Reducing the fault vulnerability of hard real-time systems

Fabien Bouquillon, Smail Niar, Giuseppe Lipari

▶ **To cite this version:**

Fabien Bouquillon, Smail Niar, Giuseppe Lipari. Reducing the fault vulnerability of hard real-time systems. Journal of Systems Architecture, 2022, 133, pp.102758. 10.1016/j.sysarc.2022.102758 . hal-03842393

HAL Id: hal-03842393
https://hal.science/hal-03842393

Submitted on 7 Nov 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Reducing the Fault Vulnerability of Hard Real-Time Systems

Fabien Bouquillon [*], Smail Niar [†], Giuseppe Lipari [‡]

November 7, 2022

Abstract

With the progress of the technology, the presence of transient faults (e.g. bit-flipping errors) in cache memories becomes a challenge, especially in embedded real-time systems. These are mission critical systems that are often subject to both fault-tolerant and real-time constraints.

To reduce the impact of transient faults, hardware protection mechanisms are usually proposed. However, these mechanisms introduce too much pessimism in the computation of the worst-case execution time of a task, decreasing the overall system performance.

In this paper, we propose a methodology to evaluate and reduce the vulnerability of hard real-time applications to soft errors in IL1 cache memories.

We use static analysis tools to analyze a binary program and compute the overall vulnerability of its instructions. Then, we propose to reduce this vulnerability by invalidating some cache blocks at specific instants during the execution, thus forcing vulnerable instruction blocks to be reloaded from higher layers of memory. Since adding invalidation points will likely increase the WCETs of the tasks, we perform a static analysis to guarantee that the application deadlines are respected

Finally, we analyze how our methodology can be combined with hardware protection mechanisms as ECC memories, and we evaluate the performance on a set of benchmarks.

Keywords: Real-Time systems, Reliability, Cache memory, Transient Faults, Vulnerability, static code analysis, WCET analysis.

1 Introduction

The design and implementation of intelligent transportation systems and autonomous vehicles is a major trend in the embedded system community. These systems are considered as hard real-time, because they include critical subsystems such as ADAS, parking assistance, engine control, etc. In turn, these subsystems are composed of many concurrent software tasks that must produce correct results within predefined time windows. If a computed result is not correct or if it is produced too late, a critical failure may happen and cause serious accidents.

Hence, a precise analysis of the temporal behavior of critical software components is required to guarantee the correct functioning. This analysis is produced in two steps. First, each task is analyzed as if it were executed alone on the system to provide a *Worst*

[*]Univ. Lille, Univ. Polytechnique Hauts-de-France, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, UMR 8201 - LAMIH, F-59313 Valenciennes, France

[†]Univ. Polytechnique Hauts-de-France, CNRS, UMR 8201 - LAMIH, F-59313 Valenciennes, France

[‡]Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Case Execution Time (WCET). This step is often produced by a static analysis WCET estimation tool, such as OTAWA [1]. The second step consists in a *schedulability analysis* to guarantee that each task will finish its execution in the correct time window, assuming it will execute for its WCET and while accounting for the interference from the other tasks. As the number of functionalities increases, these systems need to provide high performance and high reliability while not being too expensive according the targeted market.

Many Commercial off-the-shelf (or COTS) microprocessors fit the cost requirement and are increasingly used in production. They present complex architectural features as superscalar pipelining, cache memory, shared bus, etc. that are essential for providing the required level of performance. However, these features are also a source of unpredictability and unreliability which must be taken into account for the design and implementation of critical software. In this paper we focus on increasing robustness in Level-1 (or L1) cache memories for microprocessor-based real time systems.

A well-known source of problems is the presence of transient faults on L1 cache memories. 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 [2].

Transient faults are temporary unpredictable faults due to environment factors, such as temperature and radiations. They corrupt 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.

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 reloading 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 instruction-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 [3] or by delays added by the protection mechanism to detect and correct faults. Hijaz et al. [4] 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 that higher layers of memory (L2, L3 or DRAM memory) are protected by fault-tolerant 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 [5].

1.1 Original contributions

In this paper, we propose a methodology to evaluate 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. We use static analysis tools to analyze a binary

program and compute the overall vulnerability of its instructions.

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

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.

This paper is organized as follows. First, in Section 2 we discuss briefly the existing research in the field of cache reliability. Then, in Section 3 we present the system model and our assumptions.

The main contributions of this paper are presented in Section 4 and Section 5. Then, we present a case study and experiments to show the improvements in reliability with our method compared to systems without any protection in Section 6. Finally, Section 7 gives a conclusion and presents new research avenues for future works.

2 Related Work

Faults in computing systems can be classified in two categories: *permanent faults* and *transient faults*.

In Siddiqua et al. [6] the authors realized several experiment 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 faults do 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 way entirely.

When a permanent fault occurs, Agnola et al. [7] propose a method for reorganizing memory location 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. [8] 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. [9] 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 cache 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 difference between these works and our is the type of considered faults. Indeed, our work focuses only on transient faults. Furthermore, they do not consider real-time guarantees.

Sugihara et al. [10] 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. [11] develop a *lifetime* model for L1 cache. They propose to classify interval 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 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 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. [11] propose the TVF (Temporal Vulnerability Factor), a reliability score for the cache: the highest is the score, the least reliable is the 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 L1 cache from the L2 cache. To reduce the overhead, they propose to insert cache misses during the idle cycles of cache memory. In our work, we propose to insert cache misses at strategic points during the tasks execution while considering timing constraints.

Later, the same authors focused on instruction cache memories for embedded systems [2]. A new metric has been proposed, the System-level Instruction Cache Vulnerability Factor (SICFV). 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 SICFV. In comparison, our solution does not require neither new instructions in the ISA nor code re-compilation.

Song et al. [12] 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,

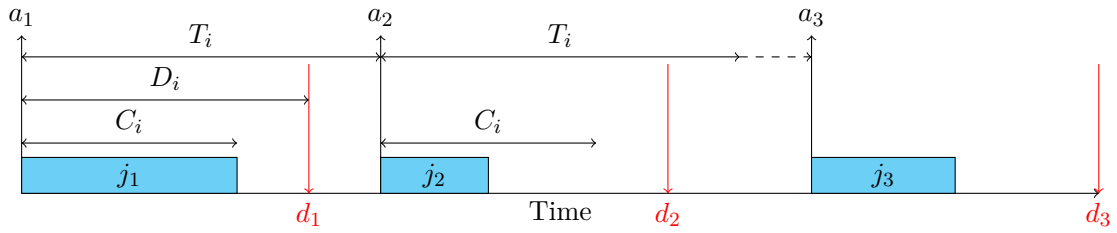


Figure 1: Task model

they consider timing constraints.

Hardware redundancy based methods such as DMR, TMR or CRC, are expensive solutions for embedded systems [13]. However, Bhat et al. observe a trend around software based approach to increase reliability. In their framework, the authors propose a novel task allocation heuristic to respect fault-tolerant requirement 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 misses insertion to improve reliability.

3 System model

3.1 Real-Time task model

In our work, a system consists of N independent real-time sporadic tasks, denoted as $\mathcal{T} = \{\tau_1, \cdots, \tau_n\}$.

A task τ_i is an infinite sequence of instances (also called *jobs*), and can be described by the tuple (C_i, D_i, T_i), with C_i the worst case execution time of its jobs, D_i the relative deadline, and T_i the minimum inter-arrival time between two of its jobs.

For simplicity, in this work we consider implicit deadline tasks, i.e. each task τ_i has $D_i = T_i$. However, our method is general, and it works also with constrained deadline tasks ($D_i \leq T_i$). The worst-case execution time C_i is estimated through a static analysis tool like OTAWA [1] assuming the task is executed alone on the processor. We consider sporadic tasks, i.e. the exact arrival times of the instances of the tasks are not known at analysis time; however, the minimum distance between two consecutive instances of the same task is T_i. An example is proposed in Figure 1. This figure represents 3 jobs of the task τ_i, denoted as j_1, j_2 and j_3, with their respective arrival time a_1, a_2 and a_3. In this example the minimum inter-arrival between all the jobs of task τ_i is between jobs j_1 and j_2, thus $T_i = a_2 - a_1$. Notice that the inter-arrival between two jobs might not be equal to T_i as for example the inter-arrival between jobs j_2 and j_3 which is greater than T_i. For each job, the box on the figure represents their execution time, as you may notice the execution time of job j_1 is equal in this example to the WCET of the task C_i contrary to the execution time of j_2. Also, all the jobs have the same relative deadline, we have for the job j_k: $d_k - a_k = D_i$, with d_k the absolute deadline of the job j_k.

We assume that the tasks are scheduled by Non-Preemptive Earliest Deadline First (NP-EDF) [14]. The job with the earliest deadline will be executed if the CPU is free.

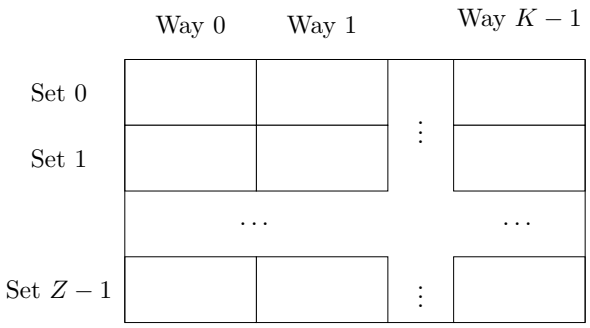


Figure 2: K-ways set-associative Cache Memory

3.2 Instruction Cache Structure

A cache memory can be modeled as a matrix of *cache lines*. Each row of the matrix is called a *cache set* and it may contain one or more cache lines. If the number of blocks within the sets is $K > 1$, the cache is said *K-way set-associative*. If $K = 1$, the cache is *direct mapped*. An example of a *K-way set-associative* cache is presented in Figure 2.

In this paper, we consider both set-associative and direct mapped caches. For simplicity, we restrict our analysis to instruction caches: we discuss the impact of dealing with data caches in Section 3.4. Also, in our model we consider that caches are flushed at the start of each instance of task execution.

Definition 1 (Cache block)**.** *The main memory is naturally divided into blocks of the size of a cache line, called cache blocks (CBs). When stored in the cache, every data of a CB is contained in the same cache line. As the binary of each task is present in the main memory, it can also be divided in CBs.*

A given CB can only be stored into one cache set. To know which set will contain the CB, a *cache set index* is computed based on its address.

In set-associative caches, a cache set may be composed of K cache lines and a *replacement policy* chooses the cache line where the block is stored. For example, the *Least Recently Used* replacement policy stores the CB in an unused cache line, or it replaces the least recently accessed cache line. In order to identify a cache block in a cache set, a *tag* is computed based on the block's address.

3.3 WCET Analysis and Cache Memory

The worst case execution time (WCET) analysis is a static analysis that computes an upper bound to the execution time of any instance of a task.

Definition 2 (Basic block)**.** *A basic block (BB) is a sequence of non-branching instructions terminated by a branch, a jump or a call.*

Definition 3 (Line block)**.** *A binary code can be split in BBs and in CBs. As a BB and a CB may not be of the same size, instructions belonging to the same BB can be present in multiple CBs. A line block (LB) is a maximal subset of a BB that is fully contained in a CB.*

Figure 3 shows the relation between CBs, BBs and LBs.

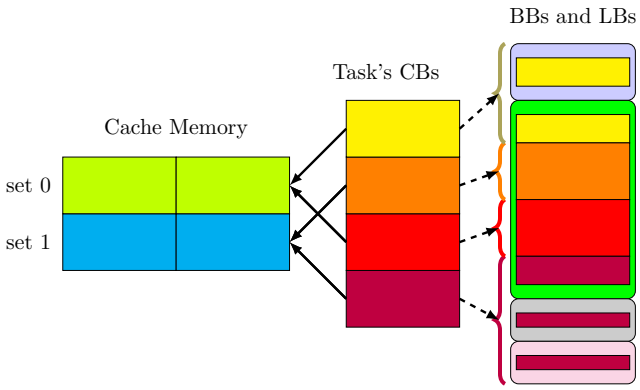


Figure 3: Example of BBs, CBs and LBs

The WCET analysis first transforms the binary code of the task into a *Control Flow Graph* (CFG). The graph is composed of nodes representing BBs. Branches, jumps or calls are represented as directed edges between nodes. An example of CFG is reported in Figure 4: it is composed of 4 BBs, $\mathbf{BB}_1$, $\mathbf{BB}_2$, $\mathbf{BB}_3$ and $\mathbf{BB}_4$. Each BB is composed of LBs, they are represented with a letter that corresponds to the CB they belong to.

Definition 4 (Vulnerability)**.** *Given a cache block 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.*

A *cache memory analysis* assigns a category to each line block. There exists 5 categories [15]: *first hit* (FH), *always hit* (AH), *first miss* (FM), *always miss* (AM) and *unknown* (U).

Since the *WCET estimation tool* must produce an estimation greater than or equal to the real WCET, the cache analysis must consider as cache hit only those accesses that can never be classified as cache misses in any possible scenario. For example, during WCET analysis, each LB classified as unknown will instead be considered as always miss.

In this paper 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. This other kind of cache analysis has been proposed by Lee et al. [16] and it is called a *may analysis*. To summarize, when computing the WCET we consider the analysis proposed by Healy et al. [15] and when computing the vulnerability, we instead use the analysis proposed by Lee et al. [16].

Definition 5 (Useful Cache Blocks)**.** *Given a point p between two BBs of a task, the set of* Useful Cache Blocks *(UCBs) represents all CBs that are present in the cache* and *may be reused sometimes later in the code.*

Notice that UCBs may be accessed much later than the moment when they are first loaded into the cache, therefore their vulnerability may be large. Our method consists in forcing the reloading of some of these blocks so to reduce their vulnerability.

3.4 Instruction cache and data cache

In this paper we only address the problem of reducing the vulnerability of instruction caches. Indeed, classifying CBs stored in data caches as CBs that can be hit later in the

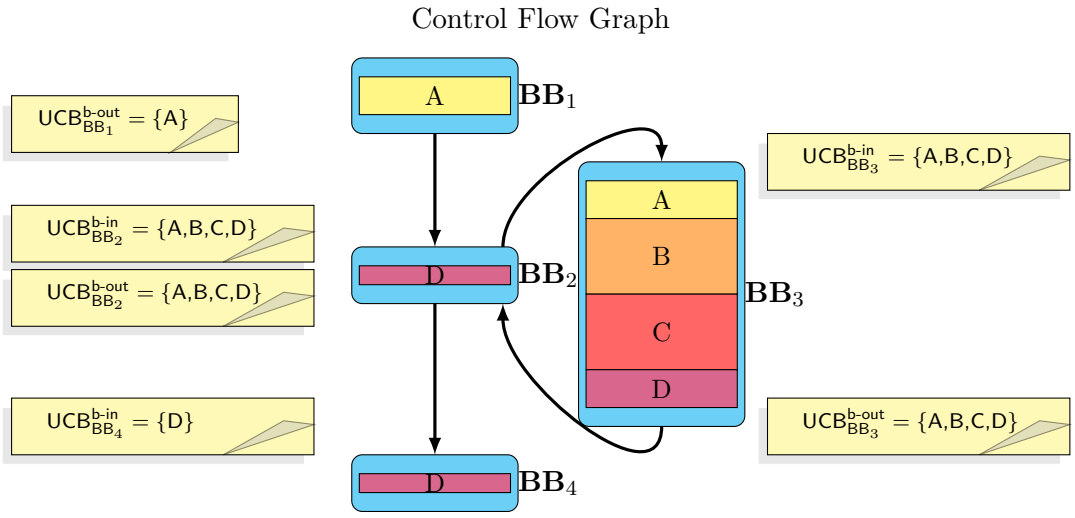


Figure 4: Example of task CFG

task execution requires a different kind of static analysis. For example, the authors of [17] propose a method to compute the useful data cache blocks of a task.

Also, data caches bring in additional issues, like the choice of the write policy (write-through vs. write-back) and its impact on timing and vulnerability. To limit the scope of this paper, we remand the analysis of data caches to a future research work.

3.5 Example

Figure 4 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 letter represent LBs belonging to the same CB. For example, the two rectangles with the letter A are two LBs belonging to the same CB A, and to two different BBs BB_1 and BB_3 (nodes of the tasks).

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 the BB_1, CB 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 $\mathsf{BB}_1 \rightarrow \mathsf{BB}_2 \rightarrow \mathsf{BB}_3$. Following the same reasoning, CBs A, B, C and D are both vulnerable along path $\mathsf{BB}_3 \rightarrow \mathsf{BB}_2 \rightarrow \mathsf{BB}_3$. On path $\mathsf{BB}_2 \rightarrow \mathsf{BB}_4$, neither A nor B and C need to be considered as vulnerable, because they are not accessed after BB_4.

4 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 combinations, is 2^λ which is exponential in the code size. Even the use of specialized solvers for exploring all possible combinations is too expensive.

This paper takes a different approach. 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* combination of artificial cache misses for each task, allowing the set of tasks to remain schedulable.

4.1 Computing the vulnerability factor of tasks

In this paper, 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. [11]. 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 [1], 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-case 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.

4.1.1 Computing the baseline vulnerability

Similarly to [11], 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 the time granularity of a BB, we consider that all instructions of a BB have the same vulnerability during their execution.

Equation (1) allows us to compute the baseline vulnerability for a LB lb.

$$\nu_{\mathsf{lb}}^{\text{bv}} = d_{\mathsf{lb}} \cdot I_{\mathsf{lb}} \cdot C_{\mathsf{BB}_{\mathsf{lb}}} \tag{1}$$

where d_{lb} corresponds to the size (in bytes) of the vulnerable instructions of the CB of lb during the execution of $\mathsf{BB}_{\mathsf{lb}}$; I_{lb} denotes the maximum number of executions of lb for

one instance of the task it belongs to; and $C_{\mathsf{BB}_{\mathsf{lb}}}$ the WCET of $\mathsf{BB}_{\mathsf{lb}}$ which represents the block that contains lb.

Equation (2) shows how d_{lb} is computed: $\mathsf{UCB}_{\mathsf{lb}}^{\text{b-out}}$ is the list of UCBs at the output of $\mathsf{BB}_{\mathsf{lb}}$; 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.

$$d_{\mathsf{lb}} = \begin{cases} |\mathsf{CB}_{\mathsf{lb}}|, & \text{if } \mathsf{CB}_{\mathsf{lb}} \in \mathsf{UCB}_{\mathsf{lb}}^{\text{b-out}} \\ |\mathsf{lb}|, & \text{otherwise} \end{cases} \tag{2}$$

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. For example, in Figure 4 the LB that belongs to CB D is vulnerable during the execution of BB_3. Furthermore, D is present in $\mathsf{UCB}_{\mathsf{BB}_3}^{\text{b-out}}$ as it can be reused later without eviction, hence 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:

$$\nu_i^{\text{bv}} = \sum_{\forall \mathsf{lb} \in \tau_i} \nu_{\mathsf{lb}}^{\text{bv}} \tag{3}$$

Theorem 4.1. *The baseline vulnerability of a task τ_i computed with Equation* (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 (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 (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 (3) is the sum of the baseline vulnerability for each LB of the task computed with Equation (1), we conclude that Equation (3) is an upper bound to the sum of the CBs vulnerability during their execution. □

4.1.2 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 are vulnerable if they belong to the UCB set at the exit of BB. However, their vulnerability is not considered by Equation (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 4 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 $\mathsf{UCB}^{\text{b-out}}_{\mathsf{BB}_1}$, $\mathsf{UCB}^{\text{b-in}}_{\mathsf{BB}_2}$, $\mathsf{UCB}^{\text{b-out}}_{\mathsf{BB}_2}$ and $\mathsf{UCB}^{\text{b-in}}_{\mathsf{BB}_3}$. 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. Therefore, the path vulnerability accounts just for the vulnerability of A during the execution of BB_2.

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

$$\nu^{\text{path}}_{\mathsf{lb}} = \begin{cases} d_{\mathsf{lb}} \cdot I_{\mathsf{lb}} \cdot \mathcal{P}^{\max}_{\mathsf{lb}}, & \text{if } \mathsf{CB}_{\mathsf{lb}} \in \mathsf{UCB}^{\text{b-in}}_{\mathsf{lb}} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

where $\mathcal{P}^{\max}_{\mathsf{lb}}$ is 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).

Similarly to the baseline vulnerability, the path vulnerability for a task τ_i can be computed as:

$$\nu^{\text{path}}_i = \sum_{\forall \mathsf{lb} \in \tau_i} \nu^{\text{path}}_{\mathsf{lb}} \tag{5}$$

Theorem 4.2. *Equation* (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 $\mathcal{P}^{\max}_{\mathsf{lb}'}$, 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 (4). Indeed, this equation consists in multiplying the length of the maximum path from $\mathcal{P}^{\max}_{\mathsf{lb}}$ 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 (4). Indeed, as the BB of lb' is inside a path through a future execution of lb, $\mathsf{CB}_{\mathsf{lb}} \in \mathsf{UCB}^{\text{b-out}}_{\mathsf{lb}'}$. In this case, the equation consists in multiplying the length of the longest path from $\mathcal{P}^{\max}_{\mathsf{lb}'}$ 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 $\sum_{\forall \mathsf{lb}' \in \mathsf{CB}_{\mathsf{lb}}} \nu^{\text{path}}_{\mathsf{lb}'}$.

As Equation (5) is the sum of ν^{path}_{lb} 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:

$$f^v_i = \frac{\nu^{\text{bv}}_i + \nu^{\text{path}}_i}{C_i \cdot |\tau_i|} \tag{6}$$

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

4.1.3 Example

In this example we consider task τ_i whose CFG is shown in Figure 4. To compute its vulnerability factor, we start by computing the size of the vulnerable instructions for each LB with Equation (2). The results are reported in the second column of Table 1.

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

Table 1: Vulnerable instructions and path vulnerability for each LB in the example of Figure 4.

The **baseline vulnerability** of a task is obtained by summing the LB baseline vulnerabilities with Equation (1). The results are reported in the third column of Table 1, and the baseline vulnerability of the 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 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.

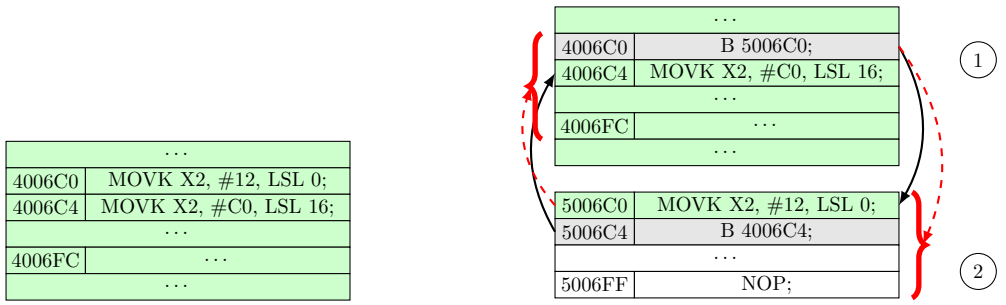
4.2 CB invalidation

Without hardware support, we can invalidate cache lines by adding special sections of code that perform the invalidation before executing the target instructions. This can be done by directly modifying the binary of the task. In this section, we present different methods to invalidate the cache depending on its architecture: direct mapped or set-associative. They are based on the technique presented in [18]. The main difficulty here is the required knowledge on the structure of the binary.

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.

Let us consider the code sample presented in Figure 6a. Here we have a CB going from address 4006C0 to 4006FC. In Figure 5b we present the code transformation to invalidate the CB in a direct-mapped cache of 16 KB.



(a) Original task instructions.

(b) Modifications to invalidate the cache line.

Figure 5: Invalidation mechanism for direct mapped cache memories.

(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 two CBs cannot be present at the same time in the cache.

The proposed scheme 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.

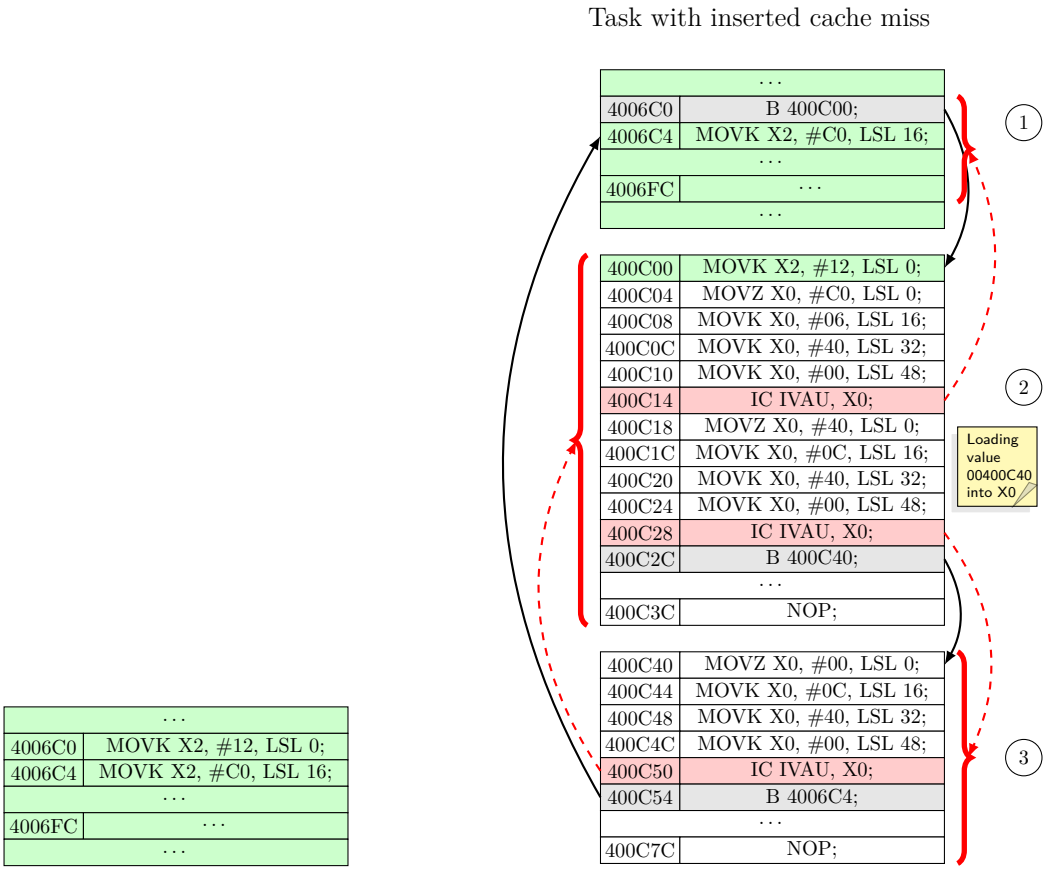
Therefore, we designed a second and a more complex strategy for set-associative caches, which we describe in Figure 6b. Here, CB (1) represents the modified task instructions, while (2) and (3) are the newly inserted CBs in the code.

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). Address 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 [19]. It is the point where instruction and data caches and translation tables are guaranteed to see the same copy of a memory location [20].

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 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.



(a) Original task instructions

(b) Modified task code.

Figure 6: CB invalidation mechanism for set-associative cache memory.

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 (7)

$$C^{\mathrm{ICM}} = \delta + \mathsf{BRT} \cdot \beta \tag{7}$$

where δ is the execution time of the code of the section and the jump instruction, BRT the block reload time 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:

$$C^{\mathrm{impact}}_{\mathsf{lb}} = I_{\mathsf{lb}} \cdot (C^{\mathrm{ICM}} + \mathsf{BRT}) \tag{8}$$

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:

$$C'_i = C_i + \sum_{\forall \mathsf{lb} \in \gamma} C^{\mathrm{impact}}_{\mathsf{lb}} \tag{9}$$

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

$$\nu^{\mathrm{ICM}} = \sum_{\forall \mathsf{lb}' \in ICM} C_{\mathsf{lb}'} \cdot |\mathsf{lb}'| \tag{10}$$

$$\nu^{\mathrm{ICM}}_{\mathsf{lb}} = I_{\mathsf{lb}} \cdot \nu^{\mathrm{ICM}} \tag{11}$$

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

4.3 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 it increases the execution time of all paths that contain lb and therefore it increases 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 to the instruction size, here the jump instruction.

$$d_{\mathsf{lb}} = \begin{cases} \sigma, & \text{if } \mathsf{lb} \text{ is invalidated} \\ |CB_{\mathsf{lb}}|, & \text{else if } \mathsf{lb} \in UCB^{\text{b-out}}_{\mathsf{lb}} \\ |\mathsf{lb}|, & \text{otherwise} \end{cases} \tag{12}$$

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:

$$\rho_{\mathsf{lb}} = \sum_{\forall m \in \Gamma_{\mathsf{lb}}} \begin{cases} C_m^{\text{impact}}, & \text{if } \exists p \in \mathcal{P}_{\mathsf{lb}} | m \in p \\ 0, & \text{otherwise} \end{cases} \tag{13}$$

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

Finally, Equation (4) is modified to account for the inserted cache misses:

$$\nu_{\mathsf{lb}}^{\text{path}} = \begin{cases} d_{\mathsf{lb}} \cdot (I_{\mathsf{lb}} \cdot \mathcal{P}_{\mathsf{lb}}^{\max} + \rho_{\mathsf{lb}}), & \text{if } CB_{\mathsf{lb}} \in UCB_{\mathsf{lb}}^{\text{b-in}} \\ 0, & \text{otherwise} \end{cases} \tag{14}$$

Example Consider again the example of Figure 4, and suppose we invalidate CB A at 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 (14), since each of their vulnerable paths contain BB_3. The new path vulnerability values are shown in Table 2.

LB	path vulnerability
$\mathsf{A}_{\mathsf{BB}_1}$	0
$\mathsf{D}_{\mathsf{BB}_2}$	$\lvert\mathsf{D}\rvert \cdot (I_{\mathsf{D}_{\mathsf{BB}_2}} \cdot 0 + \rho_{\mathsf{A}_{\mathsf{BB}_3}})$
$\mathsf{A}_{\mathsf{BB}_3}$	$\sigma \cdot (I_{\mathsf{A}_{\mathsf{BB}_3}} \cdot \lvert\mathsf{BB}_3 \rightarrow \mathsf{BB}_2 \rightarrow \mathsf{BB}_3\rvert)$
$\mathsf{B}_{\mathsf{BB}_3}$	$\lvert\mathsf{B}\rvert \cdot (I_{\mathsf{B}_{\mathsf{BB}_3}} \cdot \lvert\mathsf{BB}_3 \rightarrow \mathsf{BB}_2 \rightarrow \mathsf{BB}_3\rvert + \rho_{\mathsf{A}_{\mathsf{BB}_3}})$
$\mathsf{C}_{\mathsf{BB}_3}$	$\lvert\mathsf{C}\rvert \cdot (I_{\mathsf{C}_{\mathsf{BB}_3}} \cdot \lvert\mathsf{BB}_3 \rightarrow \mathsf{BB}_2 \rightarrow \mathsf{BB}_3\rvert + \rho_{\mathsf{A}_{\mathsf{BB}_3}})$
$\mathsf{D}_{\mathsf{BB}_3}$	$\lvert\mathsf{D}\rvert \cdot (I_{\mathsf{D}_{\mathsf{BB}_3}} \cdot 0 + \rho_{\mathsf{A}_{\mathsf{BB}_3}})$
$\mathsf{D}_{\mathsf{BB}_4}$	$\lvert\mathsf{D}_{\mathsf{BB}_4}\rvert \cdot (I_{\mathsf{D}_{\mathsf{BB}_4}} \cdot 0)$

Table 2: Path vulnerability for each LB in the example of Figure 4 after adding an ICM to a at LB $\mathsf{A}_{\mathsf{BB}_3}$.

4.4 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 miss 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 C^{bound}, thus obtaining a pareto-front of vulnerability/WCET.

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

$$C_i + \sum_{\forall \mathsf{lb}_j \in \tau_i} C_{\mathsf{lb}_j}^{\text{impact}} \cdot X_j < C^{\text{bound}}. \tag{15}$$

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 (16).

$$V_j^{\text{path}} = I_{\mathsf{lb}_j} \cdot \mathcal{P}_{\mathsf{lb}_j}^{\max} + \sum_{\forall \mathsf{lb}_k \in p | \forall p \in \mathcal{P}_{\mathsf{lb}_j}} X_k \cdot C_{\mathsf{lb}_k}^{\text{impact}}. \tag{16}$$

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

$$V_j = V_j^{\text{path}} \cdot ((1 - X_j) \cdot d_{\mathsf{lb}_j} + X_j \cdot \sigma) + X_j \cdot \nu_{\mathsf{lb}_j}^{\text{ICM}} \tag{17}$$

Factor $X_j \cdot \sigma$ from Equation (17) corresponds to the case where a cache miss is inserted into lb and the factor $(1 - X_j) \cdot d_{\mathsf{lb}_j}$ corresponds to the case when no cache miss is inserted.

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

$$fct^{\text{obj}} = min\ \nu_i^{\text{bv}} + \sum_{\forall \mathsf{lb}_j \in \tau_i} V_j \tag{18}$$

We now present the Algorithm 1 that uses the QP problem to build the task profile.

Algorithm 1 Task profile builder

Require: a task τ_i
Ensure: a task profile $\mathcal{L}$
1: $\mathcal{L} \leftarrow \emptyset$
2: $C^{\text{bound}} \leftarrow \infty$
3: **while** *continue* **do**
4: $\quad \mathcal{S} \leftarrow QP(C^{\text{bound}})$
5: $\quad$ **if** $\mathcal{S} = \emptyset$ **then**
6: $\quad\quad$ break
7: $\quad$ **else**
8: $\quad\quad \mathcal{L} \leftarrow \mathcal{L} \cup \{(\mathcal{C}_i(\mathcal{S}), \frac{\mathcal{V}_i(\mathcal{S})}{C_i \cdot |\tau_i|}, \mathcal{S})\}$
9: $\quad\quad C^{\text{bound}} \leftarrow \mathcal{C}_i(\mathcal{S})$
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 WCET upper bound C^{bound}. In Loop 3 the algorithm builds a combination of inserted cache miss $\mathcal{S}$ with a QP generated as presented earlier in this section at Line 4. If no combination of inserted cache misses is found, we exit the loop at Line 6. Otherwise, at Line 7, 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 WCET bounds is also updated with the current combination values WCET for the next iteration.

4.5 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. In [21], the authors state that ECC in DL1 increases the WCET of a task by 10% in average when adding an extra cycle per access. In the extreme case, the WCET can climb up to 20% of its value without ECC. This is because the ECC memory applies the protection mechanism to *every memory access*. While the work in [21] targets the DL1 and not the IL1, we make the same assumption that ECC in L1 cache memory are also implemented in a way that an extra cycle is required to access the IL1.

To reduce the overhead, 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 accessed normally, without using the ECC mechanism and without paying the additional delay due to decoding. However, in this case the instruction is completely vulnerable.

We observe that the mechanism described above has a 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 of a special ISA and a special ECC cache memory.

To compute the task profile using this mechanism, we substitute Equation (8) with:

$$C_{\mathsf{lb}}^{\text{impact}} = I_{\mathsf{lb}} \cdot (C^{\text{ECC}}) \tag{19}$$

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 (17) is simplified as follows:

$$V_j = V_j^{\text{path}} \cdot (1 - X_j) \cdot d_{\mathsf{lb}_j} \tag{20}$$

5 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} combinations in the profile of τ_i: If the value of this variable is 1 then the j-th combinations is selected, otherwise it is not. As only one combination must be chosen for each task profile, the first constraint of our QP problem is:

$$\forall \tau_i \in \mathcal{T}; \sum_{\forall j} X_{i,j} = 1 \tag{21}$$

Let $C_{i,j}$ and $V_{i,j}$ be respectively, the WCET of τ_i and its TAVF when the j^{th} combinations 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. [14], for the schedulability constraints:

$$\sum_{\forall \tau_i \in \mathcal{T}} \sum_{\forall j} \frac{C_{i,j} X_{i,j}}{T_i} \leq 1 \tag{22}$$

$$\forall i, 1 < i \leq n; \quad \forall L, T_1 < L < T_i$$
$$\left(\sum_{\forall j} X_{i,j} \cdot C_{i,j} \right) + \sum_{k=1}^{i-1} \left\lfloor \frac{L-1}{T_k} \right\rfloor \cdot \left(\sum_{\forall g} X_{k,g} \cdot C_{k,g} \right) \leq L \tag{23}$$

Equation 22 limits the system workload to 1 which is the schedulability bound for Earliest Deadline First Scheduler. Equation 23 verifies that each task finishes its execution before its deadline in a non-preemptive system.

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

$$fct^{\text{obj}} = min \sum_{\forall \tau_i \in \tau} \sum_{\forall j} \left(X_{i,j} \cdot V_{i,j} \cdot \frac{C_I}{T_i} \right) \tag{24}$$

6 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 subsections, we discuss the performance obtained by the invalidation and ECC mechanisms.

6.1 Experimental setting

We consider a single ARM7 core architecture with a 16 KB 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. These 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 [22] and TacleBench [23]. Details of these tasks are given in Table 3: the third

column is the WCET and the last column is the initial TAVF of the unmodified tasks computed with the OTAWA tool [1][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 3: List of programs from the 2 benchmarks. In this table we consider a 16 KB IL1 and a BRT of 20 cycles.

6.2 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 4.

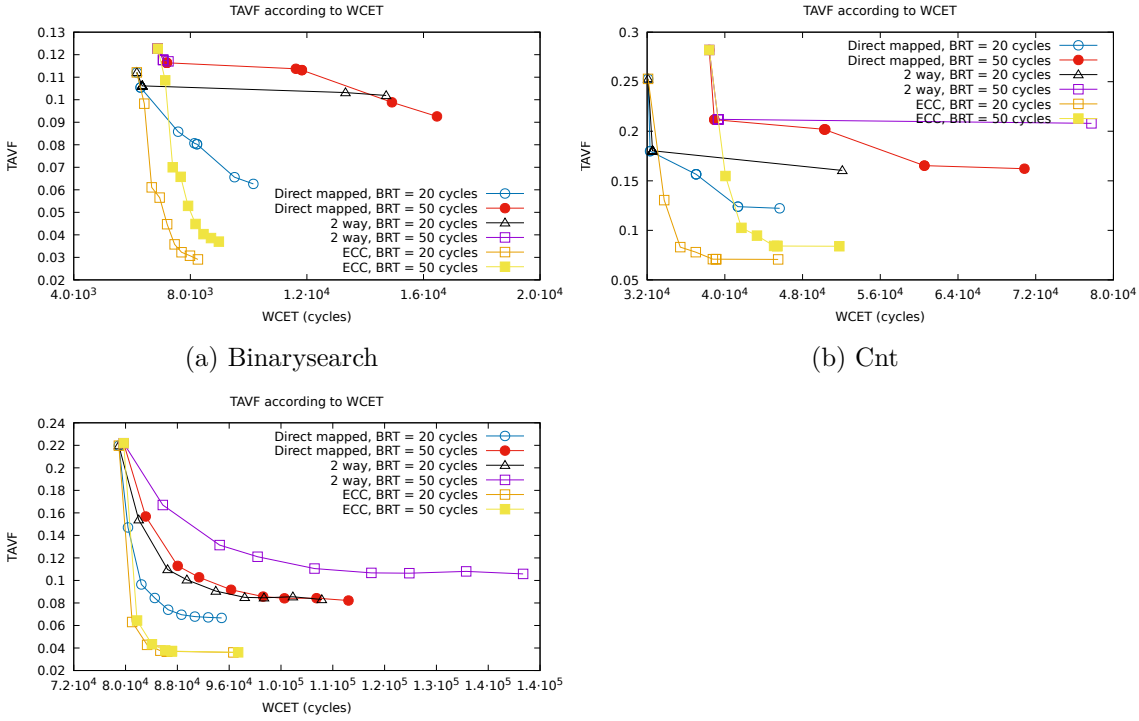
For each task in the benchmarks listed in Table 3, we built a profile using the CPLEX tool. We set a limit of 3 minutes for solving each QP problem, thus obtaining a combination of cache miss locations. Figures 7a, 7b, and 7c show the resulting task profiles respectively for the *binarysearch*, *cnt* 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 show the relation between the task WCET and its vulnerability factor for different combinations.

The cache configuration has a strong impact on the task profile. Figure 7a 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

[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 4: Cache miss mechanism impact used in the experiments.



(a) Binarysearch

(b) Cnt

(c) Select

Figure 7: Case Study task profiles

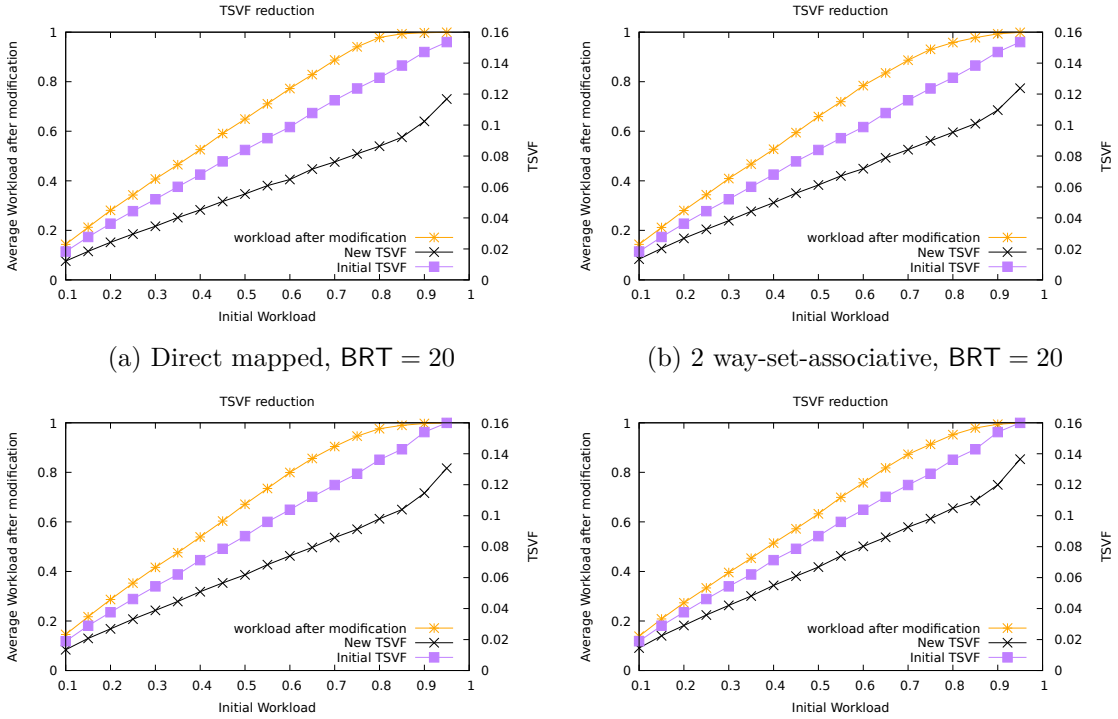
a constant. Figure 7c shows also, that the vulnerability can not be approximated by a linear function of the WCET.

These numbers show that our method can reduce the TAVF more or less effectively, depending on the task code structure and the cache architecture. For example the *cnt* TAVF can be decreased to 0.12 with a direct mapped cache and a $\mathsf{BRT} = 20$ cycles. This corresponds to a reduction of 50% of its initial TAVF. However, TAVF for *select* with a direct mapped cache and a $\mathsf{BRT} = 20$ cycles can be decreased by 73% compared to its initial value.

6.3 TSVF reduction

In the previous section we have shown the relationship between vulnerability and WCET by using the invalidation methods of Section 4.2. 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.

In the experiments, we assume that all the generated task sets contain 12 tasks randomly selected from Table 3. 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* [24] algorithm, such that the sum of all tasks' utilization is equal to the total system workload U.



(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 8: TSVF reduction after CB invalidation with a 16KB-IL1 cache

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

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

As some WCETs in Table 3 are larger than the periods in $\mathcal{G}$, the WCET values have been normalized, dividing them by the logarithm of the smallest WCET. This allows us to test a larger number of task sets. For each total workload, we generated 1000 task sets.

For each task set, we used the corresponding task profiles to perform the optimization procedure described in Section 5, which selects the best combinations of ICMs 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 8a, 8b, 8c and 8d we report the results on two 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.

We first observe 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

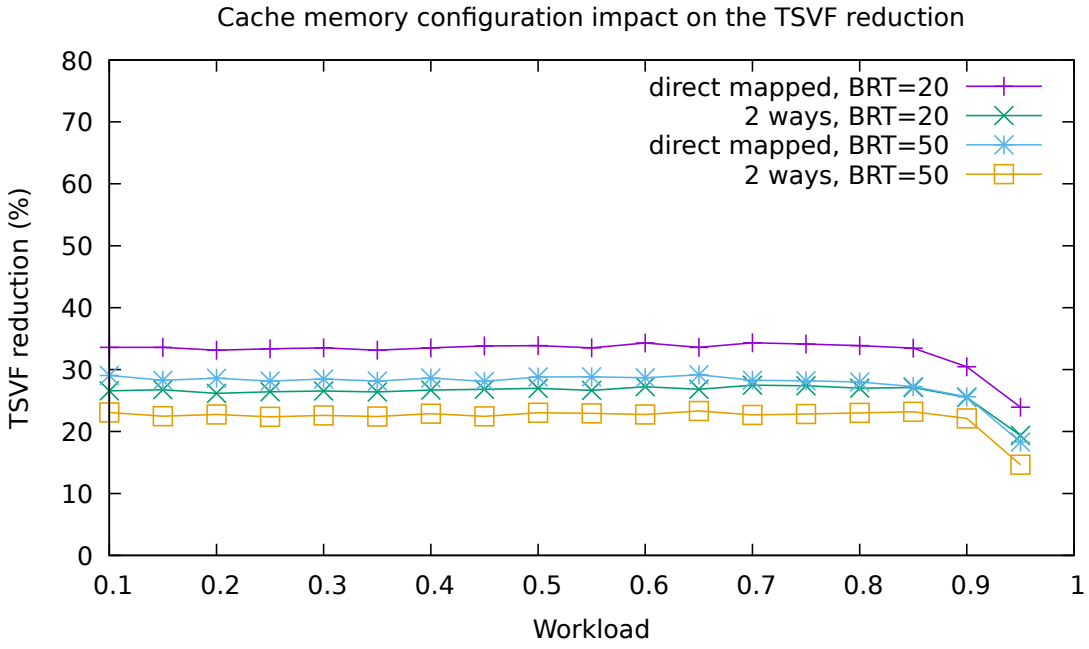


Figure 9: TSVF reduction with miss insertion. In these experiments, an average TSVF value is calculated on workloads of 1000 task sets.

in Figure 8a, 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 their initial values, we give in Figure 9 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 constrained: for these 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 9, we observe that we obtain better performance with direct mapped cache 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.

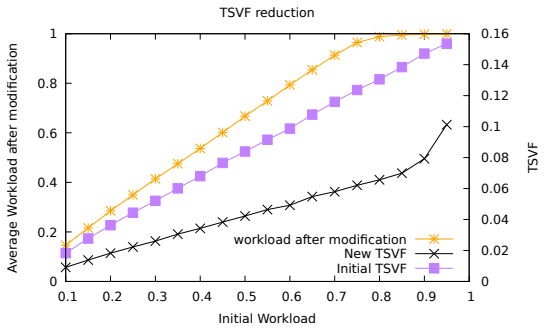
Summarizing, by using the invalidation technique, we can achieve on average between 22% and 34% reduction of the TSVF in task sets with initial workload inferior to 85%.

6.4 TSVF reduction with ECC

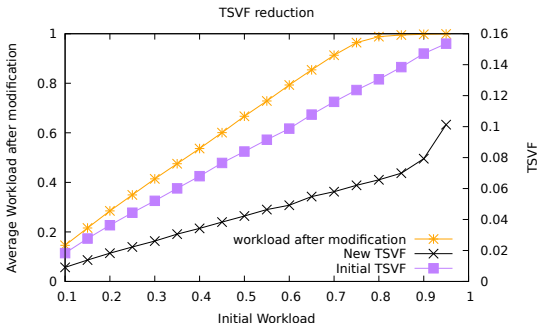
In this section, we evaluate the efficiency of the ECC mechanism discussed in Section 4.5.

Figures 10a, 10b, 10c and 10d depict the results obtained using the same configurations as those presented in Section 6.3. The ECC mechanism execution time C^{ECC} is set to 16 cycles in the experiments. This value is widely used in the literature [25] and corresponds to an average value for ECC execution time with different levels of complexity.

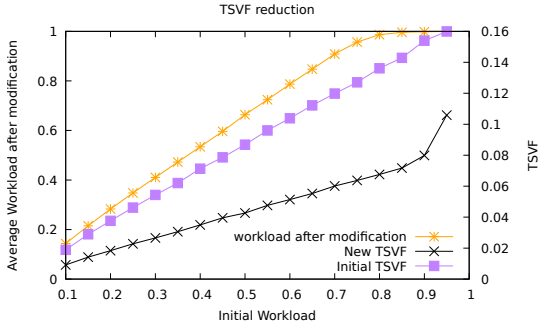
As expected, with the ECC mechanism we obtain a higher TSVF reduction compared



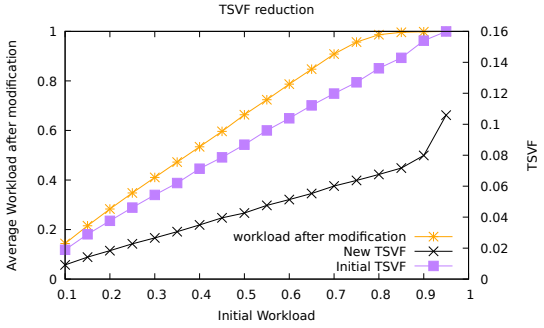
(a) Direct mapped, BRT = 20



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



(c) Direct mapped, BRT = 50



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

Figure 10: TSVF reduction with the ECC mechanism for a 16KB-IL1 cache. ECC execution time is set to 16 cycles.

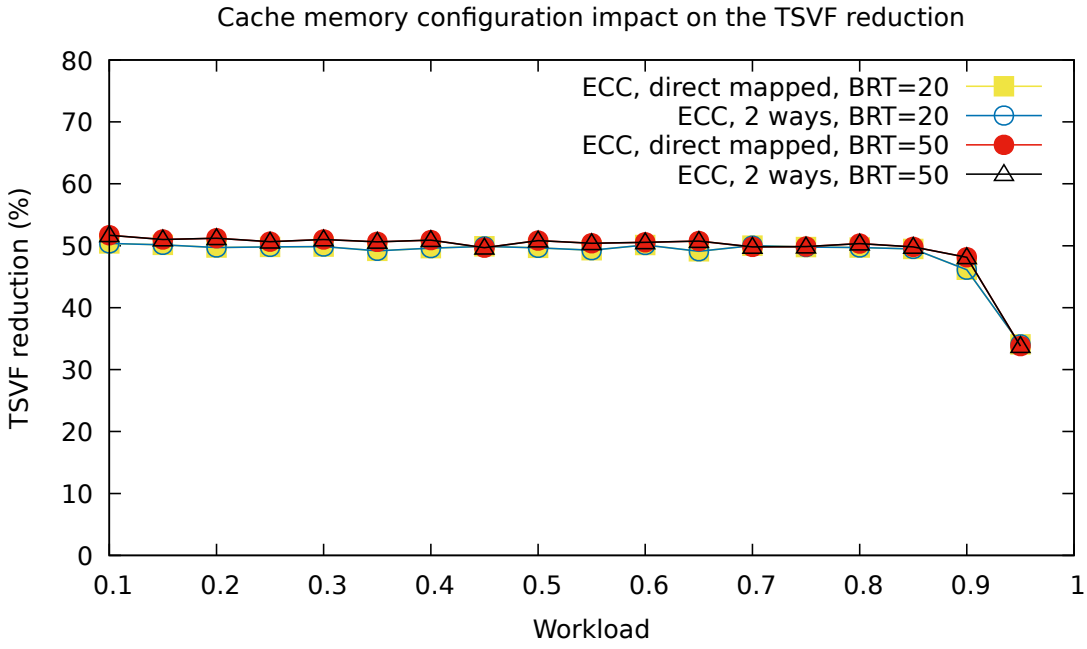


Figure 11: TSVF reduction with the ECC mechanism

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 9, Figure 11 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 the TSVF by 50% for task sets with high utilization around $U = 0.8$.

6.5 Analysis time

The average analysis time for a given number of tasks is shown in Figure 12. A total of 18000 task sets has been generated for each size of task sets, but only schedulable task sets have been taken into consideration. In this experiment, we considered a 16KB-IL1 cache with 2 ways associativity and a BRT $= 50$ cycles; the curve illustrates the relationship between the number of tasks and the average time to execute our algorithm for reducing the TSVF, along with the 95% confidence intervals.

In the experiment, we excluded the execution time of Algorithm 1, which is the most costly part of our analysis. As explained in Section 6.2, we have set a limit of 3 minutes to build a task profile configuration. Thus, computing a task profile of 10 configurations can take at most 30 minutes. Figure 12 shows that the time required for reducing the TSVF of 12 tasks is 2.75 times longer than that for 6 tasks. Furthermore, the average execution time of the algorithm is always under 250ms. Finally, we observed no relationship between workload and execution time in our experiment. In conclusion, the overall cost of the

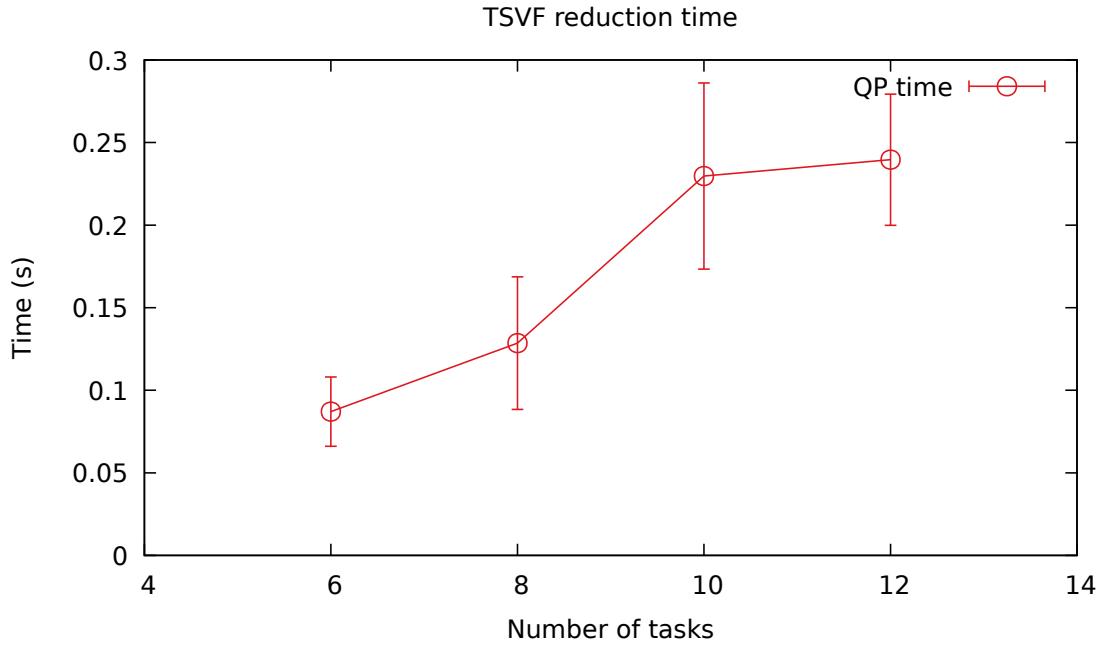


Figure 12: TSVF reduction time considering a 2 way-set-associative 16KB-IL1 cache and BRT = 50 cycles.

analysis is dominated by Algorithm 1, which needs to be run once per task to build the task profiles.

7 Conclusion

Hard-real time systems become more vulnerable to faults especially in the instruction cache memory of COTS microprocessor. We propose in this paper 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, we compute the vulnerability *profile* of each task to find the best spots where to add a protection mechanism to reduce the vulnerability while increasing the WCET. Then, we propose two alternatives: in the first one, 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. We show, by a set of experiments on real benchmarks, a reduction of the task set vulnerability factor between 22% and 34% with cache block invalidation and of 50% while using ECC.

As future work, we plan to extend our method to data caches. Addressing data cache would require a different kind of static analysis with respect to the must/may analysis we use for instruction caches. Also, data caches bring in additional problems, like the write policy (write-through vs. write-back). We also 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.

References

[1] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: An open toolbox for adaptive wcet analysis. In Sang Lyul Min, Robert Pettit, Peter Puschner, and Theo Ungerer, editors, *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[2] Shuai Wang and Guangshan Duan. On the characterization and optimization of system-level vulnerability for instruction caches in embedded processors. *Microprocessors and Microsystems*, 39(8):686–692, 2015.

[3] Michail Mavropoulos, Thodoris Lappas, Georgios Keramidas, and Dimitris Nikolos. 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.

[4] 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)*, pages 85–92. IEEE, 2013.

[5] Jeongkyu Hong and Soontae Kim. Smart ecc allocation cache utilizing cache data space. *IEEE Transactions on Computers*, 66(2):368–374, 2017.

[6] Taniya Siddiqua, Vilas Sridharan, Steven E Raasch, Nathan DeBardeleben, Kurt B Ferreira, Scott Levy, Elisabeth Baseman, and Qiang Guan. Lifetime memory reliability data from the field. In *2017 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6. IEEE, 2017.

[7] 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*, pages 117–118. IEEE, 2010.

[8] Chris Wilkerson, Hongliang Gao, Alaa R Alameldeen, Zeshan Chishti, Muhammad Khellah, and Shih-Lien Lu. Trading off cache capacity for reliability to enable low voltage operation. *ACM SIGARCH computer architecture news*, 36(3):203–214, 2008.

[9] 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)*, pages 192–202. IEEE, 2016.

[10] 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*, pages 1–6. IEEE, 2007.

[11] Shuai Wang, Jie Hu, and Sotirios G Ziavras. On the characterization and optimization of on-chip cache reliability against soft errors. *IEEE Transactions on Computers*, 58(9):1171–1184, 2009.

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

[13] Anand Bhat, Soheil Samii, and Ragunathan Rajkumar. Practical task allocation for software fault-tolerance and its implementation in embedded automotive systems. *Real-Time Systems*, 55(4):889–924, 2019.

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

[15] C.A. Healy, R.D. Arnold, F. Mueller, D.B. Whalley, and M.G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1):53–70, 1999.

[16] Chang-Gun Lee, Hoosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE transactions on computers*, 47(6):700–713, 1998.

[17] Wei Zhang, Nan Guan, Lei Ju, and Weichen Liu. Analyzing data cache related preemption delay with multiple preemptions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2255–2265, 2018.

[18] Shaun Clowes. Fixing/making holes in binaries. *BlackHat USA*, 2002.

[19] 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`.

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

[21] Pedro Benedicte, Carles Hernandez, Jaume Abella, and Francisco J Cazorla. Laec: Look-ahead error correction codes in embedded processors l1 data cache. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 818–823. IEEE, 2019.

[22] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 136–146, 2010.

[23] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sorensen, Peter Wägemann, and Simon Wegener. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, 2016.

[24] Enrico Bini and Giorgio C Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1):129–154, 2005.

[25] Irina Alam, Clayton Schoeny, Lara Dolecek, and Puneet Gupta. Parity++: Lightweight error correction for last level caches. In *2018 48th Annual IEEE/IFIP*

International Conference on Dependable Systems and Networks Workshops (DSN-W), pages 114–120, 2018.