FIFO Cache Analysis for WCET Estimation: A Quantitative Approach

Nan Guan[1,2], Xinping Yang[1], Mingsong Lv[2] and Wang Yi[1,2]

[1] Uppsala University, Sweden
[2] Northeastern University, China

***Abstract*—Although most previous work in cache analysis for WCET estimation assumes the LRU replacement policy, in practise more processors use simpler non-LRU policies for lower cost, power consumption and thermal output. This paper focuses on the analysis of FIFO, one of the most widely used cache replacement policies. Previous analysis techniques for FIFO caches are based on the same framework as for LRU caches using *qualitative* always-hit/always-miss classifications. This approach, though works well for LRU caches, is not suitable to analyze FIFO and usually leads to poor WCET estimation quality. In this paper, we propose a *quantitative* approach for FIFO cache analysis. Roughly speaking, the proposed quantitative analysis derives an upper bound on the "miss ratio" of an instruction (set), which can better capture the FIFO cache behavior and support more accurate WCET estimations. Experiments with benchmarks show that our proposed quantitative FIFO analysis can drastically improve the WCET estimation accuracy over pervious techniques (the average overestimation ratio is reduced from around 70% to 10% under typical setting).**

I. Introduction

A fundamental problem in the design and analysis of hard real-time systems is to bound the worst-case execution time (WCET) of programs [7]. 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.

In the last two decades, precise and efficient analysis techniques have been developed for caches with a particular replacement policy, LRU (Least-Recently-Used). In contrast, less work has been done for other policies like MRU [16], FIFO [8] and PLRU [13] However, in practice it is more common for commercial processors to use non-LRU caches, which are simpler in hardware implementation but still have almost as good average-case performance as LRU [1]. Therefore, hardware manufacturers tend to choose these non-LRU policies, especially for embedded processors that are subject to strict cost, power and thermal constraints.

This paper studies the analysis of FIFO (First-In-First-Out), a cache replacement policy that is widely adopted in processor architectures like Intel XScale, ARM9, ARM11 [18]. The FIFO policy is very simple, but analyzing it is much harder than analyzing LRU. The state-of-the-art cache analysis techniques for WCET estimation is based on *qualitative* memory access classifications: to determine whether the memory accesses related to a particular instruction are *always* hits or *always* misses. Such an approach is highly effective for LRU caches since most instructions under LRU indeed exhibit such a "black or white" behavior. However, as will be shown in this paper, many instructions under FIFO exhibit a more nuanced behavior: a portion of the accesses are misses while all the other accesses are hits (e.g., at most $1/3$ of the accesses are misses). By existing analysis techniques based on the qualitative classification, such a behavior has to be treated as if these accesses are all misses, which inherently leads to very pessimistic analysis results. Recently, Grund and Reineke have developed FIFO analysis techniques based on the qualitative classification [8], [9]. Although their techniques are rather sophisticated, the derived WCET bounds are still grossly over-pessimistic (as shown in Section VI).

978-3-9815370-0-0/DATE13/©2013 EDAA

In this paper we propose a *quantitative* approach to analyze FIFO caches, by which we can better capture the FIFO cache behavior and thus obtain much tighter WCET bounds for common programs. The proposed analysis derives an upper bound on the number of misses an instruction (set) may encounter through the whole program execution. As an efficient implementation, we use the cache analysis results of the same program under LRU replacement to derive the quantitative miss bound under FIFO replacement. Therefore, our technique inherits the advantages in efficiency and precision from the state-of-the-art LRU analysis techniques based on abstract interpretation [19].

The proposed analysis is based on a general metric *miss distance* of the underlying cache, and thus applies to any replacement policy as long as the miss distance of the underlying cache is known. The miss distance metric also enables an efficient *persistence* analysis to determine instructions that only encounter a *cold miss* but will always be hits afterwards, which further improves the overall analysis precision.

We have conducted experiments with benchmark programs on *instruction* caches to evaluate the quality of our proposed analysis. Experiments show that the estimated WCET by our FIFO analysis is much tighter than previous techniques (the average overestimation ratio is reduced from around 170% to 10% under typical setting), while still maintaining good analysis efficiency.

A. Relation to Previous Work

Although the always-hit/always-miss classification approach is dominating in previous work on cache analysis for WCET estimation [20], [7], recently there also have been a couple of work towards the direction of quantitative cache

analysis. Reineke and Grund [17] studied 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 this, one can use cache analysis results under one replacement policy to predict the number of cache misses (hits) of the same program under another policy. This approach differs from our proposed quantitative cache analysis in several ways: Firstly, while the relative competitiveness approach provides bounds on the number of misses of the whole program, our quantitative cache analysis bounds the number of misses at individual program points. Secondly, while the relative competitiveness computation suffers scalability problems and thus does not cover cases with great number of ways, our analysis can efficiently deal with large caches. Thirdly, the miss (hit) bounds derived by the relative competitiveness is universal to all programs and thus is much more pessimistic than our quantitative cache analysis in analyzing a concrete program. Another closely related work to this paper is [11], which analyzed MRU caches based on the k-Miss classification (at most k of an instruction's memory accesses are misses). Unfortunately, the k-Miss classification is not suitable to FIFO caches, so in this paper we have to seek more general forms of guarantees. On the other hand, the quantitative analysis in this paper involves complex constraints (regarding multiple nodes and structure information of the CFG), which introduces new difficulties in IPET [15] encoding for path analysis.

II. Preliminaries

A. 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 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. The memory content that fits into one cache line is called a *block*.

Since this work focuses on the cache behavior, we do not consider the timing effect of other components in the processor (e.g., pipeline and memory controller), but assume the execution delay of each instruction only differs depending on whether the cache access is a hit or a miss.

The program can be represented by a control-flow-graph (CFG) $G = (N, E)$, where $N = \{n_1, n_2, \cdots\}$ is the set of *nodes*, and $E = \{e_1, e_2, \cdots\}$ is the set of directed *edges*. A *loop* $\mathfrak{L}$ in the CFG is a strongly connected subgraph of G. Note that here we only provide a simple definition of the CFG and loops since the proposed cache analysis does not rely on any particular CFG or loop structure. In section V we will redefine these definitions for the presentation of path analysis.

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

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

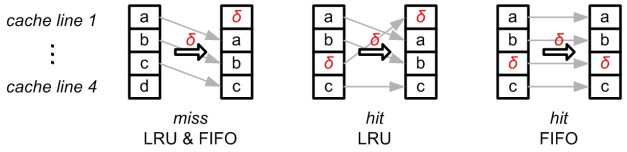


Fig. 1. Illustration of LRU and FIFO replacement.

Definition 1 (Stack Length)**.** *The stack length of an access sequence corresponding to a path p in the CFG, denoted by $\pi(p)$, is the number of different blocks accessed along p.*

For example, the stack length of access sequence "$a \to b \to c \to a \to b$" is 3, since only a, b and c are accessed.

B. LRU and FIFO

The cache update rule of LRU and FIFO is the same upon *misses*: when the program accesses a block δ that is not in the cache, all the blocks in the cache will be shifted one position to the next cache line (the block in the last cache line is removed from the cache), and δ is installed to the first cache line.

LRU and FIFO only differ in their update rules upon *hits*. Let the program access a block δ that is already in the cache. In LRU caches, δ is moved to the first cache line and all blocks that were stored before δ's old position will be shifted one position to the next cache line. In FIFO caches, δ stays at the original position and thus the whole cache keeps unchanged. Figure 1 illustrates the cache update upon an access to block δ on a 4-way LRU and FIFO cache respectively.

C. LRU Cache Analysis

As mentioned in Section I, our quantitative FIFO analysis uses the analysis results of the same program under LRU to infer the cache behavior under FIFO. Thus, we provide a brief review of the state-of-the-art LRU cache analysis technique.

WCET estimation with precise cache analysis suffers from serious state space explosion, so people resort to approximation techniques separating path analysis based on IPET (Implicit Path Enumeration Techniques) and cache analysis based on AI (Abstract Interpretation) for good scalability [19]. The AI-based LRU cache analysis uses three fix-point analyses on the abstract cache domain:

- Must analysis determines if the accesses of a node are always hits (AH);
- May analysis determines if the accesses of a node are always misses (AM);
- Persistence analysis determines if a node will at most encounter a cold miss and afterwards will be always-hit when the program executes inside a particular loop; the classification of such nodes is first-miss (FM) regarding the corresponding loop.

If a node is not determined by any of the above analyses, then it is classified as not-classified (NC). Under the problem model assumption of this paper, NC nodes are treated in the same way as AM in the path analysis to calculate safe WCET bounds. We refer to the references [19], [3], [14], [5] for details about these fix-point analyses.

III. A New Metric: Miss Distance

This section introduces a general metric *miss distance*, which will be useful to establish the quantitative FIFO cache analysis in the next section. Before formally introducing the miss distance, we first use the following example to motivate why is it an interesting metric relevant to the timing predictability of cache replacement policies:

Given a loop accessing K blocks and a K-way cache. Since the whole loop can be fit into the cache, there is a strong intuition to claim the property that each node in the loop is FM regarding this loop. However, this is not always true. It depends on the underlying replacement policy: it holds for many policies including LRU, MRU and FIFO, but not for others including PLRU.

This property is attractive since it enables a very efficient Persistence analysis by only counting the number of different blocks accessed in a loop. Since a program typically spends most of its execution time in loops, this property is highly relevant to the timing analysis of the whole program. Therefore, it is interesting to ask the following questions: What is the essence for a cache replacement policy to have this property? If it does not hold under a given policy, would it be true for a smaller loop? If yes, what is the upper limit of the loop size? Unfortunately, the existing cache replacement predictability metrics [18] cannot answer these questions.

Now we formally introduce the new metric *miss distance*:

Definition 2 (Miss Distance)**.** *The miss distance of a cache is the minimal number of different blocks being accessed between any pair of consecutive cache misses on the* same *block.*

By examining the FIFO rule, it is easy to know:

Lemma 1. *The miss distance of a K-way FIFO cache is K.*

Proof: A block is installed to the first cache line upon a miss, and other K blocks need to be accessed to evict it. ■

With this new metric, we can answer the above questions:

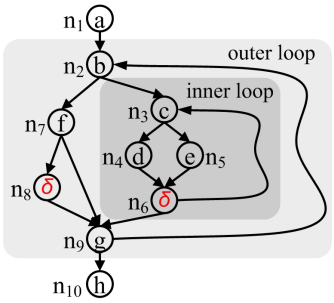
Lemma 2. *Given a cache with miss distance X, and a loop $\mathfrak{L}$ in which the number of different blocks is no larger than X. Any block in $\mathfrak{L}$ encounters at most one miss (the cold miss) every time when the program executes inside $\mathfrak{L}$.*

Proof: Since the miss distance of the underlying cache is X, after the cold miss of a block δ, at least X other different blocks are needed for the next miss on δ to happen. However, this is impossible when the program executes inside the loop $\mathfrak{L}$ since it does not contain enough blocks. ■

Thus we have obtained a very efficient Persistence analysis: Given a K-way FIFO (LRU, MRU) cache, if the total number of different blocks accessed in loop $\mathfrak{L}$ is no larger than K, then all the nodes are FM regarding $\mathfrak{L}$. Similarly all the nodes in a loop accessing at most $log_2 K+1$ different blocks are FM regarding this loop on a K-way PLRU cache.

IV. Quantitative FIFO Analysis

The idea behind the quantitative FIFO cache analysis is fairly simple. Consider the following access sequence:



$\Pi_{\mathfrak{L}_{in}}(n_6,n_6)=3$
$\Pi_{\mathfrak{L}_{out}}(n_6,n_6)=6$
$\Pi_{\mathfrak{L}_{out}}(n_6,n_8)=4$
$\Pi_{\mathfrak{L}_{out}}(n_8,n_6)=6$
$\Pi_{\mathfrak{L}_{out}}(n_8,n_8)=4$

Fig. 2. A CFG example. The letter inside each circle denotes the block accessed by this node.

$$\underline{\delta}\to a\to b\to\underline{\delta}\to c\to d\to\underline{\delta}\to e\to f\to g\to\underline{\delta}\to h\to i\to\underline{\delta}$$

Suppose the underlying FIFO cache has 4 ways, then by Lemma 2 and 1 we know that for any pair of consecutive misses to δ there are at least 4 different blocks accessed in between. In the above sequence, if the first access to δ is a miss, then the second one must be a hit since only 2 blocks are accessed in between. One can see that at most 3 out of the total 5 accesses to δ are misses. For any memory access sequence, one can calculate an upper bound on the misses for each block. However, there are exponentially many paths in the CFG and it is infeasible to do the above analysis for each individual path. In the following, we will show how to do the quantitative analysis in the context of the CFG structure, and in the next section the analysis result will be integrated into the IPET framework to efficiently calculate a WCET bound of the whole program. First define the *maximal stack distance* between two nodes accessing the same block in the scope of a certain loop:

Definition 3 (Maximal Stack Distance)**.** *Let n_i and n_j be nodes in loop $\mathfrak{L}$ accessing the same block δ (n_i and n_j may be the same node). The maximal stack distance from n_i to n_j regarding loop $\mathfrak{L}$, denoted by $\Pi_{\mathfrak{L}}(n_i,n_j)$, is defined as:*

$$\Pi_{\mathfrak{L}}(n_i,n_j)=\begin{cases}\max\{\pi(p)|p\in P_{\mathfrak{L}}(n_i,n_j)\} & \text{if } P_{\mathfrak{L}}(n_i,n_j)\neq\emptyset\\ 0 & \text{otherwise}\end{cases}$$

where $P_{\mathfrak{L}}(n_i,n_j)$ is the set of paths satisfying:

- *All nodes along the path are included in loop $\mathfrak{L}$;*
- *n_i (n_j) is the first (last) node of the path;*
- *No other nodes in the path, besides n_i and n_j, access δ.*

Fig. 2 illustrates the maximal stack distance related to block δ with an inner loop $\mathfrak{L}_{in}$ and an outer loop $\mathfrak{L}_{out}$ respectively. For example, we have $\Pi_{\mathfrak{L}_{in}}(n_6,n_6)=3$ since a "longest" path from n_6 back to n_6 in the scope of $\mathfrak{L}_{in}$ accesses 3 different blocks ($n_6\to n_3\to n_5\to n_6$), while $\Pi_{\mathfrak{L}_{out}}(n_6,n_6)=6$ since the "longest" path in the scope of $\mathfrak{L}_{out}$ accesses 6 different blocks ($n_6\to n_9\to n_2\to n_7\to n_9\to n_2\to n_3\to n_5\to n_6$).

Lemma 3. *Given a cache with miss distance K. Let Δ be the set of nodes in a loop $\mathfrak{L}$ accessing block δ, and it holds*

$$\forall n_i,n_j\in\Delta:\Pi_{\mathfrak{L}}(n_i,n_j)\leq\ell \tag{1}$$

where ℓ is a positive integer no larger than K. Then the total number of misses for nodes in Δ is bounded by:

$$\lfloor\gamma\cdot x\rfloor+y \tag{2}$$

where $\gamma = 1/(1+\lfloor (K-1)/(\ell-1)\rfloor)$, x is the total number of executions of nodes in Δ and y is the total number of times the program enters loop $\mathfrak{L}$ during the whole program execution.

Proof: The first step is to prove there are at least $\lfloor (K-1)/(\ell-1)\rfloor$ hits by nodes in Δ between any pair of consecutive misses for nodes in Δ. Since the miss distance of the underlying cache is K, after a miss of δ, at least K different blocks need to be accessed in order to evict δ from the cache. In other words, all the accesses to δ are hits as long as the number different blocks have been accessed after the first miss to δ does not exceed $K-1$. By (1) we know that when the program executes inside loop $\mathfrak{L}$, the number of different blocks accessed between any two consecutive accesses to δ (not including δ) is at most $\ell-1$. So δ will be accessed for at least $\lfloor (K-1)/(\ell-1)\rfloor$ times before it is evicted from the cache.

The program enters loop $\mathfrak{L}$ for y times. We use x_m ($1 \le m \le y$) to denote how many times δ is accessed when the program for the m^{th} time enters and executes inside $\mathfrak{L}$. Above, we have proved there are at least $\lfloor (K-1)/(\ell-1)\rfloor$ hits by nodes in Δ between any pair of consecutive misses for nodes in Δ, so the total number of misses among these x_m accesses to δ can be bounded by:

$$1+\lfloor x_m/(1+\lfloor (K-1)/(\ell-1)\rfloor)\rfloor$$

Summing up this for each x_m we get an upper bound on the total number of misses for nodes in Δ:

$$y+\sum_{i=1}^{y}\lfloor x_m/(1+\lfloor (K-1)/(\ell-1)\rfloor)\rfloor$$

By the general inequality property $\lfloor \frac{a}{c}\rfloor + \lfloor \frac{b}{c}\rfloor \le \lfloor \frac{a+b}{c}\rfloor$ and $\sum_{i=1}^{y} x_m = x$, the above expression is bounded by (2). ■

Intuitively speaking, Lemma 3 implies a "ratio" γ of the misses over all the accesses by a set of nodes when the program iterates inside a loop. We call such a node set a *γ-set regarding* $\mathfrak{L}$. Note that a node may be included by several γ-sets regarding different loops and different γ values. For example, suppose the CFG in Fig. 2 is executed with a cache of miss distance 8, then n_6 is included in a singleton $\frac{1}{4}$-set regarding $\mathfrak{L}_{in}$ ($\gamma = 1/(1+\lfloor (8-1)/(3-1)\rfloor) = 1/4$), as well as a $\frac{1}{2}$-set $\{n_6, n_8\}$ regarding $\mathfrak{L}_{out}$ ($\gamma = 1/(1+\lfloor (8-1)/(6-1)\rfloor) = 1/2$).

To use Lemma 3, one needs to compute the maximal stack distance $\Pi_{\mathfrak{L}}()$. In general, the time complexity of computing $\Pi_{\mathfrak{L}}()$ is at least exponential regarding the number of cache ways[1], so we need efficient approximation to handle real-life-size problems. Actually, computing $\Pi_{\mathfrak{L}}()$ is exactly the essential problem to solve in the analysis of LRU caches. Therefore, we can use the over-approximate AI-based LRU analysis introduced in Section II-C to efficiently bound $\Pi_{\mathfrak{L}}()$.

Lemma 4. *Given a ℓ-way* LRU *cache. Let Δ be the set of nodes in a loop $\mathfrak{L}$ accessing block δ. If all the nodes in Δ are classified as* AH *or* FM *regarding $\mathfrak{L}$ by the cache analysis, then it must hold:*

$$\forall n_i, n_j \in \Delta : \Pi_{\mathfrak{L}}(n_i, n_j) \le \ell \tag{3}$$

[1]This can be shown by a reduction from the well-known 3-SAT problem, the details of which are omitted due to the space limit.

Proof: Prove by contradiction. Assume two nodes n_i and n_j in S have $\Pi_{\mathfrak{L}}(n_i, n_j) > \ell$. Then by the definition of $\Pi_{\mathfrak{L}}(n_i, n_j)$ there is at least one path from n_i to n_j inside $\mathfrak{L}$ has stack length larger than ℓ. Now suppose this particular path is always taken when the program iterates inside the loop, then n_j will always encounter misses. This contradicts that the cache analysis claims that n_j is miss for at most once when the program executes inside this loop. ■

Now combining Lemma 1, Lemma 3 and 4, we obtain the main result of this section:

Theorem 1. *Let Δ be the set of nodes in a loop $\mathfrak{L}$ accessing block δ. If all the nodes in Δ are classified as* AH *or* FM *regarding $\mathfrak{L}$ by a safe analysis on a ℓ-way* LRU *cache, then the total number of misses for nodes in Δ on a K-way* FIFO *cache is bounded by:*

$$\lfloor \gamma \cdot x \rfloor + y$$

where $\gamma = 1/(1+\lfloor (K-1)/(\ell-1)\rfloor)$, x is the total number of executions of nodes in Δ and y is the total number of times the program enters loop $\mathfrak{L}$ during the whole program execution.

Since γ is non-decreasing with respect to ℓ, we want to find the minimal ℓ such that all nodes in Δ are classified as AH or FM regarding $\mathfrak{L}$ under LRU in order to minimize the "miss ratio". To do this, we actually only need to conduct the LRU cache analysis *once* with a K-way cache. This is because the Must and Persistence analysis for LRU maintains the information about the maximal age of a block at certain point in the CFG (when the program executes in a certain loop), which can be directly transferred to the analysis result with any LRU cache of size smaller than K. For example, suppose in the Must analysis with a 8-way LRU cache, a block δ has maximal age of 4 before the execution of a node accessing δ, then by the Must analysis with a 4-way LRU cache this node will be classified as AH. We will not recite the LRU Must and Persistence analysis details, neither explain how the age information is maintained in the analysis procedure. Interested readers can find details in the references [3], [14], [5].

V. Computation of WCET Bounds

In this section we introduce how to integrate the quantitative FIFO cache analysis results from the last section into IPET to efficiently compute a WCET bound of the analyzed program. First redefine the CFG on the basis of *basic blocks*:

Definition 4 (CFG)**.** *A CFG is a tuple $G = (B, E, b_{st})$:*

- *$B = \{b_1, b_2, \cdots\}$ is the set of* basic blocks *in the CFG;*
- *$E = \{e_1, e_2, \cdots\}$ is the set of directed* edges *connecting the basic blocks in the CFG;*
- *$b_{st} \in B$ is the unique* starting basic block *of the CFG.*

As a common restriction in structured programming [6], we assume each loop contains a single *head basic block*, and the program can jump into the loop by reaching the head basic block via some *entry edges*. The *loop bound* restricts the maximal times the loop iterates every time the program enters it. The head basic block tests whether the loop condition is satisfied. If yes, the program continues to execute the *body*

basic blocks, which are the basic blocks in the loop excluding the head basic block, otherwise the program exists the loop. Formally, a loop is defined as:

Definition 5 (Loop)**.** *A loop in the CFG is a tuple* $\mathfrak{L}_l = (\mathsf{entr}_l, \mathsf{head}_l, \mathsf{body}_l, \mathsf{lpb}_l)$ *with:*

- entr_l*: the set of entry edges of the loop;*
- head_l*: the head basic block of the loop;*
- body_l*: the set of all body basic blocks of the loop;*
- lpb_l*: the loop bound.*

The overall FIFO cache analysis results can be summarized as follows: AH nodes decided by the Must analysis in [8], [9], FM nodes (regarding some loop) decided according to Lemma 2 and γ-sets (regarding some loop) determined by Theorem 1. Finally, the nodes that do not belong to any of the above classification are treated as AM. Note that if a node n_i is FM regarding loop $\mathfrak{L}$, the number of misses with n_i is bounded by the number of times the program enters this loop, so $\{n_i\}$ can be viewed as a special case of γ-set with $\gamma = 0$ (the bound (2) becomes $y + \lfloor 0 \cdot x \rfloor = y$). For simplicity of presentation, in the following we use term γ-set to include both the original ones derived by Theorem 1 and the FM singleton sets with $\gamma = 0$.

The standard IPET for WCET computation with LRU caches is encoded as an ILP (Integer Linear Programming) problem. Since our FIFO cache analysis results involve non-integers (miss ratio γ), we encode the IPET for FIFO cache as an MILP (Mixed-Integer Linear Programming) problem. The constants used in MILP formulation include C^h (the execution delay of each node upon a cache hit), C^m (the execution delay of each node upon a cache miss) and the miss ratio γ for each γ-set.

The formulation uses the following *non-negative* variables:

- c_a: for each b_a, c_a is b_a's total execution cost,
- x_a: for each b_a, x_a is the execution count of b_a,
- y_j: for each edge e_j, y_j counts how many times this edge is taken during the whole execution,
- z_i: for each node n_i included in some γ-set, z_i counts how many times n_i executes as cache misses.

The following maximization object is a safe WCET bound of the analyzed program:

$$\textit{Maximize} \quad \left\{ \sum_{\text{all } b_a} c_a \right\}$$

The following constraints are respected to bound the object.

Cost Constraints: The overall delay of an AH (AM) node n_i in b_a is simply $C^h \cdot x_a$ ($C^m \cdot x_a$). The remaining nodes are the ones included in some γ-set (including FM nodes as stated above). For each of such nodes n_i, we use a variables z_i (s.t. $z_i \le x_a$) to denote the execution count of n_i with cache accesses being misses. So the overall delay of such a node n_i is $C^m \cdot z_i + C^h \cdot (x_a - z_i)$. Putting the above discussions together, we have the total execution cost of each basic block:

$$\forall b_a : c_a = (\pi_{ah} C^h + \pi_{am} C^m) \cdot x_a + \sum_{n_i \in b_a^*} \left(z_i \cdot C^m + (x_a - z_i) \cdot C^h \right)$$

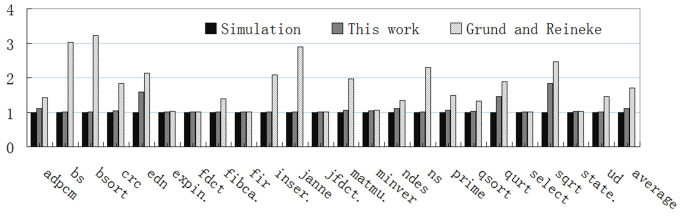


Fig. 3. Experiment results with a 0.5K 4-way cache.

where π_{ah} and π_{am} is the number of AH and AM nodes in b_a respectively, and b_a^* is the set of nodes in b_a that are involved in some γ-set.

γ-Set Constraints: The total number of misses for nodes in a γ-set regarding a loop $\mathfrak{L}_l$ is bounded by $\lfloor \gamma \cdot x \rfloor + y$, where x is the total number of executions of nodes in this γ-set and y is the total number of times the program enters $\mathfrak{L}_l$. So we can bound the number of misses incurred by a γ-set:

$$\forall (S, \mathfrak{L}_l) \text{ s.t. } S \text{ is a } \gamma\text{-set regarding } \mathfrak{L}_l : \quad \sum_{n_i \in S} z_i \le \sum_{e_j \in \mathsf{entr}_l} y_j + \sum_{n_i \in S} \lfloor x_a \cdot \gamma \rfloor$$

where entr_l is the set of entry edges of $\mathfrak{L}_l$ and y_j to denotes how many times an edge $e_j \in \mathsf{entr}_l$ is taken during the whole program execution. Recall that a node may be contained by multiple γ-sets, so each z_i may be involved in several of the above constraints.

Structure Constraints: The same as in the standard encoding of IPET with LRU caches [19].

VI. Experimental Evaluation

We assume the execution delay of each node only differs depending on whether the cache access is a hit or a miss: all instructions have the same execution delay of 1 cycle, the memory access penalty is 1 cycle upon a cache hit and 10 cycles upon a cache miss. Each instruction is 8 bytes, and each block (cache line) is 16 bytes (i.e., each block contains two instructions). The programs used in the experiments are from the Mälardalen Real-Time Benchmark [12]. Some loop bounds cannot be automatically inferred, which 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 re-targeted to the SimpleScalar [2] with -o0 option.

Simulation experiments are conducted with our in-house simulator, which is driven by the worst-case path information extracted from the solution of the MILP formulation. This approach can exclude the effects of other factors orthogonal to the cache behavior (e.g., the tightness of loop bounds), by which we can better evaluate the quality of the cache analysis itself than using traditional full-processor simulations. The solution of the MILP formulation only restricts how many times a basic block executes on the worst-case path, which allows the flexibility of arbitrarily choosing upon branches as long as the execution counts of basic blocks still comply with the MILP 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.

Figure 3 shows the WCET estimations with a 0.5K 4-way FIFO cache by the analysis of this work and Grund and Reineke's Must analyses [8], [9] (a node is classified as AH if it is determined by one of these two analyses). The must analyses in [8], [9] can only determine AH nodes. However, for a more fair comparison, we integrate these two must analyses with the VIVU technique [19] and thus they can also be used to determine FM nodes. The x-axis of the figure represents different benchmark programs (the last group is the average over all programs), and the y-axis is the *normalized* WCET estimation (the ratio between the WCET estimation and the execution time obtained by simulation). For most programs, the WCET estimation with our quantitative analysis is very close to simulation results: the normalized WCET estimation is on average 110.3%. In contrast, the WCET estimation by Grund and Reineke's Must analysis is grossly pessimistic: the normalized WCET estimation is on average 171.8%. In other words, by our new analysis, the overestimation ratio is reduced from 71.8% to 10.3%.

We also conducted experiments with various configurations: the cache size is 0.5K, 1K or 2K; the number of ways is 4, 8, 16 or 32 (the number of sets changes correspondingly, resulting in 12 different configurations). These experiments showed that our analysis is even more accurate with a greater cache size and/or a large number of cache ways, while the quality of Grund and Reineke's Must analysis is similar under different configurations. Detailed result figures with different configurations are omitted due to space limit.

Our FIFO analysis uses the analysis results of the same program under LRU to derive the quantitative guarantee, and thus is as efficient as the state-of-the-art LRU cache analysis based on abstract interpretation. The IPET with our quantitative FIFO analysis is encoded as an MILP problem and uses a greater number of variables, thus in general takes more time to solve than the standard ILP formulation in previous LRU analysis. However, some pragmatic optimizations in the MILP encoding are possible to improve the efficiency of the MILP solver performance[2]. Experiments showed that this approach has a good analysis efficiency with the benchmark programs in use. We solved the MILP formulation by lp_solve [4] on a laptop with an Intel Core i7 CPU (2.7GHZ). The computation for each program takes at most several seconds.

VII. Discussion and Conclusion

This paper presented a quantitative approach for FIFO cache analysis. Unlike the previous standard cache analysis based on qualitative AH/AM classification, this new approach quantitatively bound the number of misses cause by an instruction (set) during the whole program execution. Experiments with benchmark programs showed that the proposed analysis can significantly improve the WCET estimation accuracy over previous techniques while still maintains good efficiency.

Although the quantitative analysis approach supports a significantly better precision in predicting the *number* of cache hits/misses, we would like to point out that the guarantee provided by qualitative analysis is stronger and has the benefit of, e.g., being easier to be integrated in the analysis of architectures that are not fully timing-compositional [21]. Therefore, the next step of our work is to study how to integrate the quantitative cache analysis with the analysis of other components in the processor (e.g., pipelines). We also plan to evaluate the scalability of the proposed analysis with large-scale programs, and extend to multi-level caches and another widely used policy PLRU [10].

[2]The idea is to group as many nodes with the same γ-set characterization into "blocks", to reduce the number of variables used in the MILP encoding.

Acknowledgement

We thank Daniel Grund and Jan Reineke from Saarland University and the anonymous reviewers for their valuable comments. The third author Mingsong Lv is partially supported by NSF of China (Grant No. 60973017).

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] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 2002.

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

[4] M. Berkelaar. lp_solve: a mixed integer linear program solver. In *Relatorio Tecnico, Eindhoven University of Technology*, 1999.

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

[6] E. W. Dijkstra. Structured programming. chapter Chapter I: Notes on structured programming.

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

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

[9] D. Grund and J. Reineke. Precise and efficient FIFO-Replacement analysis based on static phase detection. In *ECRTS*, 2010.

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

[11] N. Guan, M. Lv, W. Yi, and G. Yu. WCET analysis with MRU caches: Challenging LRU for predictability. In *RTAS*, 2012.

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

[13] 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.

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

[15] Y. S. Li, S. Malik, and A. Wolfe. Performance analysis of embedded software using implicit path enumeration. In *DAC*, 1995.

[16] 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.

[17] J. Reineke and D. Grund. Relative competitive analysis of cache replacement policies. In *LCTES*, 2008.

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

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

[20] R. Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *VMCAI*, 2004.

[21] 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.