Impact of DM-LRU on WCET: a Static Analysis Approach

Renato Mancuso
Boston University, USA
rmancuso@bu.edu

Heechul Yun
University of Kansas, USA
heechul.yun@ku.edu

Isabelle Puaut
University of Rennes 1/IRISA, France
isabelle.puaut@irisa.fr

Abstract

Cache memories in modern embedded processors are known to improve average memory access performance. Unfortunately, they are also known to represent a major source of unpredictability for hard real-time workload. One of the main limitations of typical caches is that content selection and replacement is entirely performed in hardware. As such, it is hard to control the cache behavior in software to favor caching of blocks that are known to have an impact on an application's worst-case execution time (WCET).

In this paper, we consider a cache replacement policy, namely DM-LRU, that allows system designers to prioritize caching of memory blocks that are known to have an important impact on an application's WCET. Considering a single-core, single-level cache hierarchy, we describe an abstract interpretation-based timing analysis for DM-LRU. We implement the proposed analysis in a self-contained toolkit and study its qualitative properties on a set of representative benchmarks. Apart from being useful to compute the WCET when DM-LRU or similar policies are used, the proposed analysis can allow designers to perform WCET impact-aware selection of content to be retained in cache.

2012 ACM Subject Classification Computer systems organization → Real-time systems; Theory of computation → Caching and paging algorithms

Keywords and phrases real-time, static cache analysis, abstract interpretation, LRU, deterministic memory, static cache locking, dynamic cache locking, cache profiling, WCET analysis

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2019.20

1 Introduction

Most modern embedded processors include cache(s) to improve average performance by reducing average memory access cost. However, a well-known downside of using caches is that it makes timing analysis difficult because software has little, if any, control over whether a certain memory block is in the cache or not, as it is determined by the hardware—the cache replacement policy and the state of the cache. This is problematic because analyzing precise and tight worst-case timing is necessary for real-time systems. While there are timing analysis techniques for well-known cache replacement policies [42], they cannot take advantage of programmer's insights (e.g., important data used in time-critical loops), potentially resulting in pessimistic timing.

On the other hand, a scratchpad memory is similar to a cache as it offers high-speed temporary storage for a processor, but the key difference is that it is entirely managed by software. For real-time systems, the fact that software, not hardware, has full control over its management is highly beneficial because accurate timing analysis is possible. However, the downside of scratchpad is that it is generally more difficult to use than cache due to its high programming complexity [3]. Alternatively, some cache designs support selective

cache locking, which enables programmers to lock certain cache-lines in the cache at a fine-granularity (typically a cache line) [2, 7, 13]. A locked cache-line stays in the cache until it is explicitly unlocked by the programmer, which guarantees predictable timing. However, because the cache size is limited, the programmer must carefully select which cache-lines to be locked [5, 40]. Dynamic cache-locking techniques [39, 48] can help alleviate the size limitation problem of static cache-locking, but at the cost of increased complexity (for selecting locked cache lines) and overhead (to change cache contents dynamically).

In this paper, we consider a new cache architecture, which can leverage programmers' high-level insights on access frequency of memory blocks, and propose an abstract interpretation-based static analysis method to reason on the worst-case execution time (WCET) of applications. Our approach is based on a new memory abstraction, called Deterministic Memory (DM). Deterministic Memory enables classification of a program's address space into two distinct memory types—DM and non-DM [10], where the DM type indicates predictability is more important while the non-DM type indicates average performance is more important. The DM abstraction allows effective and extensible software/hardware co-designs, some of which are demonstrated in the context of providing efficient hardware isolation in multi-core [10]. In this work, we instead focus on a single-core with a private cache, and study how static guarantees on cache hits/misses can be derived for a DM-aware LRU cache replacement policy, which we call DM-LRU.

We first describe the DM-LRU cache replacement algorithm, which is a single-core adaptation of the DM-aware cache initially proposed in [10]. Next, we generalize an abstract interpretation-based analysis for LRU caches to reason on the worst-case behavior of DM-LRU. We integrated DM-LRU support in Heptane [23], an academic static WCET analysis tool, in order to evaluate the effectiveness of DM-LRU in lowering tasks' WCET. Our results show that with DM-LRU WCET improvements up to 23.7% can be achieved, compared to vanilla LRU. The WCET improvements are comparable to static and dynamic cache locking techniques while significantly lowering programming complexity. Our contributions are as follows:

- We extend LRU abstract interpretation-based analysis to perform static WCET timing analysis for DM-LRU.
- We implement DM-LRU support in the Heptane static WCET analysis tool.
- We provide experimental evaluation results showing the WCET benefits and complexity reduction of the DM-LRU based approach.
- We propose a WCET-driven heuristic approach to select content to be preferentially cached using DM-LRU.

The remainder of the paper is organized as follows. Section 2 introduces necessary background on caches and the deterministic memory abstraction. Next, the DM-LRU policy is described in Section 3 and the proposed static timing analysis is described in Section 4. A comprehensive example on how to apply the proposed analysis is presented in Section 5. Comparison and differences with cache locking techniques are briefly highlighted in Section 6, while the WCET of a set of representative benchmarks is evaluated in Section 7. Section 8 discuss related work and we conclude in Section 9.

2 Background

In this section, we provide necessary background on memory abstractions, cache replacement algorithms, and cache timing analysis.

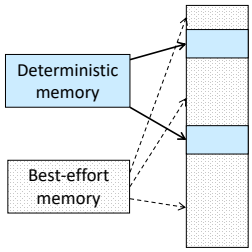

Figure 1 High-level application's memory view, where DM and BE memory coexist.

2.1 Deterministic Memory Abstraction

Traditionally, operating systems and hardware have provided a simple uniform memory abstraction to applications. While the simple abstraction is convenient for programmability, its downside is that programmer's insights on memory characteristics (e.g., time-criticality of certain data structures) cannot be explicitly expressed to enable better resource management.

Recently, a new memory abstraction, called Deterministic Memory abstraction, was proposed to explore the possibilities of more expressive memory abstractions [10]. In essence, the abstraction allows a programmer to associate (tag) a single bit of information to each memory block in the system, which classifies the memory block as either "deterministic memory" (DM) or "best-effort memory" (BE). Figure 1 shows an example address space of a task using both deterministic and best-effort memory. In [10], the memory tagging is implemented at the page granularity, although more fine-granularity tagging is also possible (e.g., [45]).

Once a task's memory blocks are tagged, the information can then be used by the operating system and the hardware to apply different resource management policies depending on the memory tag information. In [10], the DM abstraction is used to achieve hardware isolation among the cores in multicore, focusing on effective isolation of shared cache and DRAM.

2.2 DM-LRU Cache Replacement Policy

In this paper, we consider a deterministic memory-aware *private* cache design and show how such a design enables tighter static WCET cache timing analysis. We assume the cache controller has a mean to distinguish whether a certain cache-line corresponds to deterministic memory or best-effort one. This can be implemented as an additional bit in the auxiliary tag store of each cache-line, as in [10], or as a set of separately located architectural hardware range registers as in [27]. The cache implements an extended least recently used (LRU) cache replacement algorithm, which defines two *eviction classes* using the DM/BE abstractions and applies LRU-based replacement to DM lines and to BE lines separately. Allocation of a DM line can cause eviction of a BE line, but the opposite is not allowed. Note that prior work that implements a similar cache replacement policy exists [27]. In this paper, we call the extended LRU as Deterministic Memory-aware Least Recently Used, or DM-LRU for short. A more formal definition of DM-LRU is given in Section 3.

Figure 2 illustrates the difference between traditional LRU, DM-LRU, and static locking. For simplicity, the example considers a single set of a 4-way set-associative cache. In the first step, only a and d are cached, and 0 lines are allocated for DM blocks under DM-LRU. Moreover, blocks b and e are set as DM blocks under DM-LRU, and pre-allocated in cache in case of static locking. The figure tracks the evolution of the cache state for the same access sequence b, c, d, a, e, b. A miss for a DM block triggers an increase of the number of ways allocated for the DM class. This is depicted in step 2 (miss on b) and 6 (miss on e).

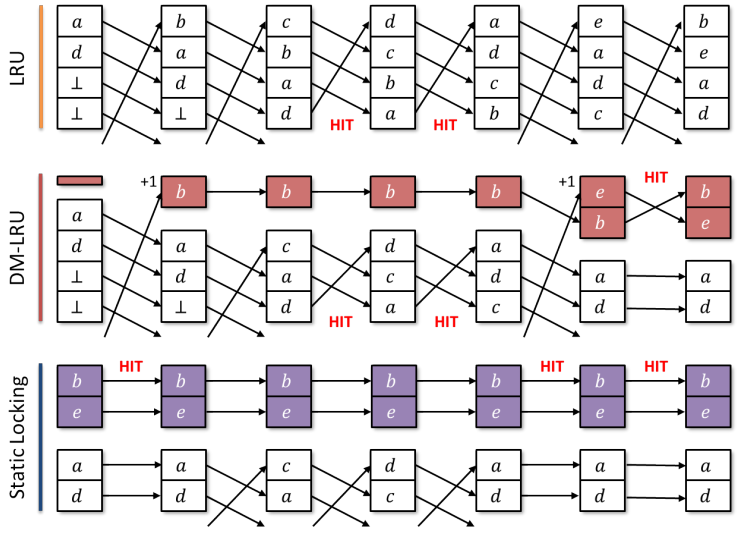

Figure 2 Comparison between traditional LRU, DM-LRU, and a statically locked LRU cache over the same access pattern b, c, d, a, e, b, where b and e are DM memory blocks, or statically locked.

Traditional LRU simply ignores the DM/BE tag of the considered memory blocks. First, note that DM-LRU results in fewer misses compared to LRU, as the DM marked memory block b was not evicted by the best-effort memory accesses. Also note that while realizing the same number of hits in the example compared to static locking, two important remarks are required. First, the figure does not include the time spent to prefetch and lock the b and e blocks. Second, static locking causes additional misses for non-locked blocks compared to DM-LRU. This exemplifies the on-demand nature of DM-LRU, which is able to retain in cache blocks as they become needed during a task's execution. We discuss the analogies and differences between DM-LRU and static/dynamic locking more extensively in Section 6.

Intuitively, it is thanks to the on-demand allocation and differential treatment of DM memory blocks that DM-LRU enables tighter worst-case cache timing analysis, as we show in the rest of the paper.

2.3 Cache Analysis via Abstract Interpretation

In this work, we extend abstract interpretation-based analysis to reason on the hit/miss classification of memory accesses when a DM-LRU cache controller is implemented in hardware. Analysis via abstract interpretation was originally proposed for LRU caches [11] and better formalized and extended to FIFO and Pseudo-LRU in [41, 14]. An excellent survey on the topic was proposed in [32]. We reuse the notation in [14, 32], while some details are omitted due to space constraints. Since this work focuses on a DM-aware extension of LRU, we introduce some of the background related to abstract interpretation-based LRU analysis.

Imagine taking a snapshot of the cache state at a given point in time. In this case, one could highlight the *state* of the cache in terms of: (i) which blocks are currently in cache, and (ii) what is the *age* of each block. In LRU, the age of a block, say block a, captures the number of memory accesses (to other blocks than a) that were performed since the last access to a. For instance, in the six steps in Figure 2, the LRU age for a is in the following sequence: 0, 1, 2, 3, 0, 1, and 2. If a has an LRU age greater than or equal to the number of ways (4 in our example), then a is not cached.

If the ages of all the cached blocks are known, the cache is in a **concrete state**. From a

concrete state, it is possible to produce a new concrete state that follows each new memory access (state update), as shown in Figure 2. In a typical program, however, execution may follow different paths. This means that at a given point in time, multiple concrete states are possible, depending on the execution path taken by the program in its control-flow graph (CFG).

Instead of keeping track of all the possible concrete states at any point of the CFG, abstract interpretation keeps track of two main pieces of information: (i) the upper-bound and (ii) the lower-bound on the age of any memory block among all the possible concrete states. Analysis on the age upper-bound and lower-bound is carried on separately. The former is referred to as *must-analysis*, while the latter goes under the name of *may-analysis*. A state that summarizes the upper-bound (resp., lower-bound) of each block in a set of possible concrete states is called an **abstract state**. For instance, consider a *must-analysis* abstract state of the form: $\bar{q} = [\{\}, \{a, b\}, \{\}, \{d, e\}]$. This corresponds to all the concrete states where blocks a, b have age at most 1, and d, e at most 3. The full *concretization* of $\bar{q}$ is the set: $\{[a, b, d, e], [b, a, d, e], [a, b, e, d], [b, a, e, d]\}$. Similarly, consider the *may-analysis* abstract state $\underline{q} = [\{\}, \{\}, \{a, b\}, \{\}]$. A concretization of $\underline{q}$ is the set $\{[\bot, \bot, \bot, \bot], [\bot, \bot, a, \bot], [\bot, \bot, \bot, a], [\bot, \bot, b, \bot], [\bot, \bot, \bot, b], [\bot, \bot, a, b], [\bot, \bot, b, a]\}$, where $\bot$ is a generic unknown block.

Given a *must*-analysis abstract state, it is possible to determine —i.e., *classify*— a memory access as *always-hit* (H). These are accesses that result in hits regardless of the path taken in the CFG. Similarly, given a *may*-analysis abstract state, it is possible to perform classification of *always-miss* memory accesses. If neither classification applies, the block is simply non-classified (NC). NC, often indicated as $\top$, represents the case in which some execution paths lead to a miss while others lead to a hit for the same memory access.

Note that for architectures without timing anomalies [31, 20], *must*-analysis is sufficient to safely compute the WCET of an application. In fact in this case NC accesses can be simply treated as misses. We developed and implemented both *must*- and *may*-analysis for DM-LRU, but we hereby focus in greater detail on *must*-analysis. Additional details about *may-analysis* are provided in the appendix.

3 Cache Model and Terminology

In this section we discuss the cache model adopted to represent the behavior of DM-LRU, and we introduce key concepts required to follow the proposed abstract interpretation analysis.

3.1 DM-LRU Model

Algorithm 1 shows the full pseudo-code of the DM-LRU cache replacement algorithm. The algorithm is defined for a generic A-way set-associative cache with S sets. The index of a set is indicated with $s \in \{0, \ldots, S-1\}$. In the algorithm, $DetMask_s$ denotes the bitmask of the set s's cache lines that contain deterministic memory. Consider a DM request ($DM = 1$) that resulted in a cache miss—see step 1 or 6 in Figure 2. The algorithm first tries to evict a BE cache line, if such a line exists (Line 3-4). This also causes an additional bit to be asserted in the $DetMask_s$ bitmap. If no BE can be evicted (i.e., all lines are deterministic ones), it chooses one of the deterministic lines (the older one in the LRU stack) as the victim (Line 6). On the other hand, consider the case where a BE memory block is requested ($DM \neq 1$), resulting in a miss—steps 1 and 2 in Figure 2. DM-LRU evicts one of the best-effort cache lines, but not any of the deterministic cache lines (Line 9).

We assume a single-core, single-level set-associative cache. We indicate with A the associativity of the cache. Since DM-LRU operates independently on each set, it is possible to describe our analysis on a single set without loss of generality. Hereafter, we consider

```
   Input  : DetMask_s - deterministic ways of Set s
   Input  : A - cache associativity
   Output : victim - the victim way to be replaced or NULL if no replacement possible
 1 if DM == 1 then
 2     if (¬DetMask_s) ≠ NULL then
           // evict a best-effort line first
 3         victim = LRU(¬DetMask_s)
 4         DetMask_s |= 1 ≪ victim
 5     else
           // evict a deterministic line
 6         victim = LRU(DetMask_s)
 7     end
 8 else
 9     if (¬DetMask_s) ≠ NULL then
           // evict a best-effort line
10         victim = LRU(¬DetMask)
11     else
           // no BE line can be allocated
12         victim = NULL
13     end
14 end
15 return victim
```

Algorithm 1: Deterministic memory-aware cache line replacement algorithm.

a single cache set. At any point in time, D is the number of cache lines allocated to DM memory blocks for the considered cache set. D is the number of bits set to "1" in the $DetMask_s$ for the set under analysis. We indicate with B the number of lines that have not been allocated for DM memory. It holds that $D + B = A$. Note that if $D < A$, and a DM line that is currently not cached as a DM line is accessed, then the new DM line is allocated and D is increased by one. This may trigger the eviction of the least recently used BE block, as per Algorithm 1.

3.2 Terminology and notations

We indicate with $\mathcal{B}$ the set of memory blocks that map to the cache set under analysis. A generic memory block $b^{CL} \in \mathcal{B}$ is comprised of an address b and an *eviction class* $CL = \{DM, BE\}$. The set of all the possible concrete states of a DM-LRU cache is denoted as Q_{DM-LRU_A}, where each state $q \in Q_{DM-LRU_A}$ is defined as follows:

$$q := \{D, [b_0^{DM}, \ldots, b_{D-1}^{DM}], [b_D^{CL}, \ldots b_{A-1}^{CL}]\}, \tag{1}$$

where $D \in [0, A]$ and $b_i^{CL} \in \mathcal{B}$. Note that the first D cache lines are allocated as DM cache lines, hence these are necessarily DM memory blocks. The remaining $A - D$ blocks are currently allocated BE memory blocks. Throughout this paper we will use the shorthand notation $b_i \in \mathcal{B}$ for blocks whose eviction class is obvious from context or unimportant. For blocks allocated as BE, we assume BE class unless specified otherwise.

An important concept is the **age** of a memory block under DM-LRU, defined as follows.

▶ **Definition 1** (DM-LRU Age). *The age of a DM memory block a^{DM} is defined as the number of distinct DM blocks accessed since the last access to a^{DM}; the age of a BE memory block b is set to the current value of D whenever b^{BE} is accessed. It is then defined as $D + K$, where K is the number of misses to DM blocks, or accesses to distinct BE blocks since the last access to b^{BE}.*

Following Definition 1, the index of a given block $b_i^{CL} \in q$ is also the age of the block in DM-LRU. The age of a block b_i^{DM} allocated as DM can increase if: (1) a new DM line is allocated (with age 0); or (2) a line b_j^{DM} already allocated as DM with age greater than b_i is accessed. Conversely, the age of a BE block b_i^{BE} can increase if: (1) a new DM line is allocated (with age 0); (2) a new BE line is allocated (with age D); or (3) a line b_j^{BE} already in cache with age greater than b_i^{BE} is accessed.

Also note that Definition 1 remains consistent for the case in which a block b^{BE} is accessed but cannot be allocated because all the sets have been reserved for DM lines. This phenomenon goes under the name of *DM takeover*, and can be resolved by imposing a hard cap on the maximum number of DM lines that can be allocated. The analysis for a DM-LRU with an allocation cap is almost identical to an unrestricted DM-LRU, and only introduces uninteresting subcases. For simplicity, we hereby focus on the analysis for unrestricted DM-LRU. We demonstrate that preventing DM takeover is indeed necessary and beneficial in Section 7.

4 DM-LRU Analysis

In this section we detail our abstract interpretation-based analysis [14, 32] for DM-LRU, i.e. when the cache controller implements the policy defined in Algorithm 1. We discuss *must*-analysis in detail. As previously mentioned, *may*-analysis is not strictly required for architectures without timing anomalies. As such we only provide the intuition behind it and defer the details to the appendix. We do not provide a persistence analysis for DM-LRU. Persistence analysis is useful to determine if memory accesses inside loops can result in hits after the first iteration. Instead, for our evaluations, we unroll the first iteration of each loop, i.e., we perform *virtual unrolling, virtual inlining* (VIVU) [34, 32].

4.1 Must-analysis

Must-analysis is performed considering abstract cache states. In this case, *must-analysis* keeps track of the upper bound on the number of allocated DM blocks indicated with $D \in \{0, \dots, A\}$, and the upper-bound on the DM-LRU age of each addressable memory block $b \in \mathcal{B}$. The abstract domain $DMLru_A^{\sqsubseteq}$ is defined as:

$$DMLru_A^{\sqsubseteq} := \{0, \dots, A\} \times \mathcal{B} \to \{0, \dots, A-1, \infty\}. \tag{2}$$

Intuitively, the domain associates a *current eviction class* (DM or BE) and an age upper bound ($0, \dots, A$ or ∞) to a memory block $b \in \mathcal{B}$ mapping to the set under analysis. We use the notation $\bar{q}(b)$ to indicate the upper-bound on the age of b in $\bar{q}$. To represent a generic abstract state $\bar{q} \in DMLru_A^{\sqsubseteq}$ we use a compact notation that highlights the distinction between DM and BE allocations. For instance, the notation

$$\bar{q} = [\{\}, \{a, b\}], [\{c\}, \{d\}] \in DMLru_A^{\sqsubseteq} \tag{3}$$

denotes an abstract state $\bar{q}$ where $D \leq 2, B \geq 2, A = 4$. Hence, blocks a and b have upper-bound $\bar{q}(a) = \bar{q}(b) = 1$ on their DM-LRU age. Similarly, c, d are BE blocks with $\bar{q}(c) = 2$ and $\bar{q}(d) = 3$, respectively.

Given an abstract state $\bar{q} \in DMLru_A^{\sqsubseteq}$, the Boolean operator $DM^{\sqsubseteq}(\bar{q}, b)$ returns *true* only if the block $b \in \mathcal{B}$ must exist as a DM-allocated block in $\bar{q}$. Formally

$$DM^{\sqsubseteq}(\bar{q}, b^{CL}) := \begin{cases} true & \text{if } CL = DM \wedge \bar{q}(b) < \infty \\ false & \text{otherwise.} \end{cases} \tag{4}$$

For instance, considering $\bar{q}$ defined as in Equation 3, we obtain $DM^{\sqsubseteq}(\bar{q}, a) = true$, $DM^{\sqsubseteq}(\bar{q}, d) = false$, and so on. We use the simpler notation $DM^{\sqsubseteq}(b)$ when the state is implicit. The operator $BE^{\sqsubseteq}(\bar{q}, b)$ is simply defined as $BE^{\sqsubseteq}(\bar{q}, b) := \neg DM^{\sqsubseteq}(\bar{q}, b)$. To prevent additional clutter in our notation, $DM^{\sqsubseteq}(\bar{q}, b^{DM})$ evaluates to *true* if and only if the DM block b^{DM} *must* be allocated in cache in $\bar{q}$. As such, if the generic DM block b^{DM} has an upper-bound on its DM-LRU age greater than $A-1$, then $BE^{\sqsubseteq}(\bar{q}, b^{DM}) = true$.

An *abstract state transformer* for the $DMLru_A^{\sqsubseteq}$ domain is an operator that takes in input an abstract state $\bar{q} \in DMLru_A^{\sqsubseteq}$ and any number of additional parameters, and returns in

output a transformed state $\bar{q}' \in DMLru_A^{\sqsubseteq}$. We consider and define two abstract transformers for $DMLru_A^{\sqsubseteq}$: an update transformer $U^{\sqsubseteq}(\bar{q}, a)$, and a join transformer $J^{\sqsubseteq}(\bar{q}, \bar{p})$. We use the operator $\lambda b.$ to represent an age update operation carried on each $b \in \mathcal{B}$ when considering a transformation from state $\bar{q}$ to $\bar{q}'$. This operator can be formally defined as:

$$\lambda b.\ f(\bar{q}(b)) := \forall b \in \mathcal{B}, \bar{q}'(b) \leftarrow f(\bar{q}(b)) \tag{5}$$

Must-analysis Update

The update abstract transformer for the *must*-analysis $U^{\sqsubseteq}(\bar{q}, a)$ is used to go from an initial abstract state, to a new abstract state after a new memory access has been performed. $U^{\sqsubseteq}(\bar{q}, a)$ takes in input an initial abstract state $\bar{q}$ and a memory block $a \in \mathcal{B}$, and returns the abstract state that results from accessing a. For ease of notation, we split the definition of $U^{\sqsubseteq}$ in two parts: the logic that corresponds to the update operation when a DM block a^{DM} is accessed, indicated with $U_D^{\sqsubseteq}$; and the update transformation when a BE block a^{BE} is accessed, namely $U_B^{\sqsubseteq}$. $U_D^{\sqsubseteq}$ is defined in Equation 6.

$$U_D^{\sqsubseteq}(\bar{q}, a^{DM}) :=$$

$$D' \leftarrow \begin{cases} D+1 & \text{if } D < A \wedge BE^{\sqsubseteq}(a) \quad \text{(a.1)} \\ D & \text{if } D = A \vee DM^{\sqsubseteq}(a) \quad \text{(a.2)} \end{cases}$$

$$\lambda b. \begin{cases} 0 & \text{if } b = a \quad \text{(b)} \\ \bar{q}(b) & \text{if } b \neq a \wedge \left|\left| \begin{array}{ll} BE^{\sqsubseteq}(b) \wedge DM^{\sqsubseteq}(a) & \text{(c.1)} \\ DM^{\sqsubseteq}(b) \wedge \bar{q}(a) \leq \bar{q}(b) & \text{(c.2)} \\ BE^{\sqsubseteq}(b) \wedge BE^{\sqsubseteq}(a) \wedge \bar{q}(a) \leq \bar{q}(b) & \text{(c.3)} \end{array} \right. \\ \bar{q}(b)+1 & \text{if } b \neq a \wedge \bar{q}(a) > \bar{q}(b) \wedge \left|\left| \begin{array}{ll} DM^{\sqsubseteq}(b) \wedge \bar{q}(b) < D'-1 & \text{(d.1)} \\ BE^{\sqsubseteq}(b) \wedge BE^{\sqsubseteq}(a) \wedge \bar{q}(b) < A-1 & \text{(d.2)} \end{array} \right. \\ \infty & \text{if } b \neq a \wedge \bar{q}(a) > \bar{q}(b) \wedge \left|\left| \begin{array}{ll} DM^{\sqsubseteq}(b) \wedge \bar{q}(b) \geq D'-1 & \text{(e.1)} \\ BE^{\sqsubseteq}(b) \wedge BE^{\sqsubseteq}(a) \wedge \bar{q}(b) \geq A-1 & \text{(e.2)} \end{array} \right. \end{cases} \tag{6}$$

Here, D' (B', resp.) is the new value of D (B, resp.) after the update. The conditions following the $||$ operator are to be considered in logical "or" with each other.

The update abstract transformer $U_B^{\sqsubseteq}$ for a best-effort memory access a can be defined as follows:

$$U_B^{\sqsubseteq}(\bar{q}, a^{BE}) :=$$

$$\lambda b. \begin{cases} D & \text{if } b = a \wedge D < A \quad \text{(a)} \\ \bar{q}(b) & \text{if } b \neq a \wedge \left|\left| \begin{array}{l} DM^{\sqsubseteq}(b) \\ BE^{\sqsubseteq}(b) \wedge \bar{q}(a) \leq \bar{q}(b) \end{array} \right. \quad \text{(b)} \\ \bar{q}(b)+1 & \text{if } b \neq a \wedge BE^{\sqsubseteq}(b) \wedge \bar{q}(a) > \bar{q}(b) \wedge \bar{q}(b) < A-1 \quad \text{(c)} \\ \infty & \text{if } \left|\left| \begin{array}{l} b = a \wedge D \geq A \\ b \neq a \wedge BE^{\sqsubseteq}(b) \wedge \bar{q}(a) > \bar{q}(b) \wedge \bar{q}(b) \geq A-1 \end{array} \right. \quad \text{(d)} \end{cases} \tag{7}$$

To clarify the update operation, consider the abstract state $\bar{q} = [\{\}, \{b, f\}], [\{c\}, \{d\}]$, where $D = 2$. Assume that deterministic block a^{DM} is accessed, which has age upper-bound ∞ in $\bar{q}$, to obtain $\bar{q}' = U^{\sqsubseteq}(\bar{q}, a) = U_D^{\sqsubseteq}(\bar{q}, a)$. First, the value of D' is computed as

$D' = D + 1 = 3$ (a.1); next, b, f both satisfy the condition $\bar{q}(a) > \bar{q}(b) = \bar{q}(f) = 1$. Moreover, we have that $DM^{\sqsubseteq}(b) = DM^{\sqsubseteq}(f) = true$, and that $\bar{q}(b) = \bar{q}(f) = 1 < D' - 1 = 2$. This corresponds to condition (d.1) in Equation 6. Hence, the age of b, f in the resulting state is $\bar{q}'(b) = \bar{q}'(f) = 2$. Similarly, block c and d satisfy condition (d.2) and (e.2), respectively. The resulting updated abstract state is: $\bar{q}' = [\{a\}, \{\}, \{b, f\}], [\{c\}]$.

An example for the abstract transformer $U^{\sqsubseteq}_{\bar{B}}$ defined in Equation 7 is provided in Section 5.

▶ **Theorem 2** (Correctness of *must*-analysis update)**.** *Consider a generic abstract state* $\bar{p} = U^{\sqsubseteq}(\bar{q}, a^{CL})$ *obtained from the* must*-analysis update state transformer when accessing a generic block* a^{CL} *from an initial abstract state* $\bar{q}$*. Then for any block* $b \in \mathcal{B}$*,* $\bar{p}(b)$ *is an upper-bound on the DM-LRU age of* b*.*

Proof Sketch. A proof can be constructed by considering two main sub-cases: (1) when $CL = DM$ for the block being accessed; and (2) the case when $CL = BE$. Due to space constraints, we provide an intuition for the former case, as the latter follows from the same reasoning. When considering $CL = DM$, the new state $\bar{p}$ is obtained as $\bar{p} = U^{\sqsubseteq}_{\bar{D}}(\bar{q}, a^{DM})$, as per Equation 6.

First let us consider the rule on the update of D. If $\bar{q}(a) = \infty$ then a is not necessarily in cache and accessing a increases the upper-bound on the number of allocated DM blocks, as long as the associativity A has not been exceeded, i.e. $D < A$. In this case, note that $BE^{\sqsubseteq}(\bar{q}, a) = true$ and condition Equation 6 (a.1) applies. D does not change in any other case (a.2). After the update, block a will have age upper-bound equal to 0 (b).

Next, consider all the blocks $b \neq a$ that had age upper-bound of infinity in $\bar{q}$ — i.e. $\bar{q}(b) = \infty$, and $BE^{\sqsubseteq}(\bar{q}, b) = true$. When a is accessed, their age upper-bound should not change. If $\bar{q}(a) = \infty$ then condition (c.3) applies. If $\bar{q}(a) \neq \infty$ then $DM^{\sqsubseteq}(\bar{q}, a) = true$ and condition (c.1) applies.

Furthermore, consider all the blocks $b^{DM}, b \neq a$ that must be allocated as DM blocks in $\bar{q}$, i.e. such that $DM^{\sqsubseteq}(\bar{q}, b) = true$. If $\bar{q}(a) = \infty$, the upper-bound on their DM-LRU age will have to increase by 1 (d.1). If however the value of $\bar{q}(b) + 1$ exceeds the updated value of D, namely D', then the block may be evicted and the new upper-bound on its DM-LRU age $\bar{p}(b) = \infty$ (e.1). The same cases apply when $\bar{q}(a) < \infty$ and $\bar{q}(a) > \bar{q}(b)$.

On the other hand, if a has an age upper-bound that is same as or lower than b's, i.e. $\bar{q}(a) \leq \bar{q}(b)$, then a concrete state where DM-LRU age of a is strictly larger than that of b cannot exist. As such, the upper-bound on the DM-LRU age of b will not change, as per condition (c.2).

Lastly, consider all the blocks $b^{BE}, b \neq a$ that must be allocated as BE blocks in $\bar{q}$, i.e. such that $BE^{\sqsubseteq}(\bar{q}, b) = true$ and $\bar{q}(b) < \infty$. The only case in which $\bar{q}(a) > \bar{q}(b)$ is if $\bar{q}(a) = \infty$. When a is accessed, the upper-bound on the age of b will have to increase by 1 (d.2), unless by doing so the associativity A is exceeded. In the latter case, $\bar{p}(b) = \infty$ (e.2). ◀

Must-analysis Join

The join abstract transformer $J^{\sqsubseteq}(\bar{q}, \bar{p})$ is used to compute a new abstract state at the merging point of two or more execution paths. There are strong similarities between the transformer defined hereby and what used in traditional LRU *must*-analysis [14]. At a high level, the joined state will consider as *must*-cached only those blocks in the intersection of the joining states, each with the maximum age in any of the two states. For the new state, D is taken as the maximum between the value of D in the joining states. Equation 8 formalizes the $J^{\sqsubseteq}(\bar{q}, \bar{p})$ abstract transformer:

$$J^{\sqsubseteq}(\bar{q}, \bar{p}) := D \leftarrow \max\{D_{\bar{q}}, D_{\bar{p}}\}, \lambda b. \max\{\bar{q}(b), \bar{p}(b)\} \tag{8}$$

If we were to join $\bar{q} = [\{\}, \{b, f\}], [\{c\}, \{d\}]$ with $\bar{q}' = [\{a\}, \{\}, \{b, f\}], [\{c\}]$, the resulting state would be $\bar{q}'' = J^{\sqsubseteq}(\bar{q}, \bar{q}') = [\{\}, \{\}, \{b, f\}], [\{c\}]$.

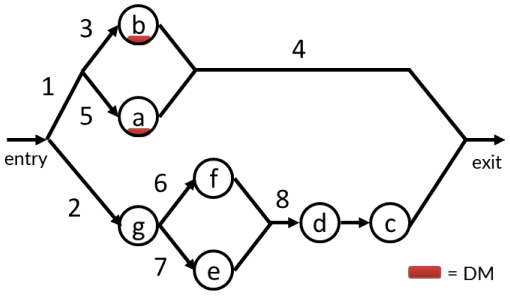

Figure 3 Fragment of process CFG. At the end of the fragment, all the cache blocks in the figure *may* be cached.

▶ **Theorem 3** (Correctness of *must*-analysis join). *Consider an abstract state* $\bar{s} = J^{\sqsubseteq}(\bar{q}, \bar{p})$ *obtained from the* must*-analysis join state transformer from two initial abstract states* $\bar{q}$ *and* $\bar{p}$*. Then for any block* $b \in \mathcal{B}$*,* $\bar{s}(b)$ *is an upper-bound on the DM-LRU age of* b*.*

Proof Sketch. A proof can simply follow from the definition of the $J^{\sqsubseteq}$ operator in Equation 8. By hypothesis $\bar{q}$ and $\bar{p}$ carry the upper-bound on the age of a generic block b along two disjoint execution sub-paths. After the two sub-paths join, the maximum between $\bar{q}(b)$ and $\bar{p}(b)$ is a safe upper-bound on the DM-LRU age of b in the resulting abstract state $\bar{s}$. Moreover, an upper-bound on the number of allocated DM blocks in $\bar{s}$ is the maximum between $D_{\bar{q}}$ and $D_{\bar{p}}$. ◀

Must-analysis Classification

Every time an access is performed, it is possible to classify a memory access using a classification function that will either return M for cache miss, H for cache hit, or $\top$ in case neither M nor H classification can be made given the current abstract state. In order to classify memory accesses, for a given $\bar{q}$ abstract state we define two helper sets $\bar{\mathcal{D}}$ and $\bar{\mathcal{B}}$ representing the deterministic and best-effort memory blocks that have finite upper bound on their DM-LRU age:

$$\bar{\mathcal{D}} := \{b^{CL} \in \mathcal{B} \mid CL = DM \wedge \bar{q}(b) < \infty\}$$
$$\bar{\mathcal{B}} := \{b^{CL} \in \mathcal{B} \mid CL = BE \wedge \bar{q}(b) < \infty\} \tag{9}$$

The classification function of the *must* analysis is defined as:

$$C^{\sqsubseteq}(\bar{q}, a^{CL}) := \begin{cases} H & \text{if } \bar{q}(a) < \infty & \text{(a)} \\ M & \text{if } \left\| \begin{array}{l} CL = DM \wedge a \notin \bar{\mathcal{D}} \wedge |\bar{\mathcal{D}}| = D \\ CL = BE \wedge a \notin \bar{\mathcal{B}} \wedge |\bar{\mathcal{B}}| = B \end{array} \right. & \text{(b)} \\ \top & \text{otherwise} & \text{(c)} \end{cases} \tag{10}$$

We provide a complete step-by-step example on how *must*-analysis can be applied to an application's CFG in Section 5.

4.2 May-analysis

The complete *may*-analysis is provided in the appendix (Section 10). We hereby provide a sketch of the approach followed in the analysis.

The goal of *may*-analysis is to track the lower-bound on the age of memory blocks. Given a *may*-analysis abstract state it is possible to classify a memory access as always leading to a miss. Let us consider the example in Figure 3 and reason on the lower bound on the age of

each block for a 4-way fully associative cache. For block a^{DM}, the best case is represented by the execution pattern 1-5-4. In this case, the block has DM-LRU age 0. A similar situation occurs for block b^{DM} and path 1-3-4. For blocks f and g, the best-case is represented by the paths 2-6-8, and 2-7-8, respectively. This leads the two blocks to have a lower-bound of 2 on their DM-LRU age. Similarly, blocks c, d, and g have lower-bound 0, 1, and 3, respectively.

We can represent the resulting *may*-analysis state obtained following the derivation above as: $[\{a,b\}],[\{c\},\{d\},\{f,e\},\{g\}]$. What happens if another access to a occurs after path 4 and 8 join? Then the best-case for block b is still 1-3-4, but its age lower-bound will be 1. At the same time, because at least one DM block was allocated regardless of the taken path, the minimum lower-bound on the age of any BE block has to be 1. Also note that regardless of the execution path taken, block g will be evicted. The result is the following *may*-analysis abstract state: $[\{a\},\{b\}],[\{\},\{c\},\{d\},\{f,e\}]$.

5 Analysis Example

In this section we provide a description of how DM-LRU *must-analysis* can be applied to a CFG once the target of each memory access is known. The original CFG of the considered C program code generated by the Heptane tool is shown in Figure 4. The program consists of a single loop with four iterations, where the first iteration has been unrolled. The program accesses 7 memory locations. These are $\mathcal{B}=\{a,b,c,d,e,f,g\}$ and are visible in the various basic blocks as operands of load/store instructions.

Figure 5 shows the same CFG as in Figure 4, but where only basic blocks in which memory accesses are being performed are kept. Moreover, basic blocks with multiple memory accesses are depicted as sequences of blocks, each with a single memory access. The nodes are annotated with their corresponding abstract states. We apply *must*-analysis starting from the entry node a. We compare the behavior of traditional LRU analysis and DM-LRU when blocks a and f have been declared as DM. We consider a fully-associative cache with 4 ways. For DM-LRU analysis, the cache state before the first access $\bar{q}_0=[],[\{\},\{\},\{\},\{\}]$ $(D_0=0)$; for LRU analysis it is $[\{\},\{\},\{\},\{\}]$. Under DM-LRU, when block a^{DM} is accessed, the performed operation is $\bar{q}_1=U^{\sqsubseteq}(\bar{q}_0,a_{DM})=U^{\sqsubseteq}_{D}(\bar{q}_0,a)$. Following Equation 6, we have $D_1=D_0+1$, then condition (a) is satisfied by a, all the other blocks $b\in\mathcal{B}$ satisfy condition (d.2). As such, we have $\bar{q}_1=[\{a\}],[\{\},\{\},\{\}]$, as reported in the figure.

Let us now follow the upper branch with access sequence $d\rightarrow e\rightarrow g$ (all of them are best-effort memory accesses). For each memory access, we apply Equation 7 to obtain a new abstract state. After accessing e, the resulting abstract state is: $\bar{q}_2=[\{a\}][\{e\},\{d\},\{\}]$. Let us now show more clearly how we obtain $\bar{q}_3=U^{\sqsubseteq}(\bar{q}_2,g^{BE})=U^{\sqsubseteq}_{B}(\bar{q}_2,g)$, when we next access g. Considering all blocks in $\mathcal{B}$ and using Equation 7 we know: block a satisfies condition (b.1) and its age remains the same; b and c satisfy (d.2) and their age remains ∞; block e satisfies (c) and its age increases by 1, from 1 to 2; the age of block d increases from 2 to 3; and finally, block g (being accessed) satisfies condition (a) and its age is set to $D_2=1$. The final state is $\bar{q}_3=[\{a\}],[\{g\},\{e\},\{d\}]$, as shown in the figure above node g. The same procedure applies to the lower branch of the CFG, and we obtain the state $\bar{q}_4=[\{a\}],[\{c\},\{b\},\{\}]$, after we access c.

Before accessing f, we need to join states $\bar{q}_3$ and $\bar{q}_4$ derived above. In this case, we apply Equation 8 to obtain $\bar{q}_5=J^{\sqsubseteq}(\bar{q}_3,\bar{q}_4)$. It follows that $D_5=1$. Moreover, the only block present in both states $\bar{q}_3$ and $\bar{q}_4$ is a. All other blocks in $\mathcal{B}$ will have age ∞ in $\bar{q}_5$. As such we have $\bar{q}_5=[\{a\}],[\{\},\{\},\{\}]$. Next, accessing f^{DM} yields $\bar{q}_6=U^{\sqsubseteq}_{D}(\bar{q}_5,f)=[\{f\},\{a\}],[\{\},\{\}]$, as shown in the figure. This is because $D_6=D_5+1$, and because a satisfies condition (c.1) in Equation 6. The same reasoning can be applied to obtain the remaining states depicted in the figure.

Consider now the state $\bar{q}_6$ and apply the *must*-analysis classifier before accessing a^{DM},

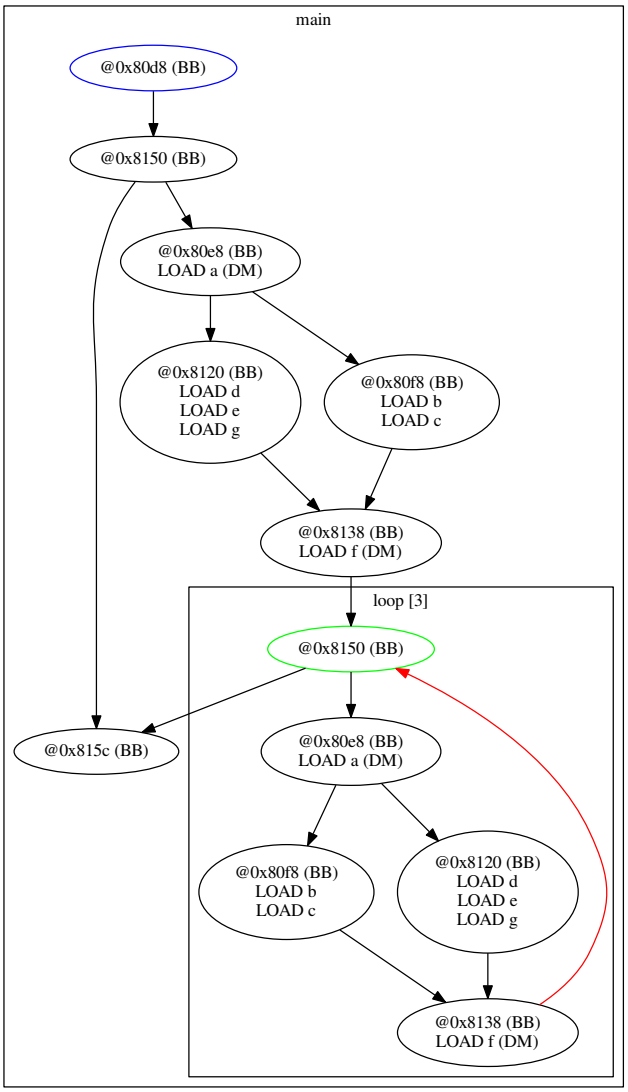

Figure 4 Original CFG of considered example as rendered by the Heptane tool, with annotated memory accesses (LOAD). Note that VIVU has been performed on the loop.

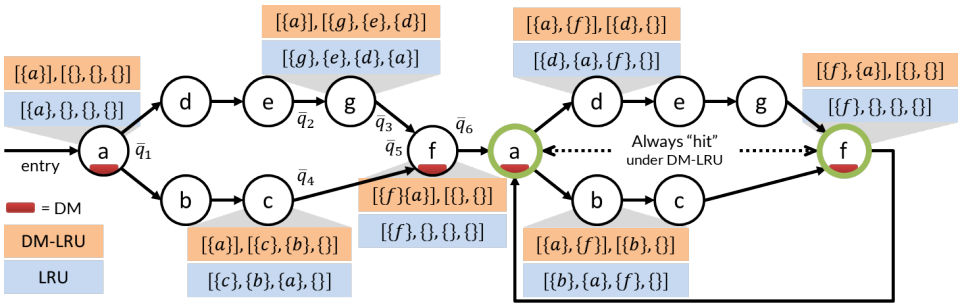

Figure 5 An example of *must* analysis under DM-LRU (orange states), compared to traditional LRU (blue states). If a and f are marked as DM, their accesses inside the loop can be classified as always hits.

i.e. compute $C^{\sqsubseteq}(\bar{q}_6, a)$ as in Equation 10. First, the sets $\bar{\mathcal{D}}_6$ and $\bar{\mathcal{B}}_6$ can be computed using Equation 9 as $\bar{\mathcal{D}}_6 = \{a, f\}$, and $\bar{\mathcal{B}}_6 = \{\}$. Hence condition (a) is satisfied and access to a is classified as H (hit). Conversely, no access can be classified as hit under LRU.

6 Analogies and Differences with Cache Locking

Cache locking refers to a technique where cache blocks that are deemed important for an application's timing are *pinned* (locked) in cache. Similar to DM-LRU, cache locking represents a way to partially override the best-effort replacement strategy offered by the hardware. And like DM-LRU, specialized hardware support is required to perform locking. With respect to WCET analysis, the big advantage provided by cache locking is that all those accesses for locked cache blocks can be immediately classified as hits. While cache locking was commonly supported in previous-generation embedded systems, the current trend in embedded SoCs is toward cache controllers that offer little or no management primitives.

Despite the strong similarities, some profound differences exist between cache locking and DM-LRU. Leveraging cache locking implies injection of additional logic —in either the application, the compiler, and/or the OS— to perform a series of prefetch&lock operations. On the contrary, a system featuring a DM-LRU cache only requires that memory blocks are tagged appropriately at task load time.

In case of static locking, prefetch&lock can be performed at initialization time. As such, the extra logic required to perform locking does not impact the task's WCET. Conversely, with dynamic cache locking, the locked cache content is modified at runtime. Depending on the available hardware support, this operation may not be directly possible in user-space, requiring instead a costly switch to kernel-space. Regardless, an online prefetch&lock routine can pollute the rest of the cache, resulting in overheads that may largely offset any benefit. In other words, additional system-level assumptions are required to make a meaningful comparison with dynamic locking. For this reason, we do not compare DM-LRU against dynamic locking.

Interestingly enough, however, the proposed DM-LRU analysis can be re-used to analyze dynamic locking schemes if additional system parameters are available. In a nutshell, consider a 4-way fully associative cache. Next, assume that the locked content is switched whenever a given branch in the CFG is taken. Then, consider the case where the new content to be locked is comprised of blocks a, b, c. A special node on the branch can be added with associated a modified update abstract transformer $Lock^{\sqsubseteq}$. This is such that the resulting *must*-analysis abstract state $\bar{q}$ after the update is: $\bar{q} = Lock^{\sqsubseteq}(\{a, b, c\}) = [\{a\}, \{b\}, \{c\}][\{\}]$.

7 Evaluation

The DM-LRU analysis presented in the previous sections provides a way to understand how the WCET of applications varies as memory blocks addressed in applications are declared as DM. We now apply DM-LRU analysis to a set of realistic embedded benchmarks. In this section, we first briefly describe our implementation. Next, we investigate three main aspects: (1) what is the degree of WCET improvement that can be achieved via DM-LRU when compared to LRU? (2) Is there an advantage in imposing a limit to the number of DM lines that can be simultaneously allocated, i.e. in preventing DM takeover? (3) how does DM-LRU compares to static cache locking?

In our evaluation we focus on the degree of WCET improvement that DM-LRU can provide compared to LRU. However, because supporting DM-LRU implies changes to the hardware cache memory and controller, it is also important to determine if a DM-LRU implementation can be efficiently carried out. In short, only one additional bit to distinguish between DM and BE lines is required per cache line. Additionally, compact changes[1] are required to the cache controller to restrict victim selection based on the classification (DM or BE) of a new line being allocated. No additional logic is required to appropriately set the DM bits at line fetch. Additional considerations on the incurred hardware costs to support DM memory are also provided in [10].

7.1 Implementation

We have implemented support for DM-LRU inside a state of the art static WCET analysis tool, namely Heptane [23]. Heptane implements Implicit Path Enumeration Technique (IPET) [29] and performs analysis for many cache architectures: e.g., LRU, FIFO, Pseudo-LRU, multi-level non-inclusive caches, and shared caches. In our setup, we consider a single-level of cache, divided into an instruction (I) cache, and a data (D) cache. For simplicity, we assume in all our experiments that both caches are selected to have the same number of sets and ways. Heptane supports two architectures: ARMv7 and MIPS. The ARMv7 target was used for this paper.

We have modified the Heptane tool to support two variants of DM-LRU, as well as static locking. Most importantly, we have extended the support for abstract interpretation-based cache analysis to implement the *must*- and *may*-analysis presented in the previous sections. The performed changes allow backward compatibility with the original set of policies supported by the tool. Next, we have integrated the logic to differentiate between BE memory and DM memory. For this purpose, we have added a table of DM addresses—the DM Table—that can be specified by an external tool, mimicking the selection of DM blocks by the OS at binary load time. Furthermore, we have added appropriate logic in Heptane to output per-memory-block statistics in terms of references, hits, and misses, as computed during WCET analysis. These statistics are then used to build a DM-block selection heuristic. Finally, we have modified Heptane's CFG extraction routines to perform VIVU—i.e., to recursively peel the first iteration of every loop.

We have developed and employed a simple heuristic to determine which memory block-s/addresses should be marked as DM and inserted into the DM Table. The heuristic initially performs WCET analysis without selecting any DM line. Next, it analyzes the per-memory-block statistics and selects as DM the block with the largest number of misses. At this point, WCET analysis is performed again with the new DM Table containing a single entry. Using

[1] Whenever a line eviction has to occur, the DM/BE bits of all the lines in the considered set form a bitmask. Victim selection for a BE access is then performed by excluding all those lines that have a bit set to 1 in the DM bitmask.

the per-memory-block statistics of the latest run, a new DM block is selected in addition to the previously selected block. The same steps are performed until no more addresses can be selected as DM. Note that when no lines are selected as DM, the behavior of the cache is indistinguishable from vanilla LRU. Similarly, when the entire memory of an applications is selected as DM, no differences exist with LRU. In practice, however, we saw no differences between DM-LRU and LRU when more than $3 \times S \times A$ lines are selected as DM, where S and A is the number of sets and ways of the cache, respectively. In our experiments, we acquired analytical results for a number of DM lines in the range $[1, 3 \times S \times A]$.

7.2 Setup

We compare two variants of DM-LRU and static cache locking against LRU. A description of the three scenarios follows.

1. **Unrestricted DM-LRU ("DM-nolim")**: in this variant, no restriction is imposed on the maximum number of cache sets that can be reserved for DM lines. It follows that the only constraint for the allocation of DM lines is the cache associativity itself. The analysis for unrestricted DM-LRU is the one presented in the previous sections.
2. **Limited DM-LRU ("DM-cap")**: in this variant, a hard cap in the maximum number of ways is imposed on the expansion of DM lines. This represents a solution to the aforementioned problem of *DM takeover*. Imposing a cap of 0 makes DM-cap to be identical to vanilla LRU. Similarly, imposing a *cap* of A makes DM-cap to be identical to DM-nolim. In our experiments, we explore all the possible values of *cap* in the range $[1, A]$.
3. **Static locking ("Static")**: this case is used to draw a comparison between the considered DM-LRU variants and static locking. In case of static locking, selection of lines to statically allocate is performed following the same heuristic used for DM lines selection. Similar to DM-cap, we impose how many ways can be dedicated to statically locked content (locked ways). The maximum number of allocated line is then $S \times locked$. Note that the main performance difference between DM-cap and Static lies in the additional flexibility that DM-cap provides. In DM-cap, in fact, more lines than $S \times cap$ can be selected, while it is not allowed in static locking.

For all the considered variants, we explore a number of cache configurations. Specifically, we vary the associativity A of the I/D caches in the set $\{2, 4, 8, 16\}$. We vary the number of cache sets S such that $S \in \{2, 4, 8, 16, 32\}$. As previously mentioned, for DM-nolim and DM-cap, we progressively select up to $3 \times S \times A$ DM lines following the heuristic described above. In each system instance, we perform WCET analysis using the modified Heptane tool. Then, we keep track of which configuration—S, A, DM-lim *cap*, *locked* ways, number of DM lines—for each of the three scenarios leads to the best reduction in WCET compared to the vanilla LRU case.

For our benchmarks, we use a subset of realistic benchmarks from the Mälardalen suite [19]. Unfortunately, vanilla HEPTANE is not able to perform WCET analysis for some of the benchmarks in the suite. As such, our evaluation only includes those benchmarks that are correctly handled by HEPTANE. Notably, the aforementioned changes to implement DM-LRU analysis did not impact the set of benchmarks that can be correctly analyzed by the tool.

7.3 Results

Figure 6 provides an overview of the obtained results. A cluster of bars is provided for each of the considered benchmarks. Reading the plot from top to bottom, the first bar corresponds to the WCET under LRU. All the results in the figure are normalized to the LRU case. The

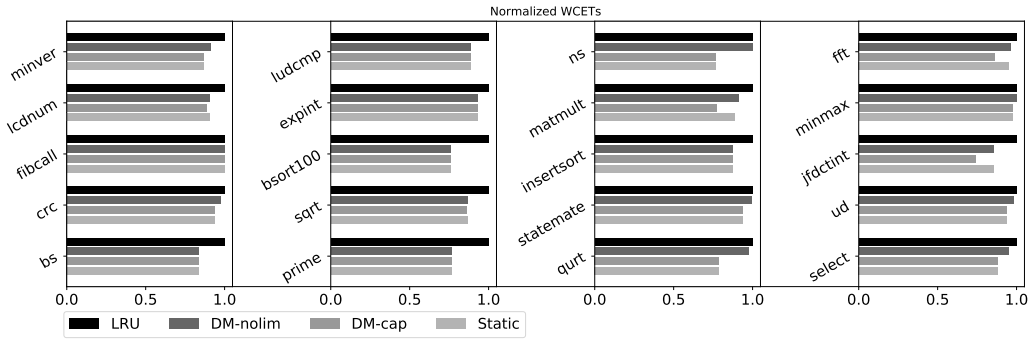

Figure 6 Computed WCETs for vanilla LRU (LRU), unrestricted DM (DM-nolim), DM limited to a subset of ways (DM-cap), and static locking (Static).

second bar represents the best WCET improvement that was observed under DM-nolim. The WCET improvement is calculated as: $\frac{WCET_{\text{DM-nolim}}}{WCET_{\text{LRU}}}$, where the WCETs under DM-nolim and under LRU are obtained in the same system configuration. A similar calculation was performed to derive the remaining two bars, i.e. for the DM-lim and Static cases.

What emerges from the plot is that in 16 out of 20 cases, DM-nolim is able to achieve WCET reductions compared to vanilla LRU. Notably, in case of `bsort100` and `prime`, it is possible to achieve a WCET reduction of around 23.73% and 23.47%, respectively. It can also be noted that DM-cap outperforms DM-nolim. Moreover, DM-cap performs generally better than static locking. For instance, the best WCET reduction achieved via static locking for the `jfdctint` benchmark is 26.09% under DM-cap (with a $S = 4$, $A = 8$, 19 DM lines, $cap = 1$). But the best WCET reduction under static locking is only 14.64%, which is achieved for a cache with parameters $S = 2$, $A = 16$ and 15 ways occupied by statically locked lines. Similar results can be observed for the benchmarks `matmult` and `fft`.

The reason for the performance improvement that can be obtained with DM-cap is twofold. On the one hand, the problem of DM takeover is solved. This prevents the case that all the accesses to BE lines result in misses. On the other hand, for applications that exhibit changes in working sets, static locking can be sub-optimal. Conversely, under DM-cap, is is possible to mark lines belonging to different working sets as DM. In this case, at working set changes over time, those DM lines belonging to a previous working set will be naturally evicted, without suffering pollution from BE lines.

A more detailed overview of the obtained experimental results is provided in Table 1. In the table, the first column reports the name of the benchmark under analysis. If multiple configurations are of interest, multiple rows are shown for a given benchmark. The second column reports the cache configuration in terms of sets S and ways W for the results on each row. Next, the WCET obtained with LRU is reported in the following column, followed by the best WCET obtained for the same configuration under DM-nolim and the relative improvement (due to the space limitation, the number of DM lines that were selected has been omitted in the table.) Similarly, the best result obtained under DM-cap is reported next, and the value of *cap* under which the result was achieved is reported in the adjacent column. Finally, the last two columns report the WCET (and the relative improvement) for static locking with the given cache configuration and number of locked ways reported in the last column.

8 Related Work

Memory Tagging and Hardware Support. In this work, we assume that hardware

Benchmark	$S \times A$	LRU	DM-nolim	DM-cap	*cap*	Static	*locked*
bs	2×2	6613	5513 (-16.63%)	5513 (-16.63%)	2	5513 (-16.63%)	2
crc	4×2	2492320	2425920 (-2.66%)	2330620 (-6.49%)	1	2330620 (-6.49%)	1
fibcall	2×2	14191	14191 (-0.00%)	14191 (-0.00%)	1	14191 (-0.00%)	1
lcdnum	4×2	16291	14791 (-9.21%)	14791 (-9.21%)	2	14791 (-9.21%)	2
	2×4	16191	16191 (-0.00%)	14391 (-11.12%)	2	15291 (-5.56%)	2
minver	4×2	126558	115758 (-8.53%)	109958 (-13.12%)	1	109958 (-13.12%)	1
prime	2×4	611425	467925 (-23.47%)	467925 (-23.47%)	3	467925 (-23.47%)	3
sqrt	2×4	54983	47552 (-13.52%)	47252 (-14.06%)	3	52552 (-4.42%)	2
	2×4	54983	47552 (-13.52%)	47252 (-14.06%)	3	47583 (-13.30%)	6
bsort100	2×2	12434700	9484580 (-23.72%)	9484580 (-23.72%)	1	9484580 (-23.72%)	1
expint	2×4	759551	709651 (-6.57%)	709651 (-6.57%)	4	709651 (-6.57%)	4
ludcmp	16×2	638233	564633 (-11.53%)	564633 (-11.53%)	2	564633 (-11.53%)	2
qurt	2×8	217555	212160 (-2.48%)	173755 (-20.13%)	6	173755 (-20.13%)	6
	4×4	217555	220355 (–1.29%)	171155 (-21.33%)	3	171155 (-21.33%)	3
statemate	2×2	616218	612918 (-0.54%)	576718 (-6.41%)	1	576718 (-6.41%)	1
	8×8	383718	382818 (-0.23%)	359118 (-6.41%)	6	359118 (-6.41%)	6
insertsort	2×2	80126	70126 (-12.48%)	70126 (-12.48%)	1	70126 (-12.48%)	1
matmult	2×2	7191620	6568220 (-8.67%)	5555520 (-22.75%)	1	6391620 (-11.12%)	1
ns	4×2	193481	193481 (-0.00%)	193481 (-0.00%)	1	193481 (-0.00%)	1
	2×2	530781	534781 (–0.75%)	406681 (-23.38%)	1	406681 (-23.38%)	1
select	4×2	170766	162266 (-4.98%)	157966 (-7.50%)	1	157966 (-7.50%)	1
	2×4	170766	162866 (-4.63%)	150966 (-11.59%)	3	150966 (-11.59%)	3
ud	4×2	226843	223243 (-1.59%)	223243 (-1.59%)	2	225243 (-0.71%)	2
	2×2	302443	354143 (–17.09%)	283843 (-6.15%)	1	283843 (-6.15%)	1
jfdctint	2×16	150234	128734 (-14.31%)	111034 (-26.09%)	2	128234 (-14.64%)	15
	4×8	150234	130334 (-13.25%)	111134 (-26.03%)	1	147134 (-2.06%)	1
	4×8	150234	130334 (-13.25%)	111134 (-26.03%)	1	130334 (-13.25%)	7
minmax	2×2	4034	4034 (-0.00%)	4034 (-0.00%)	1	4034 (-0.00%)	1
	2×4	4034	4034 (-0.00%)	3934 (-2.48%)	1	4034 (-0.00%)	1
fft	32×2	1683830	1623930 (-3.56%)	1623430 (-3.59%)	1	1623430 (-3.59%)	1
	4×4	2488230	2494360 (–0.25%)	2140830 (-13.96%)	1	2443230 (-1.81%)	1
	4×4	2488230	2494360 (–0.25%)	2140830 (-13.96%)	1	1716630 (-4.62%)	2

Table 1 Summary of notable experimental results under four strategies: (1) vanilla LRU ("LRU"); (2) unrestricted DM-LRU ("DM-nolim"); (3) restricted DM-LRU ("DM-cap"); and (4) static locking ("Static").

allows us to encode (tag) extra information (e.g., importance) on memory locations at a fine-granularity. The basic idea of memory tagging has first explored in the security community, to prevent memory corruption (e.g., buffer overflow) [6, 38] and to enforce data flow integrity [45] and capability protection [51]. Efficient hardware designs for word-granularity single-bit and multi-bit memory tagging have been investigated [24] and several real SoC prototypes have been built [45, 4], demonstrating the feasibility. In the real-time systems community, several works explored the use physical memory address based differentiated hardware designs (mostly cache) in a more coarse-grained manner (i.e., memory segments, page, and task granularity). Kumar et al, proposed a criticality-aware cache design, called Least Critical (LC), which includes a memory criticality-aware extension to LRU replacement policy [27]. The LC cache's replacement policy is similar to the replacement policy we assumed in this work (Algorithm 1), while its memory tagging mechanism, which uses a fixed number of specialized range registers, does not allow flexible and fine-grained memory tagging. Therefore, our static analysis method can be directly applicable to analyze the LC cache. PRETI [28] also proposes a criticality-aware cache design but it focuses on shared cache for SMT hardware, while we focus on private caches. More recently, OS-level page-granularity memory tagging and supporting multicore architecture designs (including a new cache design) have been explored to provide efficient hardware isolation (incl. cache isolation) in multicore [10].

Static Cache Analysis. There exists a broad literature on static cache analysis [32, 50]. With respect to existing literature, this work is closely related to approaches that propose abstract interpretation-based cache analysis. This approach was initially proposed in [1, 12]. These works illustrate LRU analysis and hit/miss classification using *may*- and *must*-analysis. The work in [12] also proposes a persistence analysis based on abstract states, which was found to be unsafe and for which a fix was proposed in [8, 25]. We base our DM-LRU extension on the *may*- and *must*-analysis proposed in [12], but use the improved formalization in [14]. In order to perform access classification in case of loops we use an approach similar to *virtual inlining & virtual unrolling* (VIVU) originally proposed in [34]. A large body of works has considered cache replacement policies other than LRU. These include FIFO [14, 15, 18], MRU [17], Pseudo-LRU [16]. Comparatively less work has been produced to analyze non-inclusive [36, 21] as well as inclusive [22] multi-level caches. With respect to these works, the proposed methodology set itself apart because it focuses on the impact on the WCET of designer-driven selection of frequently accessed memory blocks. In this sense, proposed approach can be used to analyze caches that support the definition of touch-and-lock cache lines, under the assumption that no more than A blocks are simultaneously locked on any set, where A is the associativity of the cache.

Cache Locking and Scratchpad Memory. Some COTS cache designs [2, 7, 13] support selective cache locking, which prevents evictions for certain programmer selected cache-lines. Exploting the feature, various static and dynamic cache locking schemes for both instruction and data caches have been investigated [5, 40, 39, 48, 35]. In [47, 48], for instance, cache locking statements are inserted in the task's execution flow at compilation time, when the uncertainty about the memory locations being accessed negatively impacts the static WCET analysis. Some recent works combined cache locking with cache partitioning to improve task WCET in multicore [30, 43, 33]. As an alternative to cache, scratchpad memory has received significant attention in the real-time systems community for its predictability benefits [46, 9, 49, 44]. More recently, a technique called invalidation-driven allocation (IDA) [26] was proposed to achieve the same level of determinism of a locked cache in spite of lack of hardware-assisted locking primitives. IDA can be used as long as precise invariants on the size of an application's working set hold. To overcome its high programming complexity, however, many researchers proposed various compiler-based techniques. In [44], for instance, a sophisticated compiler-based technique is proposed to break each task into intervals and at the beginning of each interval, the required memory blocks of the interval are prefetched onto a scratchpad memory via a DMA controller without blocking the task execution. Dividing a task into a sequence of well-defined memory and computation phases was originally proposed in [37, 52]. In both cache locking and scratchpad memory based techniques, a common limitation is the overhead of explicitly executing additional instructions (prefetch, lock/unlock, or data movement to/from scratchpad). Furthermore, these additional instructions are context sensitive in the sense that they must be executed before actual accesses occur, and if they are executed too early, they can negatively impact both performance and WCET. In contrast, our approach is context insensitive in the sense that, once DM blocks are flagged, actual allocation and replacement are automatically performed by the hardware (cache controller) without additional instruction execution overhead.

9 Conclusion

In this paper, we presented the DM-LRU cache replacement policy and proposed an abstract interpretation-based analysis for DM-LRU. We implemented the proposed analysis and DM-LRU support in the Heptane static WCET analysis tool. Using the Heptane, we evaluated the WCET impacts of our DM-LRU based approach on a number of benchmarks. The results show that our DM-LRU approach can provide lower task WCETs with less performance

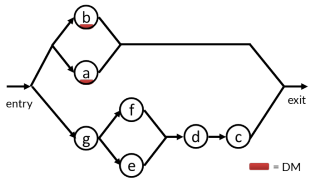

Figure 7 Fragment of process CFG that leads to abstract DM-LRU state $\underline{q} = [\{a,b\}],[\{c\},\{d\},\{e,f\},\{g\}]$.

overhead and programming complexity, compared to the standard LRU and cache locking based approaches.

Acknowledgements

We are especially grateful to Daniel Grund for making his research thesis [14] promptly available to us. This research is supported in part by NSF CNS 1718880. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF.

10 Appendix: May Analysis

In the DM-LRU analysis framework, *may*-analysis is once again performed by considering abstract cache states. Recall that *may*-analysis keeps track of the lower-bound on the age of each addressable memory block. There are a number of differences compared to the analytical tools used for *must* analysis. In *may*-analysis it is necessary to keep track of both $D \in \{0,\ldots,A\}$ and $B \in \{0,\ldots,A\}$. Here, the meaning of D and B changes. In this case, D represents the maximum lower-bound of any possibly cached DM block. Conversely, B captures the minimum lower-bound on the DM-LRU age of any BE block. It may be the case that $B + D > A$ in order to correctly abstract the age lower-bound resulting from multiple execution paths. It must hold however that $A \leq D + B \leq 2A$. It follows that the abstract domain for *may*-analysis $DMLru^{\sqsupseteq}_A$ is defined as:

$$DMLru^{\sqsupseteq}_A := \{0,\ldots,A\} \times \{0,\ldots,A\} \times \mathcal{B} \rightarrow \{0,\ldots,A-1,A\}. \tag{11}$$

An abstract state $\underline{q} \in DMLru^{\sqsupseteq}_A$ is then represented as two sets of memory blocks, for instance: $\underline{q} = [\{a,b\}],[\{c\},\{d\},\{e,f\},\{g\}] \in DMLru^{\sqsupseteq}_A$. In this example, we have $D = 1, B = 4, A = 4$. It follows that the upper-bound on the number of DM memory blocks is 1, and that blocks a and b have at least DM-LRU age 0, and *may* be marked as deterministic blocks. On the other hand, c is a best-effort memory block with DM-LRU age at least 0. It should not come as a surprise that in some states $D + B > A$. Consider the execution depicted in Figure 7 that produces $\underline{q}$. When execution reaches the end of the figure, there could be 0 or 1 DM blocks allocated in cache. Hence the upper-bound on the number of DM blocks has to be $D = 1$. On the other hand, the upper bound on the number of BE blocks is $B = 4$.

The operator $DM^{\sqsupseteq}(\underline{q},a)$ takes an abstract state $\underline{q}$ and a block a, and returns *true* if a *may* be allocated as a DM block in $\underline{q}$. For ease of notation, we simply use $DM^{\sqsupseteq}(a)$ when the considered abstract state is obvious. We define the operator $DM^{\sqsupseteq}(\underline{q},a)$ as follows:

$$DM^{\sqsupseteq}(\underline{q},b^{CL}) := \begin{cases} true & \text{if } CL = DM \wedge \underline{q}(b) < A \\ false & \text{otherwise.} \end{cases} \tag{12}$$

The update abstract transformer $U^{\sqsupseteq}_D$ for a DM memory access a can be defined as follows:

$U^{\sqsupseteq}_D(\underline{q},a) :=$

$$D' \leftarrow \begin{cases} D+1 & \text{if } D < A \wedge BE^{\sqsupseteq}(a) \\ D+1 & \text{if } D < A \wedge \exists x^{DM} \neq a \ : \ \underline{q}(x) = \underline{q}(a) = D-1 \\ D & \text{otherwise} \end{cases}, \tag{13}$$

$$B' \leftarrow \begin{cases} B-1 & \text{if } B > 0 \wedge \underline{q}(a) \geq A - B \\ B & \text{if } B = 0 \vee \underline{q}(a) < A - B \end{cases}, \tag{14}$$

$$\lambda b. \begin{cases} 0 & \text{if } b = a \quad \text{(a)} \\ \underline{q}(b) & \text{if } b \neq a \wedge \left|\left| \begin{array}{l} DM^{\sqsupseteq}(b) \wedge \underline{q}(a) < \underline{q}(b) \\ BE^{\sqsupseteq}(b) \wedge \underline{q}(a) < A - B \end{array} \right. \quad \text{(b)} \\ \underline{q}(b)+1 & \text{if } b \neq a \wedge \left|\left| \begin{array}{l} DM^{\sqsupseteq}(b) \wedge \underline{q}(a) \geq \underline{q}(b) \wedge \underline{q}(b) < D' - 1 \\ BE^{\sqsupseteq}(b) \wedge \underline{q}(a) \geq A - B \wedge \underline{q}(b) < A - 1 \end{array} \right. \quad \text{(c)} \\ A & \text{if } b \neq a \wedge \left|\left| \begin{array}{l} DM^{\sqsupseteq}(b) \wedge \underline{q}(a) \geq \underline{q}(b) \wedge \underline{q}(b) \geq D' - 1 \\ BE^{\sqsupseteq}(b) \wedge \underline{q}(a) \geq A - B \wedge \underline{q}(b) \geq A - 1 \end{array} \right. \quad \text{(d)} \end{cases} \tag{15}$$

Where D' (B', resp.) is the new value of D (B, resp.) after the update. Similarly, the update abstract transformer $U^{\sqsupseteq}_{\overline{B}}$ for a best-effort memory access a can be defined as follows:

$$U^{\sqsupseteq}_{\overline{B}}(\underline{q}, a) := \tag{16}$$

$$\lambda b. \begin{cases} A - B & \text{if } b = a \quad \text{(a)} \\ \underline{q}(b) & \text{if } b \neq a \wedge \left|\left| \begin{array}{l} DM^{\sqsupseteq}(b) \\ BE^{\sqsupseteq}(b) \wedge \underline{q}(a) < \underline{q}(b) \end{array} \right. \quad \text{(b)} \\ \underline{q}(b)+1 & \text{if } b \neq a \wedge BE^{\sqsupseteq}(b) \wedge \underline{q}(a) \geq \underline{q}(b) \wedge \underline{q}(b) < A - 1 \quad \text{(c)} \\ A & \text{if } b \neq a \wedge BE^{\sqsupseteq}(b) \wedge \underline{q}(a) \geq \underline{q}(b) \wedge \underline{q}(b) \geq A - 1 \quad \text{(d)} \end{cases} \tag{17}$$

To clarify how the $U^{\sqsupseteq}$ operation transforms a given state , consider the abstract state $\underline{q} = [\{a, b\}], [\{c\}, \{d\}, \{e, f\}, \{g\}]$, where $D = 1, B = 4$. Assume that DM block h is accessed, whose DM-LRU age is currently A or higher. First, the value of D' (B', resp.) is computed as $D' = D + 1$ ($B' = B - 1$, resp.); next, $\{a, b\}$ both satisfy the fourth condition in $U^{\sqsupseteq}_{\overline{D}}$ —Equation 15, first case of (c); block c, d, e and f satisfy the fifth condition; g the seventh. The resulting updated abstract state is: $\underline{q}' = [\{h\}, \{a, b\}], [\{\}, \{c\}, \{d\}, \{e, f\}]$. Note that in the resulting state $B = 3$, hence the least lower-bound on any BE block is $A - B = 1$.

***May*-analysis Join:** The join abstract transformer for DM-LRU *may*-analysis is symmetric to the join abstract transformer used for DM-LRU *must*-analysis. The joined state will contain all the blocks in the union of the joining states, each with the minimum age in any of the two states. Furthermore, D is taken as the maximum between the value of D in the joining states. Similarly, B is taken as the maximum between the value of B in the joining states. As such, after a join, it always holds that $D + B \leq 2A$. Equation 18 formalizes the $J^{\sqsupseteq}(\bar{q}, \bar{p})$ abstract transformer:

$$J^{\sqsupseteq}(\bar{q}, \bar{p}) := \ D \leftarrow \max\{D_{\bar{q}}, D_{\bar{p}}\}, B \leftarrow \max\{B_{\bar{q}}, B_{\bar{p}}\}, \lambda b. \min\{\bar{q}(b), \bar{p}(b)\}. \tag{18}$$

To clarify the join operation, consider the state $\underline{q} = [\{a, b\}], [\{c\}, \{d\}, \{e, f\}, \{g\}]$ obtained in Figure 7, and the state $\underline{q}' = [\{h\}, \{a, b\}], [\{\}, \{c\}, \{d\}, \{e, f\}]$ obtained as $\underline{q}' = U^{\sqsupseteq}(\underline{q}, h^{DM})$ (i.e. by accessing the DM block h). If we were to join $\underline{q}$ with $\underline{q}'$, the resulting state would be $\underline{q}'' = [\{a, b, h\}, \{\}], [\{c\}, \{d\}, \{e, f\}, \{g\}]$.

***May*-analysis Classification:** It is possible to classify a memory access using a classification function that will either return M for cache miss, or $\top$ in case access to a memory block cannot be guaranteed to be a miss given the current abstract state. The classification function of the *may* analysis is defined as:

$$C^{\sqsupseteq}(\underline{q}, a^{CL}) := \begin{cases} M & \text{if } \underline{q}(a) = A \\ \top & \text{otherwise.} \end{cases} \tag{19}$$

References

1. M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *Proceedings of the Third International Symposium on Static Analysis*, SAS '96, pages 52–66, Berlin, Heidelberg, 1996. Springer-Verlag.
2. ARM. *PL310 Cache Controller Technical Reference Manual, Rev: r0p0*, 2007.
3. R. Banakar, S. Steinke, B. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Int. Symp. Hardware/Software Codesign (CODES+ISSS)*, pages 73–78. ACM, 2002.
4. A. Bradbury, G. Ferris, and R. Mullins. Tagged memory and minion cores in the lowrisc soc. *Memo, University of Cambridge*, 2014.
5. M. Campoy, A. P. Ivars, and J. Busquets-Mataix. Static use of locking caches in multitask preemptive real-time systems. In *Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, pages 1–6, 2001.
6. S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Dependable Systems and Networks (DSN)*, pages 378–387. IEEE, 2005.
7. N. Corp. Variable SMP – A Multi-Core CPU Architecture for Low Power and High Performance. Technical report, Nvidia, 2011.
8. C. Cullmann. Cache persistence analysis: Theory and practice. *ACM Trans. Embed. Comput. Syst.*, 12(1s):40:1–40:25, Mar. 2013.
9. J.-F. Deverge and I. Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 179–190. IEEE, 2007.
10. F. Farshchi, P. K. Valsan, R. Mancuso, and H. Yun. Deterministic memory abstraction and supporting multicore system architecture. In *Euromicro Conf. Real-Time Syst. (ECRTS)*. IEEE, 2018.
11. C. Ferdinand, F. Martin, R. Wilhelm, and M. Alt. Cache behavior prediction by abstract interpretation. *Sci. Comput. Program.*, 35(2-3):163–189, Nov. 1999.
12. C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-timesystems. *Real-Time Syst.*, 17(2-3):131–181, Dec. 1999.
13. Freescale. *e500mc Core Reference Manual*, 2012.
14. D. Grund. *Static Cache Analysis for Real-Time Systems: LRU, FIFO, PLRU*. epubli, 2012.
15. D. Grund and J. Reineke. Precise and Efficient FIFO-Replacement Analysis Based on Static Phase Detection. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 155–164, July 2010.
16. D. Grund and J. Reineke. Toward precise PLRU cache analysis. In *International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 23–35, 2010.
17. N. Guan, M. Lv, W. Yi, and G. Yu. WCET Analysis with MRU Caches: Challenging LRU for Predictability. In *Real Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, April 2012.
18. N. Guan, X. Yang, M. Lv, and W. Yi. FIFO Cache Analysis for WCET Estimation: A Quantitative Approach. In *Design, Automation and Test in Europe (DATE)*, pages 296–301, San Jose, CA, USA, 2013. EDA Consortium.
19. J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In B. Lisper, editor, *Procedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 137–147, Brussels, Belgium, July 2010. OCG.
20. S. Hahn and J. Reineke. Design and analysis of sic: A provably timing-predictable pipelined processor core. In *Real-Time Systems Symposium (RTSS)*, pages 469–481. IEEE, 2018.
21. D. Hardy and I. Puaut. WCET Analysis of Multi-level Non-inclusive Set-Associative Instruction Caches. In *Real-Time Systems Symposium (RTSS)*, pages 456–466, Nov 2008.
22. D. Hardy and I. Puaut. Wcet analysis of instruction cache hierarchies. *J. Syst. Archit.*, 57(7):677–694, Aug. 2011.

23 D. Hardy, B. Rouxel, and I. Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 8 of *International Workshop on Worst-Case Execution Time Analysis*, page 12, Dubrovnik, Croatia, June 2017.

24 A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. Watson, D. Chisnall, M. Roe, B. Davis, et al. Efficient tagged memory. In *International Conference on Computer Design (ICCD)*, pages 641–648. IEEE, 2017.

25 L. Ju, S. Chakraborty, and A. Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *Design, Automation and Test in Europe (DATE)*, pages 1623–1628, San Jose, CA, USA, 2007. EDA Consortium.

26 T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna. Deterministic memory hierarchy and virtualization for modern multi-core embedded systems. In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Montreal, Canada, April 2019.

27 N. C. Kumar, S. Vyas, R. K. Cytron, C. D. Gill, J. Zambreno, and P. H. Jones. Cache design for mixed criticality real-time systems. In *Computer Design (ICCD)*, pages 513–516. IEEE, 2014.

28 B. Lesage, I. Puaut, and A. Seznec. Preti: Partitioned real-time shared cache for mixed-criticality real-time systems. In *Real-Time and Network Systems (RTNS)*, pages 171–180. ACM, 2012.

29 Y. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(12):1477–1487, Dec 1997.

30 T. Liu, Y. Zhao, M. Li, and C. J. Xue. Task assignment with cache partitioning and locking for wcet minimization on mpsoc. In *2010 39th Int. Conf. Parallel Processing*, pages 573–582, Sept 2010.

31 T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, RTSS '99, pages 12–, Washington, DC, USA, 1999. IEEE Computer Society.

32 M. Lv, N. Guan, J. Reineke, R. Wilhelm, and W. Yi. A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems*, 3(1):05–1, 2016.

33 R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni. Real-time cache management framework for multi-core architectures. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*. IEEE, 2013.

34 F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of loops. In *International Conference on Compiler Construction (CC)*, pages 80–94, London, UK, UK, 1998. Springer-Verlag.

35 S. Mittal. A survey of techniques for cache locking. *Transactions on Design Automation of Electronic Systems (TODAES)*, 21(3):49:1–49:24, May 2016.

36 F. Mueller. Timing predictions for multi-level caches. In *In ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, pages 29–36, 1997.

37 R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. A predictable execution model for COTS-based embedded systems. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS)*, pages 269–279. IEEE, 2011.

38 K. Piromsopa and R. J. Enbody. Secure bit: Transparent, hardware buffer-overflow protection. *Transactions on Dependable and Secure Computing*, 3(4):365–376, 2006.

39 I. Puaut. Wcet-centric software-controlled instruction caches for hard real-time systems. In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 10–pp. IEEE, 2006.

40 I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Real-Time Systems Symposium (RTSS)*, pages 114–123. IEEE, 2002.

41 J. Reineke. *Caches in WCET analysis: predictability, competitiveness, sensitivity.* epubli, 2008.

42 J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37(2):99–122, November 2007.

43 A. Sarkar, F. Mueller, and H. Ramaprasad. Static task partitioning for locked caches in multicore real-time systems. *ACM Trans. Embed. Comput. Syst.*, 14(1):4:1–4:30, Jan. 2015.

44 M. R. Soliman and R. Pellizzoni. WCET-Driven dynamic data scratchpad management with compiler-directed prefetching. In *Euromicro Conference on Real-Time Systems (ECRTS)*, volume 76, pages 24:1–24:23, 2017.

45 C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. HDFI: hardware-assisted data-flow isolation. In *Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

46 V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *Real-Time Systems Symposium (RTSS)*, pages 10–pp. IEEE, 2005.

47 X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. *SIGMETRICS Perform. Eval. Rev.*, 31(1):272–282, June 2003.

48 X. Vera, B. Lisper, and J. Xue. Data cache locking for tight timing calculations. *ACM Trans. Embed. Comput. Syst.*, 7(1):4:1–4:38, Dec. 2007.

49 J. Whitham and N. Audsley. Studying the applicability of the scratchpad memory management unit. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 205–214. IEEE, 2010.

50 R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst. (TECS)*, 7(3), 2008.

51 J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The cheri capability model: Revisiting risc in an age of risk. In *International Symposium on Computer Architecture (ISCA)*, 2014.

52 G. Yao, R. Pellizzoni, S. Bak, E. Betti, and M. Caccamo. Memory-centric scheduling for multicore hard real-time systems. *Real-Time Syst.*, 48(6):681–715, 2012.