Toward a predictable and secure data cache algorithm: a cross-layer approach

Philippe THIERRY∗†, Laurent GEORGE∗, Jean-François HERMANT†, Fabien GERMAIN‡, Dominique RAGOT‡ and Jean-Marc LACROIX‡

∗LiSSI/UPEC
120 rue Paul Armandot
94400 Vitry-Sur-Seine, France
†LACSC/ECE
37 quai de Grenelle
75015 Paris, France
‡Thales DSC/OPS/SAT
160 avenue de Valmy
92704 Colombes, France

Abstract—Nowadays, the gap between processor and memory speed has grown enough to make the cache usage unavoidable in order to take real advantage of the processor’s capabilities. Nevertheless, the data cache usage causes in the same time hard real-time and security concerns. As a consequence, some real-time compliant data cache algorithms have been developed. The goal of such algorithms is to reduce the WCET (Worst-Case Execution Time) of tasks. Unfortunately these solutions usually have an impact on the system security, generating breaches in the tasks partitioning. Considering security requirements might result in huge overheads, and can raise an impact on the WCETs, making the cache usage useless.

This article contributes to the definition of a data cache algorithm meeting at the same time security and real-time requirements using a cross-layering approach between the underlying hardware and the running kernel. This corresponds to some secure and real-time hybrid systems needs.

In order to contribute to a data cache memory management algorithm for such hybrid system, this article defines properties needed at the same time by hard real-time systems and partitioned secure systems and contributes to an algorithm meeting all of these requirements.

I. INTRODUCTION

A. Motivation

Recent hardware implementations have been carried out in order to simplify software architectures, by hiding the hardware complexity from the software implementations. This trend allows the software implementation to be decorrelated from the hardware specificities.

Caches are a good example of a shadowed capacity of the underlying hardware, allowing better software performance without proactive software implementation.

This simplification is done at the price of a loss of predictability in the software execution. As a consequence, most of the modern hardware improvements on general purpose processors are deactivated in hard real-time systems. In today’s architectures, deactivating the cache generates a huge overhead of memory requests on a single memory bus. This consequently increases the worst-case response time for each memory accessor and slows down the software execution.

Some solutions have been proposed in order to take advantage of the data cache and the instruction cache in order to reduce the WCETs of the tasks, by adding a predictability property to the cache content variation. This is done using static and dynamic cache locks [1], [2], [3], cache partitioning [2] or scratchpads [3].

Such techniques provide a good solution to the problem of maintaining the inconsistency of the data cache usage in real-time architectures, but usually induce security concerns, which make them unusable when security of data caches is required.

In the same way, when high security (i.e. security of Information Technology) considerations must be taken into account, cache memories, among other elements, are source of tasks partitioning breaches, through cache-based covert channels.

A cache-based covert channel is due to the variation of the data cache content from a job to another in preemptive scheduling. Because the impact on the data cache is due to the current job memory consumption, it is possible for a given job to generate more or less cache line replacements during its execution. Such behavior may allow a given task to be informed of the previous task behavior, measuring its own cache miss rate through time analysis (i.e. measuring the cost of a data access, compared with a known cache-hit access time). Such measurement can be considered as a boolean data, depending on the cache miss rate.

In [4], the author explains how data cache covert channel can be used in order to obtain a RSA key through time analysis. In order to disallow such covert channel, caches are usually voluntarily flushed by the scheduler, or cache are deactivated for some sensitive tasks.

In the following, the usage of the term security refers to the information partitioning guarantee concerns.
B. Scope and limitations

In this paper, shared data cache and multi-level caches are not taken into account. This paper contributes, through a cross-layer approach, to a predictable and secure usage of an unshared, single-level data cache memory. This article describes the implementation of a data cache memory management algorithm, used through a cross-layered approach done at the kernel level, as a first element of a better predictable hardware usage.

As shown in the case of heterogeneous multi-cores [5] or for better ad-hoc wireless controller power control [6], a cross-layer approach allows the system to optimize its behavior better than independent hardware and software layers.

One goal is to propose a secure cache management algorithm offering better performances than systems where data cache is deactivated, without losing the control of the data cache content and permitting any violation of the task partitioning. The data cache memory management proposed in this article is predictable, allowing an optimized measurement of the WCET which takes into account cache-hits for memory access cost. This article proposes a better synergy between hardware and software implementations in order to maintain predictability of the data cache behavior from the software point of view. In order to maintain a control of the cache behavior at the software level, the cache algorithm described in this paper is controlled by the software scheduler. The data cache memory management implementation also takes into account security requirements in order to prevent the usage of data cache-based covert channels [7], [4]. In order to define a new algorithm, the following properties are proposed. We show that the data cache memory management algorithm must meet each of them to be compliant with a hard real-time and secure system.

\textbf{Property 1:} The usage of the data cache by a task must not increase its WCET.

\textbf{Property 2:} Cache-related preemption due to inter-task interference should be null. When a task is loaded by the scheduler, the cache content should be known and should not depend on other tasks’s behavior.

\textbf{Property 3:} Cache load/cache flush cost must be at least upper-bounded.

\textbf{Property 4:} The impact of the cache on the memory access cost must be quantifiable in WCET measurement.

Properties 3 and 4 are needed in order to consider the data cache impact in the WCET measurement. If one of them is not supported by a data cache memory management algorithm, the data cache impact can’t be taken into account in WCET measurement. Property 1 is needed in order to take a real advantage of the data cache usage. This property can be obtained through a data cache memory management property or through a capacity to predict the data cache overhead for each task. Property 2 is a consequence of the data cache covert channels threats [7], [4].

In Section II, existing data caches are presented. A state of the art describes the existing problems and solutions for real-time and security in data cache management. For each of them, its compliance to Properties 1, 2, 3 and 4 is discussed.

In Sections III and IV, the real-time and secure data cache memory management implementation is described and its impact formalized. This section also describes the impact on software implementations, specifying how the data cache could be managed. A macroscopic measurement is then proposed.

Sections V and VI propose a new definition of a fixed priority event-driven preemptive scheduling runtime cost measurement, and a feasibility analysis on a simple task set. Section VI shows how the data cache algorithm makes it possible to schedule an unschedulable tasks set using a classical fixed priority event-driven preemptive scheduler.

II. ABOUT EXISTING DATA CACHE MEMORY MANAGEMENT IMPLEMENTATIONS

A. Generic hardware architecture

Today’s systems usually use multi-level cache architecture, for better performance. The Level 1 (L1) cache is a specialized memory, separated in a data cache memory (the one studied in this paper), and an instruction cache memory, for instruction pre-fetch. L2 and sometimes L3 caches are general purpose fast access memory, used as a backup memory for L1 cache. Accessing the main memory is done through a memory bus. Its access depends on an external agent, which can be a bus controller or an interconnect. This agent schedules the access to the memory bus for all the bus accessor(s), etc to avoid any collision. Cache access is faster than main memory access because of its proximity to the CPU core. L1 caches are accessible without using shared bus, avoiding shared access overhead.

Cache memories can be a necessary step for the core to access the memory bus (case of cache look-through architectures
B. Predictable and probabilist L1 data cache memory management implementations

Cache algorithms have three main properties which are:

- **the cache associativity**, defining, for each memory cache line, the number of possible location in cache, from 1 (direct mapped) to \(N\) (\(N\)-way associative cache), or any (fully associative).
- **the cache line replacement algorithm**, using several techniques\(^2\) like MRU, LRU, Pseudo-LRU, Segmented-LRU (SLRU) or LFU, when the cache is set-associative.
- **the cache consistency** support, which can be:
  - snooping: spying the address bus, detecting modification on main memory. The cache may then deactivate the associated cache line, or anticipate a reload before the next cache line access
  - snarfing: spying the address and data bus in order to maintain the cache content versus the main memory consistency
- **the memory Synchronisation method**, which can be:
  - write-back: waiting as long as possible before writing back the modification in the main memory.
  - write-through: synchronize the cache content with the main memory at each modification, avoiding the problematic of multi-core caches incoherency.

Direct-mapped caches use a predictable algorithm. For a given cache line, the caching time is fixed as soon as the access to the main memory is given by the memory bus. Usually, implemented algorithms are set-associative, generating unpredictability of the memory caching time due to the cache internal memory management algorithm. Although, such algorithms duration is usually upper-bounded. The set-associative data caches are based on probabilistic algorithms in order to generate the best average cache-hit ratio (e.g. (P)LRU, (S)MRU).

Whatever the data cache algorithm is, the cache content directly depends on the overlying software behavior. For sporadic or aperiodic task sets, when using preemptive scheduling policy, the variation of the data cache content is very difficult not to say impossible to predict. As a consequence, even with predictable data cache algorithms like direct mapped, the WCET computation can’t take into account a data cache usage in order to reduce the task set execution cost.

Moreover, probabilistic cache memory management implementations never flush the entire cache memory without an explicit software request. The first consequence is that a part of the data cache may be left unchanged from a job execution to another, allowing the last one to be informed of the amount of cache consumption of the previous one, using advanced time measurements. Such covert channels are common in generic hardware implementations.

**Compliance of common data cache memory management algorithms to Properties 1, 2, 3 and 4** Depending on the hardware implementation, cache-based preemption delay (cache-miss and memory access) may be longer than a direct memory access (Worst-Case Memory Access) time. This depends on the position of the data cache (look-aside cache or look-through cache) and how the verification of the content in the cache is done. As a consequence, common cache algorithms are not always compliant with Property 1 depending on the internal cache controller implementation.

None of the above cache algorithms are compliant with Property 2. Though voluntary software-based cache flush avoid inter-task cache preemption. Direct-mapped caches meet Property 3. Set-associative cache only allows the measurement of an upper-bounded cache load time, which still meets Property 3.

As described in \^[8] Set-associative cache algorithms are extrinsic. They are function of the software execution environment. Direct-mapped caches are intrinsic, their behavior is not impacted by the software environment. Although, in both case, there is no guarantee, for a copied cache line, to still be present in preemptive systems. The associated worst-case memory cost is then always the main memory access cost. Thus, none of them are compliant with Property 4.

As a consequence, none of the commonly used data cache memory management algorithms meet the four Properties at the same time, and then can’t be solution to the dual problematic of this paper.

C. Integration of predictability in data cache access

Software behavior impact on cache state makes them unusable in hard real-time systems \^[9]. As a consequence, hard real-time software sometimes need specific ASIC or specific FPGA implementations in order to be at the same time predictable and efficiently used. There are two main problems to the above situation:

- the low frequency of FPGAs in comparison to standard unpredictable general purpose processors
- the power consumption of FPGAs, w.r.t. low powered GPP like ARM based cores
- the financial cost of FPGA implementations and specific ASIC, w.r.t. a standardized general purpose processor.

There are two techniques \^[2], \^[10] in order to recover some predictability in data cache usage: cache partitioning and cache locking. These techniques have been studied \^[11], measuring their impact in hard real-time systems. Scratchpads, because they correspond to a close-to-core SRAM, are not considered in this article.

**Cache partitioning** This technique defines partitions in all or a part of the data cache in order to avoid inter task interferences, generating cache preemption. The number of partitions, depending on the task set properties like the maximum number

\(^2\)MRU: Most Recently Used, LRU: Least Recently Used, LFU: Least Frequently Used
of simultaneous jobs, reduce the amount of cache accessible to a given task. Cache partitioning avoids inter task cache preemption but raises the intra task cache preemption due to the over-consumption of the cache partition [10].

Compliance of cache partitioning to Properties 1, 2, 3 and 4 Cache partitioning may not meet Property 1. Depending on the task set properties, some tasks may be incompatible with a reduced data cache size, depending on its behavior. Such task may generate successive cache-refill instead of taking advantage of its amount of allocated cache lines. When using big task sets, only a subset of the tasks can be cached using data cache partitioning, using selection algorithm based on task priority or memory consumption behavior.

Cache partitioning meets Property 2. For a given task, the partition content is maintained during the preemption period. Cache partitioning is not predictable because the partition content variation depends on the task’s behavior. Though the variation can be upper bounded through static analysis of the task, it is then not compliant with Property 3. Although, Cache Partitioning is compliant with Property 4, because the cache-miss rate due to intra-task memory consumption can be upper-bounded through static analysis.

Cache lock is implemented in numerous processors like PowerPC 604e, 405 & 440 families, some Intel x86 and Motorola MPC7400. The cache locking can be done statically or dynamically.

Static data cache lock is done by locking the cache at the system boot-up. Such behavior guarantees that a part of the cache contains needed data all along the system execution. This solution doesn’t respond to this article’s problematic, because it aims to lock usually the data of one task in order to speed it up. This is a good solution in order to speed up, for example, the scheduler, but not a complete task set. Dynamic data cache lock is done by allowing tasks to lock/unlock parts of the data cache. This lock avoids intra-task cache line override. Such locking algorithm can be done for the task lifetime, but with performance issues [12], or being unlocked at each preemption, reducing predictability benefits in preemptive scheduling.

In dynamic cache lock behavior, the WCMA to locked data is fixed to worst-case cache-hit access but is cleared by preemption. As a consequence, the cache load impact depends on the number of preemptions, which can be upper-bounded. Using task’s lifetime cache lock, a lock request may be refused, depending on the current free space in the data cache. The lock threshold policy is hardware specific. Such refusal can also happen when a task tries to lock more than the allowed threshold.

Compliance of dynamic cache locks to Properties 1, 2, 3 and 4 Dynamic cache lock is done at tasks request. This solution is compliant with Property 1 because it is under the task control, and then optimized by the task itself, depending on its current execution graph.

When using cache locking, the task is able to consume a certain amount of cache lines, generating inter-task cache preemption. When using cache locks having a lifetime up to a job’s lifetime, the inter-task cache preemption is heightened. As a consequence, dynamic cache lock is not compliant with Property 2, and may make time analysis more easy because the task has a direct access to the cache update policy using lock requests.

Dynamic cache lock is not compliant with Property 3 because it depends on the current job behavior and the lock threshold, but it can be measured and integrated in dynamic WCET measurement in the scheduler. As a consequence, dynamic cache lock is compliant with Property 4. The main issue with dynamic cache locking is a security concern. As a consequence, it can’t be considered as a solution to hybrid real-time and security problems.

D. Integration of security in data cache usage

As for now, there is no hardware implementation of security algorithms in data caches. As a consequence, disallowing data cache time analysis based threats is fully done at software level. This is done by Flushing both data caches and TLB at each preemption. The TLB is the Translation Lookaside Buffer, used by MMU, which speed up the translation between virtual addresses and physical addresses when using memory paging.

Because the TLB content, like the data cache, depends on the software behavior, the same timing analysis threats can be done using the TLB-miss measurement. A TLB-miss generates a main memory access request in order to read back the requested page information in the Page Table. Avoiding time analysis threats is done through flushing both TLB and data cache content. n this paper, TLB is not considered.

Conclusion Today’s existing solutions respond only to real-time issue or to security issue but not both. Moreover, solutions to one problematic may sometimes generate huge incompatibility with the other one. Nevertheless, dynamic cache lock solution is a good start in order to respond in the same time to real-time and security concerns. It needs to be completed in order to correct its security weakness.

III. DEFINITION OF A PREDICTABLE AND SECURE DATA CACHE ALGORITHM

A. General description of the data cache algorithm

In order to meet Properties 1, 2, 3 and 4, this paper provides a flush-and-preload-based cache algorithm. For a more comprehensive explanation, this algorithm is described in the following as the SFPL (Secured Flush and PreLoad) data cache algorithm. This algorithm is proposed as a software-based algorithm integrated in the scheduler, and requires that the underlying hardware supports Hypothesis 1, 2, 3, 4, 5 and 6.

Hypothesis 1: The data cache supports direct mapped algorithm.

Hypothesis 2: The data cache supports cache lock with a known threshold.
Hypothesis 3: The data cache supports overall cache lock and unlock requests. Such request have a bounded execution time.

Hypothesis 4: The data cache can be dynamically activated and deactivated.

Hypothesis 5: The TLB flush execution cost can be upper-bounded.

Hypothesis 6: The memory bus worst-case access time can be upper-bounded.

The data cache is fully controlled by the scheduler, to avoid security threats. The scheduler has then the possibility to activate or deactivate the cache before each job execution. It also has the possibility to generate loading, preparing (see bellow), and flush requests.

The SFPL algorithm is based on a systematic flush and preload of the cache contents. As a consequence, the cached memory should be preallocated by the kernel for each task. These preallocated memories are named as memory pools. Each task has its own memory pool, reused for each job.

For each job scheduling, the associated memory pool is preloaded into the cache. The pool is then mapped into the job’s address space, allowing it to use its pool like any other allocated memory. The preload of this memory in cache reduces all memory access to a cache-hit access.

Because the SFPL algorithm generates an overhead due to successive cache flush and cache load, it may degrade the performance of some tasks, depending on the scheduling policy and the task profile. As a consequence, the SFPL algorithm alone is not compliant with Property 1. Though, the SFPL algorithm is associated with an election sequence in order to guarantee that its usage generates an effective task speed up. The SFPL algorithm is enabled only when the associated overhead is counter balanced by the task’s execution speed-up. The task election to the SFPL algorithm is explained in Section VI.

A basic implementation of the SFPL algorithm into the scheduler will have the following behavior for a task $\tau_i$ being selected for execution:

1) Data cache memory purge and preparation. If the previous task used the data cache, locally modified content should be written back to the memory, and the whole cache content should be prepared. This second part can be considered from a security or safety (i.e. faultsafe) point of view. In this paper, the cache content preparation can be, for example, a randomization of the cache lines content. In the same time, other hardware buffer like the TLB should be flushed. The overhead of the cache memory preparation is considered in this paper as a symbolic value, in order to leave the applied security (resp. safety) algorithm generic.

If the previous task wasn’t elected for the SFPL algorithm, the data cache is only re-enabled. This step cost is denoted $C_{\text{prepare},i}$.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{fig2.png}
\caption{Temporal diagram of predictable data cache usage}
\end{figure}

2) New task(s) management and task election depending on the scheduler model.

3) If the newly elected task uses data cache, its associated memory pool is loaded into the cache. The cost of this step is named $C_{\text{load},i}$. Otherwise the cache is disabled.

Figure 2 shows how the SFPL algorithm impacts the scheduler.

External events and interrupt consideration All executed tasks are not controlled by the software scheduler. Interrupts preempt the running task without notifying the software scheduler.

In order to guarantee the data cache memory content at the end of the interrupt handler execution, the SFPL data cache deactivates itself in the Interrupt Service Routine. Its content remains unchanged during all the interrupt handler execution. At the end of the Interrupt Service Routine, the data cache is reactivated. This paper doesn’t contribute to a speed-up of external events handlers, though it is possible to consider mixed SFPL cache locking and cache partitioning for event management. This could be done using a specific cache partition for trustworthy software elements like interrupt handlers. This issue is left to future works.

From a security point of view, flushing and reloading the data cache content before each job instantiation, puts the data cache in a well-known state. The SFPL algorithm is therefore compliant to Property 2. For tasks for which the cache is disabled because of a too high cache management overhead, cache-based covert-channels are nonexistent.

B. Toward a first software implementation

The study of the software implementation of the SFPL algorithm is out of the scope of this paper. This article provides some SFPL algorithm support requirements:

- The pool of memory that should be cached for a given task $\tau_i$ should be allocated before the first task execution, by the kernel. It also should be maintained during all the
life cycle of the task and all its jobs.
In order to maintain the overall system security, the memory pool should be initialized in a known state, memsetting them with zeros, for example.

- The memory pool should be accessible to the task. In order to avoid reimplementation of all real-time software, the cache usage should be done without code source modification.

This can be done through a modification of the tool chain (the compiler is in charge of positioning usually used data in a specific memory area) coupled with a modification of the task load implementation of the kernel. This can also be done through mapping the pool on the heap section of the task. Such usage make the SFPL algorithm usage compliant with the Property 4.

- Using the pool for storing I/O data (e.g. network packets, loaded to main memory through DMA) requires that the cache controller supports snarfing method in order to maintain cache consistency.

IV. DEFINING IF A TASK SHOULD USE OR NOT THE SFPL DATA CACHE

Considering the SFPL algorithm implies that task profile measurements like static WCET computations [13], [14] should be updated considering the data-cache impact in WCMA (Worst-Case Memory Access) integration. Such integration generates a new WCET $C_i$, which can be higher or lower than the original one without cache WCET measurement. This article doesn’t detail how the SFPL algorithm should be taken into account in such measurement. Updated values of $C_i$ are considered as input values in Section VI.

The predictable data cache algorithm has a certain impact on the scheduler’s execution cost. This algorithm is divided into two parts:

- Preparing the cache memory in order to receive the following task’s memory pool content. This cost is defined as the value $C_{\text{prepare},i}$.
- Loading the following task’s cache pool into the data cache. This cost is defined as the value $C_{\text{load},i}$.

The total cost overhead of the data cache usage on the scheduler execution is then the summation of the $C_{\text{prepare},i}$ and $C_{\text{load},i}$ costs:

$$C_{\text{cache},i} = C_{\text{prepare},i} + C_{\text{load},i} \tag{1}$$

The value $C_{\text{cache},i}$ depends on the following hardware properties:

- $S_{\text{data-bus}}$ is the width of the data bus (in bits).
- $W$ is the temporal window (in cycles) resulting from the memory bus controller or the interconnect, allocated to the processor for accessing the main memory.
- $t_{\text{CPU}}$, including the time needed by the CPU to set (resp. receive) data on (resp. from) the bus.

It is considered in this paper that the read cost may differ from the write cost. As a consequence, read and write access are formalized separately.

A. Description of the $C_{\text{prepare},i}$ cost

The data cache is a copy of a subset of the memory. When using data copied in cache, the modifications done by the task in cached memory are not automatically synchronized with the main memory. As a consequence, the locally modified lines of cache (dirty cache lines) need to be written back to the memory. This synchronization generates an overhead, which can be upper-bounded, considering that all of the cache content needs to be synchronized. The synchronization with the main memory corresponds to successive write() requests to the main memory, as the cache fulfill corresponds to successive read() requests. To complete the $C_{\text{prepare},i}$ cost, some local treatments on the data cache memory can be done, for security or safety needs. As explained in Section III, the cost of these treatments depends on the security or safety policy chosen. These treatment costs are then defined as a symbolic value $t_{\text{supp}}$. $C_{\text{prepare},i}$ is formalized in Equation 7.

B. Description of the $C_{\text{load},i}$ cost

The $C_{\text{load},i}$ cost corresponds to the loading of the memory pool into the data cache, before starting the task. It corresponds to a succession of read requests to the memory controller. Its cost depends on $S_{\text{cache}}$, and may be null if the elected task is not eligible for the SFPL data cache algorithm. $C_{\text{load},i}$ is formalized in Equation 8.

C. Memory reading and writing overhead

1) CPU latency overhead: The bus access cost is not only due to the latency generated for each memory access. Before accessing the bus, the cache has its own latency corresponding to the bus access delay, equal to $t_{\text{CPU}}$.

It happens when pushing bits on the bus for a memory writing or reading, and when getting back the bus content. This latency is in the order of the cycle and can vary for reading and writing requests. The $t_{\text{CPU}}$ value is composed of two latencies, possibly equal, denoted $t_{\text{CPU}_{\text{W}}}$, and $t_{\text{CPU}_{\text{R}}}$.

Each of them corresponds to the summation of all the request delays generated for a read or a write request.

For a read request, there are the read-request latency, and the receive-request latency. For a write request, there are the write-request latency, and the send-request latency. Therefore, the CPU latency is defined as follows:

$$t_{\text{CPU}_{\text{Rd}}} = t_{\text{CPU}_{\text{Rd-req}}} + t_{\text{CPU}_{\text{Rd-send}}} \tag{2}$$

$$t_{\text{CPU}_{\text{Wr}}} = t_{\text{CPU}_{\text{Wr-req}}} + t_{\text{CPU}_{\text{Wr-send}}} \tag{3}$$

Row Address Strobe

Column Address Strobe
2) Memory controller overhead: The duration of a memory access depends on the memory controller, memory bus and interconnect specificities. The memory controller must first be informed of the base position of data to read or write to memory, using a matrix based structure (columns and rows). Then it starts to push or receive data on the memory bus using a synchronization mechanism based on the control bus. The memory bus width defines how many bits are sent during one access.

Considering a full access time to the memory controller, the time needed to read or write a given amount of data depends on the memory controller capabilities.

All dynamic RAM architectures have a refresh period to maintain the information, which is specific to each technology. Theses refreshments are managed by the memory controller and may interfere with memory access. They have a specific frequency and duration.

Usually, the refresh period of the DRAM is around 10 milliseconds. This period is very large compared to a complete data cache load time. This overhead is not considered in this article.

Figure 3 describes the time cost to access the main memory in the case of DDR1 SDRAM physical architecture. In this paper, the memory technology considered is the DDR1 (one access per cycle).

Considering a DDR1 SDRAM architecture, for one read or write access, the DDR controller needs to set its coordinates in the matrix cell where the data is positioned. Positioning itself is done using the RAS and CAS signals, generating a delay named \( t_{RCD} \) (RAS to CAS delay).

Moreover, before starting to send or receive data on the bus, the memory controller needs another delay, named CAS-latency \( (CL) \).

Initializing data transfer is done using the WE and the RAS lines at the same time. The RAS line usage has a minimum period, named RAS period \( (t_{RP}) \). If after this period all the data are not sent, an additional delay, at most equal to \( t_{RP} - (t_{RCD} + CL) \) is added before starting the data transfer.

As a consequence, before starting data transfer with the memory controller, the CPU needs to wait \( max(t_{RCD} + CL, t_{RP}) \).

Figure 3 describes how those memory controller specific delays impact memory transfers.

Each read or write access to the memory generates a complete cycle including row and column configuration, generating an overhead. In our case, the cache reads (resp. writes) one cache line per read (resp. write) access, which generates as many write cycles as needed cache lines to fulfill. Depending on the data bus size, the time needed in order to read \( S_{cache-line} \) bits varies. Due to the data bus size, the number of cycles needed to read \( S_{cache-line} \) bits is \( \frac{S_{cache-line}}{S_{data-bus}} \).

Once the data are read, a new memory request cycle is done, separated by at least one cycle. The new cycle starts with a RAS request, which implies that the RAS minimum period is satisfied. Depending on the amount of cycles the data transfer has cost, a new delay is needed before starting a new request. The cost in cycles of this delay is then \( max(t_{RP} - \frac{S_{cache-line}}{S_{data-bus}}, 1) \).

The overall memory access cost \( \Psi \), described above, may differ for a read or a write access, depending on the memory controller properties. As a consequence, we define a read access cost, \( \Psi_{Rd} \), and a write access cost, \( \Psi_{Wr} \):

\[
\Psi_{Rd} = max(t_{RCD} + CL, t_{RP}) + \frac{S_{cache-line}}{S_{data-bus}} + max(t_{RP} - \frac{S_{cache-line}}{S_{data-bus}}, 1)
\]

\[
\Psi_{Wr} = max(t_{RCD} + CL, t_{RP}) + \frac{S_{cache-line}}{S_{data-bus}} + max(t_{RP} - \frac{S_{cache-line}}{S_{data-bus}}, 1)
\]

\( \Psi_{Rd} \) (resp. \( \Psi_{Wr} \) is the cost (in cycles) of a cache line size read (resp. write) in the memory, without taking into account the interconnect scheduling policy. Although, the interconnect generates an overhead corresponding to the waiting time before accessing the bus. \( \Psi_{Rd} \) and \( \Psi_{Wr} \) define the access time for reading or writing a single cache line. For a given task \( \tau \), the number of successive read access of \( C_{load,i} \) is:

\[
N_{tot,i} = \frac{S_{cache}}{S_{cache-line}}
\]  \( \text{(4)} \)

In the same way, for a given task \( \tau \), the maximum number of write accesses to write back to memory during a flush corresponds to the write back of all the task cache lines, which is the same value \( N_{tot,i} \).

3) Bus and interconnect overhead: Accessing the memory is done through a bus access. The memory bus access is controlled by a bus controller or an interconnect, in order to manage the multiple accessors. Temporal windows are defined in order to allocate time slices to each accessor. The temporal window size (in cycles) is specific to the interconnect, and is defined in this paper using the symbolic value \( W \).

We consider that the temporal window allocated for a read access can differ from the one given for a write access. We
then define $W_{Rd}$ (resp. $W_{Wr}$) defining the temporal window size for read access (resp. for write access). Depending on $W_{Rd}$ and $W_{Wr}$, we define two values:

- $NW_{Rd}$, the amount of cache lines loaded during one temporal window
- $NW_{Wr}$, the amount of cache lines written back during one temporal window

with the following definitions:

$$NW_{Rd} = \frac{W_{Rd}}{\Psi_{Rd}} \quad (5)$$

$$NW_{Wr} = \frac{W_{Wr}}{\Psi_{Wr}} \quad (6)$$

The number of temporal windows needed in order to get the total amount $N_{tot,i}$ of requested cache lines is then:

- In the case of cache lines load: $\left\lfloor \frac{N_{tot,i}}{NW_{Rd}} \right\rfloor$
- In the case of cache lines back writing: $\left\lfloor \frac{N_{tot,i}}{NW_{Wr}} \right\rfloor$

At each end of a temporal window $W$, the interconnect preempts the bus usage access in order to elect another bus user. Its internal scheduling policy is hardware specific. As a consequence, the more temporal windows are needed, the higher the interconnect impact is.

The interconnect overhead corresponding to the active wait of the memory bus access can be defined as a symbolic variable $\Phi$. Depending on the interconnect, this value can be upper-bounded to $\Phi_{max}$.

Depending on the number of interconnect temporal windows needed to achieve the data cache load, the $C_{prepare,i}$ cost would be (mod stands for modulo):

$$C_{prepare,i} = \left\lfloor \frac{N_{tot,i}}{NW_{Wr}} \right\rfloor \times (t_{CPU_{Wr}} + \Phi_{max}) + (N_{tot,i} \mod NW_{Wr}) \times (\Psi_{Wr} + t_{CPU_{Wr}}) + \Phi_{max} + t_{supp} \quad (7)$$

In the same way, the $C_{load,i}$ cost would be:

$$C_{load,i} = \left\lfloor \frac{N_{tot,i}}{NW_{Rd}} \right\rfloor \times (t_{CPU_{Rd}} + \Phi_{max}) + (N_{tot,i} \mod NW_{Rd}) \times (\Psi_{Rd} + t_{CPU_{Rd}}) + \Phi_{max} \quad (8)$$

### 6. Toward a macroscopic formalism of the impact on the scheduler execution

We now provide a simplified bound on $C_{cache,i}$ for a task $\tau_i$, when the temporal windows for read and write access are the same. In the same way, the CPU latency for accessing the bus ($t_{CPU}$) is bounded considering the maximum time between a read and write access:

$$t_{CPU} = \max(t_{CPU_{Rd}}, t_{CPU_{Wr}}) \quad (9)$$

In Equations 7 and 8, the usage of the last temporal window is measured taking into account that its duration may be smaller, depending on the number of the residual cache lines to read. Equation 10 provides a bound on $C_{cache,i}$.

The simplified formalism of the data cache impact on the scheduler execution is then:

$$C_{cache,i} \leq 2 \times \left( \left\lfloor \frac{N_{tot,i}}{NW} \right\rfloor \times (\max(t_{CPU}) + \Phi_{max} + W) \right) + t_{supp} \quad (10)$$

Because $C_{cache,i}$ and $C_{load,i}$ are upper-bounded, the SFPL algorithm is compliant to the Property 3.

### 7. Impact of the SFPL Cache Algorithm on a Real-time Event-Driven Fixed Priority Scheduling

#### A. Definition of a task model

The considered task model is a real-time sporadic task set. Each task $\tau_i$ has the following attributes:

- Its minimal period (or sporadicity interval) $T_i$
- Its relative deadline, $D_i$, with the following constraint: $D_i \leq T_i$
- Its WCET $C_i$, with the following constraint: $C_i \leq D_i$

In the following, we consider a fixed-priority event-driven preemptive scheduler. Considering a task $\tau_i$, we defined a set $H_i$, containing all the tasks having a priority higher than that of $\tau_i$. We now introduce $I_i$, a bound on the maximum number of preemptions a task $\tau_i$ can experience due to the activation of tasks in $H_i$. This bound is valid for any scenario and is considered in Equation 13 for the computation of the worst-case response time ok a task $\tau_i$ using cache memory.

$$I_i = \sum_{j \in H_i} \left\lceil \frac{D_i}{T_j} \right\rceil \quad (11)$$

We are aware of the pessimism of this bound compared to the exact number of preemptions. Nevertheless, this number is used to compute the duration of cache management overhead, function of $C_{cache,i}$. $C_{cache,i}$ is usually very small compared to the duration of a task. The pessimism is thus limited. Furthermore, in [15], the authors have shown that the synchronous scenario is not necessarily the one that maximizes the number of preemptions.

#### B. Measuring the predictable cache usage impact

Two new attributes are defined:

- $C_{i}^{op}$, which corresponds to the WCET of task $\tau_i$, using predictable data cache. This value is the result of a WCET computation based on the SFPL data cache usage.
- $R_{i}^{op}$, which corresponds to the worst-case response time to execute $\tau_i$ job with the SFPL cache algorithm. A simulation of the impact of the cache algorithm on $R_{i}^{op}$ is done in Section VI

As $C_{i}^{op}$ is an input attribute of a task, $R_{i}^{op}$ depends on the scheduling policy and the task set. When no cache algorithm is used, for a given task $\tau_i$, [10] have shown how to compute the worst-case response time of a task with Equation 12. This recursive equation converges if and only if $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$.

$$R_i = C_i + \sum_{j \in H_i} \left[ \frac{R_j}{T_j} \right] \times C_j \quad (12)$$
Using the SFPL cache algorithm, the worst-case response time of a task $\tau_i$ should take into account the cost $C_{cache,i}$ of cache management. This cost is due to the load and prepare costs (see Section 3.1) that occur at each task activation and preemption and the cost that results in the load and prepare of $\tau_i$ before starting $\tau_i$ after a preemption of a task in $H_i$ (at most $I_i$ preemptions for $\tau_i$).

Equation 12 is therefore updated as follows in Equation 13:

$$R^{op}_i = C^{op}_i + C_{cache,i} + \sum_{j \in H_i} \left[ \frac{R_j}{T_j} \right] \times (C^{op}_j + C_{cache,j}) + I_i \times C_{cache,i}$$  \hspace{1cm} (13)

Considering $P = LCM(T_1, \cdots T_n)$, a sufficient condition to guarantee the convergence of Equation 13 is that the workload requested by tasks in time interval $[0, P)$ should be less than or equal to $P$. We take into account a bound on the maximum number of preemptions assuming a preemption of cost $C_{cache,i}$ at each task activation, which is:

$$\forall i, \sum_{i=1 \rightarrow n} \left[ \frac{P}{T_i} \right] \times (C^{op}_i + C_{cache,i}) + \sum_{i=1 \rightarrow n} \left[ \frac{P}{T_i} \right] \times C_{cache,i} \leq P$$  \hspace{1cm} (14)

$$\Rightarrow \sum_{i=1 \rightarrow n} \frac{C^{op}_i + 2 \times C_{cache,i}}{T_i} \leq 1$$  \hspace{1cm} (15)

Equation 14 is an extension of the classical necessary condition required for the convergence of Equation 12.

We now study with an example the benefits of our data cache management approach.

VI. EMULATING THE SFPL DATA CACHE IMPACT

A. introduction

As described in Section IV the SFPL cache algorithm might not be efficient for every task profile and then its impact should be integrated in WCET measurements. Its impact may be extremely valuable, or seriously penalizing, depending on the task’s memory behavior.

As described in Equation 13, the SFPL cache algorithm overhead on a task $\tau_i$ depends on its priority and on the memory consumption of $\tau_i$. This memory consumption depends on two attributes that enable us to define the WCET $C^{op}_i$ obtained for a task $\tau_i$ when using the data cache: the worst-case number of memory accesses and the rate of cache-hit memory accesses, in $M_i$.

The shorter $C^{op}_i / C_i$ the more interesting data cache for $\tau_i$. The exact computation of $C^{op}_i$ is out of the scope of the paper. We show in subsection VI-B the worst-case response time $R^{op}_i$ obtained for task $\tau_i$ for different cases of $C^{op}_i / C_i$ ratios: 0.2, 0.4, 0.6 and 0.8.

Fig. 4. $R^{op}_i$ values for task $\tau_4$, depending on $C_{cache,i}$ cost

B. Example of the SFPL impact on a given task set

The following task set is used in order to simulate the SFPL cache impact:

<table>
<thead>
<tr>
<th>Task</th>
<th>$\tau_0$</th>
<th>$\tau_1$</th>
<th>$\tau_2$</th>
<th>$\tau_3$</th>
<th>$\tau_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>1st</td>
<td>2nd</td>
<td>3rd</td>
<td>4th</td>
<td>5th</td>
</tr>
<tr>
<td>$C_i$</td>
<td>0.01</td>
<td>0.05</td>
<td>0.1</td>
<td>0.2</td>
<td>0.7</td>
</tr>
<tr>
<td>$T_i$</td>
<td>0.1</td>
<td>1.5</td>
<td>1</td>
<td>1.1</td>
<td>2</td>
</tr>
<tr>
<td>$D_i$</td>
<td>0.05</td>
<td>0.2</td>
<td>0.5</td>
<td>0.7</td>
<td>1</td>
</tr>
</tbody>
</table>

This task set is not schedulable using a fixed priority event driven preemptive scheduling policy. This is shown in Figure 4, where $R_4$ is higher than $D_4$.

The simulation has been done using a first $C_{cache,i}$ value of 50$\mu$s, using a step of 50$\mu$s for each successive measurement. 20 points have been measured using a $C_{cache,i}$ cost from 50$\mu$s to 1ms.

Figure 4 compares the worst-case response of task $\tau_4$ for different cases of $C^{op}_i / C_i$ ratios (0.2, 0.4, 0.6 and 0.8) as a function of $C_{cache,i} / C_i$ ratio. Tasks $\tau_3$, $\tau_2$, $\tau_1$ and $\tau_0$ are schedulable even without using the SFPL algorithm.

Using the SFPL algorithm, this task set may be scheduled, depending on the impact of the cache algorithm on the tasks. Figure 4 show that when the SFPL data cache algorithm reduces $C_i$ with a factor 1.67 and above, the SFPL data cache algorithm always leads to a schedulable task set.

When the SFPL data cache algorithm reduces $C_i$ with only a factor 1.25, the SFPL data cache algorithm leads to a feasible task set up to $C_{cache,i} = \frac{1}{1000} \times C_i$ (see Figure 4).
VII. CONCLUSION

In this paper we have proposed a cross-layer approach for a secure and deterministic management of a L1 data cache by the scheduler. We have characterized the parameters defining the cost involved by the data cache management. We have then proposed a new deterministic and secure data cache management denoted SFPL. We have shown in an example how to integrate the cost of cache management in classical feasibility conditions for sporadic tasks when tasks are scheduled with a fixed-priority preemptive scheduling algorithm. The example shows that the SFPL algorithm may allow the scheduling of some unschedulable task sets in cache-disabled systems.

REFERENCES