FIFO Cache Analysis for WCET Estimation: A Quantitative Approach

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 previous techniques (the average overestimation ratio is reduced from around 170% 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 [19]. 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, much less work has been done for other policies like MRU [14], FIFO [7] and PLRU [11]. 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 [16]. 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 [7], [8]. Although their techniques are rather sophisticated, the derived WCET bounds are still grossly over-pessimistic (as shown in Section VI).

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 [17].

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 [18], [19], recently there also have been a couple of work towards the direction of quantitative cache analysis. Reineke and Grund [15] 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 do 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 bounds derived by the relative competitiveness computation suffers scalability problems and thus can efficiently deal with large caches. Fourthly, the miss (hit) classification of such nodes is first-miss (FM) regarding the corresponding loop.

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:

**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 “\( a \rightarrow b \rightarrow c \rightarrow a \rightarrow b \)” 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 \( \delta \) 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 \( \delta \) is installed to the first cache line.

LRU and FIFO only differ in their update rules upon hits. Let the program accesses a block \( \delta \) that is already in the cache. In LRU caches, \( \delta \) is moved to the first cache line and all blocks that were stored before \( \delta \)’s old position will be shifted one position to the next cache line. In FIFO caches, \( \delta \) stays at the original position and thus the whole cache keeps unchanged.

Figure 1 illustrates the cache update upon an access to block \( \delta \) 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 [17]. 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 [17], [3], [12], [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 it is 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 [16] 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 replacement 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.

The miss distance is $K$ for a $K$-way LRU or MRU cache, and is $\log_2 K + 1$ for a $K$-way PLRU cache (proof omitted). LRU, MRU and FIFO are optimal regarding this metric:

**Lemma 2.** The miss distance of a $K$-way cache with any replacement policy is no larger than $K$.

**Proof:** Assume a $K$-way cache with miss distance $K' > K$. Given a loop accessing $K'$ different blocks, by Lemma 3 each of these blocks only encounters a cold miss and then be always hits when the program iterates inside the loop. However, this is impossible since a $K$-way cache cannot store more than $K$ blocks at the same time.

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

**Lemma 3.** Given a cache with miss distance $X$, and a loop $\Sigma$ in which the number of different blocks is no larger than $X$. Any block in $\Sigma$ encounters at most one miss (the cold miss) every time when the program executes inside $\Sigma$.

**Proof:** Since the miss distance of the underlying cache is $X$, after the cold miss of a block $\delta$, at least $X$ other different blocks are accessed before the next miss on $\delta$ happens. This is impossible when the program executes inside the loop $\Sigma$ 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 $\Sigma$ is no larger than $K$, then all the nodes are FM regarding $\Sigma$. 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: $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow h \rightarrow i \rightarrow \delta$

Suppose the underlying FIFO cache has 4 ways, then by Lemma 3 and 1 we know that for any pair of consecutive misses to $\delta$ there are at least 4 different blocks accessed in between. In the above sequence, if the first access to $\delta$ is a miss, then the second one must be a hit since only 2 blocks are accessed in between. By a simple calculation, one can see that at most 3 out of the total 5 accesses to $\delta$ are misses. If the underlying FIFO cache is 8-way, then there are at least two hits with $\delta$ between any two consecutive misses with $\delta$, and in total at most 2 accesses to $\delta$ are misses.

For any memory access sequence, it is not difficult to 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 $\Sigma$ accessing the same block $\delta$ ($n_i$ and $n_j$ may be the same node). The maximal stack distance from $n_i$ to $n_j$ regarding loop $\Sigma$, denoted by $\Pi_{\Sigma}(n_i, n_j)$, is defined as:

$$
\Pi_{\Sigma}(n_i, n_j) = \left\{ \begin{array}{ll}
\max \{ \pi(p) | p \in P_{\Sigma}(n_i, n_j) \} & \text{if } P_{\Sigma}(n_i, n_j) \neq \emptyset \\
0 & \text{otherwise}
\end{array} \right.
$$

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

- All nodes along the path are included in loop $\Sigma$;
- $n_i$ and $n_j$ are the first (last) node of the path;
- No other nodes in the path, besides $n_i$ and $n_j$, access $\delta$.

Fig. 2 illustrates the maximal stack related to block $\delta$ with an inner loop $\Sigma_{in}$ and an outer loop $\Sigma_{out}$ respectively. For example, we have $\Pi_{\Sigma_{in}}(n_6, n_6) = 3$ since the “longest” path

1A counter-example for interested readers who are familiar with PLRU: in the initial state of a 4-way cache all tree-bits are all 0 and all cache lines are empty; the loop is $a \rightarrow b \rightarrow c \rightarrow b \rightarrow d$ and then goes back to $a$. 

Fig. 2. A CFG example. The letter inside each circle denotes the block accessed by this node.
from \( n_6 \) back to \( n_6 \) in the scope of \( \mathfrak{L}_{in} \) accesses 3 different blocks \((n_6 \rightarrow n_3 \rightarrow n_5 \rightarrow n_6)\), 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 \((n_6 \rightarrow n_9 \rightarrow n_2 \rightarrow n_7 \rightarrow n_9 \rightarrow n_2 \rightarrow n_3 \rightarrow n_5 \rightarrow n_6)\).

**Lemma 4.** Given a cache with miss distance \( K \). Let \( \Delta \) be the set of nodes in a loop \( \mathfrak{L} \) accessing block \( \delta \), and it holds

\[
\forall n_i, n_j \in \Delta: \Pi_{\mathfrak{L}}(n_i, n_j) \leq \ell
\]

where \( \ell \) is a positive integer no larger than \( K \). Then the total number of misses caused by nodes in \( \Delta \) is bounded by:

\[
[\gamma \cdot x] + y
\]

where \( \gamma = 1/(1 + \left[(K - 1)/(\ell - 1)\right]) \), \( x \) is the total number of executions of nodes in \( \Delta \) 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 \( [(K - 1)/(\ell - 1)] \) hits by nodes in \( \Delta \) between any pair of consecutive misses by nodes in \( \Delta \). Since the miss distance of the underlying cache is \( K \), after a miss of \( \delta \), at least \( K \) different blocks need to be accessed in order to evict \( \delta \) from the cache. In other words, all the accesses to \( \delta \) are hits as long as the number of different access blocks have been accessed after the first miss to \( \delta \) 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 \( \delta \) (not including \( \delta \)) is at most \( \ell - 1 \). So \( \delta \) will be accessed for at least \( [(K - 1)/(\ell - 1)] \) times before it is evicted from the cache.

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

\[
1 + \left[x_m/(1 + [(K - 1)/(\ell - 1)])\right]
\]

Summing up this for each \( x_m \) we get an upper bound on the total number of misses by nodes in \( \Delta \):

\[
y + \sum_{i=1}^{y} \left[x_m/(1 + [(K - 1)/(\ell - 1)])\right]
\]

By the general inequality property \( \left[a\right] + \left[b\right] \leq \left[a + b\right] \) and \( \sum_{i=1}^{y} x_m = x \), the above expression is bounded by (2). \( \blacksquare \)

Approximately speaking, Lemma 4 implies a “ratio” \( \gamma \) 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 \( \gamma \)-set regarding \( \mathfrak{L} \). Note that a node may be included by several \( \gamma \)-sets regarding different loops and different \( \gamma \) 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 \( 1/2 \)-set regarding \( \mathfrak{L}_{in} \) \((\gamma = 1/(1 + [(8 - 1)/(3 - 1)]) = 1/4\)), as well as a \( 1/2 \)-set \{n_6, n_6\} regarding \( \mathfrak{L}_{out} \) \((\gamma = 1/(1 + [(8 - 1)/(6 - 1)]) = 1/2\)).

To use Lemma 4, 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\(^2\), 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 5.** Given a \( \ell \)-way LRU cache. Let \( \Delta \) be the set of nodes in a loop \( \mathfrak{L} \) accessing block \( \delta \). If all the nodes in \( \Delta \) are classified as AH or FM regarding \( \mathfrak{L} \) by any safe analysis, then it must hold:

\[
\forall n_i, n_j \in \Delta: \Pi_{\mathfrak{L}}(n_i, n_j) \leq \ell
\]

**Proof:** Prove by contradiction. Assume two nodes \( n_i \) and \( n_j \) in \( \Delta \) 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 \( \ell \). 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 a safe analysis claims that \( n_j \) is miss for at most once when the program executes inside this loop.

Now combining Lemma 1, Lemma 4 and 5, we obtain the main result of this section:

**Theorem 1.** Let \( \Delta \) be the set of nodes in a loop \( \mathfrak{L} \) accessing block \( \delta \). If all the nodes in \( \Delta \) are classified as AH or FM regarding \( \mathfrak{L} \) by a safe analysis on a \( \ell \)-way LRU cache, then the total number of misses caused by nodes in \( \Delta \) is bounded by:

\[
[\gamma \cdot x] + y
\]

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

Since \( \gamma \) is non-decreasing with respect to \( \ell \), we want to find the minimal \( \ell \) such that all nodes in \( \Delta \) 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 \( \delta \) has maximal age of 4 before the execution of a node accessing \( \delta \), 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], [12], [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:

\(^2\)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.
Definition 4 (CFG). A CFG is a tuple $G = (B,E,b_0)$:

- $B = \{b_1,b_2,\ldots\}$ 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;
- $b_0 \in B$ 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 $\mathcal{L}_l = (\text{entr}_l,\text{head},\text{body}_l,\text{lpb}_l)$ with:

- $\text{entr}_l$: the set of entry edges of the loop;
- $\text{head}$: the head basic block of the loop;
- $\text{body}_l$: the set of all body basic blocks of the loop;
- $\text{lpb}_l$: the loop bound.

The overall FIFO cache analysis results can be summarized as follows: AH nodes decided by the Must analysis in [7], [8], FM nodes (regarding some loop) decided according to Lemma 3 and $\gamma$-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 $\mathcal{L}_l$, the number of misses caused by $n_i$ is bounded by the number of times the program enters this loop, so $\{n_i\}$ can be viewed as a special case of $\gamma$-set with $\gamma = 0$ (the bound (2) becomes $y + |0 \cdot x| = y$). For simplicity of presentation, in the following we use term $\gamma$-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 uses is encoded as an ILP (Integer Linear Programming) problem. Since our FIFO cache analysis results involve non-integers (miss ratio $\gamma$), 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 $\gamma$ for each $\gamma$-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 $\gamma$-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:

$$\text{Maximize } \left\{ \sum_{a} c_a \right\}$$

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 $\gamma$-set (including FM nodes as stated above). For each of such nodes $n_i$, we use a variables $z_i$ (s.t. $z_i \leq x_a$) 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:

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

where $\pi_{ah}$ and $\pi_{am}$ 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 $\gamma$-set.

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

$$\forall (S, \mathcal{L}_l) \text{ s.t. } S \text{ is a } \gamma\text{-set regarding } \mathcal{L}_l : \sum_{n_i \in S} z_i \leq \sum_{e_j \in \text{entr}_l} y_j + \sum_{n_i \in S} |x_a \cdot \gamma|$$

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

Structure Constraints: Each basic block should have balanced input and output:

$$\forall b_a : x_a = \sum_{e_j \in \text{input}(b_a)} y_j = \sum_{e_j \in \text{output}(b_a)} y_j$$

where input($b_a$) and output($b_a$) is the set of input edges and output edges of $b_a$ respectively. The start basic block $b_{st}$ only executes once so we have $x_{st} = 1$.

Each time the program enters the loop, each body basic block executes for at most $\text{lpb}_l$ times, so we have:

$$\forall \mathcal{L}_l, \forall b_a \in \text{body}_l : x_a \leq \text{lpb}_l \cdot \sum_{e_j \in \text{entr}_l} y_j$$

The head basic block may execute one more time to realize that the loop condition is not satisfied and thus the program exists the loop, so for each head basic block $x_a$ the loop bound constraint is $x_a \leq (\text{lpb}_l + 1) \cdot \sum_{e_j \in \text{entr}_l} y_j$.

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 [10]. 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 teens to about 4000 lines of C code, or from
 teens to about 8000 assembly instructions compiled by a gcc
 compiler re-targeted to the SimpleScalar architecture [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 analysis [7], [8]. 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 277.7%. In other words, by our new analysis, the
 overestimation ratio is reduced from 177.7% 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, experiments showed that our
 approach also have a good analysis efficiency. 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. CONCLUSION AND FUTURE WORK

 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.

 An important future work is to study how to integrate the
 quantitative cache analysis with the analysis of other compo-
 nents in the processor (e.g., pipeline and memory controller).
 We also plan to apply the quantitative analysis approach to the
 multi-level caches and another widely used policy PLRU.

 REFERENCES

 of cache replacement policies for the spec cpu2000 benchmark
 structured programming.
 In SAS, 2009.
 The influence of processor architecture on the design and the results of wcet
 implementing a pseudo-lru cache memory replacement scheme with a
 prediction by separated cache and path analyses. In Real-Time Systems,
 2000.
 [18] R. Wilhelm. Why ai + ilp is good for wcet, but mc is not, nor ilp alone.
 In VMCAI, 2004.
 [19] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whal-
 ley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller,
 I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The worst-case
 execution-time problem overview of methods and survey of tools. ACM