WCET Analysis with MRU Cache: Challenging LRU for Predictability

NAN GUAN, Northeastern University, China, and Uppsala University, Sweden
MINGSONG LV, Northeastern University, China
WANG YI, Northeastern University, China, and Uppsala University, Sweden
GE YU, Northeastern University, China

Most previous work on 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, while 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 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 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 the advantages in efficiency and precision of the state-of-the-art LRU analysis techniques based on abstract interpretation. Experiments with instruction caches show that our proposed MRU analysis has both good precision and high efficiency, and the obtained estimated WCET is rather close to (typically 1%$\sim$8% more than) that obtained by the state-of-the-art LRU analysis, which indicates that MRU is also a good candidate for cache replacement policies in real-time systems.

Categories and Subject Descriptors: C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*

General Terms: Verification, Reliability

Additional Key Words and Phrases: Hard real time, worst-case execution times, cache replacement, MRU

ACM Reference Format:
Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. 2014. WCET analysis with MRU cache: Challenging LRU for predictability. ACM Trans. Embedd. Comput. Syst. 13, 4s, Article 123 (March 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2584655

123

1. INTRODUCTION

For hard real-time systems, one must ensure that all timing constraints are satisfied. To provide such guarantees, a key problem is to bound the worst-case execution time (WCET) of programs [Wilhelm et al. 2008]. To derive safe and tight WCET bounds, the analysis must take into account the timing effects of various microarchitecture features of the target hardware platform. Cache is one of the most important hardware components affecting the timing behavior of programs: the timing delay of a cache miss could be several orders of magnitude greater than that of a cache hit. Therefore,

M. Lv was partially sponsored by NSFC project (61100023).
Authors' address: N. Guan and W. Yi, Uppsala University, Sweden; email: {nan.guan, yi}@it.uu.se; M. Lv and G. Yu, Northeastern University, China; email: {lvmingsong, yuge}@ise.neu.edu.cn.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2014 ACM 1539-9087/2014/03-ART123 $15.00
DOI: http://dx.doi.org/10.1145/2584655

analyzing the cache access behavior is a key problem in WCET estimation. However, the cache analysis problem of statically determining whether each memory access is a hit or a miss is challenging.

Much work has been done on cache analysis for WCET estimation in the last two decades. Most of the published 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 [Wilhelm et al. 2008]. In contrast, much less attention has been paid to other replacement policies like MRU (Most-Recently-Used)[1] [Malamy et al. 1994], FIFO (First-In-First-Out) [Grund and Reineke 2009] and PLRU (Pseudo-LRU) [Heckmann et al. 2003]. In general, research in the field of real-time systems assumes LRU as the default cache replacement policy. Non-LRU policies in general, in fact, are considered to be much less predictable than LRU, and it would be very difficult to develop precise and efficient analyses for them. It is recommended to only use LRU caches when timing predictability is a major concern in the system design [Wilhelm et al. 2009].

However, most commercial processors actually do not employ the LRU cache replacement policy. The reason is that the hardware implementation logic of LRU is rather expensive [Hennessy and Patterson 2006], which results in higher hardware cost, power consumption and thermal output. On the other hand, non-LRU replacement policies like MRU, FIFO, and PLRU enjoy simpler implementation logic, but still have almost as good average-case performance as LRU [Al-Zoubi et al. 2004]. Therefore, hardware manufacturers tend to choose these non-LRU replacement policies in processor design, especially for embedded systems subject to strict cost, power and thermal constraints.

In this article, we study one of the most widely used cache replacement policies MRU. MRU uses a mechanism similar to the clock replacement algorithm in virtual memory mapping [Tanenbaum 2007]. It only uses one bit for each cache line to maintain age information, which is very efficient in hardware implementation. MRU has been employed in mainstream processor architectures like Intel Nehalem (the architecture codename of processors like Intel Xeon, Core i5 and i7) [Eklov et al. 2011] and UltraSPARC T2 [Kongetira et al. 2005]. A previous work comparing the average-case performance of cache replacement policies with the SPEC CPU2000 benchmark showed that MRU has almost as good average-case performance as LRU [Al-Zoubi et al. 2004]. To the best of our knowledge, there has been no previous work dedicated to the analysis of MRU in the context of WCET estimation. The only relevant work was performed by Jan Reineke et. al. [Reineke et al. 2007; Reineke and Grund 2008], which studies general timing predictability properties of different cache replacement policies. The cited work argues that MRU is a very unpredictable policy.

However, this article shows that the predictability of MRU actually has been significantly underestimated. The state-of-the-art cache analysis techniques are based on *qualitative* classifications, to determine whether the memory accesses related to a particular point in the program are always hits or not (except the first access that may be a cold miss). This approach is highly effective for LRU since most memory accesses indeed exhibit such a "black or white" behavior under LRU. In this work, we show that the memory accesses may have more nuanced behavior under MRU: a small number of the accesses are misses while all the other accesses are hits. By the existing analysis

[1]The name of the MRU replacement policy is inconsistent in the literature. 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 [Reineke et al. 2007; Reineke 2008], and to distinguish it from another Pseudo-LRU policy PLRU [Heckmann et al. 2003] which uses tree structures to store access history information.

framework based on qualitative classifications, such a behavior has to be treated as if all the accesses are misses, which inherently leads to very pessimistic analysis results.

In this article, we introduce a new cache hit/miss classification k-Miss (at most k accesses are misses while all the others are hits). In contrast to qualitative classifications, k-Miss can *quantitatively* bound the number of misses incurred at certain program points, hence it can more precisely capture the nuanced behavior in MRU. As our main technical contribution, we establish formal conditions to determine k-Miss memory accesses, and develop techniques to efficiently check these conditions. Notably, our technique uses the cache analysis results of the same program under LRU to derive k-Miss classification under MRU. Therefore, our technique inherits the advantages in both efficiency and precision from the state-of-the-art LRU analysis based on *abstract interpretation* (AI) [Theiling et al. 2000].

We conduct experiments with benchmark programs with *instruction* caches to evaluate the quality of our proposed analysis, which show that our MRU analysis has both good precision and efficiency: the estimated WCET obtained by our MRU analysis is on average 2%∼10% more than that obtained by simulations, and the analysis of each benchmark program terminates within 0.1 second on average. Moreover, the estimated WCET by our MRU analysis is close to (typically 1%∼8% 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 policies in real-time systems, especially considering MRU's other advantages in hardware cost, power consumption and thermal output.

Although the experimental evaluation in this article only considers instruction caches, the properties of MRU disclosed in this article also hold for data caches and our analysis techniques can be directly applied to systems with data caches. We didn't include experiments with data caches because predicting data cache behaviors heavily relies on value analysis [Wilhelm et al. 2008], which is another important topic in WCET estimation but orthogonal to the cache analysis issue studied in this paper. Since our prototype does not yet support high-quality value analysis functionalities, we currently cannot provide a meaningful evaluation with data caches. The evaluation of the proposed MRU analysis with data caches is left as our future work.

The rest of the article is organized as follows. Related work on cache analysis in WCET estimation is surveyed in Section 2. Section 3 introduces the problem model and basic concepts, and Section 4 reviews the state-of-the-art LRU cache analysis technique. Section 5 proposes a new classification k-Miss and presents our MRU cache analysis techniques. Experimental results are given in Section 6. Finally, Section 7 concludes the article and discusses possible future work.

2. RELATED WORK

Most previous work on cache analysis for static WCET estimation assumes the LRU replacement policy. Li et al. [Li and Malik 1995; Li et al. 1996] uses ILP-only approaches where the cache behavior prediction is formulated as part of the overall *integer linear programming* (ILP) problem. These approaches suffer from serious scalability problems due to the exponential complexity of ILP, and thus cannot handle realistic programs on modern processors. Mueller et al. [Arnold et al. 1994; Mueller 1994, 2000] proposed a technique called *static cache simulation*, which iteratively calculates the instructions that *may* be in the cache at the entry and exit of each basic block until the collective cache state reaches a fixed point, and then uses this information to categorize the caching behavior of each instruction.

A milestone in the research of static WCET estimation is establishing the framework combining micro-architecture analysis by *abstract interpretation* (AI) and path analysis by *implicit path enumeration technique* (IPET) [Theiling et al. 2000]. The AI-based cache analysis statically categorizes the caching behavior of each instruction by sound

Must, May, and Persist analyses, which have both high efficiency and good precision for LRU caches. The IPET-based path analysis uses the cache behavior classification to derive a delay invariant for each instruction and encodes the WCET calculation problem in into ILP formulation. Such a framework forms the common foundation for later research in cache analysis for WCET estimation. For example, it has been refined and extended to deal with nested loops [Ballabriga and Casse 2008; Cullmann 2011], data caches [Ferdinand and Wilhelm 1998; Sen and Srikant 2007; Huynh et al. 2011], multilevel caches [Hardy and Puaut 2008; Sondag and Rajan 2010], shared caches [Liang et al. 2012; Chattopadhyay et al. 2010] and cache-related preemption delay [Staschulat and Ernst 2007; Altmeyer et al. 2010].

In contrast, much less work has been done for non-LRU caches. Although some important progress has been made in the analysis of policies like FIFO [Grund and Reineke 2009, 2010a] and PLRU [Grund and Reineke 2010b], in general these analyses are much less precise than for LRU. To the best of our knowledge, there has been no work dedicated to the analysis of MRU in the context of WCET estimation.

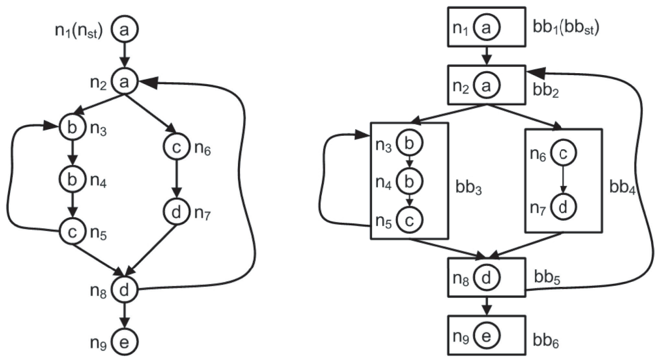
Jan Reineke et al. [Reineke et al. 2007; Reineke and Grund 2008, 2013; Reineke 2008] have conducted a series of fundamental studies on predictability properties of different cache replacement policies. Reineke et al. [2007] defines several *predictability* metrics, regarding the minimal number of different memory blocks that are needed to (i) completely clear the original cache content (evict), (ii) reach a completely known cache state (fill), (iii) evict a block that has just been accessed (mls). Reineke and Grund [2013] studies the *sensitivity* of different cache replacement policies, which expresses to what extent the initial state of the cache may influence the number of cache hits and misses during program execution. According to all the above metrics, LRU appears significantly more predictable than other policies like MRU, FIFO, and PLRU. Reineke and Grund [2008] studies the *relative competitiveness* between different policies by providing upper (lower) bounds of the ratio on the number of misses (hits) between two different replacement policies during the whole program execution. By such information, one can use the cache analysis result under one replacement policy to predict the number of cache misses (hits) of the program under another policy. This approach is different in many ways from our proposed analysis based on k-Miss classification. First, while the relative competitiveness approach provides bounds on the number of misses of the *whole program*,[2] the k-Miss classification bounds the number of misses at individual program points. Second, while the bounds on the number of misses provided by the relative competitiveness analysis is linear with respect to the total number of accesses, our k-Miss analysis provides constant bounds. Third, the k-Miss classification for MRU does not necessarily rely on the analysis result of LRU, and one can identify k-Miss by other means, for example, directly computing the maximal stack distance as defined in Section 4. Overall, our proposed analysis based on k-Miss can better capture MRU cache behavior and support a much more precise WCET estimation than the relative competitiveness approach.

Finally, we refer to Puschner and Burns [2000] and Wilhelm et al. [2008] for comprehensive surveys on WCET analysis techniques and tools, which cover many relevant references that are not listed here.

3. BASIC CONCEPTS

In this article, we assume an abstract processor architecture model: The processor has a perfect pipeline and instructions are fetched sequentially. The processor has a

[2] The relative competitiveness also can be used as Must/May analysis to predict the cache access behavior at individual program points. However, this relies on the analysis under other policies with typically a much smaller cache sizes (to get 1-competitiveness), which generally yields very pessimistic results.



(a) on the basis of nodes. (b) on the basis of basic blocks.

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

cache between the processing core and the main memory. The execution delay of each instruction only depends on whether the corresponding memory content is in the cache or not, and the time to deliver data from the main memory to the cache is constant. Other factors affecting the execution delay are not considered in this article.

We assume that the cache is *set-associative* or *fully-associative*. In set-associative caches, the accesses to memory references mapped to different cache sets do not affect each other, and each cache set can be treated as a fully-associative cache and analyzed independently. We present the cache analysis techniques in the context of a fully-associative cache for simplicity of presentation, and the experiments in Section 6 are all conducted with *set-associative* caches. Let the cache have L ways, that is, the cache consists of L *cache lines*. The memory content that fits into one cache line is called a *memory block*.

We consider the common class of programs represented by control-flow-graphs (CFG). Programs that are difficult to be modeled by CFGs, for example, self-modified programs, are usually not suitable for safe-critical systems and out of the scope of this article. A CFG can be defined on the basis of individual nodes as follows.

Definition 3.1 (*CFG on the Basis of Nodes*). A CFG is a tuple $G = (N, E, n_{st})$:

—$N = \{n_1, n_2, \ldots\}$ is the set of *nodes* in the CFG;
—$E = \{e_1, e_2, \ldots\}$ is the set of directed *edges* in the CFG;
—$n_{st} \in N$ is the unique *starting node* of the CFG.

A CFG can also be represented as a digraph of *basic blocks* [Allen 1970].

Definition 3.2 (*CFG on the Basis of Basic Blocks*). A CFG is a tuple $G = (B, E, \mathsf{bb}_{st})$:

—$B = \{\mathsf{bb}_1, \mathsf{bb}_2, \ldots\}$ is the set of *basic blocks* in the CFG;
—$E = \{e_1, e_2, \ldots\}$ is the set of directed *edges* connecting the basic blocks in the CFG;
—$\mathsf{bb}_{st} \in B$ is the unique *starting basic block* of the CFG.

Figure 1 shows a CFG example on the basis of individual nodes and basic blocks respectively. Letter $a, b, \ldots$ inside each node denotes the memory block accessed by the node. When we mention the CFG in the rest of this article, it is by default on the basis of nodes unless otherwise specified.

At runtime, when (a node of) the program accesses a memory block, the processor first checks whether the memory block is in the cache. If yes, it is a *hit*, and the program directly accesses this memory block from the cache. Otherwise, it is a *miss*, and this memory block is first installed in the cache before the program accesses it.

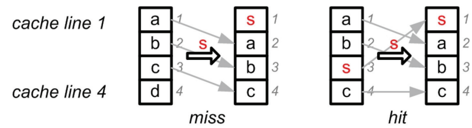


Fig. 2. Illustration of LRU cache update with $L = 4$, where the left part is a miss and the right part is a hit.

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

Definition 3.3 (*Stack Length*). The Stack Length of a memory access sequence corresponding to a path p in the CFG, denoted by $\mathsf{sl}(p)$, is the number of pairwise different memory blocks accessed along p.

For example, the stack length of the access sequence

$$a \to b \to c \to a \to d \to a \to b \to d$$

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

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

3.1. 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. Figure 2 illustrates the update upon an access to memory block s in an LRU cache of four 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 figures in this article follow this convention.

A metric defined in Reineke et al. [2007] to evaluate the predictability of a replacement policy is the *minimal-life-span* (mls), the minimal number of pairwise 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). The following is known [Reineke et al. 2007].

LEMMA 3.4. *The* mls *of* LRU *is* L.

Recall that L is the number of lines in the cache. 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

$$a \to b \to c \to c \to d \to a \to e \to b$$

on a 4-way LRU cache, we can easily conclude that the second access to memory block a is a hit since the sequence between two accesses to a is $b \to c \to c \to d$, which has stack length 3. The second access to b is a miss since the stack length of the sequence

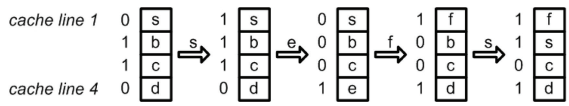


Fig. 3. An example illustrating MRU and its mls.

$c \to c \to d \to a \to e$ is 4. Clearly, replacement policies with larger mls are preferable, and the upper bound of mls is L.

3.2. 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 at 1 indicates that this line was recently visited, while at 0 indicates the opposite. Whenever a cache line is visited, its MRU-bit will be set to 1. Eventually, there will be only one MRU-bit at 0 in the cache. When the cache line with the last MRU-bit at 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 replace the originally stored memory block in it by s and set its MRU-bit to 1. After these operations, if there still exists some MRU-bit at 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 $\mathsf{C} : \{1, \ldots, L\} \to M \times \{0, 1\}$. We use $\mathsf{C}(i)$ to denote the state of the ith 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 such that $\mathsf{C}(h).\omega = s$, then let $\delta \leftarrow h$, otherwise let $\delta = h$ such that $\mathsf{C}(h).\beta = 0$ and $\mathsf{C}(j).\beta = 1$ for all $j < h$.
(2) $\mathsf{C}'(\delta) \leftarrow (s, 1)$.
(3) If $\mathsf{C}(h).\beta = 1$ for all $h \neq \delta$, 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 $j \neq \delta$.

Figure 3 illustrates MRU replacement with a 4-way cache. First, the program accesses memory block s, which is already in the cache. So s still stays in the same cache line, and the corresponding MRU-bit is changed to 1. Then the program accesses e, which is not in the cache yet. Since only the 4th cache line's MRU-bit is 0, e is installed in that line and triggers the global-flip, after which the 4th cache line's MRU-bit is 1 and all the other MRU-bits are changed to 0. Then, the program accesses f and s in order, which are both not in the cache, so they will be installed to the first and second cache line with MRU-bits at 0 and change these bits to 1.

In MRU caches, an 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 [Reineke et al. 2007].

LEMMA 3.5. *The* mls *of* MRU *is* 2.

The example in Figure 3 illustrates this lemma, 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 arbitrarily many cache lines, where we still only need two memory blocks to evict s. Partly due to this property, MRU has been believed to be a very unpredictable replacement policy, and to the best of our knowledge it has never been seriously considered as a good candidate for timing-predictable architectures.

4. A REVIEW OF THE ANALYSIS FOR LRU

As we mentioned in Section 1, the MRU analysis proposed in this article uses directly the results of the LRU analysis for the same program. 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. Hence, researchers resort to approximation techniques separating path analysis and cache analysis for good scalability [Theiling et al. 2000]. Path analysis requires an upper bound on the timing delay of a node whenever it is executed. Therefore, the main purpose of the LRU cache analysis is to decide the cache hit/miss classification (CHMC) for each node [Arnold et al. 1994; Theiling et al. 2000].

—AH (always hit). The node's memory access is always hit whenever it is executed.
—FM (first miss). The node's memory access is miss for the first execution, but always hit afterwards. This classification is useful to handle "cold miss" in loops.
—AM (always miss). The node's memory access is always miss whenever it is executed.
—NC (non-classified). Cannot be classified into any of the above categories. This category has to be treated as AM in the path analysis.

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

Recall that the mls of LRU is L, and one can directly use this property to decide the hit/miss of a node with linear access sequences. However, a CFG is generally a digraph, and there may be multiple paths between two nodes.

The following concept captures the maximal number of pairwise different memory blocks between two nodes accessing the same memory block in the CFG.

Definition 4.1 (*Maximal Stack Distance*). Let n_i and n_j be two nodes 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 defined as:

$$\mathsf{dist}(n_i, n_j) = \begin{cases} \max\{\mathsf{sl}(p) \mid p \in \Pi(n_i, n_j)\} & \text{if } \Pi(n_i, n_j) \neq \emptyset \\ 0 & \text{if } \Pi(n_i, n_j) = \emptyset \end{cases}$$

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

—n_i and n_j is the first and last node of the path, respectively;
—None of the nodes in the path, except the first and last, accesses s.

Note that the maximal stack distance between two nodes is direction sensitive, that is, $\mathsf{dist}(n_i, n_j)$ may not be equal to $\mathsf{dist}(n_j, n_i)$. The example in Figure 4 illustrates the maximal stack distance using a CFG with three nodes n_1, n_3 and n_7 accessing the same

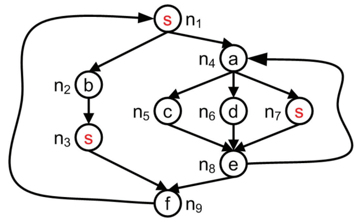


Fig. 4. Illustration of Maximal Stack Distance.

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 and 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 $\mathsf{dist}(n_3, n_7) = 0$ since any path from n_3 to n_7 has to go through n_1 which also accesses s.

Now one can use the maximal stack distance to judge whether the CHMC of a node n_i is positive: n_j falls into the positive classification (AH or FM), if $\mathsf{dist}(n_i, n_j) \leq L$ holds for any node n_i that accesses the same memory block s as n_j. This is because there are not enough pairwise different memory blocks to evict s along any path to n_i since the last access to s.

However, computing the exact maximal stack distance is, in general, very expensive. Therefore, the LRU analysis resorts to overapproximation 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 fixed-point analyses:

—Must analysis to determine AH nodes,
—May analysis to determine AM nodes,
—Persist analysis to determine FM nodes.

A node is a NC if it cannot be classified by any of these analyses. We refer to Ferdinand [1997] and Huynh et al. [2011] for details of these fixed-point analyses.

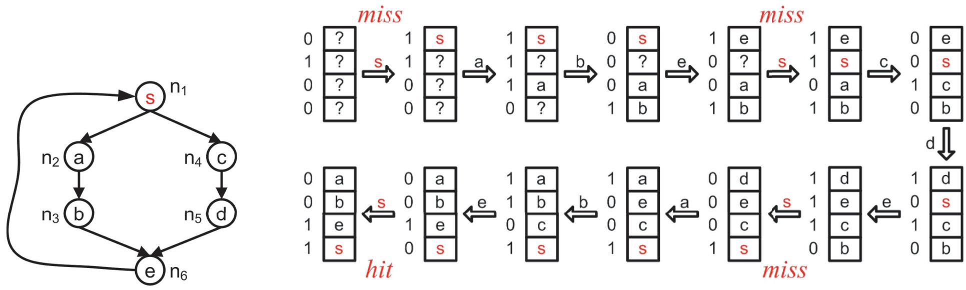
5. THE NEW ANALYSIS OF MRU

In this section, we present our new analysis for MRU. First, we show that the existing CHMC in the LRU analysis as introduced in last section is actually not suitable to capture the cache behavior under MRU, and thus we introduce a new classification k-Miss (Section 5.1). After that, we introduce the conditions for nodes to be k-Miss (Section 5.2), and show how to efficiently check these conditions (Section 5.3). Then, the k-Miss classification is generalized to more precisely analyze nested-loops (Section 5.4). Finally, we present how to apply the cache analysis results in the path analysis to obtain the WCET estimation (Section 5.5).

5.1. New Classification: *k*-Miss

First, we consider the example in Figure 5(a). We can easily see that $\mathsf{dist}(n_1, n_1) = 4$, that is, 4 pairwise different memory blocks appear in each iteration of the loop no matter which branch is taken. Since $\mathsf{dist}(n_1, n_1)$ is larger than 2 (the mls of MRU), n_1 cannot 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 Figure 5(b). Assume that the memory blocks initially stored in the cache (denoted by "?") are all



(a) A CFG example. (b) Cache update when the two branches are taken alternatively.

Fig. 5. An example motivating the k-Miss classification.

different from the ones that appear in Figure 5(a), and initial MRU-bits are shown in the first cache state of Figure 5(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. Indeed, node n_1 cannot be determined as AH or FM, and one has to put it into the negative classification and treat it as being always 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 sequence, but 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 nodes like n_1 in this example, which only encounters a smaller number of misses, but will eventually go into a stable state of being always hits. Such behavior is actually quite common under MRU. Therefore, the analysis of MRU will be inherently very pessimistic if one only relies on the AH and FM classification to claim cache hits.

This phenomenon shows the need for a more precise classification to capture the MRU cache behavior. As we show in Section 5.2, the number of misses under MRU may be bound not only for individual nodes, but also for a set of nodes that access the same memory block. This leads us to the definition of the k-Miss classification as follows.

Definition 5.1 (k-Miss). A set of nodes $S = \{n_1, \ldots, n_i\}$ is k-Miss iff at most k accesses by nodes in S are misses while all the other accesses are hits.

The traditional classification FM can be viewed as a special case of k-Miss with a singleton node set and $k = 1$. Note that although the k-Miss classification can bound the number of misses for a set of nodes, it does not say anything about when do these k times of misses actually occur. The misses do not necessarily occur at the first k accesses of these nodes. It allows the misses and hits to appear alternatively, as long as the total number of misses does not exceed k.

5.2. Conditions for *k*-Miss

In this section, we establish the conditions for a set of nodes to be k-Miss. We start with an important property of MRU.

LEMMA 5.2. *At least k pairwise different memory blocks are needed to evict a memory block in cache line k with MRU-bit at* 1.

PROOF. Only the memory block in a cache line with MRU-bit at 0 can be evicted, so before the eviction of s there must be a global-flip to change the MRU-bit of cache line

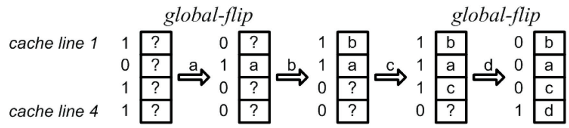


Fig. 6. Illustration of Lemma 5.3.

k from 1 to 0. Right after the global flip, the number of 0-MRU-bits among cache lines $\{1, \ldots, k\}$ is at least $k - 1$, so $k - 1$ pairwise different memory blocks (which are also different from the one triggering the global-flip) are needed to fill up these 0-MRU-bit cache lines. In total, the number of pairwise different memory blocks required is at least k. □

Lemma 5.2 indicates that the minimal-life-span of memory blocks installed to different cache lines are asymmetric: a cache line with a greater index provides a larger minimal-life-span guarantee (while the mls metric does not distinguish different positions but simply captures the worst case). To provide a better analysis than the mls approach, one needs information about where a memory block is installed. However, under MRU a memory block may be installed to any cache line without restricting the cache state beforehand. Since the initial cache state is unknown, and the precise cache state information is lost quickly during the abstract analysis, it is difficult to precisely predict the position of a memory block in the cache.

However, Lemma 5.2 indeed gives us opportunities to do a better analysis. When a memory block is installed to a cache line with a larger index, it becomes more difficult to be evicted. So the main idea of our analysis is to verify whether a memory block will eventually be installed to a "safe position" (a cache line with large enough index) and stay there afterwards (as long as it executes in the scope of the program under analysis). The k times of misses in k-Miss happens before the memory block is installed to the "safe position", and after that all the accesses will be hits. In the following, we show the condition for a memory block to have such behavior. We first introduce an auxiliary lemma.

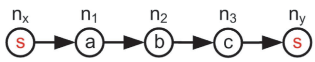
LEMMA 5.3. *On an L-way* MRU *cache, L pairwise different memory blocks are accessed between two successive global-flips (including the ones triggering these two global-flips).*

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, that is, it needs $L - 1$ pairwise different memory blocks that are also different from the one causing the first global-flip. So in total L pairwise different memory blocks are involved in the access sequence between two successive global-flips. □

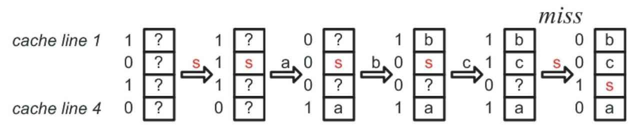
Lemma 5.3 is illustrated by the example in Figure 6 with $L = 4$. The access to memory block a triggers the first global-flip, after which three MRU-bits are 0. To trigger the next global-flip, these three MRU-bits have to be changed to 1, which needs three pairwise different memory blocks. So, in total, four pairwise different memory blocks are involved in the access sequence between these two global-flips. With this auxiliary lemma, we are able to prove the following key property.

LEMMA 5.4. *Suppose that under* MRU *at some point a memory block s is accessed by node n_x at cache line i (either hit or miss), 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:*

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



(a) a path from n_x to n_y



(b) s is moved to a larger index when it is loaded back

Fig. 7. Illustration of Lemma 5.4.

Figure 7 illustrates Lemma 5.4, where n_x and n_y are two nodes accessing the same memory block s and satisfying Condition (1). We focus on a particular path as shown in Figure 7(a). Figure 7(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. Let event ev_x be the access to s at cache line i by n_x as stated in the lemma, and event ev_y the installation of s to cache line j by n_y. We prove the lemma by contradiction, assuming $j \leq i$.

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

Before ev_y, s has to be first evicted out of the cache. Let event ev_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* ev_v.

On the other hand, we also know that $\mathsf{C}(i).\beta = 1$ *right after* event ev_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 $\{ev_{x+1}, \ldots, ev_{v-1}\}$.

Then we focus on the event sequence $\{ev_v, \ldots, ev_{y-1}\}$. We distinguish two cases.

—$i = j$. Right after the eviction of s at cache line i (event ev_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 ev_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 $\{ev_{v+1}, \ldots, ev_{y-1}\}$, 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 ev_v, it must be true that $\forall h < i : \mathsf{C}(h).\beta = 1$, and hence $\mathsf{C}(j).\beta = 1$. On the other hand, just before the installation of s in event ev_y, the MRU-bit of cache line j must be 0. Therefore, there must be at least one global-flip among the events $\{ev_v, \ldots, ev_{y-1}\}$, 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 $\{ev_v, \ldots, ev_{y-1}\}$.

Therefore, we can conclude that there are at least two global-flips among the events $\{ev_{x+1}, \ldots, ev_{y-1}\}$. By Lemma 5.3, we know that at least L pairwise different memory blocks are accessed in $\{ev_{x+1}, \ldots, ev_{y-1}\}$. Since ev_y is the first access to memory block s after ev_x, there is no access to s in $\{ev_{x+1}, \ldots, ev_{y-1}\}$, so at least $L+1$ pairwise different memory blocks are accessed in $\{ev_x, \ldots, ev_y\}$.

On the other hand, let p be the path that leads to the event sequence $\{ev_x, \ldots, ev_y\}$. Clearly, p starts with n_x and ends with n_y. We also know that no other node along p, apart from n_x and n_y, accesses s, since ev_y is the first event accessing s after ev_x. So p is a path in $\Pi(n_x, n_y)$ (Definition 4.1), and we know $\mathsf{dist}(n_x, n_y) \geq \mathsf{sl}(p)$. Combining this with Condition (1), we have $\mathsf{sl}(p) \leq L$, which contradicts with that at least $L+1$ pairwise different memory blocks are accessed in $\{ev_x, \ldots, ev_y\}$ as we concluded here. □

Fig. 8. An example illustrating the usage of Lemma 5.4.

To see the usefulness of Lemma 5.4, we consider a special case where only one node n in the CFG accesses memory block s and $\mathsf{dist}(n, n) \leq L$ as shown in Figure 8(a). In this case, by Lemma 5.4, we know that each time s is accessed (except the first time), 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 the access to s can only be 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. Moreover, we can combine Lemma 5.2 and Lemma 5.4 to have a stronger claim: if condition $\mathsf{dist}(n, n) \leq k$ holds for some $k \leq L$, then the access to s can only be miss for at most k times, since the number of pairwise different memory blocks along the path from n back to n is not enough to evict s as soon as it is installed to cache line k.

However, in general, there could be more than one node in the CFG accessing the same memory block, where Lemma 5.4 cannot be directly applied to determine the k-Miss classification. Consider the example in Figure 8(b), where two nodes 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 cannot classify n_2 as a k-Miss, although Lemma 5.4 still applies to the path from n_1 to n_2. This is because Lemma 5.4 only guarantees the position of s will move to larger indices each time n_2 encounters a miss, but the position of s may move to smaller indices upon misses of n_1 (since $\mathsf{dist}(n_2, n_1) > L$), which breaks down the memory block's movement monotonicity.

In order to use Lemma 5.4 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 nodes. This can be done by examining the condition of Lemma 5.4 for all node pairs in a *strongly connected component* (maximal strongly connected subgraph) together, as described in the following theorem.

THEOREM 5.5. *Let SCC be a strongly connected component in the CFG, let S be the set of nodes in SCC accessing the same memory block s. The total number of misses incurred by all the nodes in S is at most k if the following condition holds:*

$$\forall n_x, n_y \in S : \mathsf{dist}(n_x, n_y) \leq k, \tag{2}$$

where k is bounded by the number of cache lines L.

PROOF. Let ev_f and ev_l be the first and last events triggered during program execution. Since S is a subset of the strongly connected component SCC, any event accessing s in the event sequence $\{ev_f, \ldots, ev_l\}$ has to be also triggered by some node in S (otherwise, there will be a cycle including nodes both inside and outside SCC, which contradicts SCC is a strongly connected component).

By $k \leq L$, Condition (2), and Lemma 5.4, we know that among the events $\{ev_f, \ldots, ev_l\}$ whenever the access to s is a miss, s will be installed to a cache line with a strictly larger index than before. Since every time after s is accessed in the cache (either hit or miss), the corresponding MRU-bit is 1, so by Condition (2) and Lemma 5.2, we further know that among the events $\{ev_f, \ldots, ev_l\}$, as soon as s is installed to a cache line with index equal to or larger than k, it will not be evicted. In summary, there are at most k misses of s among events $\{ev_f, \ldots, ev_l\}$, that is, the nodes in S have at most k misses in total. □

5.3. Efficient *k*-Miss Determination

Theorem 5.5 gives us the condition to identify k-Miss node sets. The major task of checking this condition is to calculate the maximal stack distance dist(). As mentioned in Section 4, the exact calculation of dist() is very expensive, which is the reason why the analysis of LRU relies on AI to obtain an over-approximate classification. For the same reason, we also resort to over-approximation to efficiently check the conditions of k-Miss. The main idea is to use the analysis result for the same program under LRU to infer the desired k-Miss classification under MRU.

LEMMA 5.6. *Let n_y be a node that accesses memory block s and is classified as* AH / FM *by* Must / Persist *analysis with a k-way* LRU *cache. For any node n_x that also accesses s, if there exists a cycle in the CFG including n_x and n_y, then the following must hold:*

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

PROOF. We prove the lemma by contradiction. Let n_x be a node that also accesses s and there exists a cycle in the CFG including n_x and n_y. We assume that $\mathsf{dist}(n_x, n_y) > k$. Then, by the definition of $\mathsf{dist}(n_x, n_y)$, we know that there must exist a path p from n_x to n_y satisfying (i) $\mathsf{sl}(p) > k$ and, (ii) no other node accesses s apart from the first and last node along this path (otherwise, $\mathsf{dist}(n_x, n_y) = 0$). This implies that under LRU, 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 n_y is classified as AH/FM by the Must/Persist analysis with a k-way LRU cache (Must/Persist yields *safe* classification, so in the real execution, an AH node will never miss and an FM node can miss for at most once). □

THEOREM 5.7. *Let SCC be a strongly connected component in the CFG, and S the set of nodes in SCC that access the same memory block s. If all the nodes in S are classified as* AH *by* Must *analysis or* FM *by* Persist *analysis with a k-way* LRU *cache, then the node set S is k-*Miss *with an L-way* MRU *cache for $k \leq L$.*

PROOF. Let n_x, n_y be two arbitrary nodes in S, so both of them access memory block s and are classified as AH/FM by the Must/Persist analysis with a k-way LRU cache. Since S is a subset of a strongly connected component, we also know n_x and n_y are included in a cycle in the CFG. Therefore, by Lemma 5.6, we know $\mathsf{dist}(n_x, n_y) \leq k$. Since n_x, n_y are arbitrarily chosen, this conclusion holds for any pair of nodes in S. Therefore, S can be classified as k-Miss according to Theorem 5.5. □

Theorem 5.7 tells that we can identify k-Miss node sets with a particular k by doing Must/Persist analysis with an LRU cache of the corresponding number of ways. Actually, we only need to do the Must and Persist analysis once with an L-way LRU cache, to identify k-Miss node sets with *all* different k ($\leq L$). This is because the Must and Persist analysis for LRU cache maintains the information about the maximal age of a memory block at certain point in the CFG, which can be directly transferred to the analysis result with any cache size smaller than L. For example, suppose by the Must analysis with an L-way LRU cache, a memory block s has maximal age of k before the access of a node n, then by the Must analysis with a k-way LRU cache this node n will be classified as AH. We will not recite the details of Must and Persist analysis for LRU cache or

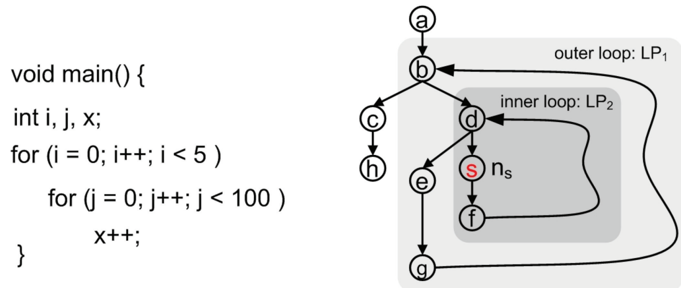


Fig. 9. A program with nested loop and its (simplified) CFG.

explain how the age information is maintained in these analysis procedures, but refer interested readers to Theiling et al. [2000] and Huynh et al. [2011].

Moreover, the maximal age information in the Must and Persist analysis with an 2-way LRU cache can also be used to infer traditional AH and FM classification under MRU according to the relative competitiveness property between MRU and LRU [Reineke 2008]: an L-way MRU cache is 1-competitive relative to a 2-way LRU cache, so a Must (Persist) analysis with a 2-way LRU cache can be used as a sound Must (Persist) analysis with an L-way MRU cache. Therefore, if the maximal age of a node in a Must (Persist) analysis with an L-way LRU cache is bounded by 2 ($L \geq 2$), then this node can be classified as AH (FM) with an L-way MRU cache. Adding this competitiveness analysis optimization helps us to easily identify AH nodes when several nodes in a row access the same memory block. For example, if a memory block (i.e., a cache line) contains two instructions, then in most cases the second instruction is accessed right after the first one, so we can conclude that the second node is AH with a 2-way LRU cache, and thus is also AH with an L-way MRU. Besides dealing with this easy case, the competitiveness analysis optimization sometimes can do more for *set-associative* caches with a relatively large number of cache sets. For example, consider a path accessing 16 pairwise different memory blocks, and a set-associative cache of 8 sets. On average only 2 memory blocks on this path are mapped to each set, so competitiveness analysis may have a good chance to successfully identify some AH and FM nodes.

5.4. Generalizing *k*-Miss for Nested Loops

Precisely predicting the cache behavior of loops is very important for obtaining tight WCET estimations. In this article, we simply define a loop LP_ℓ as a *strongly connected subgraph* in the CFG.[3] (Note the difference between a strongly connected *subgraph* and a strongly connected *component*.)

The ordinary CHMC may lead to over-pessimistic analysis when loops are nested. For example, Figure 9 shows a program containing two-level nested loops and its (simplified) CFG. Suppose the program executes with a 4-way LRU cache. Since $\mathsf{dist}(n_s, n_s) = 6 > 4$ (see $s \rightarrow f \rightarrow d \rightarrow e \rightarrow g \rightarrow b \rightarrow d \rightarrow s$), the memory block s can be evicted out of the cache repeatedly, and thus we have to put n_s into the negative classification according to the the ordinary CHMC, and treat it as being always miss whenever it is accessed. However, by the program semantics, we know that every time the program enters the inner loop it will iterate for 100 times, during which s will not be evicted out of the cache since the inner loop can be fit into the cache entirely. So

[3]In realistic programs, loop structures are usually subject to certain restrictions (e.g., a *natural* loop has exactly one header node which is executed every time the loop iterates, and there is a path back to the header node [Aho et al. 1986]). However, the properties presented in this section is not specific to any particular structure, so we define a loop in a more generic way.

node n_s has only 5 misses out of the total 500 cache accesses during the whole program execution. Putting n_s into the negative classification and treating it as being always miss is obviously over-pessimistic.

To solve this problem, Ferdinand [1997] and Ballabriga and Casse [2008] reloaded the FM classification by relating it to certain loop scopes.

Definition 5.8 (FM *Regarding a Loop*). A node is FM regarding a loop LP_ℓ iff it has at most one miss (at the first access) and otherwise will be always hit when the program executes inside LP_ℓ.

In this example, node n_s is FM regarding the inner loop LP_2.

The same problem also arises for MRU. Suppose the program in Figure 9 runs with a 4-way MRU cache. For the same reason as under LRU, node n_s has to be put into the negative classification category. However, we have $\mathsf{dist}(n_s, n_s) = 3$ if only looking at the inner loop, which indicates that n_s can be miss for at most 3 times every time it executes inside the inner loop. As with FM, we can reload the k-Miss classification to capture this locality.

Definition 5.9 (k-Miss *Regarding a Loop*). A node is k-Miss regarding a loop LP_ℓ of the CFG iff it has at most k misses and all the other accesses are hits when the program executes inside LP_ℓ.

The sought k-Miss classification under MRU for a loop can be inferred from applying the FM classification under LRU to the same loop.

THEOREM 5.10. *Let LP_ℓ be a loop in the CFG, and S the set of nodes in the loop that access the same memory block s. If all the nodes in S are classified as* FM *regarding* LP_ℓ *with a k-way* LRU *cache ($k \leq L$), then the node set S is k-*Miss *regarding* LP_ℓ *with an L-way* MRU *cache.*

PROOF. Similar to the proof of Theorem 5.5 and 5.7. □

A node may be included in more than one k-Miss node sets regarding different loops. This typically happens across different levels in nested loops. For example, if the program in Figure 9 executes with an 8-way MRU cache, then by Theorem 5.10 $\{n_s\}$ is classified as 3-Miss regarding the inner loop and 6-Miss regarding the outer loop. The miss number constraints implied by k-Miss with different k and different loops are generally incomparable. For example, with the loop bound setting in Figure 9, 3-Miss regarding the inner loop allows at most $3 \times 5 = 15$ misses during the whole execution, which is "looser" than the outer loop 6-Miss which allows at most 6 misses. However, if we change the outer loop bound to 1, then the inner loop 3-Miss actually poses a "tighter" constraint as it only allows 3 misses while the outer loop 6-Miss still allows 6 misses. Although it is possible to explore program structure information to remove redundant k-Miss, we simply keep all the k-Miss classifications in our implementation since the ILP solver for path analysis can automatically and efficiently exclude such redundancy, as we illustrate in the next section.

5.5. WCET Computation by IPET

By now, we have obtained the cache analysis results for MRU:

—k-Miss node sets that are identified by Theorem 5.7 and Theorem 5.10,
—AH and FM nodes that are identified using the relative competitiveness property between MRU and LRU as stated at the end of Section 5.3,
—all the nodes not included in these two categories are NC.

Note that a node classified as AH by the relative competitiveness property may also be included in some k-Miss node set. In this case, we can safely exclude this node from the k-Miss node set, since AH provides a strong guarantee and the total number of misses incurred by other nodes in that k-Miss set is still bounded by k.

In the following, we present how to apply these results in the path analysis by *implicit path enumeration technique* (IPET) to obtain the WCET estimation. The path analysis adopts a similar ILP formulation framework to the standard, but it is extended to handle k-Miss node sets. All the variables in the following ILP formulation are nonnegative, which will not be explicitly specified for simplicity of presentation.

To obtain the WCET, the following maximization problem is solved:

$$Maximize \quad \left\{ \sum_{\forall \mathsf{bb}_a} cost_a \right\},$$

where $cost_a$ denotes the overall execution cost of basic block bb_a (on the worst-case execution path). Since a basic block typically contains multiple nodes with different CHMC, the execution cost for each basic block is further refined as follows.

We assume the execution delay inside the processing unit is constant for all nodes, and the total execution delay of a node only differs depending on whether the cache access is a hit or a miss: C^h upon a hit and C^m upon a miss. Since the accesses of an AH node are always hits, the overall execution delay of an AH node n_i in bb_a is simply $C^h \times x_a$ where the variable x_a represents the execution count of bb_a. Similarly, the overall execution delay of a NC node is $C^m \times x_a$. The remaining nodes are the ones included in some k-Miss node sets (regarding some loops). For each of such nodes n_i, we use a variables z_i ($\leq x_a$) to denote the execution count of n_i with cache access being miss. So the overall execution delay of a node n_i in some k-Miss node set is $C^m \times z_i + C^h \times (x_a - z_i)$. Putting these discussions together, we have the total execution cost of a basic block bb_a:

$$cost_a = (\pi_{\mathsf{AH}} \times C^h + \pi_{\mathsf{NC}} \times C^m) \times x_a + \sum_{n_i \in \mathsf{bb}_a^*} \big(C^m \times z_i + C^h \times (x_a - z_i)\big),$$

where π_{AH} and π_{NC} is the number of AH and NC nodes in bb_a respectively, and bb_a^* is the set of nodes in bb_a that are contained in some k-Miss node sets (regarding some loops). Since at most k misses are incurred by a k-Miss node set regarding a loop LP_ℓ every time the program enters and iterates inside the loop, we have the following constraints to bound z_i:

$$\forall (S, \mathsf{LP}_\ell) \text{ such that } S \text{ is } k\text{-Miss regarding } \mathsf{LP}_\ell : \sum_{n_i \in S} z_i \leq k \times \sum_{e_j \in \mathsf{entry}_\ell} y_j,$$

where entry_ℓ is the set of edges through which the program can enter LP_ℓ and we use variable y_j to denote how many times an edge $e_j \in \mathsf{entry}_\ell$ is taken during program execution. Recall that a node may be contained by multiple k-Miss sets (e.g., k-Miss regarding both the inner and outer loop with different k), so each z_i may be involved in several of these constraints.

Besides these constraints, the formulation also contains *program structural constraints* which are standard components of the IPET encoding. The WCET of the program is obtained by solving this maximization problem, and the execution count for each basic block along the worst-case path is also returned.

6. EXPERIMENTAL EVALUATION

The main purpose of the experiments is to evaluate

(1) the precision of our proposed MRU analysis, and
(2) the predictability comparison between LRU and MRU.

To evaluate (1), we compare the estimated WCET obtained by our MRU analysis and the measured WCET obtained by simulation with MRU caches. To evaluate (2), we compare the estimated WCET obtained by our MRU analysis and that by the state-of-the-art LRU analysis based on abstract interpretation (Must and May analysis in Theiling et al. [2000] and Persist analysis in Huynh et al. [2011]). The smaller is the difference between the estimated WCET by our MRU analysis and by the LRU analysis, the more confident we are to claim that MRU is also a good candidate for cache replacement policies in real-time embedded systems, especially taking into account MRU's other advantages in hardware cost, power consumption and thermal output.

6.1. Experiment Setup

Hardware Configuration. As presented in Section 5.5, we assume the execution delay of each node only differs depending on whether the cache access is a hit or miss. The programs execute with a $1K$ bytes set-associative instruction cache. Each instruction is 8 bytes, and each cache line (memory block) is 16 bytes (i.e., each memory block contains two instructions). All instructions have a fixed latency of 1 cycle. The memory access penalty is 1 cycle upon a cache hit, and 10 cycles upon a cache miss. To conduct experiments with cache of different number of ways, we keep the total cache size fixed and change the number of cache sets correspondingly. Although the experiments in this article are conducted with *instruction* caches, the theoretical results of this work also directly apply to data caches, and we leave the evaluation for data caches as our future work.

Benchmark. The programs used in the experiments are from the Mälardalen Real-Time Benchmark suite [Gustafsson et al. 2010]. Some programs in the benchmark are not included in our experiments since the CFG construction engine (from Chronos [Li et al. 2007]) used in our prototype does not support programs with particular structures like recursion and switch-case very well. The loop bounds in the programs that cannot be automatically inferred by the CFG construction engine are manually set to be 50. The size of these programs used in our experiments ranges from several tens to about 4000 lines of C code, or from several tens to about 8000 assembly instructions compiled by a gcc compiler retargeted to the SimpleScaler simulator [Austin et al. 2002] with $-O0$ option (no optimization is allowed in the compilation).

Simulation Methodology. Since the benchmark programs have been compiled by a gcc compiler retargeted to SimpleScalar, a straightforward way of doing the simulation is to execute the compiled binary on SimpleScalar (configured and modified to match our hardware configuration). However, the comparison between the measured execution time by this approach and the estimated WCET may be meaningless to evaluate the quality of our MRU analysis since (i) simulations may only cover program paths that are much shorter than the actual worst-case path, and (ii) the precision of the estimated WCET also depends on other factors, for example, the tightness of the loop bounds, which is out of the interest of this article. In other words, the estimated WCET can be always significantly larger than the measured execution time obtained by this approach, regardless the quality of the cache analysis.

In order to provide meaningful quality evaluation of our MRU cache analysis, we built an in-house simulator, which is driven by the worst-case path information extracted from the solution of the IPET ILP formulation and only simulates the cache update upon each instruction. This enables us to get closer to the worst-case path in the simulation and exclude effects of other factors orthogonal to the cache behavior. Note that the solution of the IPET ILP formulation only restricts how many times a basic block executes on the worst-case path, which allows the flexibility of arbitrarily choosing among branches as long as the execution counts of basic blocks still comply with the ILP solution. In order to obtain execution paths that are as close to the worst-case path as possible, our simulator always takes different branches alternatively which leads to more cache misses. The manual and source code of the simulator are online available [Lv 2012].

6.2. Results and Discussions

Table I is the simulation and analysis results with 4-way caches. In the simulation with each cache, for each program we record the measured execution time (column "sim. WCET") and the number of hits and misses. In the analysis with each cache, for each program we record the estimated WCET (column "est. WCET") and the number of memory accesses that can and cannot be classified as hit (column "hit" and "miss") respectively. We calculate the over-estimation ratio of the LRU and MRU analysis respectively (column "over est."). For example, the "sim. WCET" and "est. WCET" of program *bs* under LRU is 3911 and 3947 respectively, then the overestimation ratio is $(3947 - 3911)/3911 = 0.92\%$. Finally, we calculate the excess ratio of MRU analysis over LRU analysis (column "exc. LRU"). For example, the estimated WCET of program *bs* under LRU and MRU is 3947 and 4089, respectively, then the excess ratio is $(4089 - 3947)/3947 = 3.60\%$.

Table I shows that the WCET estimation with our MRU analysis has very good precision: the overestimation comparing with the simulation WCET is on average 2.06%. We can also see that the estimated WCETs with MRU and LRU caches are very close: the difference is 1.17% on average.

For several benchmark programs, the simulated WCETs are exactly the same under LRU and MRU. The reason is that MRU is designed to imitate the LRU policy with a cheaper hardware logic. In some cases, the cache miss/hit behavior under MRU could be exactly the same as that under LRU, and thereby we may obtain exactly the same simulated WCET with MRU and LRU for some programs. Moreover, the total number of memory accesses in the simulation may be different with two policies for the same program. This is because our simulator simulates the program execution with each policy according to the "worst-case" path information obtained from the solution of the corresponding ILP formula for WCET calculation. Sometimes, the ILP solutions with these two policies may correspond to different paths in the program, which may lead to different total numbers of memory accesses.

Then we conduct experiments with 8-way and 16-way caches (with the same total cache size but different number of cache sets). Note that it is rare to see set-associative caches with more than 16 ways in embedded systems, since a large number of ways significantly increases hardware cost and timing delay but brings little performance benefit [Hennessy and Patterson 2006]. So we did not conduct experiments with caches with more than 16 ways. Figure 10 summarizes the results with 8-way and 16-way caches, where the WCETs are normalized as the ratio versus the simulation results under LRU. The overestimation by our MRU analysis is 4.59% and 9.81% for 8-way and 16-way caches respectively, and the difference between the MRU and LRU analysis is 3.56% and 8.38%. Overall, our MRU analysis still provides quite good precision on 8-way and 16-way caches.

We observe that for most programs the overestimation ratio of the WCET by our MRU analysis scales about *linearly* with respect to the number of ways, the reason of which can be explained as follows. The k times of misses of k-Miss nodes is merely a

Table I. Experiment Results with 4-Way Caches

Program	policy	Simulation hit	Simulation miss	Simulation sim. WCET	Analysis hit	Analysis miss	Analysis est. WCET	Analysis over est.	Analysis exc. LRU
adpcm	LRU	1,161,988	56,622	2,946,818	1,158,440	60,170	2,978,750	1.08%	
	MRU	1,162,890	55,920	2,490,900	1,155,008	63,802	3,011,838	2.41%	1.11%
bs	LRU	1,741	39	3,911	1,737	43	3,947	0.92%	
	MRU	1,740	39	3,909	1,720	59	4,089	4.61%	3.60%
bsort	LRU	146,781	69	294,321	146,773	77	294,393	0.02%	
	MRU	146,781	69	294,321	146,718	132	294,888	0.19%	0.17%
cnt	LRU	198,496	103	398,125	198,489	110	398,188	0.02%	
	MRU	198,496	103	398,125	198,443	156	398,602	0.12%	0.10%
crc	LRU	104,960	222	212,362	104,947	235	212,479	0.06%	
	MRU	104,947	227	212,391	104,759	415	214,083	0.80%	0.75%
edn	LRU	8,506,404	395,838	21,367,026	8,503,690	398,552	21,391,452	0.11%	
	MRU	8,506,398	395,844	21,367,080	8,413,450	488,792	22,203,612	3.92%	3.80%
expint	LRU	43,633	65	87,981	43,627	71	88,035	0.06%	
	MRU	43,633	65	87,981	43,530	168	88,908	1.05%	0.99%
fdct	LRU	15,269	15,421	200,169	15,268	15,422	200,178	<0.01%	
	MRU	15,269	15,421	200,169	15,268	15,422	200,178	<0.01%	0
fibcall	LRU	1,009	27	2,315	1,006	30	2,342	1.17%	
	MRU	1,009	27	2,315	1,006	30	2,342	1.17%	0
fir	LRU	58,034	74	116,882	58,029	79	116,927	0.04%	0.61%
	MRU	58,028	74	116,870	57,942	160	117,644	0.66%	
insertsort	LRU	128,844	53	258,271	128,841	56	258,298	0.01%	
	MRU	128,844	53	258,271	128,811	86	258,568	0.12%	0.10%
janne	LRU	60,794	37	121,995	60,788	43	122,049	0.04%	
	MRU	60,793	37	121,993	60,779	51	122,119	0.10%	0.06%
jfdctint	LRU	17,302	15,540	205,544	17,299	15,543	205,571	0.01%	
	MRU	17,302	15,540	205,544	17,290	15,552	205,652	0.05%	0.04%
matmult	LRU	6,331,762	130	12,664,954	6,331,737	155	12,665,179	<0.01%	
	MRU	6,331,760	132	12,664,972	6,331,606	286	12,666,358	0.01%	0.01%
minver	LRU	21,841,458	9,655	43,789,121	21,840,200	10,913	43,800,443	0.03%	
	MRU	21,841,102	10,011	43,792,325	21,827,892	23,221	43,911,215	0.27%	0.25%
ndes	LRU	968,253	20,448	2,161,434	958,951	29,750	2,245,152	3.87%	
	MRU	968,333	20,262	2,159,548	947,654	40,941	2,345,659	8.62%	4.48%
ns	LRU	245,408,124	77	490,817,095	245,408,119	82	490,817,140	<0.01%	
	MRU	245,408,124	77	490,817,095	245,408,075	126	490,817,536	<0.01%	<0.01%
nsichneu	LRU	198,708	198,858	2,584,854	195,706	201,860	2,611,872	1.05%	
	MRU	198,708	198,858	2,584,854	195,706	201,860	2,611,872	1.05%	0
prime	LRU	3,617	63	7,927	3,585	95	8,215	3.63%	
	MRU	3,617	63	7,927	3,574	106	8.314	4.88%	1.21%
qsort	LRU	4,209,245	63	8,419,183	4,206,704	2,604	8,442,052	0.27%	
	MRU	4,209,245	63	8,419,183	4,205,204	4,104	8,455,552	0.43%	0.16%
qurt	LRU	8,417	250	19,584	8,341	326	20,268	3.49%	
	MRU	8,432	235	19,449	8,227	440	21,294	9.49%	5.06%
select	LRU	3,931,319	7,921	7,949,769	3,930,818	8,422	7,954,278	0.06%	
	MRU	3,931,319	7,921	7,949,769	3,927,618	11,622	7,983,078	0.42%	0.36%
sqrt	LRU	2,990	52	6,552	2,984	58	6,606	0.82%	
	MRU	2,986	52	6,544	2,936	102	6,994	6.88%	5.87%

Continued

Table I. Continued

Program	policy	Simulation hit	Simulation miss	Simulation sim. WCET	Analysis hit	Analysis miss	Analysis est. WCET	over est.	exc. LRU
statemate	LRU	15,976	17,781	227,543	15,570	18,187	231,197	1.02%	
	MRU	15,926	17,831	227,993	15,565	18,192	231,242	1.43%	0.02%
ud	LRU	11,683,773	5,472	23,427,738	11,683,266	5,979	23,432,301	0.02%	
	MRU	11,683,765	5,480	23,427,810	11,671,443	17,802	23,538,708	0.47%	0.45%
average	LRU							0.78%	
	MRU							2.06%	**1.17%**

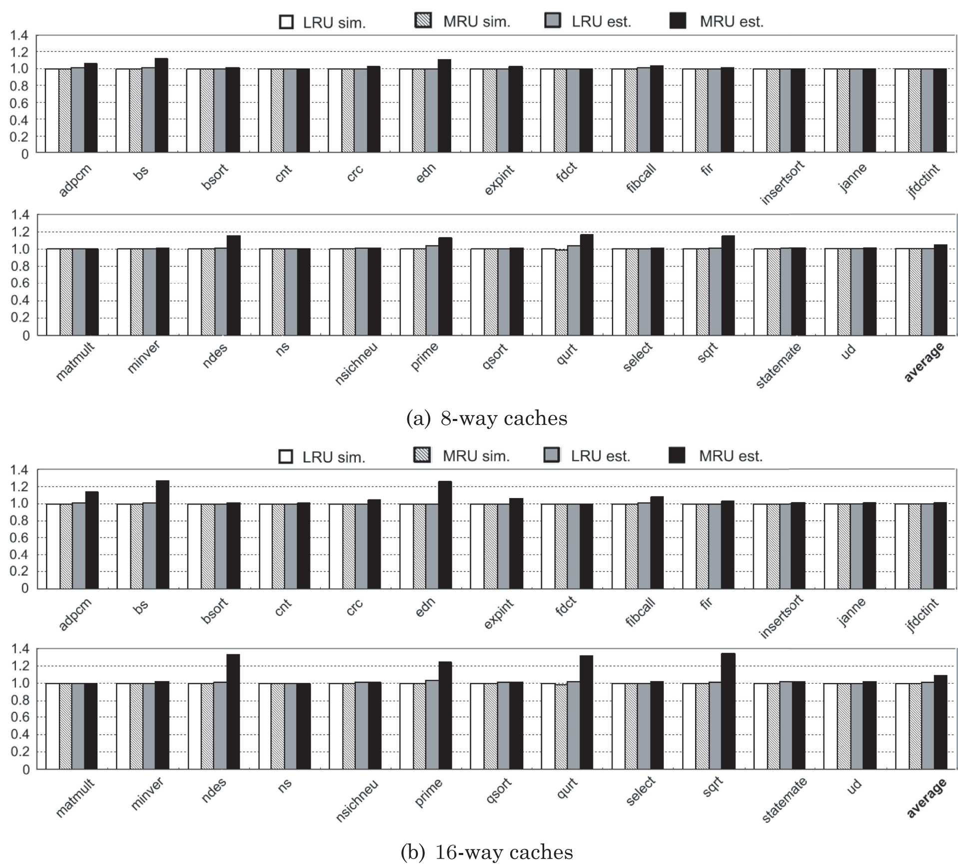


Fig. 10. Experiment results with 8-way and 16-way caches.

theoretical bound for extreme worst-cases. In the simulation experiments, we observe that it hardly happens that a k-Miss node really encounters k times of misses. Most k-Miss nodes actually only incur one miss and exhibit similar behavior to FM nodes under LRU. For example, suppose a loop that contains k nodes accessing different memory blocks executes with k-way caches. Under LRU, the maximal ages of these nodes are all k, so our MRU analysis will classified each of these nodes as k-Miss, and $k \times k = k^2$ misses have to be taken into account for the WCET estimation. However, in the simulation, these k nodes can be entirely fit into the cache, and each of them typically only incur one miss, so the number of misses reflected in the simulation WCET

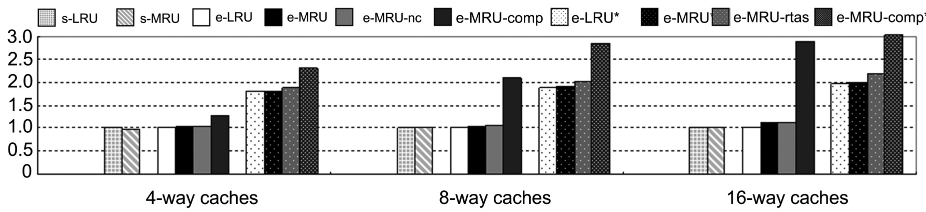


Fig. 11. Comparison of different analyses.

is typically k, which is k times smaller than that claimed by the analysis. So the ratio of overestimated misses increases linearly with respect to the number of cache ways, and thus the overestimation ratio in terms of WCET also scales about linearly with respect to the number of cache ways.

In this experiments, while our MRU analysis has a precision close to that of LRU analysis for most programs, it obtains relatively worse performance for several programs (*bs*, *edn*, *ndes*, *prime*, *qurt* and *sqrt*). While various program structures may lead to pessimism in our MRU analysis, there is a common reason behind that phenomenon, which can be explained as follows. The precision of our MRU analysis is sensitive to the ratio between the k value of k-Miss nodes and the number of times for which the loops containing these nodes iterate. For example, suppose a node is classified as 6-Miss with respect to a loop under MRU. If this loop iterates for 10 times, then the total execution cost of this node is estimated by $11 \times 6 + 2 \times 4 = 74$, where 11 is the execution cost upon a miss, 6 is the number of misses of this node, 2 is the execution cost upon a cache hit, and 4 is the number of hits of this node. On the other hand, this node is an FM with respect to the same loop under LRU, and the total execution cost is $11 \times 1 + 2 \times 9 = 29$. The estimated execution cost under MRU is about 2.5 times of that under LRU. However, if this loop iterates for 100 times, the total execution cost of this node under MRU is $11 \times 6 + 2 \times 94 = 254$, which is only 1.2 times of that under LRU ($11 \times 1 + 2 \times 99 = 209$). The high precision of our MRU analysis relies on the big amount of hits predicted by k-Miss. If a program contains many k-Miss nodes with comparatively large k values but iterates for a small number of times, the estimated WCET by our MRU analysis is less precise. This implies that, from the predictability perspective, MRU caches are more suitable for programs with relatively "small" loops that iterate for a great amount of times, for example, with large loop bounds or nested-loops inside.

Figure 11 shows comparisons among the LRU analysis, the state-of-the-art MRU analysis (competitiveness analysis) and our k-Miss-based MRU analysis with various combinations of different optimization. Each column in the figure represents the normalized WCET (the ratio versus the simulated WCET under LRU) averaged over all benchmark programs. With each cache setting, the first two columns are simulations, the next four columns are analyses with nested-loop optimization and the last four columns are analyses without nested-loop optimization.

—**s-LRU**: Simulated WCET under LRU
—**s-MRU**: Simulated WCET under MRU
—**e-LRU**: Estimated WCET under LRU
—**e-MRU**: Estimated WCET under MRU by the analysis in this article
—**e-MRU-nc**: Estimated WCET under MRU by the analysis in this article but excludes the competitiveness analysis optimization
—**e-MRU-comp**: Estimated WCET under MRU only by competitiveness analysis, which is the state-of-the-art MRU analysis before our k-Miss-based analysis
—**e-LRU***: Estimated WCET under LRU but excludes the nested-loop optimization

—**e-MRU***: Estimated WCET under MRU by the analysis in this article but excludes the nested-loop optimization
—**e-MRU-rtas**: Estimated WCET under MRU by the analysis in the previous conference version of this article [Guan et al. 2012]
—**e-MRU-comp***: Estimated WCET under MRU only by competitiveness analysis, but excludes the nested-loop optimization

By comparing **e-MRU** with **e-MRU-comp**, we can see that our new MRU analysis greatly improves the precision over the state-of-the-art technique for MRU analysis (competitiveness analysis), and the improvement is more significant as the number of cache ways increases. Recall that the competitiveness analysis relies on the analysis results for the same program with a 2-way LRU cache (with the number of cache sets unchanged, and thus the cache size scaled down to $\frac{2}{L}$ of the original L-way cache), so its results are more pessimistic when L is larger.

By the comparison among **e-MRU**, **e-MRU-nc**, **e-MRU***, and **e-MRU-rtas**, we can see that both the competitiveness analysis and nested-loop optimization help to improve our MRU analysis precision. However, the contribution by the nested-loop optimization is much more significant.

By comparing columns $3 \sim 6$ with columns $7 \sim 10$, we see that in general adding nested-loop optimization can significantly improve the analysis precision. The only exception is **e-MRU-comp** with more cache ways (thus, fewer cache sets, as we keep the total cache size unchanged), where even the memory blocks mapped to one cache set in an *inner* loop are too many to fit into 2 cache ways.

By comparing **e-MRU** with **e-LRU** and comparing **e-MRU-rtas** with **e-LRU***, we can see that the nested-loop optimization, which greatly affects the precision of each analysis, does not significantly affect the ratio between the estimated WCET under LRU and MRU. This is because our MRU analysis directly uses the LRU analysis results to find k-Miss nodes. With a more precise LRU analysis, our MRU analysis also becomes correspondingly more precise. This is why do this article and its earlier conference version [Guan et al. 2012] draw similar conclusions about the precipitability comparison between LRU and MRU, although the analysis results in them are different.

We also evaluate the efficiency of our analysis. As presented in previous sections, our MRU analysis only requires to do the LRU cache analysis *once* to infer all the cache access classifications, so the MRU cache analysis procedure is as efficient as the state-of-the-art LRU cache analysis based on abstract interpretation. The interesting problem is the efficiency of the IPET-based path analysis, where more variables are used to support the constraints for k-Miss nodes. We solve the ILP formulation with an open source solver *lp_solve* [Berkelaar] on a desktop machine with a 3.4GHZ Core i7 2600 processor. The ILP formulation can be solved very efficiently: the calculation for each program takes on average 0.1 seconds and at most 0.8 seconds.

In summary, the experiment results show that our MRU analysis has both good precision and high efficiency. The estimated WCET by our MRU analysis is quite close to that by LRU analysis under common hardware setting, which indicates that MRU is a good candidate for cache replacement policies in real-time embedded systems, especially considering MRU's other advantages in hardware, power and thermal efficiency.

7. CONCLUSIONS

This article studies the problem of WCET analysis with MRU caches. MRU was considered to be a very unpredictable replacement policy in the past, due to the lack of effective techniques to predict its hit/miss behavior. In this article, we disclose important properties of MRU, and develop efficient techniques to precisely bound the number of misses and thereby support high-quality WCET estimations with MRU caches.

Experiments with benchmark programs indicate that the estimated WCET with MRU caches is rather close to that with LRU. This suggests a great potential for MRU to be used as the cache replacement policy in real-time embedded systems, especially considering the MRU's advantages in better cost, power and thermal efficiency.

The experiments in this article only consider instruction caches. The reason is that our WCET analysis prototype does not support high-quality value analysis, so currently we cannot provide a meaningful evaluation with data caches. However, the properties of MRU disclosed in this article also hold for data caches, and our proposed analysis techniques can be directly applied to MRU data caches. As future work, we will implement state-of-the-art value analysis techniques in our WCET analysis prototype and evaluate the proposed approach in this article with data caches.

We also plan to study the integration between our MRU analysis and the analysis of other microarchitecture components such as pipelines and memory controllers. Although the k-Miss classification can bound the number of misses that may occur at certain program points, it does not tell when do these k times of cache misses exactly occur. It would be an interesting topic to study how to efficiently use the miss number bounds in the analysis of other components in the presence of timing anomalies. The potential is that the k-Miss classification can significantly prune away the state space of other component's behavior by considerably reducing the number of possible misses.

REFERENCES

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. 2004. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42nd Annual Southeast Regional Conference*. ACM-SE 42. ACM, New York, 267–272.

Frances E. Allen. 1970. Control flow analysis. In *Proceedings of the Symposium on Compiler Optimization*.

Sebastian Altmeyer, Claire Maiza, and Jan Reineke. 2010. Resilience analysis: tightening the crpd bound for set-associative caches. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'10)*. ACM, New York, 153–162.

R. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. 1994. Bounding worst-case instruction cache performance. In *Proceedings of RTSS*.

Todd Austin, Eric Larson, and Dan Ernst. 2002. Simplescalar: An infrastructure for computer system modeling. *Computer* 35, 2, 59–67.

Clément Ballabriga and Hugues Casse. 2008. Improving the first-miss computation in set-associative instruction caches. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS'08)*. IEEE Computer Society, Los Alamitos, CA, 341–350.

M. Berkelaar. lp_solve: (Mixed Integer) linear programming problem solver. `ftp://ftp.es.ele.tue.nl/pub/lp_solve`.

Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. 2010. Modeling shared cache and bus in multi-cores for timing analysis. In *Proceedings of the 13th International Workshop on Software Compilers for Embedded Systems (SCOPES'10)*. ACM, New York, 6:1–6:10.

Christoph Cullmann. 2011. Cache persistence analysis: A novel approachtheory and practice. In *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'11)*. ACM, New York, 121–130.

David Eklov, Nikos Nikoleris, David Black-Schaffer, and Erik Hagersten. 2011. Cache pirating: Measuring the curse of the shared cache. In *Proceedings of the International Conference on Parallel Processing (ICPP'11)*. IEEE Computer Society, Los Alamitos, CA, 165–175.

C. Ferdinand. 1997. Cache behavior prediction for real-time systems. Ph.D. Thesis, Universitat des Saarlandes.

Christian Ferdinand and Reinhard Wilhelm. 1998. On predicting data cache behavior for real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, Frank Mueller and Azer Bestavros, Eds., Lecture Notes in Computer Science, vol. 1474, Springer, 16–30.

Daniel Grund and Jan Reineke. 2009. Abstract interpretation of FIFO replacement. In *Proceedings of the 16th International Symposium on Static Analysis (SAS'09)*. Springer-Verlag, Berlin, 120–136.

Daniel Grund and Jan Reineke. 2010a. Precise and efficient FIFO-replacement analysis based on static phase detection. In *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems, (ECRTS'10)*. IEEE Computer Society, Los Alamitos, CA, 155–164.

Daniel Grund and Jan Reineke. 2010b. Toward precise PLRU cache analysis. In *Proceedings of 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*. B. Lisper, Ed., 28–39.

Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. 2012. WCET analysis with MRU caches: Challenging LRU for predictability. In *Proceedings of the IEEE 18th Real Time and Embedded Technology and Applications Symposium (RTAS'12)*. IEEE Computer Society, Los Alamitos, CA, 55–64.

Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The mälardalen WCET benchmarks: Past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET'10)*. B. Lisper, Ed., OASICS Series, vol. 15, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 136–146.

Damien Hardy and Isabelle Puaut. 2008. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 2008 Real-Time Systems Symposium (RTSS'08)*. IEEE Computer Society, Los Alamitos, CA, 456–466.

Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. 2003. The influence of processor architecture on the design and the results of WCET tools. *Proc. IEEE* 91, 7, 1038–1054.

John L. Hennessy and David A. Patterson. 2006. *Computer Architecture: A Quantitative Approach*. 4th Ed. Morgan Kaufmann Publishers Inc., San Francisco, CA.

Bach Khoa Huynh, Lei Ju, and Abhik Roychoudhury. 2011. Scope-aware data cache analysis for WCET estimation. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'11)*. IEEE Computer Society, Los Alamitos, CA, 203–212.

Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. 2005. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro* 25, 2, 21–29.

Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. 2007. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.* 69, 1–3, 56–67.

Y. S. Li, S. Malik, and A. Wolfe. 1996. Cache modeling for real-time software: Beyond direct mapped instruction caches. In *Proceedings of the 1996 Real-Time Systems Symposium (NTSS'96)*. IEEE Computer Society, Los Alamitos, CA.

Yau-Tsun Steven Li and Sharad Malik. 1995. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference (DAC'95)*. ACM, New York, 456–461.

Yun Liang, Huping Ding, Tulika Mitra, Abhik Roychoudhury, Yan Li, and Vivy Suhendra. 2012. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Syst.* 48, 6, 638–680.

M. Lv. 2012. CATE: A simulator for Cache Analysis Technique Evaluation in WCET estimation. `http://faculty.neu.edu.cn/ise/lvmingsong/cate/`.

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

F. Mueller. 1994. Static cache simulation and its applications. Ph.D. thesis, Florida State University.

Frank Mueller. 2000. Timing analysis for instruction caches. *Real-Time Syst.* 18, 2/3, 217–247.

Peter P. Puschner and Alan Burns. 2000. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Syst.* 18, 2/3, 115–128.

J. Reineke. 2008. Caches in WCET analysis - predictability, competitiveness, sensitivity. In Ph.D. thesis, Saarland University.

Jan Reineke and Daniel Grund. 2008. Relative competitive analysis of cache replacement policies. In *Proceedings of the ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'08)*. ACM, New York, 51–60.

Jan Reineke and Daniel Grund. 2013. Sensitivity of cache replacement policies. *ACM Trans. Embed. Comput. Syst.* 12, 1s, 42:1–42:18.

Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. 2007. Timing predictability of cache replacement policies. *Real-Time Syst.* 37, 2, 99–122.

R. Sen and Y. N. Srikant. 2007. WCET estimation for executables in the presence of data caches. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT'07)*. ACM, New York, 203–212.

Tyler Sondag and Hridesh Rajan. 2010. A more precise abstract domain for multi-level caches for tighter WCET analysis. In *Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS'10)*. IEEE Computer Society, Los Alamitos, CA, 395–404.

Jan Staschulat and Rolf Ernst. 2007. Scalable precision cache analysis for real-time software. *ACM Trans. Embed. Comput. Syst.* 6, 4.

Andrew S. Tanenbaum. 2007. *Modern Operating Systems* 3rd Ed. Prentice-Hall.

Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. 2000. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Syst.* 18, 2/3, 157–179.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem - Overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, 36:1–36:53.

Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. 2009. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comput.-Aided Des. Integ. Cir. Sys.* 28, 7, 966–978.

Received June 2012; revised January 2013; accepted April 2013