WCET Analysis with MRU Caches: Challenging LRU for Predictability

Nan Guan
Uppsala University, Sweden
Email: nan.guan@it.uu.se

Mingsong Lv
Northeastern University, China
Email: lvmingsong@ise.neu.edu.cn

Wang Yi
Uppsala University, Sweden
Email: yi@it.uu.se

Abstract—**Most previous work in cache analysis for WCET estimation assumes a particular replacement policy called LRU. In contrast, much less work has been done for non-LRU policies, since they are generally considered to be very "unpredictable". However, most commercial processors are actually equipped with these non-LRU policies, since they are more efficient in terms of hardware cost, power consumption and thermal output, but still maintaining almost as good average-case performance as LRU.**

In this work, we study the analysis of MRU, a non-LRU replacement policy employed in mainstream processor architectures like Intel Nehalem. Our work shows that the predictability of MRU has been significantly underestimated before, mainly because the existing cache analysis techniques and metrics, originally designed for LRU, do not match MRU well. As our main technical contribution, we propose a new cache hit/miss classification, k-Miss, to better capture the MRU behavior, and develop formal conditions and efficient techniques to decide the k-Miss memory accesses. A remarkable feature of our analysis is that the k-Miss classifications under MRU are derived by the analysis result of the same program under LRU. Therefore, our approach inherits all the advantages in efficiency, precision and composability of the state-of-the-art LRU analysis techniques based on abstract interpretation. Experiments with benchmarks show that the estimated WCET by our proposed MRU analysis is rather close to ($5\% \sim 20\%$ more than) that obtained by the state-of-the-art LRU analysis, which indicates that MRU is also a good candidate for the cache replacement policy in real-time systems.

I. Introduction

Hard real-time systems are subjected to strict timing analysis, in which a fundamental problem is to bound the worst-case execution time (WCET) of programs [25]. To derive safe and tight WCET bounds, the analysis must take into account the cache architecture of the target processor. However, the cache analysis problem of statically determining whether each memory access is a hit or a miss is a challenging problem.

Much work has been done on cache analysis for WCET estimation in the last two decades. A vast majority of these works assume a particular cache replacement policy, called LRU (Least-Recently-Used), for which researchers have developed successful analysis techniques to precisely and efficiently predict cache hits/misses [24]. In contrast, much less attention has been paid to other replacement policies like MRU [17], FIFO [6] and PLRU [11]. In general, research in the field of real-time systems assumes LRU as the default cache replacement policy. The main reason is that these non-LRU policies are considered to be much less predictable than LRU [18], [26], and it would be very difficult to develop precise and efficient analyses for them.

However, most commercial processors actually do not employ the LRU cache replacement policy. The reason is that LRU requires more complex logic in hardware implementation, which results in higher hardware cost, power consumption, thermal output and latency. On the other hand, the non-LRU replacement policies like MRU, FIFO and PLRU have simpler implementation logic, but still have almost as good average-case performance as LRU. Therefore, hardware manufacturers tend to choose these non-LRU replacement policies in the processor design, especially for embedded systems that are subject to strict cost, power and thermal constraints.

In this work, we study the analysis of one of the most widely used cache replacement policies, called MRU[1]. MRU has been employed in mainstream processor architectures like Intel Nehalem (the architecture codename of processors like Intel Xeon, Core i5 and i7) [4]. A previous work comparing the average-case performance of cache replacement policies with the SPEC CPU2000 benchmark showed that MRU has as good average-case performance as LRU, and is superior to other non-LRU policies like FIFO and PLRU [1]. To the best of our knowledge, there has been no previous work on the analysis of MRU in the context of WCET estimation. The only relevant work is [18], a general study on the timing predictability of different cache replacement policies, which indicates that MRU is a very "unpredictable" one.

However, this work will show that the predictability of MRU actually has been significantly underestimated. This is mainly because the state-of-the-art cache analysis techniques, originally designed for LRU, are not suitable for MRU. The bottom line of the LRU analysis is to determine whether the accesses related to a particular memory reference are always hits or not (except the first access that may be a cold miss). Such a classification is highly effective for LRU caches since most memory references under LRU indeed exhibit such a "black or white" behavior. In this work we will show that

[1]The name of the MRU replacement policy is inconsistent in literatures. Sometimes, this policy is called Pseudo-LRU because it can be seen as a kind of approximation of LRU. However, we use the name MRU to keep consistency with previous works in WCET research [18], [19], and to distinguish it from another Pseudo-LRU policy PLRU [11] which uses tree structures to store access history information.

many memory references under MRU exhibit a more nuanced behavior: a small number of the accesses are misses while all the other accesses are hits. By the existing analysis framework based on the "black or white" classification, such a behavior has to be treated as if all the accesses are misses, which inherently leads to very pessimistic analysis results.

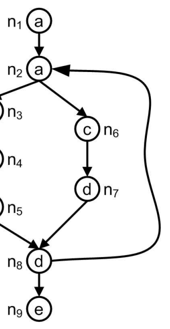
In this paper, we introduce a new cache hit/miss classification k-Miss (at most k accesses are misses while all the others are hits), to more precisely capture the nuanced behavior in MRU. As our main technical contribution, we establish formal conditions to determine whether a memory reference is a k-Miss, and develop techniques to efficiently check these conditions and thus determine the k-Miss memory references. A remarkable feature of our technique is using the cache analysis results of the same program under LRU replacement to derive the k-Miss classifications under MRU replacement. Therefore, our technique inherits the advantages in efficiency, precision and composability from the state-of-the-art LRU analysis techniques based on *abstract interpretation* (AI) [24].

We have conducted experiments with benchmark programs on *instruction* caches to evaluate the effectiveness of our proposed analysis, as well as the predictability of the MRU replacement policy. The experiments show that the estimated WCET by our MRU analysis is very close to ($5\% \sim 20\%$ more than) that obtained by the state-of-the-art LRU analysis. This suggests that MRU is also a good candidate for instruction cache replacement in real-time systems, especially considering MRU's other advantages in hardware cost, power consumption and thermal output. Although the experimental evaluation in this paper focuses on instruction caches, we believe this work is a significant step towards the efficient and precise analysis of MRU for both instruction and data caches, since the properties of MRU disclosed in this paper and our analysis technique generally hold for both. The evaluation and refinement of the techniques in this paper for data caches is left as future work.

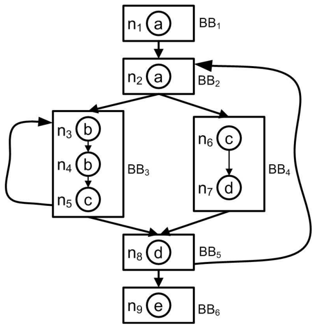
A. Related Work

Most previous work on cache analysis for static WCET estimation assumes the LRU replacement policy. Early work [16] uses the ILP-only approach where the cache behavior prediction is formulated as part of the overall linear program. This approach suffers from serious scalability problems due to the exponential complexity of ILP, and can not handle realistic programs on modern processors. A milestone in the research of static WCET estimation is separating cache analysis by *abstract interpretation* (AI) and path analysis by *implicit path enumeration* based on basic blocks [24]. Such a framework, which is very efficient and precise for LRU caches, forms the common foundation for later research in cache analysis for WCET estimation. For example, it has been extended and refined to deal with loops [2], [3], data caches [21], [13], multi-level caches [10], [22], and shared caches [15], [20].

In contrast, much less work has been done for non-LRU caches. Although some progress has been made in the analysis of policies like FIFO [6], [7] and PLRU [8], in general these analyses are significantly less precise than for LRU.



(a) On the basis of individual entries.



(b) On the basis of basic blocks.

Fig. 1. An example control-flow-graph.

To the best of our knowledge, there has been no work on the analysis of MRU. The only relevant work is [18], which is a general study of the predictability of different policies. This work defines several metrics to evaluate the predictability of different replacement policies including LRU, MRU, FIFO and PLRU. Based on these metrics, all the non-LRU policies appear to be significantly less predictable than LRU.

Finally, we refer to [25] for a comprehensive survey on WCET analysis techniques and tools, which covers many relevant references that are not listed here due to the space limit.

II. Basic Concepts

For simplicity of presentation, we assume a *fully-associative* cache. However, the analysis techniques of this paper are directly applicable to *set-associative* caches, since the memory references mapped to different cache sets do not affect each other, and each cache set can be treated as a fully-associated cache and analyzed independently.

The cache consists of L *cache lines*. The memory content that fits into one cache line is called a *memory block*. Each instruction or data accessed by the program is called a *memory reference*, and we assume that in general a memory block may contain multiple memory references.

The program is represented by a control-flow-graph (CFG) $G = (N, E)$, where N is the set of entries and E the set of directed edges. We use n_i to denote an entry. The CFG can also be represented as a digraph of basic blocks. Each basic block BB_i contains a number of sequentially executed entries. Fig. 1 shows the two different forms of CFG. The letters a, b, $\cdots$ inside each entry denote the memory block accessed by the entry. Note that several entries may access the same memory block since a memory block in general contains multiple memory references and a memory reference may be visited at different places in the program.

When the program accesses a memory reference (an entry in the CFG), the processor first checks whether the memory block containing this memory reference is in the cache. If yes, it is a *hit*, and the program directly accesses this memory reference from the cache. Otherwise, it is a *miss*, and the memory block containing this memory reference is first installed in the cache before the program accesses it.

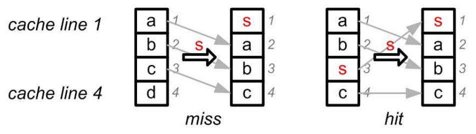


Fig. 2. Illustration of LRU cache update, where the left part is a cache miss and the right part is a cache hit.

A memory block only occupies one cache line regardless how many times it is accessed. So the number of *different* memory blocks in an access sequence is important to the cache behavior. We use the following concept to reflect this:

Definition II.1 (Stack Length)**.** *The Stack Length of a memory access sequence p, denoted by $\mathsf{sl}(p)$, is the number of different memory blocks accessed in p.*

For example, the stack length of the access sequence

$$a \rightarrow b \rightarrow c \rightarrow a \rightarrow d \rightarrow a \rightarrow b \rightarrow d$$

is 4, since only a, b, c and d are accessed in this sequence.

The number of memory blocks in a program is typically far greater than the number of cache lines, so a replacement policy must decide which block to replace upon a miss. In the following we describe the LRU and MRU replacement policies respectively.

A. LRU *Replacement*

The LRU replacement policy always stores the most recently accessed memory block in the first cache line. When the program accesses a memory block s, if s is not in the cache (miss), then all the memory blocks in the cache will be shifted one position to the next cache line (the memory block in the last cache line is removed from the cache), and s is installed to the first cache line. If s is in the cache already (hit), then s is moved to the first cache line and all memory blocks that were stored before s's old position will be shifted one position to the next cache line. Fig. 2 illustrates the update upon an access to memory block s in an LRU cache of 4 lines. In the figure, the uppermost block represents the first (lowest-index) cache line and the lowermost block is the last (highest-index) one. All the figures in this paper will follow this convention.

A metric defined in [18] to evaluate the predictability of a replacement policy is the *minimal life span* (mls), the minimal number of different memory blocks required to evict a just visited memory block out of the cache (not counting the access that brought the just visited memory block into the cache). It is known that [18]:

Lemma II.2. *The* mls *of* LRU *is L.*

The mls metric can be directly used to determine cache hits/misses for a memory access sequence: if the stack length of the sequence between two successive accesses to the same memory block is smaller than mls, then the later access must be a hit. For example, for a memory access sequence

$$a \rightarrow b \rightarrow c \rightarrow c \rightarrow d \rightarrow a \rightarrow e \rightarrow b$$

on an LRU cache with $L = 4$, we can easily conclude that the second access to memory block a is a hit since the sequence between two accesses to a is $b \rightarrow c \rightarrow c \rightarrow d$, which has stack length 3, while the second access to b is a miss since the stack length of the sequence $c \rightarrow c \rightarrow d \rightarrow a \rightarrow e$ is 4. Clearly, replacement policies with larger mls are preferable, and the upper bound of mls is L.

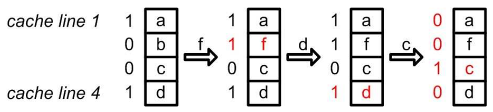


Fig. 3. Illustration of MRU cache update.

B. MRU *Replacement*

For each cache line, the MRU replacement policy stores an extra MRU-bit, to approximately represent whether this cache line was recently visited. An MRU-bit being 1 indicates that this line was recently visited, while being 0 indicates the opposite. Whenever a cache line is visited, it's MRU-bit will be set to 1. Eventually there will be only one MRU-bit being 0 in the cache. When the cache line with the last MRU-bit being 0 is visited, this MRU-bit is set to 1 and all the other MRU-bits change back from 1 to 0, which is called a *global-flip*.

More precisely, when the program accesses a memory block s, MRU replacement first checks whether s is already in the cache. If yes, then s will still be stored in the same cache line and its MRU-bit is set to 1 regardless of its original state. If s is not in the cache, MRU replacement will find the first cache line whose MRU-bit is 0, then replaces the originally stored memory block in it by s and set its MRU-bit to 1. After the above operations, if there still exists some MRU-bit being 0, the remaining cache lines' states are kept unchanged. Otherwise, all the remaining cache lines' MRU-bits are changed from 1 to 0, which is a global-flip. Note that the global-flip operation guarantees that at any time there is at least one MRU-bit in the cache being 0.

In the following we present the MRU replacement policy formally. Let M be the set of all the memory blocks accessed by the program plus an element representing emptiness. The MRU cache state can be represented by a function $\mathsf{C} : \{1, \cdots, L\} \rightarrow M \times \{0, 1\}$. We use $\mathsf{C}(i)$ to denote the state of the i^{th} cache line. For example, $\mathsf{C}(i) = (s, 0)$ represents that cache line i currently stores memory block s and its MRU-bit is 0. Further, we use $\mathsf{C}(i).\omega$ and $\mathsf{C}(i).\beta$ to denote the resident memory block and the MRU-bit of cache line i. The update rule of MRU replacement can be described by the following steps, where C and C' represent the cache state before and after the update upon an access to memory block s, respectively, and δ denotes the cache line where s should be stored after the access:

1) If there exists h s.t. $C(h).\omega = s$, then let $\delta \leftarrow h$, otherwise let $\delta = h$ s.t. $\mathsf{C}(h).\beta = 0$ and $\mathsf{C}(j).\beta = 1$ for all $j < h$.
2) $\mathsf{C}'(\delta) \leftarrow (s, 1)$
3) If $C(h).\beta = 1$ for all h, then let $\mathsf{C}'(j) \leftarrow (\mathsf{C}(j).\omega,\ 0)$ for all $j \neq \delta$ (i.e., global-flip), otherwise $\mathsf{C}'(j) \leftarrow \mathsf{C}(j)$ for all $j \neq \delta$.

Fig. 3 illustrates the MRU cache replacement with a cache of 4 lines. First the program accesses memory block f, which

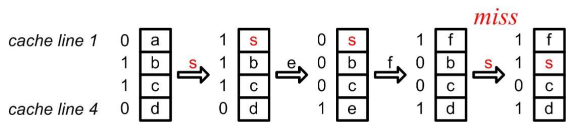


Fig. 4. An example showing that the mls of MRU is 2.

is not in the cache yet. The first cache line with MRU-bit being 0 is cache line 2, so f will replace b and the corresponding MRU-bit is set to 1. Since there are still other cache lines with MRU-bit being 0, all the other cache lines stay unchanged. Then the program accesses d, which is already in the 4^{th} cache line. In this case, d is still stored in the same cache line, and its MRU-bit keeps 1. Finally, the program accesses c, which is also in the cache. So c stays in the same cache line, and its MRU-bit is changed to 1. However, at this moment all the other cache lines' MRU-bits are 1, so the global-flip is triggered, which changes all the other MRU-bits from 1 to 0.

In the MRU cache, the MRU-bit can roughly represent how old the corresponding memory block is, and the replacement always tries to evict a memory block that is relatively old. So MRU can be seen as an approximation of LRU. However, such an approximation results in a very different mls [18]:

Lemma II.3. *The* mls *of* MRU *is* 2.

This is illustrated in Fig. 4, where only two memory blocks e and f are enough to evict a just-visited memory block s. It is easy to extend this example to a cache with an arbitrarily large number of cache lines, where we still only need two memory blocks to evict s. Due to this property, MRU has been believed to be a very unpredictable replacement policy, and to our best knowledge it has never been seriously considered as a good candidate for timing-predictable architectures.

III. A Review of the Analysis for LRU

As mentioned in Section I, a remarkable feature of the MRU analysis proposed in this paper is using the analysis results of the same program under LRU to derive the cache behavior under MRU. Thus, before presenting our new analysis technique, we first provide a brief review of the state-of-the-art analysis technique for LRU.

Exact cache analysis suffers from a serious state space explosion problem, so researchers focused on approximation techniques, separating path analysis and cache analysis for scalability [24]. The path analysis requires an upper bound on the timing delay of an entry whenever it is executed. Therefore, the main purpose of the LRU cache analysis is to decide cache hit/miss classification (CHMC) for each entry:

- AH: Always-Hit. The entry's memory access is always a hit whenever it is executed.
- FM: First-Miss. The entry's memory access is a miss for the first execution, but always a hit afterwards. This classification is useful to handle "cold misses" in loops.
- AM: Always-Miss. The entry's memory access is always a miss whenever it is executed.

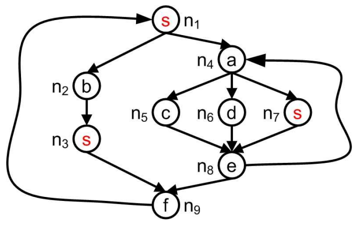


Fig. 5. Illustration of Maximal Stack Distance.

- NC: Non-Classified. The entry can not be classified into any of the above categories. These entries have to be treated as AM in later phases of the analysis.

Among the above CHMC, we call AH and FM *positive classification* since they ensure that (the major portion of) the memory accesses of an entry to be hits, while we call AM and NC *negative classification*.

Recall that the mls of LRU is L (Lemma II.2), and that one can directly use this property to decide the CHMC for an entry in a linear access sequence. However, a CFG is generally an arbitrary digraph, and there may be multiple paths between two entries. We use the following concept to capture the maximal number of different memory blocks that are accessed between two entries accessing the same memory block in the CFG.

Definition III.1 (Maximal Stack Distance)**.** *Let n_i and n_j be two entries accessing the same memory block s. The Maximal Stack Distance from n_i to n_j, denoted by $\mathsf{dist}(n_i, n_j)$, is:*

$$\mathsf{dist}(n_i, n_j) = \begin{cases} \max\{\mathsf{sl}(p) \mid p \in \Pi(n_i, n_j)\} & \Pi(n_i, n_j) \neq \emptyset \\ -1 & \Pi(n_i, n_j) = \emptyset \end{cases}$$

where $\Pi(n_i, n_j)$ is the set of paths that:

- *Start with n_i and end with n_j,*
- *Do not contain other entries accessing s apart from n_i and n_j.*

For example, the CFG in Fig. 5 contains three entries n_1, n_3 and n_7 accessing the same memory block s. We have $\mathsf{dist}(n_1, n_7) = 5$ since $\Pi(n_1, n_7)$ contains a path

$$n_1 \to n_4 \to n_5 \to n_8 \to n_4 \to n_6 \to n_8 \to n_4 \to n_7$$

in which $\{s, a, c, d, e\}$ are accessed. We have $\mathsf{dist}(n_1, n_3) = 2$ since $n_1 \to n_2 \to n_3$ is the only path in $\Pi(n_1, n_3)$ (any other path from n_1 to n_3 does not satisfy the second condition for Π). We have $\mathsf{dist}(n_3, n_7) = -1$ since any path from n_3 to n_7 has to go though n_1 which also accesses s.

Now one can use the maximal stack distance to decide the CHMC: we can decide n_j to be a positive classification (AH or FM), if $\mathsf{dist}(n_i, n_j) \leq L$ holds for any entry n_i that accesses the same memory block s as n_j. This is because no matter which path n_i is visited via, there are not enough different memory blocks to evict s since the last access to s.

However, the problem of computing the maximal stack distance with a digraph is highly intractable. Therefore, the LRU analysis resorts to an over-approximation by abstract interpretation. The main idea is to define an abstract cache state and iteratively traverse the program until the abstract state converges to a fixed point, and use the abstract state of this fixed point to determine the CHMC. There are mainly three fix-point analyses:

- Must analysis to determine AH entries.
- May analysis to determine AM entries.
- Persistence analysis to determine FM entries.

An entry is a NC if it can not be classified by any of the above analyses. We refer to the reference [5] for a detailed introduction to these fix-point analyses.

Finally, the CHMCs are used to calculate the timing delay of each basic block, which will be encoded into *implicit path enumeration* by ILP formulas, or other constraint forms, to calculate the WCET of the program. The encoding for AH and AM/NC is rather straightforward: we count the cache hit delay for AH entries and cache miss delay for AM/NC entries. To encode the FM entries, one can use the VIVU technique [23], which unrolls a loop into the *head* part representing the first iteration, and the *body* part representing the remaining iterations. The FM entry is counted as a miss in the head part and a hit in the body part. We refer to the reference [23] for a detailed introduction to implicit path enumeration.

IV. The New Analysis of MRU

In this section we present our new analysis for MRU. First we will show that the existing CHMC in the LRU analysis as introduced in the last section is actually not suitable to capture the cache behavior under MRU, and thus we introduce a new positive classification k-Miss. Then we introduce the conditions for an entry to be k-Miss, and show how to efficiently check these conditions. Finally we present the overall flow of our proposed analysis based on the above results.

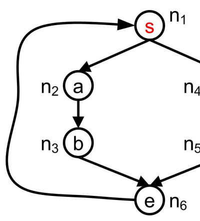
A. New Classification: k-Miss

First we consider the example in Fig. 6-(a). We can easily see that $\text{dist}(n_1, n_1) = 4$, i.e., at most 4 different memory blocks appear in each iteration of the loop, no matter which branch is taken. Since 4 is larger than 2 (the mls of MRU), n_1 can not be decided as a positive classification using mls.

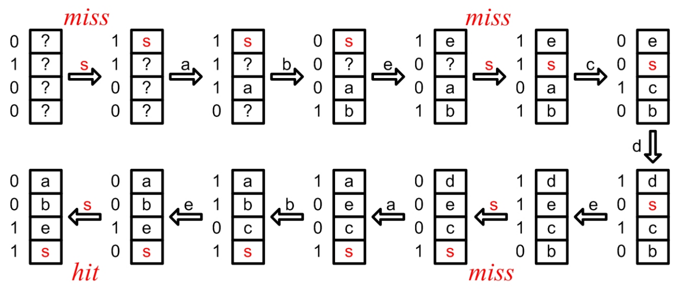
Now we have a closer look into this example, considering a particular execution sequence in which the two branches are taken alternatively, as shown in Fig. 6-(b). Assume that the memory blocks initially stored in the cache (denoted by "?") are all different from the ones that appear in Fig. 6-(a), and initial MRU-bits are shown in the first cache state of Fig. 6-(b).

We can see that the first three executions of s are all misses. The first miss is a cold miss which is unavoidable anyway under our initial cache state assumption. However, the second and third accesses are both misses because s is evicted by other memory blocks of the program. Indeed, entry n_1 can not be determined as AH or FM, and one has to put it into the negative classification category and assume that it is always a miss whenever it is executed.

However, if the sequence continues, we can see that when n_1 is visited for the fourth time, s is actually in the cache, and most importantly, *the access of n_1 will always be a hit afterwards* (we do not show a complete picture of this



(a) A CFG example.



(b) Cache update with a possible execution sequence where the two branches are taken alternatively.

Fig. 6. An example motivating the k-Miss classification.

sequence, however, this can be easily seen by simulating the update for a long enough sequence until a cycle appears).

The existing positive classification AH and FM are inadequate to capture the behavior of entries like n_1 in the above example, which only encounters a smaller number of misses, but will eventually go into a stable state of being always hit. Such behavior is actually quite common under the MRU replacement. Therefore, the analysis of MRU will be inherently very pessimistic if one only relies on the AH and FM classification to claim cache hits.

Motivated by the above phenomenon, we define a new positive classification to more precisely capture the cache behavior under MRU:

k-Miss: Among all the memory accesses of the entry, at most k of them are misses, while all the others are hits.

Note that for a k-Miss entry, the k times misses do not necessarily occur at the first k accesses of the entry. It allows the case that the misses and the hits appear alternatively, as long as the total number of misses does not exceed k.

B. Conditions for k-Miss

In the following we will introduce the conditions for an entry to be classified as k-Miss. We start with the following properties of the MRU replacement:

Lemma IV.1. *If the number of cache lines is L, then there are L different memory blocks between two successive global-flips (including the ones triggering these two global-flips).*

Proof: Right after a global-flip, there are $L - 1$ cache lines whose MRU-bits are 0. In order to have the next flip, all these cache lines of which the MRU-bits are 0 need to be accessed, i.e., it needs $L - 1$ different memory blocks that are also different from the one causing the first global-flip. So

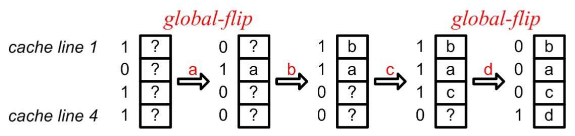
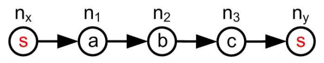
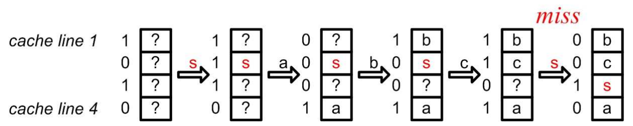


Fig. 7. Illustration of Lemma IV.1.



(a) a particular path from n_x to n_y



(b) the position of s is moved one step down when it is loaded back to the cache

Fig. 8. Illustration of Lemma IV.2

in total L different memory blocks are involved in the access sequence between two successive global-flips. ■

We illustrate the intuition of Lemma IV.1 by the example in Fig. 7 with $L = 4$. The access to memory block a triggers the first global-flip, after which there are 3 MRU-bits being 0. To trigger the next global-flip, these three MRU-bits have to be changed to 1, which needs another 3 different memory blocks. So the total number of different memory blocks involved in the access sequence between these two flips is 4.

Lemma IV.2. *Suppose that under the* MRU *replacement, at some point memory block s is accessed by n_x at cache line i (either miss or hit), and the next access to s is a miss caused by n_y upon which s is installed to cache line j. We have $j > i$ if the following condition holds:*

$$\mathsf{dist}(n_x, n_y) \le L. \tag{1}$$

Fig. 8 illustrates the intuition of Lemma IV.2, where n_x and n_y are two entries accessing the same memory block s and satisfying Condition (1). We focus on a particular path $n_x \to n_1 \to n_2 \to n_3 \to n_y$. Fig. 8-(b) shows the cache update along this path: first n_x accesses s in the second cache line. After s is evicted out of the cache and is loaded back again, it is installed to the third cache line, which is one position below the previous one. In the following we give a formal proof of the lemma.

Proof: We use the term *event* to refer to a cache access. Let event e_x be the access to s at cache line i by n_x as stated in the lemma, and event e_y the installation of s to cache line j by n_y. We prove the lemma by contradiction, assuming $j \le i$.

The first step is to prove that there are at least two global-flips in the event sequence $\{e_{x+1}, \cdots, e_{y-1}\}$ (e_{x+1} denotes the event right after e_x and e_{y-1} the event right before e_y).

Before e_y, s has to be first evicted out of the cache. Let event e_v denote such an eviction of s, which occurs at cache line i. By the MRU replacement rule, a memory block can be evicted from the cache only if the MRU-bit of its resident cache line is 0. So we know $\mathsf{C}(i).\beta = 0$ *right before* e_v.

On the other hand we also know that $\mathsf{C}(i).\beta = 1$ *right after* event e_x. And since only a global-flip can change an MRU-bit from 1 to 0, we know that there must exist at least one global-flip among the events $\{e_{x+1}, \cdots, e_{v-1}\}$.

Then we focus on the event sequence $\{e_v, \cdots, e_{y-1}\}$. We distinguish two cases:

- $i = j$. Right after the eviction of s at cache line i (event e_v), the MRU-bit of cache line i is 1. On the other hand, just before the installation of s to cache line j (event e_y), the MRU-bit of cache line j must be 0. Since $i = j$, there must be at least one global-flip among the events $\{e_{v+1}, \cdots, e_{y-1}\}$, in order to change the MRU-bit of cache line $i = j$ from 1 to 0.
- $i > j$. By the MRU replacement rule, we know that just before s is evicted in event e_v, it must be true that $\forall h < i : \mathsf{C}(h).\beta = 1$, and particularly $\mathsf{C}(j).\beta = 1$. On the other hand, just before the installation of s in event e_y, the MRU-bit of cache line j must be 0. Therefore, there must be at least one global-flip among the events $\{e_v, \cdots, e_{y-1}\}$, in order to change the MRU-bit of cache line j from 1 to 0.

In summary, there is at least one global-flip among the events $\{e_v, \cdots, e_{y-1}\}$.

Therefore, we can conclude that there are at least two global-flips among the events $\{e_{x+1}, \cdots, e_{y-1}\}$. By Lemma IV.1 we know that at least L different memory blocks are accessed in $\{e_{x+1}, \cdots, e_{y-1}\}$. Since e_y is the first access to memory block s after e_x, there is no access to s in $\{e_{x+1}, \cdots, e_{y-1}\}$, so at least $L + 1$ different memory blocks are accessed in $\{e_x, \cdots, e_y\}$.

Let p be the path that leads to the event sequence $\{e_x, \cdots, e_y\}$. Clearly, p starts with n_x and ends with n_y. We also know that no other entry along p, apart from n_x and n_y, accesses s, since e_y is the first event accessing s after e_x. So p is a path in $\Pi(n_x, n_y)$ (Definition III.1), and we know $\mathsf{dist}(n_x, n_y) \ge \mathsf{sl}(p)$. Combining this with Condition (1) we have

$$\mathsf{sl}(p) \le L. \tag{2}$$

This contradicts with that at least $L + 1$ different memory blocks are accessed in $\{e_x, \cdots, e_y\}$ as we concluded above. ■

To see the significance of Lemma IV.2, we consider a special case where a CFG only contains one entry n accessing the memory block s and $\mathsf{dist}(n, n) \le L$ as shown in Fig. 9-(a). In this case, by Lemma IV.2 we know that each time s is accessed, there are only two possibilities:

- the access to s is a hit, or
- the access to s is a miss and s is installed to a cache line with a strictly larger index than before.

So we can conclude that s can only be a miss for at most L times since the position of s can only "move downwards" for a limited number of times which is bounded by the number of cache lines.

However, in general there could be more than one entry in the CFG accessing the same memory block, where Lemma IV.2 can not be directly applied to determine the k-Miss classification. Consider the example in Fig. 9-(b), where two

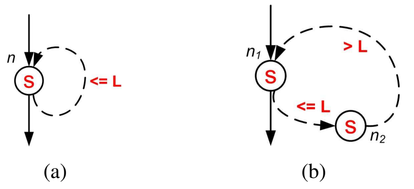


Fig. 9. An example illustrating the usage of Lemma IV.2

entries n_1 and n_2 both access the same memory block s, and we have $\mathsf{dist}(n_1, n_2) \leq L$ and $\mathsf{dist}(n_2, n_1) > L$. In this case, we can not classify n_2 as a k-Miss, although Lemma IV.2 still applies to the path from n_1 to n_2. The is because Lemma IV.2 only guarantees the position of s will "move downwards" each time n_2 encounters a miss, but the position of s may "move upwards" (since $\mathsf{dist}(n_2, n_1) > L$), which breaks down the memory block's movement monotonicity.

In order to use Lemma IV.2 to determine the k-Miss classification in the general case, we need to guarantee a global movement monotonicity of a memory block among all the related entries. This can be done by examining the condition of Lemma IV.2 for all the entries in a *strongly connected component* (maximal strongly connected subgraph) together, as described in the following theorem:

Theorem IV.3. *Let SCC be a strongly connected component in the CFG, let S be the set of entries in SCC accessing the memory block s. The total number of misses caused by all the entries in S is at most L if the following condition holds:*

$$\forall n_x, n_y \in S : \mathsf{dist}(n_x, n_y) \leq L. \tag{3}$$

Proof: Let e_f and e_l be the first and last event launched by some entry in S during the whole program execution. Since S is a subset of the strongly connected component SCC, any event accessing s in the event sequence $\{e_f, \cdots, e_l\}$ has to be also launched by some entry in S (otherwise there will be a cycle including entries both inside and outside SCC, which contradicts that SCC is a strongly connected component).

By Condition (3) and Lemma IV.2 we know that, among the events $\{e_f, \cdots, e_l\}$ whenever the access to s is a miss, s will be installed to a cache line with a strictly larger index than before. Since the cache contains only L cache lines, there are at most L misses of s in $\{e_f, \cdots, e_l\}$. Therefore, the entries in S have at most L misses in total. ■

Theorem IV.3 provides the condition to have an upper bound on the *total* number misses of a group of entries, which can be easily relaxed to that each of these entries can be individually classified as an L-Miss (k-Miss with $k = L$):

Corollary IV.4. *Let SCC be a strongly connected component in the CFG, let S be the set of entries in SCC accessing the memory block s. Each entry in S can be classified as L-Miss if the following condition holds:*

$$\forall n_x, n_y \in S : \mathsf{dist}(n_x, n_y) \leq L. \tag{4}$$

We do such a relaxation to facilitate a simple implicit path enumeration formulation based on basic blocks. It is also possible to directly encode Theorem IV.3 into implicit path enumeration to obtain less-pessimistic analysis results at the cost of a more complicated (thus less scalable) ILP formulation, which will not be further explored in this paper.

C. Efficient k-Miss Determination

Corollary IV.3 gives us the condition to determine whether an entry is an L-Miss (k-Miss with $k = L$). The major task of checking this condition is to calculate the Maximal Stack Distance $\mathsf{dist}()$. As we mentioned in Section III, the exact calculation of $\mathsf{dist}()$ is highly intractable (that is why the analysis of LRU relies on AI to obtain over-approximate classification). For the same reason, we will also resort to over-approximation to efficiently check the conditions of L-Miss.

The main idea is to first do the Must/Persistence *analysis for the same program under* LRU *replacement, and then use the* AH/FM *classification under* LRU *to derive the L-*Miss *classification under* MRU.

Lemma IV.5. *Let n_y be an entry that accesses memory block s and is classified as* AH/FM *by the* Must/Persistence *analysis under the* LRU *replacement. For any entry n_x that also accesses s, if there exists a cycle in the CFG including n_x and n_y, then it must be true that*

$$\mathsf{dist}(n_x, n_y) \leq L.$$

Proof: We prove by contradiction. Let n_x be an entry that also accesses s and there exists a cycle in the CFG including n_x and n_y. We assume that

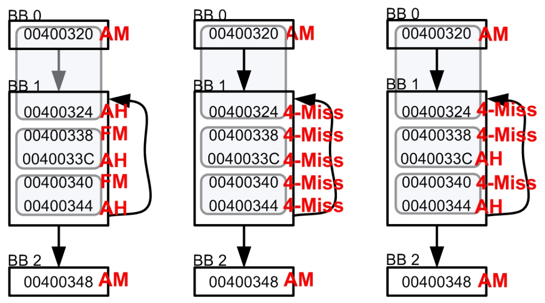
$$\mathsf{dist}(n_x, n_y) > L. \tag{5}$$

By the definition of $\mathsf{dist}(n_x, n_y)$ we know that there must exists a path from n_x to n_y which does not contain any other entry accessing r apart from n_x and n_y (otherwise $\mathsf{dist}(n_x, n_y) = -1$). Let p the path that leads to the maximal stack distance from n_x to n_y, i.e., $\mathsf{sl}(p) = \mathsf{dist}(n_x, n_y)$.

By our assumption (5), we know there are strictly more than L different memory blocks along p. This implies that under the LRU replacement, whenever n_y is reached via path p, s is not in the cache. Furthermore, n_y can be reached via path p repeatedly since there exists a cycle including n_x and n_y. This contradicts with that n_y is classified as AH/FM by the Must/Persistence analysis under LRU, since Must/Persistence yields *safe* classification (in the real execution, an AH entry does not have cache misses and a FM entry only has at most one miss) [24]. ■

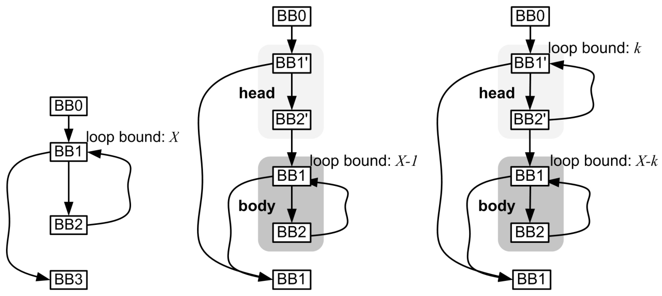
Theorem IV.6. *Let SCC be a strongly connected component in the CFG, and S the set of entries in SCC that access the same memory block s. If all the entries in S are classified as* AH/FM *by the* Must/Persistence *analysis under the* LRU *replacement, then each of these entries can be classified as L-*Miss *under the* MRU *replacement.*

Proof: Let n_x, n_y be two arbitrary entries in S, so both of them access the memory block s and are classified as AH/FM by the Must/Persistence analysis under LRU. Since S is a subset of a strongly connected component, we also know n_x



(a) After step 1). (b) After step 2). (c) After step 3).

Fig. 10. Illustration of the overall analysis flow.



(a) original CFG (b) unrolled for FM (c) unrolled for k-Miss

Fig. 11. VIVU for FM and k-Miss

and n_y are included in a cycle in the CFG. Therefore, by Lemma IV.5 we know

$$\mathsf{dist}(n_x, n_y) \leq L.$$

Since n_x, n_y are arbitrarily chosen, the above conclusion holds for any pair of entries in S. Therefore, by Corollary IV.4 we know that each entry in S can be classified as L-Miss. ∎

Actually, Theorem IV.6 can be extended to the form of using the maximal memory block ages produced in the Must/Persistence analysis under LRU to derive k-Miss classifications under MRU with smaller k values. However, in this work we only use the original form of Theorem IV.6 for the L-Miss classification, which already gives very precise WCET estimation as we will show in Section V.

D. *The Overall Analysis of* MRU

Now we introduce the overall analysis flow for the MRU cache, which are in the following four major steps:

1) Do the AI analysis for the program under LRU.
2) Use the results obtained in step 1) to derive the $k-$Miss classification entries under MRU (Theorem IV.6).
3) Revise the classifications of the "prefetched" entries that are obviously AH.
4) Encode the classifications obtained from the above steps into the implicit path enumeration ILP formulation to calculate the WCET.

In the following we introduce each step in detail.

In the first step, we do the AI cache analysis for the CFG assuming an LRU cache with the same number of cache lines. For example, Fig. 10-(a) shows the classification for the entries in the CFG under LRU with $L = 4$. We assume that each memory block contains two memory references, and in the figure the entries sharing the same memory block are covered by the same grey box.

Then in the second step, we transform the result obtained in step 1) into the k-Miss classifications for MRU using Theorem IV.6. Fig. 10-(b) shows the result of the transformation from Fig. 10-(a). Note that although both the entries "00400320" and "00400324" access the same memory block and "00400320" is an AM, we still can use Theorem IV.6 to classify "00400324" as a 4-Miss since they are not in the same strongly connected component.

The third step is to revise the classification of the entries that are obviously AH due to the "prefetching" by the preceding entries accessing the same memory block. If an entry is in the same basic block as its preceding entry that access the same memory block, this entry can be immediately classified as AH under either LRU or MRU. Fig. 10-(c) shows the results of the revision. Note that although "00400320" and "00400324" also share the same memory block, "00400324" can not be classified as AH since they are in different basic blocks.

By now, we have determined the CHMC for all the entries: the k-Miss entries determined in step 2), the AH entries determined in step 3) and remaining entries which are AM/NC.

In the last step, we encode CHMC obtained from the above steps into the implicit path enumeration ILP formulation. The k-Miss entries can be handled by a similar approach as the VIVU in the LRU analysis (see Section III). The new problem is that the cache misses of a k-Miss entry do not necessarily occur in the first k iterations of the loop, and if a basic block contains more than one k-Miss entry, the cache-misses of these entries may occur in different iterations. So it seems the implicit path enumeration has to be done on the individual entry level, instead of the basic block level as the original LRU analysis [24]. However, actually we can still encode the implicit path enumeration on the basis of basic blocks, assuming that all the misses of a k-Miss entry occur in the first k iterations, as shown in Fig. 11-(c). This particular scenario exactly captures the worst case among all the possible cases and indeed yields a safe WCET estimation.

V. Experimental Evaluation

A. *Methodology*

We evaluate the performance of our proposed MRU analysis with programs[2] from the Mälardalen Real-Time Benchmark suite [9]. All the experiments are conducted with *instruction* caches. However, the theoretical results of this work also directly apply to data caches, and we leave the evaluation for data caches as future work.

[2]Some programs in the benchmark are not included in our experiments since the CFG construction engine (from Chronos [14]) used in our prototype does not support programs with particular structures like recursion and switch-case very well. Some loop bounds in the programs can not be automatically inferred by the CFG construction engine, which are manually set to be 50.

TABLE I
EXPERIMENT RESULTS.

Program	LRU analysis by AI				MRU analysis in this work					Baseline analysis				
	AH	FM	AM +NC	WCET	AH	k-Miss	AM +NC	WCET	exceeding LRU analy.	AH	FM	AM +NC	WCET	exceeding LRU analy.
adpcm	1262	80	1319	14,302,899	1159	84	1419	14,482,260	1.25%	1159	0	1503	14,601,582	2.09%
bs	36	28	14	3,938	35	29	14	4,658	18,28%	35	0	43	12,278	211.78%
bsort	61	54	22	587,350	55	60	22	588,835	0.25%	55	0	82	2,046,718	248,47%
cnt	94	63	45	398,170	85	72	45	399,919	0.44%	85	0	117	1,371,981	244,57%
crc	225	173	64	212,398	198	200	64	216,532	1.95%	198	0	264	801,804	277,50%
edn	427	157	312	58,321,260	415	166	312	58,348,791	0.05%	415	0	481	59,119,344	1.37%
expint	103	89	35	88,026	91	101	35	89,439	1.61%	91	0	136	309,966	252.13%
fdct	321	76	250	166,662	320	77	250	168,723	1.24%	320	0	327	200,187	20.12%
fibcall	25	12	17	1,503	20	17	17	1,845	22.75%	20	0	34	4,356	189.82%
fir	66	51	28	116,927	62	55	28	118,322	1.19%	62	0	83	402,728	244.43%
insertsort	51	33	22	258,289	48	36	22	259,261	0.38%	48	0	58	873,169	238.06%
janne	32	26	16	122,040	29	29	16	122,772	0.60%	29	0	45	463,850	280.08%
jfdctint	341	22	328	205,571	339	24	328	206,156	0.28%	339	0	352	214,868	4.52%
matmult	133	114	40	12,665,170	124	123	40	12,668,329	0.02%	124	0	163	42,420,529	234.94%
minver	376	101	325	60,177,371	350	113	339	60,318,356	0.23%	350	0	452	42,420,529	234.94%
ndes	570	97	454	6,131,554	501	117	503	6,665,202	8.70%	501	0	620	6,793,656	10.80%
ns	73	39	43	500,833,778	68	44	43	500,834,957	<0.01%	68	0	87	1,716,732,788	242.78%
nsichneu	3920	1	4044	2,611,872	3606	3	4356	2,752,290	5.38%	3606	0	4359	2,752,290	5.38%
prime	165	26	59	8,097	99	92	59	9,807	21.12%	99	0	151	27,537	240.09%
qsort	217	0	241	29,830,054	212	0	246	29,853,463	0.08%	212	0	246	29,853,463	0.08%
qurt	362	270	109	20,043	295	337	109	23,508	17.29%	295	0	446	66,816	233.36%
select	189	0	215	26,811,978	174	2	228	29,198,796	8.90%	174	0	228	29,198,796	8.90%
sqrt	49	37	21	6,588	42	44	21	7,638	15.94%	42	0	65	22,090	235.37%
statemate	656	6	810	231,197	561	18	893	250,655	8.42%	561	0	911	250,655	8.42%
ud	255	137	155	68,705,001	241	144	162	69925032	1.78%	241	0	306	78,495,453	14.25%
average									**5.74%**					**135.03%**

We compare the estimated WCET obtained by the state-of-the-art LRU analysis based on abstract interpretation (Must and May analysis in [24], Persistence analysis in [3]) and that obtained by our MRU analysis. If the estimated WCET by our MRU analysis is sufficiently close to that by the LRU analysis, then it is fair to claim that MRU is also a good candidate for the instruction cache replacement policy in real-time embedded systems. At the same time, such a comparison can also evaluate the precision of our MRU analysis, since it is known that i) LRU and MRU have essentially the same performance [1] and, ii) the LRU analysis is very precise (typically with $5\% \sim 10\%$ WCET over-estimation) [24].

To do such experiments, it may be unauthentic to only compare the estimated WCET by the LRU analysis and our MRU analysis, since under certain hardware configurations such a comparison may not provide useful information. Consider an extreme case: the cache is very small and all the memory accesses are always misses regardless of the underlying replacement policy. In this case, the estimated WCET by the LRU analysis and our MRU analysis will be exactly the same. However, such a result provides no information about the predictability of MRU or the precision of our MRU analysis.

To solve this problem, we introduce a "baseline" analysis into the comparison, to show the actual gain of our MRU analysis. The baseline analysis only explores the AH entries due to the "prefetching" by their preceding entries in the same memory block, and assumes all other entries to be AM. The comparison between the LRU analysis and our MRU analysis is considered to be meaningful only if the gap between the LRU analysis and the baseline is sufficiently large.

B. Results

The experiments of Table I use the same hardware setting as in [24]: We assume a $1K$ bytes set-associative instruction cache, and each set has 4 ways. Each instruction is 4 bytes, and each cache line (memory block) is 8 bytes. We assume a perfect pipeline in the processor, and all the instructions have the same execution delay of 1 cycle. The memory access penalty is 1 cycle upon a hit, and 10 cycles upon a miss.

Table I summarizes the number of entries with different classifications and the estimated WCETs of the LRU analysis by abstract interpretation [24], [3], the MRU analysis in this work and the baseline WCET estimation mentioned above. For the MRU and baseline analysis, we also calculate the corresponding excesses over the LRU analysis in percentage (the "exceeding LRU" columns in the table). For example, the estimated WCET by our MRU analysis of program *adpcm* is $14,482,260$, which is $(14,482,260 - 14,302,899)/14,302,899 = 1.25\%$ more than the estimation $14,302,899$ by the LRU analysis.

From the table we can see that most positive classifications (AH and FM) in the LRU analysis still remain positive in our MRU analysis (AH and k-Miss). For most programs the estimated WCET by our MRU analysis is very close to that by the LRU analysis. The difference is 5.74% on average. On the other hand, the gap between the baseline and the LRU analysis is indeed very large: the estimated WCET by the baseline analysis is on average 135.03% *more than* that obtained by the LRU analysis.

The k-Miss classification used in our MRU analysis is sensitive to k, which is the number of ways (cache lines) in each cache set according to Theorem IV.6. In general, a larger k may increase the estimated WCET by our MRU analysis since it leads to more misses (in the estimation) for each k-Miss entry. Therefore, we also conduct experiments with 8-way and 16-way caches (with the same total cache size but different number of cache sets). The estimated WCET by our MRU analysis is on average 10.42% and 20.47% more than that obtained by the LRU analysis for 8-way and 16-way caches respectively, and in both cases the gap between the baseline analysis and the LRU analysis is similar with the case of 4-way caches. Note that it is very uncommon to use set-associative caches with more than 16 ways in realistic processors, especially the ones used in embedded systems, since a large number of ways will significantly increase the hardware cost and power consumption, but provides little performance benefit [12]. By the above experiment results we can see that the estimated WCET by our MRU analysis is quite close to that by LRU analysis under common hardware setting, which indicates that MRU is also a good candidate for the cache replacement policy in real-time embedded systems, especially considering MRU's other advantages in hardware, power and thermal efficiency.

As presented earlier in the paper, our analysis is based on the framework with separated cache and path analysis, and uses the analysis results under LRU to derive cache classifications under MRU. Therefore, our MRU analysis is as efficient as the state-of-the-art LRU analysis. In our experiments, the analysis of each program can terminate within one second.

VI. Conclusions and Future Work

In this work, we study the WCET analysis with the MRU cache replacement policy, which is used in mainstream processor architectures like Intel Nehalem. The MRU replacement has received little attention in real-time system research, as it is generally believed to be very unpredictable. This work discloses that actually the predictability of MRU has been significantly underestimated before, which is mainly because the cache hit/miss classification in existing cache analysis techniques, originally designed for LRU, does not match the MRU cache behavior well. The main technical contribution of this work is to define a new cache hit/miss classification k-Miss to more precisely capture the MRU behavior, and develop formal conditions and efficient techniques to determine the k-Miss memory references. A remarkable feature of our technique is to derive the k-Miss classifications under MRU by the analysis results of the same program under LRU. Experiments with programs from the Mälardalen Real-Time Benchmark suite on instruction caches indicate that our analysis yields precise cache hit/miss prediction. And most importantly, the experiments show that the predictability of MRU, explored by our analysis technique, is rather close to LRU. This suggests that processor platforms with MRU cache replacement are also a good candidate for real-time systems.

As future work, we will improve our MRU analysis precision by further exploring the k-Miss classifications with smaller k values (as discussed at the end of Section IV-C), and bounding the total number of misses for a group of entries (directly using Theorem IV.3). We also plan to evaluate and refine our analysis for data caches.

References

[1] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *ACM-SE*, 2004.

[2] C. Ballabriga and H. Casse. Improving the first-miss computation in set-associative instruction caches. In *ECRTS*, 2008.

[3] C. Cullmann. Cache persistence analysis: A novel approach theory and practice. In *LCTES*, 2011.

[4] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache pirating: Measuring the curse of the shared cache. In *ICPP*, 2011.

[5] C. Ferdinand. Cache behavior prediction for real-time systems. In *PhD Thesis, Universitat des Saarlandes*, 1997.

[6] D. Grund and J. Reineke. Abstract interpretation of fifo replacement. In *SAS*, 2009.

[7] D. Grund and J. Reineke. Precise and efficient fifo-replacement analysis based on static phase detection. In *ECRTS*, 2010.

[8] D. Grund and J. Reineke. Toward precise plru cache analysis. In *WCET*, 2010.

[9] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The mälardalen wcet benchmarks: Past, present and future. In *WCET*, 2010.

[10] D. Hardy and I. Puaut. Wcet analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.

[11] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of wcet tools. In *Proceedings of the IEEE*, 2003.

[12] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach (4. ed.)*. 2007.

[13] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-aware data cache analysis for wcet estimation. In *RTAS*, 2011.

[14] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. In *Sci. Comput. Program.*, 2007.

[15] Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *RTSS*, 2009.

[16] Y. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *RTSS*, 1996.

[17] A. Malamy, R. Patel, and N. Hayes. Methods and apparatus for implementing a pseudo-lru cache memory replacement scheme with a locking feature. In *United States Patent 5029072*, 1994.

[18] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. In *Real-Time Systems*, 2007.

[19] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Caches in wcet analysis - predictability, competitiveness, sensitivity. In *Ph.D. Thesis, Saarland University*, 2008.

[20] A. Roychoudhury S. Chattopadhyay and T. Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In *SCOPES*, 2010.

[21] R. Sen and Y. N. Srikant. Wcet estimation for executables in the presence of data caches. In *EMSOFT*, 2007.

[22] T. Sondag and H. Rajan. A more precise abstract domain for multi-level caches for tighter wcet analysis. In *RTSS*, 2010.

[23] H. Theiling. Control flow graphs for real-time system analysis: Reconstruction from binary executables and usage in ilp-based path analysis. In *Ph.D. Thesis in Universitat des Saarlandes*, 2002.

[24] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise wcet prediction by separated cache and path analyses. In *Real-Time Systems*, 2000.

[25] 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. Stenstrom. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transaction on Embedded Computing Systems*, 7, May 2008.

[26] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand. Memory hierarchies, pipelines and buses for future architectures in time-critical embedded systems. *IEEE Transaction on Computer-Aided Design of Integrated Circus Systems*, 28, July 2009.