; \tag{5.24}$$

$$(\sum_{\forall j} X_{i,j} \cdot C_{i,j}) + \sum_{k=1}^{i-1} \lfloor \frac{L-1}{T_k} \rfloor \cdot (\sum_{\forall g} X_{k,g} \cdot C_{k,g}) \leq L \tag{5.25}$$

The objective function of the QP is the minimization of the TSVF. It is corresponding to the sum of the TAVFs of the tasks multiplied by their initial utilization:

$$fct^{\text{obj}} = min \sum_{\forall \tau_i \in \tau} \sum_{\forall j} (X_{i,j} \cdot V_{i,j} \cdot \frac{C_I}{T_i}) \tag{5.26}$$

5.4 Evaluation

In this section, we present the experimental results obtained using the proposed method on a set of benchmarks. We first explain the settings of our experiments. Then, we present and discuss some representative task profiles obtained with our analysis. In the last two sub-sections, we discuss the performance obtained by the invalidation and ECC mechanisms.

5.4.1 Experimental setting

We consider a single ARM7 core architecture with a 16KB IL1 cache memory. Each cache line has a size of 64 bytes. Two IL1 configurations are explored: direct mapped and 2 way set-associative. We consider two different values for the BRT: 20 and 50 cycles. Theses values have been chosen because they correspond to a typical embedded micro-architecture. We assume that the tasks are scheduled non-preemptively according to the

np-EDF scheduling policy. Since the size of the task code of the benchmarks are relatively small, we observed that increasing the total IL1 size does not impact the WCET or the vulnerability factor.

In the experiments, we use tasks from two benchmarks: the Malärdalen benchmark suite [39] and TacleBench [35]. Details of these tasks are given in Table 5.2: the third column is the WCET and the last column is the initial TAVF of the unmodified tasks computed with the OTAWA tool [17][1].

Name	Benchmark	WCET (cycles)	TAVF
binarysearch	TacleBench	6164	0.112
bs	Malärdalen	1044	0.301
cnt	Malärdalen	32097	0.253
fibcall	Malärdalen	5348	0.268
insertsort	Malärdalen	21236	0.191
janne_complex	Malärdalen	4640	0.137
ludcmp	Malärdalen	64624	0.074
matmult	Malärdalen	1337090	0.15
minver	Malärdalen	46627	0.081
ndes	TacleBench	633538	0.054
ns	Malärdalen	128321	0.109
qurt	Malärdalen	24685	0.172
select	Malärdalen	78961	0.22
sqrt	Malärdalen	7656	0.25

Table 5.2: List of programs from the 2 benchmarks. In this table we consider a 16KB IL1 and a BRT of 20 cycles.

5.4.2 Task profiles for the cache invalidation method

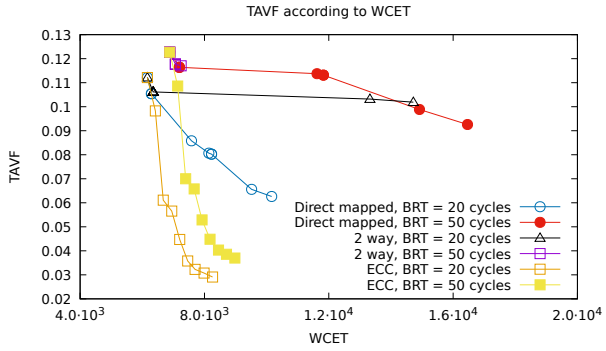
The impact of the inserted cache misses on the execution time and vulnerability used in the experiments are presented in Table 5.3.

For each task in the benchmarks listed in Table 5.2, we built a profile using the CPLEX tool. We set a limit of 3 minutes for solving each QP problem, thus obtaining a combination of cache miss locations. Figures 5.4a, 5.4b, and 5.4c show the resulting

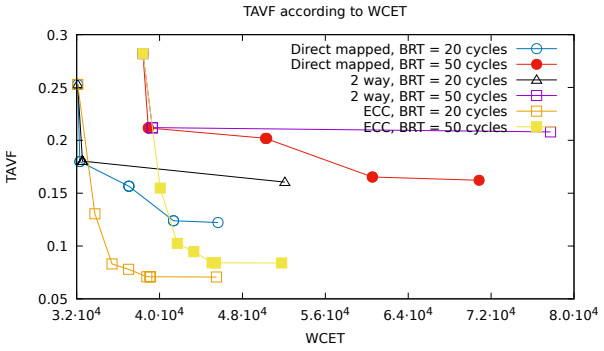
[1]The Vulnerability plugin for OTAWA can be found at the following address: https://gitlab.cristal.univ-lille.fr/otawa-plugins/plugin_cache_blocks.git

Cache configuration	β	δ	ν^{ICM}
Direct mapped	1	3	24
Set-associative	2	33	1320

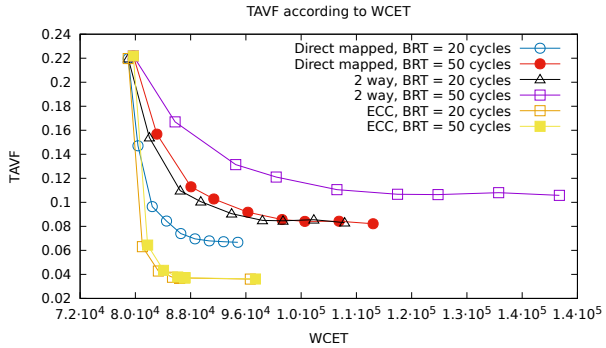
Table 5.3: Cache miss mechanism impact used in the experiments.



(a) Binarysearch



(b) Cnt



(c) Select

Figure 5.4: Case Study task profiles

task profiles respectively for the *binarysearch*, *cnt* and *select* benchmarks. Each of these figures presents six configurations. Each configuration is made up of a BRT value and one of the proposed protection mechanisms: the direct mapped invalidation mechanism, the set-associative invalidation mechanism, and the ECC mechanism. These figures shows the relation between the task WCET and its vulnerability factor for different combinations.

As we can see in these figures, the cache configuration has a strong impact on the task profile. Figure 5.4a shows that the profile of *binarysearch* with a direct mapped cache and a BRT of 20 cycles corresponds to a nearly linear relation between the WCET and the vulnerability factor. However, in the case of a set-associative cache with a BRT of 20 cycles, the vulnerability is more like a constant. Figure 5.4c shows also, that the vulnerability can not be approximated by a linear function of the WCET.

Notice that the executed tasks have different initial TAVF and the impact of the WCET is different from one task to another. For example, *binarysearch* with a BRT = 20 cycles (Figure 5.4a) has an initial TAVF of 0.011. This value is small compared to *cnt* with a direct mapped cache and a BRT = 20 cycles (Figure 5.4b) and an initial TAVF around 0.25. Furthermore, for *cnt* TAVF decreases to 0.12. This corresponds to a reduction of 50% of its initial TAVF. However, TAVF for *select* with a direct mapped cache and a BRT = 20 cycles decreases by 73% compared to its initial value.

5.4.3 TSVF reduction

In the previous section we have shown the relationship between vulnerability and WCET by using the invalidation methods of Section 5.2.2. To do this, we built individual profiles for each task. However, when a set of task is executed concurrently in a system, we have to consider the impact of the WCET on the schedulability of the system. If a task's WCET increases too much, one of the system's tasks can miss its deadline. In this section, we try to reduce the global vulnerability of the system without missing any deadline by using the optimization method of Section 5.3.

In the experiments, we assume that all the generated task set contains 12 tasks randomly selected from Table 5.2. To generate a task set, we proceed as follows: For each experiment, we first fix the initial system workload $U \in [0.1, 1]$ using steps of 0.05. Then, the utilization U_i of each task is randomly assigned with the *UUnifast* [23] algorithm, such that the sum of all tasks' utilization is equal to the total system workload U.

Then, each task is assigned a period from the list $\mathcal{G} = \{20, 50, 100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000, 100000, 200000, 500000\}$, such that

$$T_i = \underset{\forall t \in \mathcal{G}, t \geq (C_i/U_i)}{min}(t).$$

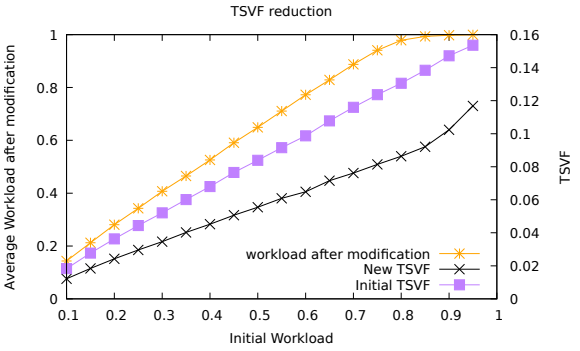
As some of the WCETs in Table 5.2 are larger than their periods, the WCETs values have been normalised, dividing them by the logarithm of the smallest WCET. For each total workload, we generate 1000 task sets.

For each task set, we use the corresponding task profiles to perform the optimization procedure described in Section 5.4.3, which selects the best combination of profile points (vulnerability/WCET) such that the system remains schedulable. If a solution is found, we report the reduction in vulnerability and the total workload after insertion of the cache misses. In the following Figures 5.5a, 5.5b, 5.5c and 5.5d we report the results on 2 y-axis: the y-axis on the left represents the average value, among the schedulable task sets, of the workload after inserting the cache misses. The right y-axis depicts the average TSVF value. On the x-axis we show the initial workload of the generated task set.

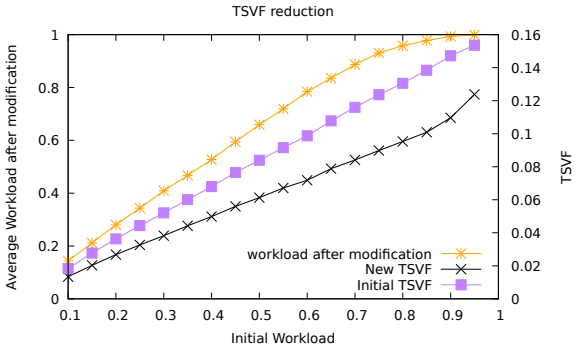
In these figures, *Initial vulnerability factor* and *New vulnerability factor* (respectively) corresponds to the TSVF before and after cache misses insertion. The *Workload after modification* represents the workload of the task set after inserting cache misses.

A first observation shows that the task set workload after inserting cache misses is not always equal to 1, and correspondingly, vulnerability is not reduced to 0. This means that, by using the invalidation method, we can only reduce the vulnerability to a certain limit, even when the system is largely underutilized. For example, for an initial workload of 0.3 in Figure 5.5a, the TSVF is reduced from 0.052 to 0.034 while the workload increases from 0.3 to 0.41.

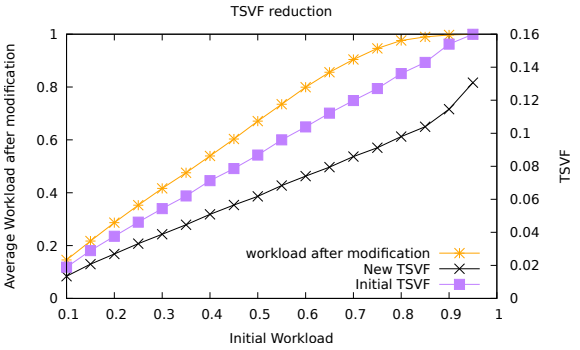
To have a better view of the evolution of the new vulnerability factor curves compared to the initial value, we give in Figure 5.6 the average task set vulnerability factor reduction in percentage of the initial value. Each curve represents a different cache configuration. Notice that the percentage reduction in the TSVF is almost constant until a certain point. It should be noted that task sets with an initial workload greater than 0.8 are very



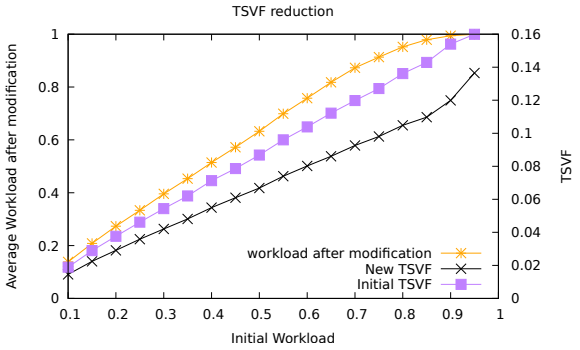
(a) Direct mapped, BRT = 20



(b) 2 way-set-associative, BRT = 20



(c) Direct mapped, BRT = 50



(d) 2 way-set-associative, BRT = 50

Figure 5.5: TSVF reduction after CB invalidation with a 16KB-IL1 cache

constrained: for this task sets, even a small increase in the WCET could deem the system unschedulable. Therefore, our optimization algorithm has a small impact on vulnerability.

From Figure 5.6, we observe that with direct mapped cache we obtain better performance than with set associative cache. This phenomenon is explained by the different additional costs of cache invalidation between these cache memories. This figure depicts also the strong impact of the BRT on the performance of our method. Again, this is intuitively due to the additional cost of every cache miss.

Summarizing, by using the invalidation technique, we can achieve on average between 22% and 34% reduction of the TSVF in task sets with initial workload inferior to 85%.

5.4.4 TSVF reduction with ECC

In this section, we evaluate the efficiency of the ECC mechanism discussed in Section 5.2.5.

Figures 5.7a, 5.7b, 5.7c and 5.7d depict the results obtained using the same configurations as those presented in Section 5.4.3. The ECC mechanism execution time C^{ECC} is set to 16 cycles in the experiments. This value is widely used in the literature [3] and corresponds to an average value for ECC execution time with different levels of complexity.

As expected, with the ECC mechanism we obtain a higher TSVF reduction compared

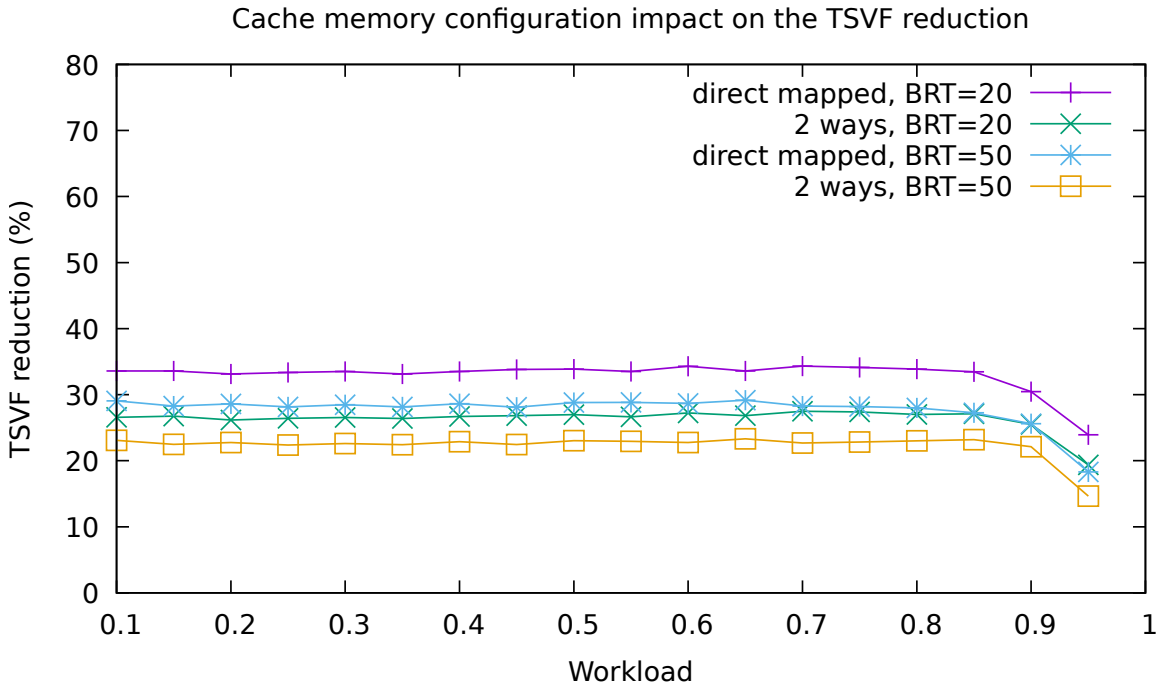


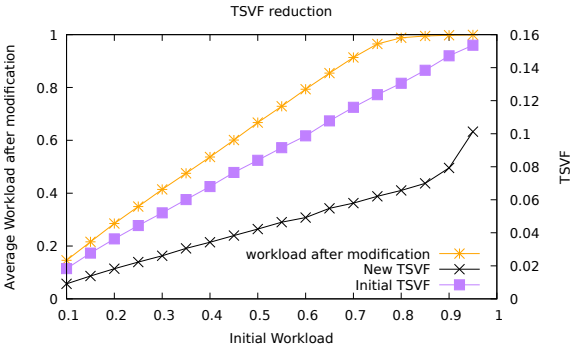
Figure 5.6: TSVF reduction with miss insertion. In these experiments, an average TSVF value is calculated on workloads of 1000 task sets.

to the CB invalidation mechanism. These results can be explained by two factors: first, the ECC mechanism has a lower impact on the WCET and second the lower additional vulnerability of the mechanism itself. Also, the efficiency of the ECC mechanism is almost constant regardless of the initial workload of the task set.

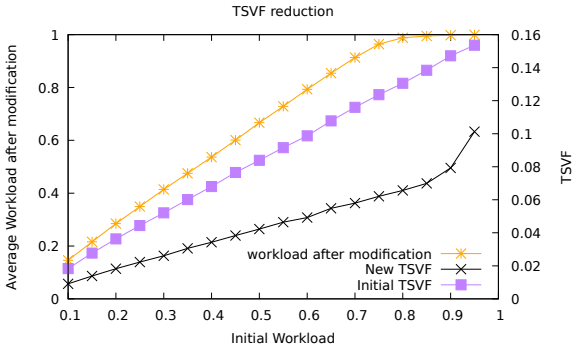
Similarly to Figure 5.6, Figure 5.8 shows the average percentage of task set vulnerability factor reduction for the different cache configurations according to the initial workload of the task set, this time using the ECC mechanism. This figure shows that the efficiency of our method with the ECC mechanism is the same regardless the cache memory configuration. Furthermore, it also shows that our method can reduce by 50% the TSVF for high utilization task sets around $U = 0.8$.

5.5 Conclusion

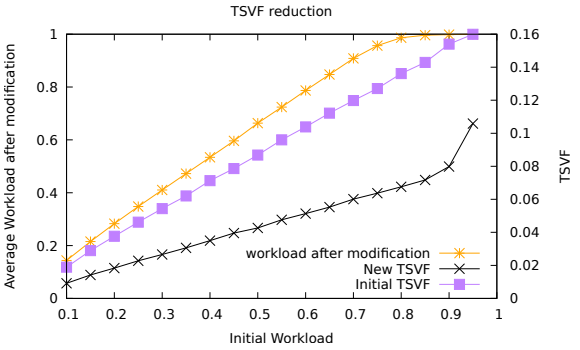
Hard-real time systems become more vulnerable to faults especially in the instruction cache memory of COTS microprocessor. We proposed in this Chapter a software based method that guarantees the real-time constraints while increasing the reliability of the task set. In particular, our method consists of two steps: in the first step, by using a static analysis, the vulnerability *profile* of each task is computed to find the best spots where to add a protection mechanism to reduce the vulnerability. This is done without significantly increasing the WCET. Then we proposed two alternatives: in the first one,



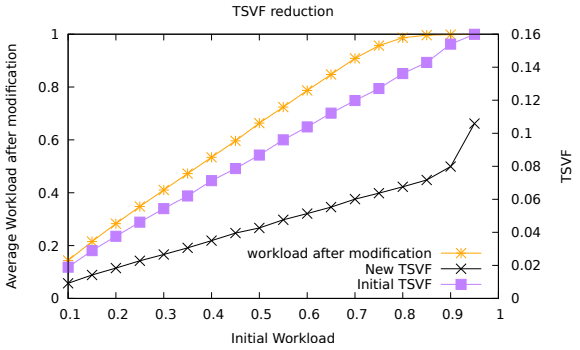
(a) Direct mapped, BRT = 20



(b) 2 way-set-associative, BRT = 20



(c) Direct mapped, BRT = 50



(d) 2 way-set-associative, BRT = 50

Figure 5.7: TSVF reduction with the ECC mechanism for a 16KB-IL1 cache. ECC execution time is set to 3 cycles.

we invalidate the cache in some of these spots using a software-only method; in the second one, we propose to modify the ISA of the processor to selectively enable ECC to specific instructions. A reduction of the task set vulnerability factor between 22% and 34% with CB invalidation and of 50% while using ECC has been obtained on real benchmarks.

As future work, we plan to adapt this method for schedulers that allow preemptions. We think that our method can be more performant on such systems as the vulnerable paths in the task may become larger due to preemptions.

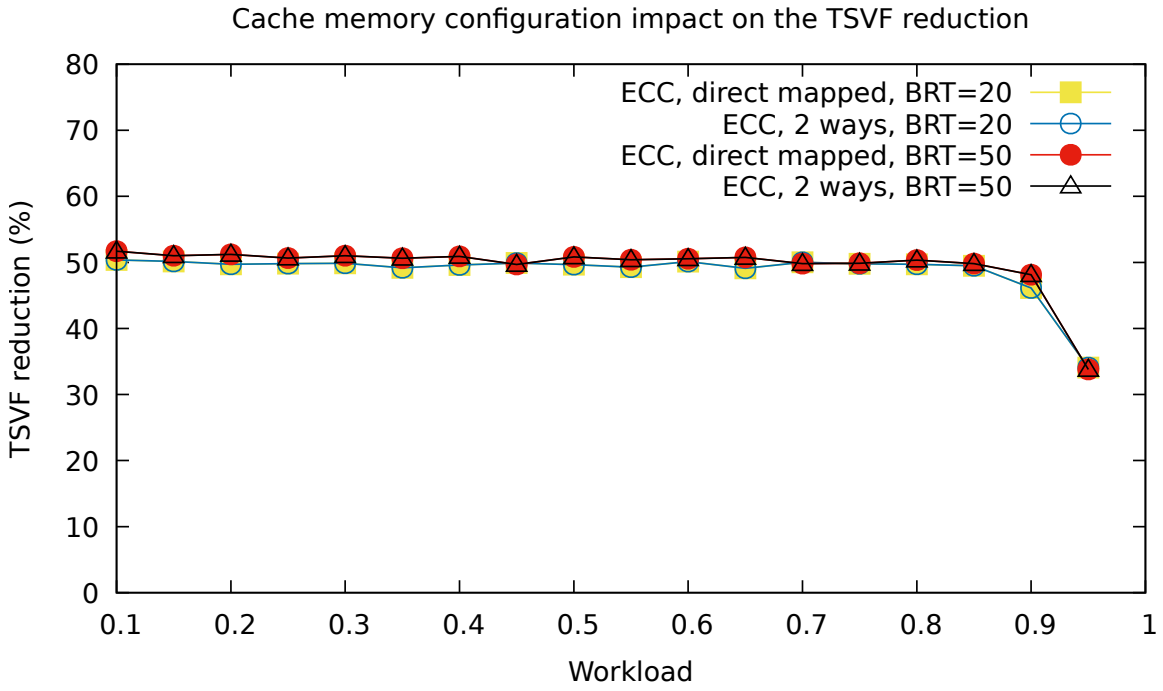


Figure 5.8: TSVF reduction with ECC mechanism

Conclusion

In this chapter, we summarize the contributions presented in this thesis, along with their adantages and their limitations as well as the various research perspectives.

5.6 Summary

In this thesis, I presented contributions related to cache analysis in two domains: real-time schedulability of preemptive systems and reliability of non-preemptive systems.

5.6.1 Contributions on Cache Related Preemption Delay

In Chapter 4:

- A less pessimistic method for computing the maximum number of preemptions between real-time periodic or sporadic tasks under EDF scheduling.
- An algorithm that allows a trade-off between precision and complexity when computing the Cache Related Preemption Delay under EDF scheduling. The algorithm realises a trade-off between two different types of analysis: the one that considers a single multiset of UCB per task and the one employing a multiset of UCB per each basic block. Our technique involves limiting the quantity of multisets for each task. Starting with the most complete model, which considers one UCB multiset for each BB of a task, our algorithm selects two UCB multisets to fusion so that the resulting multiset is the smallest of all possible multisets that could have resulted from a fusion. When the desired number of multisets has been obtained, the procedure ends.

These two contributions aim to improve the accuracy of schedulability analysis that takes into account the CRPD, we have demonstrated that these methods succeed in ensuring that task sets deemed non-schedulable by state-of-the-art analyses are, in fact, schedulable. As an example, we have demonstrated that 40% of the randomly generated task sets of 12 tasks, with a system utilization of 0.9 using a 2 ways cache memory of 4KB and a $\mathsf{BRT}= 50$ are considered schedulable by our method, whereas they are not by the state of the art methods.

In Chapter 4, we provided a counterexample to two schedulability analyses from the literature, Partitioning-ver1 and Partitioning-ver2. The counterexample demonstrates

that partitioning approaches to the enumeration of preemptions are not safe, in the sense that they are not able to correctly compute an upper bound to the number of preemptions in the general case. Hence, we need to resort to approximating the number of preemptions using simple techniques (as the one presented in this thesis) or enumerate all possible preemptions (which is exponential in the number of tasks in the general case).

Another known limitation of current CRPD analyses is that current models with one or more multisets per task do not permit to estimate an upper bound to the CRPD for caches using a different replacement policy than LRU. Further research is needed in this direction to improve the techniques and provide correct upper bounds for different replacement policies.

5.6.2 Contribution on Cache Reliability in Real-Time Embedded Systems

In Chapter 5, we presented contributions to improve cache reliability.

- The first contribution consists of a method that enables the selection of specific locations within the source code of a task to insert a protection mechanism, thereby reducing the task's vulnerability and ensuring that its time constraints are met. This strategy can be viewed as a compromise between the reliability of a system to operate despite the presence of transient faults and the reliability of a system with respect to real-time constraints.

- In addition, we have proposed two protection mechanisms, which are software-only and consist of inserting artificial cache misses to reload data from a less vulnerable memory. One of these two cache miss insertion mechanisms is only compatible with direct mapped caches, while the other is also compatible with associative set caches, but takes longer to execute.

- Also, we proposed a mechanism that uses ECCs to protect the system instead of inserted cache misses.

Through a series of experiments on real benchmarks, we have demonstrated that our contributions to increasing the system's reliability are effective. In particular, we observed a reduction of the task set vulnerability factor between 22% and 34% with cache line invalidations and 50% with ECCs.

The limitations are primarily hardware-based, and the platforms must provide protection features, as well as the ability to restrict their use if they interfere with the system's time constraints.

5.7 Perspectives

Currently, our contributions are intended for single-core systems, but the plan is to adapt these techniques for multicore partitioned systems. A possible first step could be adapting

our contributions to reduce the vulnerability of a system to preemptive systems using our contributions for schedulability analysis considering CRPD. The second step would be to consider multiple cache levels for the CRPD schedulability analysis. And finally, based on these two steps, propose a method for protecting multicore partitioned systems while taking time constraints into account.

Another important future direction is to research schedulability analyses for caches employing alternative replacement policies that take into account the CRPD.

Personal publications

- Fabien Bouquillon, Giuseppe Lipari, and Smail Niar. "Reducing the fault vulnerability of hard real-time systems". Journal of Systems Architecture, volume 133. 2022.
- Fabien Bouquillon, Giuseppe Lipari, and Smail Niar. "Improving Instruction Cache Memory Reliability under Real-Time constraints". The 2nd European Automotive Reliability, Test and Safety workshop. 2022.
- Fabien Bouquillon, Giuseppe Lipari, and Smail Niar. "Improving CRPD Analysis for EDF Scheduling: Trading Speed for Precision". The 37th ACM/SIGAPP Symposium On Applied Computing. 2022.

Appendix A

List of Symbols

Symbols are listed in order of their first appearance in the manuscript.

Symbol	Description	pages
N	Number of tasks in $\mathcal{T}$.	24
$\mathcal{T}$	Set of independent tasks.	24
τ_i	i^{th} task in $\mathcal{T}$.	24
$j_{i,k}$	k^{th} job of τ_i.	24
$a_{i,k}$	Arrival time of the k^{th} job of τ_i.	24
$d_{i,k}$	Absolute deadline of the k^{th} job of τ_i.	25
$c_{i,k}$	Execution time of the k^{th} job of τ_i.	25
C_i	WCET of task τ_i.	25
D_i	Relative deadline of τ_i.	25
T_i	Minimum inter-arrival time between two jobs of τ_i.	25
U_i	The utilization of τ_i.	25
BB_i	A Basic Block.	27
t	An instant of time.	29
P_i	Priority of τ_i.	29
U_{lub}	Maximum utilization for which we are certain the system can be scheduled with FP.	30
R_i	Response time of τ_i.	30
S_i	Worst start execution time of τ_i.	31
B_i	Blocking time of τ_i.	31
$lp(i)$	Set of tasks of lower preemption-level or priority than τ_i.	31
$hp(i)$	Set of tasks of higher preemption-level than τ_i.	31
π_i	Preemption-level of τ_i.	31
L	Upper bound on the first idle time.	31
$\mathsf{dbf}_i(t)$	Demand bound of τ_i on interval $[0, t]$.	31

$\eta(i,t)$	An upper bound to the number of instances of the sporadic task τ_i that have arrival and deadline in interval $[0,t]$.	31
$\mathsf{dbf}(t)$	Demand bound of the task set$\mathcal{T}$ on interval $[0,t]$.	31
K	The number of ways in the cache memory.	34
p	A preemption point.	43
$\mathcal{E}_i$	The multiset representing the ECBs of τ_i.	43
$\mathcal{U}_i^p$	The multiset representing the UCBs of τ_i at the preemption point p.	44
$\overline{\mathcal{U}}_i$	The set of multiset (SOM) representing the set of $\mathcal{U}$ of τ_i.	44
A	A multiset.	45
$\mathsf{rep}_{\mathsf{A}}(a)$	The number of instances of a in A.	45
$\lvert\mathsf{A}\rvert$	The size of multiset A.	45
$\mathsf{A} \uplus \mathsf{B}$	The multiset union between A and B.	45
$\mathsf{A} \bigtriangledown \mathsf{B}$	The multiset fusion between A and B.	45
$\mathsf{A} \sqcap \mathsf{B}$	The multiset intersection between A and B.	45
$\overline{\mathsf{A}}$	A SOM.	45
$\mathsf{MS}(\overline{\mathsf{A}})$	The largest size of any multiset in $\overline{\mathsf{A}}$.	45
$\mathrm{hp}(\mathcal{T})$	The hyperperiod of task set $\mathcal{T}$.	46
$\gamma(t)$	Upper bound on the total CRPD in interval $[0,t]$.	46
$(A)^x$	The result between the multiset union of x multiset A.	46
$\mathcal{U}_i$	The multiset representing the UCBs of τ_i for any preemption point.	46
$\mathbf{Pr}_j(D_i)$	The maximum number of preemptions by task τ_j on one instance of τ_i.	46
$\gamma_j^{\mathbf{ucbm}}(t)$	Upper bound computed with UCB-union multiset of CRPD provoked by τ_j on interval $[0,t]$ for EDF.	46
BRT	The block reload time.	46
$\gamma_j^{\mathbf{ecb}\boldsymbol{SOM}}(t)$	Upper bound computed with ECB-union $SOM$of CRPD provoked by τ_j on interval $[0,t]$ for EDF.	47
$dbf^{combinedSOM}(t)$	Demand bound function on interval $[0,t]$ with CRPD cost computed with the combined SOMapproach.	47
$dbf^{\mathbf{ucbm}}(t)$	Demand bound function on interval $[0,t]$ with CRPD cost computed with the UCB-union multiset approach.	47
$dbf^{\mathbf{ecb}\boldsymbol{SOM}}(t)$	Demand bound function on interval $[0,t]$ with CRPD cost computed with the ECB-union SOMapproach.	47
TVF_{cache}	Temporal Vulnerability Factor of the cache (TVF).	49

$data\_item_i$	A data item.	49
$vul\_inter_{i,j}$	The size of the j^{th} vulnerable interval of the i^{th} data item.	49
$E_j^h(t)$	The number of preemptions during interval $[0,t]$ for FP.	59
$\Lambda_{i,t}$	A multiset of preemption partitions considering only the i highest priority tasks.	59
λ_r	A preemptions partition.	59
$\gamma_i(\lambda_r)$	The CRPD cost of partition λ_r between the i highest priority tasks for FP.	60
$aff(i,h,\lambda_r)$	The set of tasks with priorities higher than or equal to P_i which can be preempted by τ_h according λ_r.	60
$hp(h,\lambda_r)$	The set of tasks with higher priorities than task τ_h that, according to λ_r, can preempt τ_h.	60
$\gamma(i,t)$	The CRPD cost between the i highest priority tasks on interval [0,t] for FP.	61
I_i	The size of the preemption interval of τ_i.	63
$\eta'_i(j,t)$	The number of preemptions by higher preemption-level task τ_j on task τ_i on any contiguous interval of size t inside $[0,D_i]$ (with $t \leq D_i$).	64
$\mathbf{Pr}'_j(D_i)$	The maximum number of preemption by task τ_j on τ_i considering the preemption interval of τ_i.	65
M_{reduce}	The maximum number of elements in a *SOM*.	65
ω	An upper bound to the maximum number of instances of any task contained in the hyperperiod $\text{hp}(\mathcal{T})$.	65
ψ	An upper bound of the maximum number of preemptions by any task on any job from another task.	66
Z	The number of cache sets.	66
λ	The number of potential locations in the code where to invalidate the cache for a set of tasks.	75
lb	A LB.	76
$\nu_{\mathsf{lb}}^{\text{bv}}$	The baseline vulnerability of lb.	76
$\mathsf{wcet}_{\mathsf{BB}}$	The WCET of BB.	76
$\mathsf{BB}_{\mathsf{lb}}$	The BB that contains the lb.	76
I_{lb}	The maximum number of executions of lb for one instance of a task.	76
d_{lb}	The size (in bytes) of the vulnerable instructions of the lb's CB during the execution of lb's BB.	76
$\mathsf{UCB}_{\mathsf{lb}}^{\text{b-out}}$	The list of UCB at the exit of $\mathsf{BB}_{\mathsf{lb}}$.	76
$\mathsf{UCB}_{\mathsf{lb}}^{\text{b-in}}$	The list of UCB at the entry of $\mathsf{BB}_{\mathsf{lb}}$.	76
$\mathsf{CB}_{\mathsf{lb}}$	The CB that contains lb.	76

$\lvert\mathsf{lb}\rvert$	The size in bytes of lb.	76
$\lvert\mathsf{CB}_{\mathsf{lb}}\rvert$	The size in bytes of $\mathsf{CB}_{\mathsf{lb}}$.	76
ν_i^{bv}	The baseline vulnerability of τ_i.	77
$\mathsf{UCB}_{\mathsf{BB}}^{\text{b-out}}$	The list of UCB at the exit of BB.	77
$\mathsf{UCB}_{\mathsf{BB}}^{\text{b-in}}$	The list of UCB at the entry of BB.	77
$\nu_{\mathsf{lb}}^{\text{path}}$	The path vulnerability of lb.	78
$\mathcal{P}_{\mathsf{lb}}^{\max}$	The WCET of the longest path from the last access to the $\mathsf{CB}_{\mathsf{lb}}$ until $\mathsf{BB}_{\mathsf{lb}}$. We consider only the paths where $\mathsf{CB}_{\mathsf{lb}}$ is in the cache along the entire path without being used (and it used at the end of the path).	78
ν_i^{path}	The path vulnerability of τ_i.	78
f_i^v	The task vulnerability factor (TAVF) of τ_i.	78
$\lvert\tau_i\rvert$	The size of the code of τ_i in bytes.	78
C^{ICM}	The WCET of an inserted cache miss (ICM).	82
$C_{\mathsf{lb}}^{\text{impact}}$	The impact of an ICM at lb on the WCET of the task.	82
δ	The execution time of the code of one instance of the protection mechanism.	82
β	The number of CBs used by one instance of the protection mechanism.	82
C_i'	WCET of the task considering the impact of the inserted cache misses.	82
ν^{ICM}	The vulnerability of an ICM.	82
$\nu_{\mathsf{lb}}^{\text{ICM}}$	The impact of an ICM at lb on the vulnerability of the task.	82
σ	The size of a jump instruction.	82
Γ_{lb}	The list of LBs different from lb that contain an invalidation.	82
ρ_{lb}	The impact of the inserted cache misses on the path vulnerability of lb.	83
$\mathcal{P}_{\mathsf{lb}}$	The list of paths from the last access to the $\mathsf{CB}_{\mathsf{lb}}$ until $\mathsf{BB}_{\mathsf{lb}}$. We consider only the paths where $\mathsf{CB}_{\mathsf{lb}}$ is in the cache along the entire path without being used (and it used at the end of the path).	83
C^{bound}	An upper bound to the WCET.	83
$V^{\text{Task-max-vul}}$	An upper bound to bound to the task vulnerability.	84
X_j	The decision variable representing the insertion of a protection mechanism for lb_j. This variable is used to compute a TAVF.	83

V_j^{path}	The variable representing the length of vulnerable path of lb_j. This variable is used to compute a TAVF.	83
V_j	The variable representing the path vulnerability of lb_j. This variable is used to compute a TAVF.	83
C^{ECC}	The extra delay of the ECC mechanism expressed in cycles.	85
$X_{i,j}$	The decision variable representing the selection of the j^{th} point in the profile of τ_i. This variable is used to compute a TSVF.	86
$C_{i,j}$	The WCET of τ_i when the j^{th} combination of the task profile of τ_i is selected.	86
$V_{i,j}$	The TAVF of τ_i when the j^{th} combination of the task profile of τ_i is selected.	86

Table A.1: List of symbols used in the thesis

Appendix B

List of Abbreviation of Terms

The following abbreviations are listed in alphabetic order.

Abbreviation	Description
ABS	Antilock Braking Systems
ADAS	Advanced Driver Assistance Systems
AEB	Autonomous Emergency Braking
AH	Always Hit
AM	Always Miss
BB	Basic Block
BIST	Built-in Self Test
CACC	Cooperative Adaptive Cruise Control
CAS	Collision Avoidance Systems
CB	Cache Block
CCI	Clean Cache line Invalidation
CFG	Control Flow Graph
COTS	Component-Off-The-Shelf
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CRPD	Cache Related Preemption Delay
DL1	Level 1 Data cache memory
DM	Deadline Monotonic
DMA	Direct Memory Access
DMR	Double-Modular Redundancy
DRAM	Dynamic Random Access Memory
ECB	Evicting Cache Block
ECC	Error Correcting Codes
EDF	Earliest Deadline First
FH	First Hit
FIFO	First In First Out
FM	First Miss

FP	Fixed Priority
GPU	Graphics Processing Unit
GPS	Global Positioning System
ICM	Inserted Cache Miss
IL1	Level 1 Instruction cache memory
ILP	Integer Linear Program
IPET	Implicit Path Enumeration Technique
ISA	Instruction Set Architecture
ITS	Intelligent Transportation Systems
L2	Level 2 cache memory
L3	Level 3 cache memory
LB	Line Block
LLF	Least Laxity First
LRU	Least Recently Used
MMU	Memory Management Unit
MIP	Mixed Integer Programming
NP	Non-Preemptive
PIPT	Physically Indexed, Physically Tagged
PLRU	Pseudo Least Recently Used
QP	Quadratic Programming
RAM	Random Access Memory
RM	Rate Monotonic
SICVF	System-level Instruction Cache Vulnerability Factor
SRAM	Static Random Access Memory
TAVF	Task Vulnerability Factor
TMR	Triple-Modular Redundancy
TSVF	Task Set Vulnerability Factor
TVF	Temporal Vulnerability Factor
U	Unknown
UCB	Useful Cache Block
VIVT	Virtually Indexed, Virtually Tagged
VIPT	Virtually Indexed, Physically Tagged
WCET	Worst Case Execution Time

Bibliography

[1] National Highway Traffic Safety Administration (NHTSA). *Automated vehicles for-safety.* URL: https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety.

[2] Liviu Agnola, Mircea Vlăduţiu, and Mihai Udrescu. "Self-Adaptive mechanism for cache memory reliability improvement". In: *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*. IEEE. 2010, pp. 117–118.

[3] Irina Alam et al. "Parity++: Lightweight Error Correction for Last Level Caches". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. 2018, pp. 114–120. DOI: 10.1109/DSN-W.2018.00048.

[4] Ihsen Alouani. "Conception de Systèmes Embarqués Fiables et Auto-réglables, Applications sur les Systèmes de Transport Ferroviaire". PhD thesis. Valenciennes, Université Polytechnique Hauts-de-France, 2016.

[5] S. Altmeyer and C. Burguière. "A New Notion of Useful Cache Block to Improve the Bounds of Cache-Related Preemption Delay". In: *2009 21st Euromicro Conference on Real-Time Systems (ECRTS)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2009, pp. 109–118. DOI: 10.1109/ECRTS.2009.21. URL: https://doi.ieeecomputersociety.org/10.1109/ECRTS.2009.21.

[6] Sebastian Altmeyer, Robert I Davis, and Claire Maiza. "Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems". In: *Real-Time Systems* 48.5 (2012), pp. 499–526.

[7] Sebastian Altmeyer, Claire Maiza, and Jan Reineke. "Resilience analysis: tightening the CRPD bound for set-associative caches". In: *ACM Sigplan Notices* 45.4 (2010), pp. 153–162.

[8] Sheng-hai An, Byung-Hyug Lee, and Dong-Ryeol Shin. "A Survey of Intelligent Transportation Systems". In: *2011 Third International Conference on Computational Intelligence, Communication Systems and Networks*. 2011, pp. 332–337. DOI: 10.1109/CICSyN.2011.76.

[9] arm. *IC IVAU, Instruction Cache line Invalidate by VA to PoU*. https://developer.arm.com/documentation/ddi0595/2021-06/AArch64-Instructions/IC-IVAU--Instruction-Cache-line-Invalidate-by-VA-to-PoU.

[10] arm. *Point of coherency and unification*. https://developer.arm.com/documentation/den0024/a/Caches/Point-of-coherency-and-unification.

[11] Neil Audsley et al. "Applying new scheduling theory to static priority pre-emptive scheduling". In: *Software engineering journal* 8.5 (1993), pp. 284–292.

[12] Verband der Automobilindustrie. "Automation: From driver assistance systems to automated driving". In: *VDA Magazine-Automation* (2015), p. 2.

[13] Algirdas Avizienis et al. "Basic concepts and taxonomy of dependable and secure computing". In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.

[14] Philip Axer et al. "Building Timing Predictable Embedded Systems". In: *ACM Trans. Embed. Comput. Syst.* 13.4 (Mar. 2014). ISSN: 1539-9087. DOI: 10.1145/2560033. URL: https://doi.org/10.1145/2560033.

[15] Zhenyu Bai et al. "A Framework for Calculating WCET Based on Execution Decision Diagrams". In: *ACM Trans. Embed. Comput. Syst.* (July 2021). Just Accepted. ISSN: 1539-9087. DOI: 10.1145/3476879. URL: https://doi.org/10.1145/3476879.

[16] T. Baker. "A stack-based resource allocation policy for realtime processes". In: *Proceedings 11th Real-Time Systems Symposium*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 1990, pp. 191, 192, 193, 194, 195, 196, 197, 198, 199, 200. DOI: 10.1109/REAL.1990.128747. URL: https://doi.ieeecomputersociety.org/10.1109/REAL.1990.128747.

[17] Clément Ballabriga et al. "OTAWA: An Open Toolbox for Adaptive WCET Analysis". In: *Software Technologies for Embedded and Ubiquitous Systems*. Ed. by Sang Lyul Min et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 35–46. ISBN: 978-3-642-16256-5.

[18] Claude Baron and Vincent Louis. "Towards a continuous certification of safety-critical avionics software". In: *Computers in Industry* 125 (2021), p. 103382. ISSN: 0166-3615. DOI: https://doi.org/10.1016/j.compind.2020.103382. URL: https://www.sciencedirect.com/science/article/pii/S0166361520306163.

[19] S. Baruah, A. Mok, and L. Rosier. "Preemptively scheduling hard-real-time sporadic tasks on one processor". In: *Proceedings 11th Real-Time Systems Symposium*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 1990, pp. 182, 183, 184, 185, 186, 187, 188, 189, 190. DOI: 10.1109/REAL.1990.128746. URL: https://doi.ieeecomputersociety.org/10.1109/REAL.1990.128746.

[20] Sanjoy K. Baruah. "Resource Sharing in EDF-Scheduled Systems: A Closer Look". In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. 2006, pp. 379–387. DOI: 10.1109/RTSS.2006.41.

[21] Anand Bhat, Soheil Samii, and Ragunathan Rajkumar. "Practical task allocation for software fault-tolerance and its implementation in embedded automotive systems". In: *Real-Time Systems* 55.4 (2019), pp. 889–924.

[22] E. Bini and G. C. Buttazzo. "Biasing Effects in Schedulability Measures". In: *2012 24th Euromicro Conference on Real-Time Systems*. Los Alamitos, CA, USA: IEEE Computer Society, July 2004, pp. 196–203. DOI: 10.1109/EMRTS.2004.1311021. URL: https://doi.ieeecomputersociety.org/10.1109/EMRTS.2004.1311021.

[23] Enrico Bini and Giorgio C Buttazzo. "Measuring the performance of schedulability tests". In: *Real-Time Systems* 30.1 (2005), pp. 129–154.

[24] Roman Bourgade et al. "Accurate analysis of memory latencies for WCET estimation". In: *16th International Conference on Real-Time and Network Systems (RTNS 2008)*. Ed. by Giorgio Buttazzo and Pascale Minet. Isabelle Puaut. Rennes, France, Oct. 2008. URL: https://hal.inria.fr/inria-00336530.

[25] R. J. Bril et al. "Integrating Cache-Related Pre-Emption Delays into Analysis of Fixed Priority Scheduling with Pre-Emption Thresholds". In: *2014 IEEE Real-Time Systems Symposium (RTSS)*. Los Alamitos, CA, USA: IEEE Computer Society, Dec. 2014, pp. 161–172. DOI: 10.1109/RTSS.2014.25. URL: https://doi.ieeecomputersociety.org/10.1109/RTSS.2014.25.

[26] Claire Burguière, Jan Reineke, and Sebastian Altmeyer. "Cache-Related Preemption Delay Computation for Set-Associative Caches - Pitfalls and Solutions". In: *9th International Workshop on Worst-Case Execution Time Analysis (WCET'09)*. Ed. by Niklas Holsti. Vol. 10. OpenAccess Series in Informatics (OASIcs). also published in print by Austrian Computer Society (OCG) with ISBN 978-3-85403-252-6. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009, pp. 1–11. ISBN: 978-3-939897-14-9. DOI: 10.4230/OASIcs.WCET.2009.2285. URL: http://drops.dagstuhl.de/opus/volltexte/2009/2285.

[27] Anastasiia Butko et al. "Full-System Simulation of big.LITTLE Multicore Architecture for Performance and Energy Exploration". In: *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*. 2016, pp. 201–208. DOI: 10.1109/MCSoC.2016.20.

[28] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Vol. 24. Springer Science & Business Media, 2011.

[29] Alexandre Chabot. "Exploration des architectures des systèmes embarqués dirigée par la fiabilité". PhD thesis. Valenciennes, Université Polytechnique Hauts-de-France, 2020.

[30] On-Road Automated Driving (ORAD) committee. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. Apr. 2021. DOI: https://doi.org/10.4271/J3016_202104. URL: https://doi.org/10.4271/J3016_202104.

[31] P. Czech, K. Turoń, and J. Barcik. "Autonomous vehicles: basic issues". In: *Zeszyty Naukowe. Transport / Politechnika Śląska* 100100 (2018), pp. 15–22.

[32] Robert I Davis, Sebastian Altmeyer, and Jan Reineke. "Response-time analysis for fixed-priority systems with a write-back cache". In: *Real-Time Systems* 54.4 (2018), pp. 912–963.

[33] Robert I Davis et al. "Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised". In: *Real-Time Systems* 35.3 (2007), pp. 239–272.

[34] George Dimitrakopoulos and Panagiotis Demestichas. "Intelligent Transportation Systems". In: *IEEE Vehicular Technology Magazine* 5.1 (2010), pp. 77–84. DOI: 10.1109/MVT.2009.935537.

[35] Heiko Falk et al. "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research". In: *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*. Ed. by Martin Schoeberl. Vol. 55. OpenAccess Series in Informatics (OASIcs). 2016, 2:1–2:10. ISBN: 978-3-95977-025-5. DOI: 10.4230/OASIcs.WCET.2016.2. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6895.

[36] DRIVER Fatigur. "Driver datigue and road accidents: a literature review and position paper". In: *Edgbaston: The Royal Society for the Prevention of Accidents* (2001).

[37] Mark S. Fineberg and Omri Serlin. "Multiprogramming for Hybrid Computation". In: *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference*. AFIPS '67 (Fall). Anaheim, California: Association for Computing Machinery, 1967, pp. 1–13. ISBN: 9781450378963. DOI: 10.1145/1465611.1465613. URL: https://doi.org/10.1145/1465611.1465613.

[38] V Dankan Gowda et al. "Modelling and performance evaluation of anti-lock braking system". In: *J. Eng. Sci. Technol* 14.5 (2019), pp. 3028–3045.

[39] Jan Gustafsson et al. "The Mälardalen WCET Benchmarks: Past, Present And Future". In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Vol. 15. OpenAccess Series in Informatics (OASIcs). 2010, pp. 136–146. ISBN: 978-3-939897-21-7. DOI: 10.4230/OASIcs.WCET.2010.136. URL: http://drops.dagstuhl.de/opus/volltexte/2010/2833.

[40] Umar Zakir Abdul Hamid et al. "Autonomous emergency braking system with potential field risk assessment for frontal collision mitigation". In: *2017 ieee conference on systems, process and control (icspc)*. IEEE. 2017, pp. 71–76.

[41] Sarah L. Harris and David Harris. "8 - Memory Systems". In: *Digital Design and Computer Architecture*. Ed. by Sarah L. Harris and David Harris. Morgan Kaufmann, 2022, pp. 498–541. ISBN: 978-0-12-820064-3. DOI: https://doi.org/10.1016/B978-0-12-820064-3.00008-8. URL: https://www.sciencedirect.com/science/article/pii/B9780128200643000088.

[42] C.A. Healy et al. "Bounding pipeline and instruction cache performance". In: *IEEE Transactions on Computers* 48.1 (1999), pp. 53–70. DOI: 10.1109/12.743411.

[43] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[44] Farrukh Hijaz, Qingchuan Shi, and Omer Khan. "A private level-1 cache architecture to exploit the latency and capacity tradeoffs in multicores operating at near-threshold voltages". In: *2013 IEEE 31st International Conference on Computer Design (ICCD)*. IEEE. 2013, pp. 85–92.

[45] Jeongkyu Hong and Soontae Kim. "Smart ECC Allocation Cache Utilizing Cache Data Space". In: *IEEE Transactions on Computers* 66.2 (2017), pp. 368–374. DOI: 10.1109/TC.2016.2595570.

[46] Kevin Jeffay, Donald F Stanat, and Charles U Martel. "On non-preemptive scheduling of periodic and sporadic tasks". In: *IEEE real-time systems symposium*. US: IEEE. 1991, pp. 129–139.

[47] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. "FERRARI: A flexible software-based fault and error injection system". In: *IEEE Transactions on computers* 44.2 (1995), pp. 248–260.

[48] H. Kim, A. Kandhalu, and R. Rajkumar. "A Coordinated Approach for Practical OS-Level Cache Management in Multi-core Real-Time Systems". In: *2013 25th Euromicro Conference on Real-Time Systems (ECRTS)*. Los Alamitos, CA, USA: IEEE Computer Society, July 2013, pp. 80–89. DOI: 10.1109/ECRTS.2013.19. URL: https://doi.ieeecomputersociety.org/10.1109/ECRTS.2013.19.

[49] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. "Real world automotive benchmarks for free". In: *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 2015.

[50] Swadhesh Kumar and PK Singh. "An overview of modern cache memory and performance analysis of replacement policies". In: *2016 IEEE International Conference on Engineering and Technology (ICETECH)*. IEEE. 2016, pp. 210–214.

[51] Chang-Gun Lee et al. "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling". In: *IEEE transactions on computers* 47.6 (1998), pp. 700–713.

[52] John D Lee, Kristie L Young, and Michael A Regan. "Defining driver distraction". In: *Driver distraction: Theory, effects, and mitigation* 13.4 (2008), pp. 31–40.

[53] Joseph Y.-T. Leung and Jennifer Whitehead. "On the complexity of fixed-priority scheduling of periodic, real-time tasks". In: *Performance Evaluation* 2.4 (1982), pp. 237–250. ISSN: 0166-5316. DOI: https://doi.org/10.1016/0166-5316(82)90024-4. URL: https://www.sciencedirect.com/science/article/pii/0166531682900244.

[54] Yau-Tsun Steven Li and Sharad Malik. "Performance analysis of embedded software using implicit path enumeration". In: *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*. 1995, pp. 88–98.

[55] Chung Laung Liu and James W Layland. "Scheduling algorithms for multiprogramming in a hard-real-time environment". In: *Journal of the ACM (JACM)* 20.1 (1973), pp. 46–61.

[56] Sarah M Loos, André Platzer, and Ligia Nistor. "Adaptive cruise control: Hybrid, distributed, and now formally verified". In: *International Symposium on Formal Methods*. Springer. 2011, pp. 42–56.

[57] W. Lunniss et al. "Integrating cache related pre-emption delay analysis into EDF scheduling". In: *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2013, pp. 75–84. DOI: 10.1109/RTAS.2013.6531081. URL: https://doi.ieeecomputersociety.org/10.1109/RTAS.2013.6531081.

[58] Will Lunniss, Sebastian Altmeyer, and Robert Davis. "A Comparison between Fixed Priority and EDF Scheduling accounting for Cache Related Pre-emption Delays". In: *Leibniz Transactions on Embedded Systems* 1.1 (2014), 01–1-01:24. ISSN: 2199-2002. DOI: 10.4230/LITES-v001-i001-a001. URL: https://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v001-i001-a001.

[59] Will Lunniss, Sebastian Altmeyer, and Robert I Davis. "Optimising task layout to increase schedulability via reduced cache related pre-emption delays". In: *Proceedings of the 20th International Conference on Real-Time and Network Systems*. ACM. 2012, pp. 161–170.

[60] 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)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2013, pp. 45–54. DOI: 10.1109/RTAS.2013.6531078. URL: https://doi.ieeecomputersociety.org/10.1109/RTAS.2013.6531078.

[61] Filip Marković et al. "Improving the Accuracy of Cache-Aware Response Time Analysis Using Preemption Partitioning". In: *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*. Ed. by Marcus Völp. Vol. 165. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 5:1–5:23. ISBN: 978-3-95977-152-8. DOI: 10.4230/LIPIcs.ECRTS.2020.5. URL: https://drops.dagstuhl.de/opus/volltexte/2020/12368.

[62] Michail Mavropoulos et al. "Use Them-Don't Waste Them. Recruiting Strong ECC in L1 Caches for Hard Error Recovery Without the Penalty". In: *11th European Dependable Computing Conference (EDCC 2015)*. 2015.

[63] Vicente Milanés et al. "Cooperative adaptive cruise control in real traffic situations". In: *IEEE Transactions on intelligent transportation systems* 15.1 (2013), pp. 296–305.

[64] Aloysius Ka-Lau Mok. "Fundamental design problems of distributed systems for the hard-real-time environment". PhD thesis. Massachusetts Institute of Technology, 1983.

[65] Amir Mukhtar, Likun Xia, and Tong Boon Tang. "Vehicle Detection Techniques for Collision Avoidance Systems: A Review". In: *IEEE Transactions on Intelligent Transportation Systems* 16.5 (2015), pp. 2318–2338. DOI: 10.1109/TITS.2015.2409109.

[66] Fadia Nemer et al. "PapaBench: a Free Real-Time Benchmark". In: *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*. Ed. by Frank Mueller. Vol. 4. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2006. ISBN: 978-3-939897-03-3. DOI: 10.4230/OASIcs.WCET.2006.678. URL: http://drops.dagstuhl.de/opus/volltexte/2006/678.

[67] Ludovic Pintard. "From safety analysis to experimental validation by fault injection-case of automotive embedded systems". PhD thesis. 2015.

[68] Krithi Ramamritham and John A Stankovic. *Hard Real-time Systems: Tutorial*. 1988.

[69] TA Ranney et al. "Driver distraction research: past, present and future". In: *17th International Technical Conference of Enhanced Safety of Vehicles, Amsterdam. Recuperado el*. Vol. 29. 07. 2000, p. 2010.

[70] Semeen Rehman. "Reliable software for unreliable hardware-a cross-layer approach". In: (2015).

[71] Jakob 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)*. 2007, pp. 49–60. DOI: 10.1109/RTSS.2007.24.

[72] sakharamgawade1. *Virtually Indexed Physically Tagged (VIPT) Cache*. https://www.geeksforgeeks.org/virtually-indexed-physically-tagged-vipt-cache/. Oct. 2021.

[73] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. "Priority inheritance protocols: An approach to real-time synchronization". In: *IEEE Transactions on computers* 39.9 (1990), pp. 1175–1185.

[74] Darshit Shah, Sebastian Hahn, and Jan Reineke. "Experimental Evaluation of Cache-Related Preemption Delay Aware Timing Analysis". In: *18th International Workshop on Worst-Case Execution Time Analysis (WCET 2018)*. Ed. by Florian Brandner. Vol. 63. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 7:1–7:11. ISBN: 978-3-95977-073-6. DOI: 10.4230/OASIcs.WCET.2018.7. URL: http://drops.dagstuhl.de/opus/volltexte/2018/9753.

[75] Taniya Siddiqua et al. "Lifetime memory reliability data from the field". In: *2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE. 2017, pp. 1–6.

[76] Elizabeth Skinner. *Modern Technology in the Automotive Industry*. https://industrytoday.com/modern-technology-in-the-automotive-industry/. Dec. 2019.

[77] Jiguo Song, John Wittrock, and Gabriel Parmer. "Predictable, efficient system-level fault tolerance in C^ 3". In: *2013 IEEE 34th Real-Time Systems Symposium*. IEEE. 2013, pp. 21–32.

[78] Jane C Stutts et al. "The role of driver distraction in traffic crashes". In: (2001).

[79] Makoto Sugihara, Tohru Ishihara, and Kazuaki Murakami. "Task scheduling for reliable cache architectures of multiprocessor systems". In: *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE. 2007, pp. 1–6.

[80] Yudong Tan and Vincent Mooney. "Timing analysis for preemptive multitasking real-time systems with caches". In: *ACM Transactions on Embedded Computing Systems (TECS)* 6.1 (2007), p. 7.

[81] Valentin Touzeau. "Static analysis of least recently used caches : complexity, optimal analysis, and applications to worst-case execution time and security". Theses. Université Grenoble Alpes, Oct. 2019. URL: https://tel.archives-ouvertes.fr/tel-02510350.

[82] Shuai Wang and Guangshan Duan. "On the characterization and optimization of system-level vulnerability for instruction caches in embedded processors". In: *Microprocessors and Microsystems* 39.8 (2015), pp. 686–692.

[83] Shuai Wang, Jie Hu, and Sotirios G Ziavras. "On the characterization and optimization of on-chip cache reliability against soft errors". In: *IEEE Transactions on Computers* 58.9 (2009), pp. 1171–1184.

[84] Bryan C Ward et al. "Making shared caches more predictable on multicore platforms". In: *2013 25th Euromicro Conference on Real-Time Systems*. IEEE. 2013, pp. 157–167.

[85] Reinhard Wilhelm et al. "The worst-case execution-time problem—overview of methods and survey of tools". In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008), pp. 1–53.

[86] Chris Wilkerson et al. "Trading off cache capacity for reliability to enable low voltage operation". In: *ACM SIGARCH computer architecture news* 36.3 (2008), pp. 203–214.

[87] Chao Yan and Russ Joseph. "Enabling deep voltage scaling in delay sensitive l1 caches". In: *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2016, pp. 192–202.