FIFO Cache Analysis for WCET Estimation: A Quantitative Approach

Xinping Yang
Abstract

FIFO Cache Analysis for WCET Estimation: A Quantitative Approach

Xinping Yang

Although most previous work in cache analysis for WCET estimation assumes the LRU (Latest Recently Used) replacement policy, in practice more processors use simpler non-LRU policies for lower cost, power consumption and thermal output. This paper focuses on the analysis of FIFO (First In First Out), 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 it 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 178% to 6.5% under typical setting).
Acknowledgements

I would like to thank my supervisor Nan Guan for providing me his nice guidance from time to time as needed. From the thesis topic choosing, research, experimental design, his valuable comments and suggestions help me to successful completed the paper. I am really thankful to him for his patient guiding.

I would also like to thank my reviewer Philipp Rüemmer for providing his support and useful comments which led to the paper better. And I would also like to thank Mingsong Lv for proving me the program platform for the thesis.

Finally, I would like to thank everyone else who give a small but valuable comment which helped me make a better thesis. Last but not the least, I wish to thank my family, thanks for their strong support for my study.
# Contents

1 Introduction  
   1.1 Motivation of the thesis  
   1.2 Background  
   1.3 Related Work  
   1.4 Contributions and Structure of the Thesis  

2 Introduction to Caches and Cache Replacement Policies  
   2.1 Basic Concepts  
   2.2 LRU Replacement  
   2.3 FIFO Replacement  

3 A Review of the Analysis for LRU and FIFO  
   3.1 Abstract Interpretation  
      3.1.1 Partial Orders and Lattices  
      3.1.2 Collecting Semantics  
      3.1.3 Abstract Semantics  
      3.1.4 Establishing Soundness  
      3.1.5 Computing the Abstract Semantics  
   3.2 The analysis for LRU  
   3.3 The analysis for FIFO  
      3.3.1 Challenges when predicting hits  
      3.3.2 FIFO analysis by phase detection  

4 The New Analysis of FIFO  
   4.1 A New Metric: Miss Distance  
   4.2 Quantitative FIFO analysis  
   4.3 Computation of WCET Bounds  

5 Experimental Evaluation  
   5.1 Experiment Setup  
   5.2 Results  

6 Conclusion and Future works
List of Figures

2.1 Illustration of LRU and FIFO replacement . . . . . . . . . . . 9
2.2 An example illustrating the mls of FIFO is 1 . . . . . . . . . 11
3.1 Illustration of the soundness proof . . . . . . . . . . . . . . . 15
4.1 A CFG example. The letter inside each circle denotes the block accessed by this node. . . . . . . . . . . . . . . . . . . . 24
Chapter 1

Introduction

1.1 Motivation of the thesis

A fundamental problem in the design and analysis of hard real-time systems is to bound the worst-case execution time (WCET) of programs [1]. To derive safe and tight WCET bounds, the analysis must take into account the cache architecture of the target processor. The actual execution time depends on the architectural state in which the instruction is executed, i.e. the contents of the caches, the occupancy of the pipeline units. 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. In contrast, much less work has been done for other policies like MRU [2], FIFO [3] and PLRU [4]. 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 [5]. 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, a cache replacement policy that is widely adopted in processor architectures like Intel XScale, ARM9 and ARM11 [6]. The FIFO policy is very simple, but analyzing it is hard [3]. 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 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 [3], [7]. Although their techniques are rather sophisticated, the derived WCET bounds are still grossly over-pessimistic (as shown in Chapter 5).

In this paper we propose a quantitative approach to analysis 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 [8].

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 178% to 6.5% under typical setting), while still maintaining good analysis efficiency.

1.2 Background

An embedded system is a computer system completely embedded within the device, and designed for a specific application dedicated computer system, often with real time constraints. Embedded systems are widespread in consumer, industrial, commercial and military applications, such as Mp4, Aerospace variation, air traffic control, medical monitoring and military equipment. Because they are as a part of the devices, and often require real time interaction with the outside world, real time constraints play a key role on them. For example, consider an auto-mobile security subsystem, the
correctness of embedded real time systems not only depends on the correct calculated result of the systems, but also ensures the calculation time of the result to meet the systems' requirements. This requires the systems to ensure the external events response in a predictable period. If the time does not meet these requirements, it could cause the system to fail or disastrous consequences, even if the calculations are correct.

In order to build reliable real-time systems, it is necessary to know the worst case execution time (WCET) of the procedures in advance. This makes WCET estimation become an important research field in real-time system, and WCET for real time analysis become a research hot spot in the last two decades. In order to meet the timing requirements of real-time systems, the system must know the time of the execution of a program in the worst-case, so WCET is one of the most important parameters to measure timeliness of a real-time system. Many applications have the requirement of WCET estimation, like system scheduling and calculable detection. Most of the scheduling algorithm and schedule analysis algorithms need to know the WCET of the tasks in advance. Usually, the tasks are implemented by a set of programs which are periodically or sporadically executed on a shared hardware platform. The scheduling analysis will consider all possible execution orders of the programs to guarantee the timeliness of each task. Another application is that WCET analysis can be used to help system designers to select equipment. According to the running applications' WCET values on different hardware platforms, the system designers are able to determine the most reasonable hardware configuration to meet the system performance requirements.

Different kinds of time analyses are being used today. Dynamic measurement, static analysis and hybrid methods are the most prominent.

- Dynamic measurement methods directly measure the execution time of the program on the target platform. Currently used methods are random methods, evolutionary methods [9] and statistical methods [10]. Random methods allowed choosing arbitrary programs as input and measuring the execution time of the program. Evolutionary method studies the WCET use-case in the input domain and takes the maximum execution time of the program as optimization criterion. Statistical method is based on a large number of measurement results (program execution time), using extreme statistical methods to infer WCET of the program.

- Static analysis methods estimate the program WCET by the flow of information of the program and the property of the processor. However, due to the complex program flow and the complex characteristics
of modern processors, the corresponding static analysis and calculation also becomes very complicated. But static program methods can ensure the obtained results to be safe, so that it might become the mainstream technology for WCET analysis study. The WCET static analysis method will be detailed described in chapter 3.

- Hybrid method combines the static analysis and dynamic measurement. One hybrid method to get the static analysis WCET is based on measure program fragments [11],[12], such as basic block; another hybrid method is the dynamic measurement based on static analysis [13],[14]. For example, it finds all the possible paths which cause WCET at first, then measures the WCET on the path.

Dynamic measurement method is difficult to obtain the WCET of the program accurately; this makes it hard to obtain the real executing time value on the worst case, so an obtained WCET value is often below the real WCET value. Static analysis can overcome these difficulties in dynamic measurement; it does not need to run the program, and it can get both of the WCET for the task and the code fragment program. But static analysis methods often overestimate the WCET of the program. Our WCET analysis is also focus on static analysis, the later mentioned WCET are default looked as static cache analysis.

Research status and the problems

WCET analysis gradually gets the attention of the academic community since Klingerman and Stoyenko [15] have published the first WCET literature. There are some famous institutions, such as Saarbrucken University in Germany, Florida State University in the United States, Princeton University, the Technical University of Vienna, Malardalen University in Sweden, University of York UK. In the current industry, at least three WCET analysis tools have been put into the industrial market, including aiT [16], Bound-T [17] and RapiTime [14]. aiT is used in the aeronautics and automotive industries and can obtain precise bounds for the execution times of real-time programs. Bound-T and RapiTime can also obtain the precise upper bound WCET for a determined program.

WCET analysis has made great achievements, but there are still a lot of problems to be solved:

- WCET analysis of the degree of automation. Most of the existing WCET analysis tools require manual intervention.
- To improve the accuracy of the WCET analysis. Precise WCET analysis conserves system resource. Although the precision of analysis has been constantly improved, more accurate WCET analysis is always a goal in academic research.
• WCET analysis for a combination of new processor features or characteristics. As the advance of the new technology, the new processor features will emerge, this put forward new demands for WCET analysis.

1.3 Related Work

Most previous work on cache analysis for static WCET estimation focuses on the LRU replacement policy. Li et. al. [18], [19] use integer linear programming (ILP) approaches to predict cache behavior by formulating an ILP problem. There are serious scalability problems due to the time complexity of ILP, and cannot handle realistic programs on modern processors. A technique called static cache simulation is proposed by Mueller et. al. [20]. It calculates the instructions that may be in the cache at entry and exit of each basic block iteratively, until the collective cache state reaches a fixed point, and categorizes the caching behavior of each instruction by using this information.

The establishing of a WCET estimation framework, which is separated abstract interpretation (AI) and path analysis by implicit path enumeration technique (IPET) [8] is a milestone in the research of static WCET estimation. To get both high efficiency and good precision for LRU caches, the AI-based cache analysis uses Must, May and Persistence analyses to categorize the caching behavior of each instruction. The IPET-based path analysis uses the cache behavior classification to derive a delay invariant for each instruction and encodes the WCET calculation problem 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 [21], data caches [22], [23], [24], multilevel caches [25], [26], shared caches [27], [28] and cache-related preemption delay [29], [30], [31].

In general, few works has been done for non-LRU caches. Although some important progress have been made in the analysis of policies like FIFO [3], PLRU [32] and MRU [33], these analyses are much less precise than LRU analysis.

Jan Reineke et al. [34], [35], [36] have conducted a series of fundamental studies on predictability properties of different cache replacement policies. [6] 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). [35] 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. [34] Considers 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 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 for the caches which have high associations, 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 [33], 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 cache, 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 [37] encoding for path analysis.

1.4 Contributions and Structure of the Thesis

The main contribution of this thesis is we propose a quantitative approach for FIFO cache analysis. It can better capture the FIFO cache behavior and drastically improve the WCET accuracy 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 178% to 6.5% under typical setting).

Structure of the thesis In Chapters 2, we present the fundamentals on the method of investigation, i.e. basic concepts and the simple introduction of LRU and FIFO policy. Chapter 3 gives a review for LRU and FIFO cache analysis. The main Chapters 4 we present cache analyses for the new FIFO replacement policies, respectively. Chapter 5 we present Experimental Evaluation to evaluate the precision of our proposed FIFO analysis and the
predictability comparison between Daniel Grund and Jan Reineke’s and ours. We close with Chapter 6, show the test result of benchmarks for our FIFO and future work, and provide conclusion.
Chapter 2

Introduction to Caches and Cache Replacement Policies

2.1 Basic Concepts

A cache can be characterized by three major parameters: capacity is the number of bytes it may contain. Line size is the number of contiguous bytes that are transferred from memory on a cache miss. Association is the number of cache locations where a particular block may reside. If a block can reside in any cache location, then the cache is called fully associative. If a block can reside in exactly set locations, then the cache is called set-associative cache. 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) $G = (B, E, b_{st})$, where $B = \{b_1, b_2, \cdots\}$ is the set of nodes, and $E = \{e_1, e_2, \cdots\}$ is the set of directed edges. A loop $L$ in the CFG is a strongly connected subgraph of $G$. $b_{st}$ is the unique starting basic block of the CFG. 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. The instructions in the programs are fetched from the instruction cache. The types of instructions are the same as SimpleScalar architecture [38]. For ex-
ample, control instruction, jump (j); load/store instruction, load word (lw); integer arithmetic instruction, integer add (add) and so on. In Chapter 4 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:

**Definition 2.1** (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 \( p \).

The access sequence is a sequence of CPU instructions which access the cache by following a path \( p \) in the CFG. 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.

### 2.2 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 \( s \) and \( s' \) old position will be shifted one position to the next cache line. Figure 2.1 illustrates the update upon an access to memory block \( s \) in an LRU and FIFO cache of 4 lines.

![Figure 2.1: Illustration of LRU and FIFO replacement](image-url)
the figure, the uppermost block represents the first (lowest-index) cache line and the lowermost block is the last (highest-index) one. A metric defined in [6] to evaluate the predictability of a replacement policy is the minimal-life-span (mls), the minimal number of different memory blocks required to evict a just visited memory block out of the cache (not counting the access that brought the just visited memory block into the cache). It is known that in [6].

**Lemma 2.1** The mls of LRU is $L$.

$L$ 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 $a \rightarrow b \rightarrow c \rightarrow c \rightarrow d \rightarrow a \rightarrow e \rightarrow b$ 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 $b \rightarrow c \rightarrow c \rightarrow d$, which has stack length 3. The second access to $b$ is a miss since the stack length of the sequence $c \rightarrow c \rightarrow d \rightarrow a \rightarrow e$ is 4. Clearly, replacement policies with larger mls are preferable, and the upper bound of mls is $L$.

### 2.3 FIFO Replacement

The FIFO replacement policy stores the accessed memory block by the rule with the first coming first out. When the program accesses a memory block $s$, if $s$ is not in the cache (miss), then all the memory blocks in the cache set will be shifted one position to the next cache line, the memory block in the last cache line will remove from the cache set, and the memory block $s$ will be installed to the first cache line. If $s$ is in the cache set already (hit), there is nothing to update for the cache set. In figure 2.1, the first $\delta$ is a miss, all the memory blocks will move one position to bottom, the bottom line $d$ will evict from the cache set. The second $\delta$ is a hit, all the cache set will stay the same as before. In FIFO caches, because the character of a cache hit does not change the constancy of the cache set, it is harder to predict the cache behavior. Such a behavior results in a different mls [6].

**Lemma 2.2** The mls of FIFO is 1.
Figure 2.2: An example illustrating the mls of FIFO is 1

The example in Figure 2.2 illustrates this lemma, where only one block $e$ is enough to evict a just-visited memory block $s$. Extends this example to arbitrarily many cache line, we only need one memory block to evict $s$. 
Chapter 3

A Review of the Analysis for LRU and FIFO

3.1 Abstract Interpretation

Abstract Interpretation was proposed by P. Cousot and R. Cousot (1977), and it is widely used in program analysis. It performs program computations by abstract values instead of the concrete values. This Section we only present some part of abstract interpretation, which is relevant for our applications simply. We start in Section 3.1.1 by giving the basic definitions from lattice theory. Section 3.1.2 and 3.1.3 introduce the collecting semantics and abstract semantics respectively. Section 3.4 shows how the establish soundness can be established. Finally, Section 3.1.4 presents the Maximal fixed point solution theorem to approximate compute the abstract semantics.

3.1.1 Partial Orders and Lattices

Definition 3.1 (Partial order). Let $P$ be a set. The partial order on $P$ represents a binary relation on $P$, if it is reflexive, antisymmetric and transitive.

For example, $x, y, z \in P$. If 1). $x \subseteq x$; 2). $x \subseteq y$ and $y \subseteq x$ implies $x = y$; 3). $x \subseteq y$ and $y \subseteq z$ implies $x \subseteq z$. The set $P$ with a partial order $(P, \subseteq)$ can be called partially ordered set.

Definition 3.2 (Lattice, Complete lattice). Let $(P, \subseteq)$ be a non-empty partially ordered set. $S$ is a set.

(i) If $x \cup y$ and $y \cap x$ exist for all $x, y \in P$, the $P$ is called a lattice.

(ii) If $\cup S$ and $\cap S$ exist for all $S \subseteq P$, then $P$ is called a complete lattice.

$\cup$ and $\cap$ are the intersection and union operations. $\cup S$ denotes the least upper bound of $S$, $\cap S$ is the greatest lower bound.
### 3.1.2 Collecting Semantics

The CFG \( (B, E, b_{st}) \) which we will be mentioned in Chapter 4 represents a imperative program, the nodes \( B \) stand for the program statements, and the edges \( E \) represent the possible control flow. \( b_{st} \) represents an unique start node. We give a concrete function \( f : B \rightarrow D_{conc} \rightarrow D_{conc} \) to describe the effect of a program statement \( b \in B \) on a concrete state \( b_{st} \in D_{conc} \), then a path \( \pi = (b_1, \cdots, b_k) \) in the control flow graph can be defined by the path semantics \([\pi]_{coll}\).

**Definition 3.3 (Path semantics).**

\[
[\pi]_{coll} := \begin{cases} 
\text{id} & \text{if } \pi \text{ is the empty path} \\
[(b_2, \cdots, b_k)_{coll} \circ f(b_1)] & \text{otherwise}
\end{cases}
\]  

(3.1)

The collecting semantics \( \text{Coll} \) maps program points to sets of states \( D_{coll} = 2^{D_{conc}} \):

\[
\text{Coll} : B \rightarrow D_{coll}
\]  

(3.2)

Let \( \text{Init} \) be the set of all possible initial program states, i.e., all program states that may occur before the execution of \( b_{st} \):

\[
\text{Coll}(n) = \bigsqcup_{i \in \text{Init}} \bigsqcup_{\pi \text{ is a path from } s \text{ to } n} [\pi]_{conc(i)}
\]  

(3.3)

The set \( D_{coll} \) forms a complete lattice, i.e. the power-set lattice \( (D_{coll}, \subseteq, \cup, \cap, \perp_{coll}, \top_{coll}) \) is a completed lattice, its partial order is set inclusion, \( \subseteq \), the least upper bound with the set union, \( \cup \), the greatest lower bound with set intersection, \( \cap \), \( \perp_{coll} = \emptyset \) for the least element of \( D_{coll} \), and \( \top_{coll} = 2^{D_{conc}} \) for the greatest element.

The main idea of abstract interpretation is to use descriptions of sets of values, but not the values themselves. The elements of a new abstract domain \( D_{abs} \) describe the sets of states. Abstract interpretation describes abstract states of the executing program with incomputable collecting semantics lattice. Each element of the abstract domain is defined by a concretization function:

\[
\beta : D_{abs} \rightarrow D_{coll}
\]

And an abstraction function can describes the relation of a set of elements from the concrete domain to the abstract domain:
\( \alpha : D_{coll} \rightarrow D_{abs} \)

### 3.1.3 Abstract Semantics

As similar to collecting semantics, the effect of program statements can be computed by abstract semantics with a monotone abstract function \( f' : B \rightarrow D_{abs} \rightarrow D_{abs} \).

**Definition 3.4** (Abstract semantics).

\[
\llbracket \pi \rrbracket_{abs} := \begin{cases} 
  id & \text{if } \pi \text{ is the empty path} \\
  (b_2, \ldots, b_k)_{abs} \circ f'(b_1) & \text{otherwise} 
\end{cases} 
\]  

(3.4)

An abstract to the collecting path semantics and the program points to elements of the abstract domain is mapped by the abstract collecting semantics \( \Lambda : B \rightarrow D_{abs} \).

\[
\Lambda = \bigsqcup \{ \llbracket \pi \rrbracket_{abs}(\text{Init}) \mid \pi \text{ is a path from } b_{st} \text{ to } b \} 
\]

Again, the Init denotes the set of all possible initial program states. \( b_{st} \) is the start node and \( b \) is end node of the collecting path.

### 3.1.4 Establishing Soundness

Before to introduce the soundness of the abstraction interpretation, we will give two definitions local consistency and adjoining. The local consistency describes a notion of approximation in the concrete and abstract domain.

**Definition 3.5** (Monotone). A function is monotone, if the function between ordered sets that preserves the given order.

It is easy to know that \( \alpha \) and \( \beta \) are both partial order sets, so they are both monotone.

**Definition 3.6** (Local consistency). \( f \) and \( f' \) are locally consistent, if

\[
\forall x \in D_{coll} : f(x) \subseteq \beta(f'(\alpha(x))). 
\]  

(3.5)

And P. Cousot and R. Cousot add the adjoining conditions on \( \alpha \) and \( \beta \).

**Definition 3.7** (Adjoint). \( \alpha \) and \( \beta \) are strongly adjoining, if

\[
\forall x \in D_{coll} : x \subseteq \beta(\alpha(x)) \text{ and} 
\]  

(3.6)
\[ \forall x \in D_{abs} : x = \alpha(\beta(x)). \]  \hspace{1cm} (3.7)

If both \( \alpha \) and \( \beta \) are monotone and strongly adjoint, then a Galois insertion can be defined between \( D_{coll} \) and \( D_{abs} \). In order to predict any state of a program point at run-time by the analysis, the abstract interpretation should be sound (i.e., the analysis should predict any state for each program point at run-time). But abstract interpretation cannot exact to predict all the states due to the undesirability problems, it usually will predict some unreachable states at run-time, i.e., it is conservative. The precision which is used to measure the quality of the computed information should be as good as possible.

Theorem for soundness of program analyses is as follows:

**Theorem 3.1** (Soundness of program analyses) Consider an abstract interpretation satisfying:

- \( \langle D_{coll}, \subseteq, \cup, \bot_{coll} \rangle \) and \( \langle D_{abs}, \subseteq, \cup, \bot_{abs} \rangle \) are complete join semilattices;
- \( \alpha \) and \( \beta \) are monotone and strong adjoint;
- the abstract operation \( f' \) is locally consistent with the concrete operation \( f \).

Then, \( Coll \subseteq \beta(\Lambda) \). The proof soundness for analysis refer to P. Cousot and R. Cousot (1977). And the proof is illustrated by the following diagram:

![Figure 3.1: Illustration of the soundness proof](image)

### 3.1.5 Computing the Abstract Semantics

Due to the abstract collecting semantics \( \Lambda \) [39], an approximate algorithm, which is called maximal fixed point solution (MFP) [39] can be used in data-flow analysis.
**Definition 3.7** (Maximal fixed point solution, $MFP$).

\[ \Lambda' = \begin{cases} [b_{st}]_{abs} (\alpha(\text{Init})) & \text{if } b = b_{st} \\ [b]_{abs} (\bigsqcup \{\Lambda'(m) | m \text{ predecessor of } b\}) & \text{otherwise} \end{cases} \] (3.8)

The correctness of this approach has been shown in Kam and Ullman (1977), Martin (1995). According to items 1-3 of Theorem 3.1, it guarantees the correctness of the analysis.

### 3.2 The analysis for LRU

As mentioned in Chapter 1, 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 [8], 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 [8]. Before AI analysis, the categorizations of memory references should be defined: 1). always hit ($AH$), The memory reference will always result in a cache hit; 2). always miss ($AM$), the memory reference will always result in a cache miss; 3). persistent ($FM$), the first memory reference could neither be classified as $AH$ nor $AM$. But the second and all further executions of the memory reference will always result in a cache hit; 4). not classified ($NC$), the memory reference could neither be classified as $AH$, $AM$ nor $FM$. 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 \[8\], \[10\], \[11\], \[12\] for details about these fix-point analyses.

### 3.3 The analysis for FIFO

#### 3.3.1 Challenges when predicting hits

Because the mls of FIFO cache is lower than LRU cache, it poses additional challenges for FIFO analysis. Below is one example to illustrate the challenge.

\[ q_1 = [\bot, \bot, \bot, \bot] \xrightarrow{\mathcal{M}} [x, \bot, \bot, \bot] \xrightarrow{\mathcal{M}} [y, x, \bot, \bot] \xrightarrow{\mathcal{M}} [z, y, x, \bot] = q_1' \]
\[ q_2 = [x, a, y, z] \xrightarrow{\mathcal{H}} [x, a, y, z] \xrightarrow{\mathcal{M}} [x, a, y, z] \xrightarrow{\mathcal{H}} [x, a, y, z] = q_2' \]
\[ q_3 = [a, b, c, x] \xrightarrow{\mathcal{H}} [a, b, c, x] \xrightarrow{\mathcal{M}} [y, a, b, c] \xrightarrow{\mathcal{H}} [z, y, a, b] = q_3' \]
\[ q_4 = [a, b, y, c] \xrightarrow{\mathcal{M}} [x, a, b, y] \xrightarrow{\mathcal{M}} [x, a, b, y] \xrightarrow{\mathcal{M}} [z, x, a, b] = q_4' \]

In the example, \( q_i \) denotes the initial cache set, \( q_i' \) denotes the cache set, which is carried out on an access sequence on the initial cache set, \( H \) denotes a cache hit, \( M \) denotes a cache miss. A same access sequence \( s = <x, y, z> \) is carried out on different cache set \( q_i \), although three different memory blocks are accessed, but some of the results of \( q_i' \) do not contain all of the accessed blocks. But for LRU policy, a \( k \)-way cache set always contains \( k \) most recently used memory blocks, so \( \{x, y, z\} \) should be cached after the accessed sequences \( s \). Due to this difference, FIFO analysis is hard to predict.

In order to overcome this challenge problem, Grund and Reineke proposed two different method to predict cache hits \[3\] \[7\]. In \[3\] they define a hit after miss domain which consider to predict hit after misses. Continuously, they present a policy-independent cache analysis framework in which any number of independent cache analysis can cooperate, so the must analysis can cooperate with may analysis, and it uses the hit after miss domain to predict cache hits. To further precise predict cache hits, they propose a FIFO analysis, which is based on phase detection \[7\]. It defines a phase detection must domain in the must analysis which can predict a significant amount of cache hits without relying on may information. We will review of this method in the next section.
3.3.2 FIFO analysis by phase detection

The main contribution for Grund and Reineke’s precise analysis is that they propose a method, which is called phase detection to predict hit or miss for FIFO cache analysis. We will introduce the phase detection in this section.

**Lemma 3.1** Let \( b \in B \), \(|A(b)| = n \leq k \) (the access sequence \( b \) contains at most \( k \) pair wise different blocks). Then, for all cache sets \( q \in Q_k \) and \( q' := U_{Q_k}(q, b) \):

\[
A(b) \subseteq C(q') \lor C_1(q') \subseteq A(b) \tag{3.9}
\]

\( k \) is the association of the cache, \( q \) denotes the cache set, \( Q_k \) denotes the cache, \( U_{Q_k}(q, b) \) is the update state after the \( b \) access the cache. If all the access are hits, all blocks were cache in \( q \). All the blocks are still cached after an update state, because the FIFO cache keep the same state upon a hit. Otherwise, at least one miss must be happened. In this case, the last in position of the \( q' \) contains a block due to miss happen in FIFO cache(\( C_1 \subseteq A(b) \)).

**Definition 3.8 (Phase)** For a set of memory blocks \( B \), a \( B\)-phase is an access sequence \( b \) such that \( A(b) = B \). The phase blocks of a \( B\)-phase are the memory blocks in \( B \). If \(|B| = |A(b)| = n \), we say that \( s \) is of size \( n \).

For example access sequence \( b = <a, b, b, a, c> \), the size of \( b \) is 3, we call this is \( \{a, b, c\}\)-phase.

**Theorem 3.2** Let \( b \in B \) be a \( B\)-phase of size \( n \leq k \) that can be partitioned into \( j \leq n \) \( B\)-phases, \( b_i \), i.e. \( b := b_1 \circ \cdots \circ b_j \). Then for each \( q \in Q_k \), and \( q' := U_{Q_k}(q, b) \):

It is similar to Lemma 3.1, the difference is that theorem 3.2 departs the access sequence \( b \) into many sub sequences; each sub sequence can be looked as a \( B\)-phase. With the multiple phases, if only hits in \( b \), the cache state will keep the same, it is the same as Lemma 1. Otherwise, accumulate in last-in positions by the number of departed phases, the access element at accumulated last-in positions will belong to the coordinated \( B\)-phase.

\[
[d, c, b, a] \xrightarrow{(a,b,e)} [e, d, b, c] \xrightarrow{(b,a,e)} [a, e, d, c] \xrightarrow{(a,b,e)} [b, a, e, d]
\]

Above fig shows Theorem 3.2 to predict multiple phases. The access
sequence \( b = \langle a, b, e, b, a, e, a, b, e \rangle \), but it can depart to three sub-sequence \( \langle a, b, e \rangle, \langle b, a, e \rangle \) and \( \langle a, b, e \rangle \), to predict the multiple phase, the accumulated last-in positions will belong to the coordinated \( B \)-phase. For the first \( \langle a, b, e \rangle \) sequence, the accumulated last-in position is 1, so the last position element \( e \) belong to \( B_1 \)-phase. The similar to \( \langle b, a, e \rangle \) and \( \langle a, b, e \rangle \) sequence. Due to this special property, FIFO must analysis can use it to predict cache hits. Assume that the association of the cache is \( K, n = 1 \ldots K \), the must analysis maintains:

- phase blocks \( B_n \in 2^R \);
- phase progress, to record the already accessed blocks in current phase, \( P_n \in 2^B \);
- phase counter the phase counter, which is record the number of the detected \( B \)-phases, \( p_{cn} \in N \).

Phase blocks \( B_n \) are the \( n \) most-recently-used blocks, and also \( i \leq j \), \( B_i \subseteq B_j \), so all the \( B_n \) can be encoded in a single LRU stack. And every block \( B_i \) has the property: \( P_i \subseteq B_i \), \( P_n \) can be encoded as “pointers” into the stack. It can be predicted hits for each block by the condition \( p_{ci} = |B_i| \), if the accessing tag belong the \( B_i \)-phase. Must analysis perform multiple analyses for different \( B_n \) sets in parallel.

The main idea of the FIFO analysis is to partition the access sequence into many phases, and then it uses the Theorem 3.2 to predict hits by the condition \( p_{ci} = |B_i| \). The analysis can be proceeded in three stages: At first stage, it determined the phase blocks \( B \), for the memory accesses can be defined as \( B \)-phase. The second stage, the analysis try to detect \(|B| - 1 \) \( B \)-phases, it will increase the number phase progress, when it detects \(|B| \) \( B \)-phases in total successfully, then it can continue to proceed at the third stage. If the accessed block is not contained in \( B \), it will restart from stage one to determine the new phase, because such a block may evict any block which is contained in the phase block. In the third stage, the analysis can make use of the accumulated information for all the blocks. If the memory accessing block is contained in the \( B_i \)-phase, the hits prediction condition of each block \( B_i \) is issued as \( p_{ci} = |B_i| \).

We assume that the phase block is \( B = \{x, y\} \), the process to detect the single phase block is showed bellow:

From Table 3.1, the second and third \( y \) are not be detected as \( Hit \), this because the value of phase counter is less than the size of the phase block \(|B| = 2 \). But the fourth \( y \), the value of the phase counter is equal to the size of the phase block, so it can be predicted as \( Hit \). When the phase
Table 3.1: Example for $B = \{ x, y \}$

<table>
<thead>
<tr>
<th>phase</th>
<th>$\emptyset$</th>
<th>${x}$</th>
<th>$\emptyset$</th>
<th>${y}$</th>
<th>${y, x}$</th>
<th>$\emptyset$</th>
<th>${y}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Hit</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

counter reach to the size of the phase block, it will restart to detect the accessing blocks, like after the first $y$ tag in the table, phase should be set to empty set. Here we will only give a simple example to describe how the phase detection works. The detail implemented algorithm for FIFO must analysis described in [7].
Chapter 4

The New Analysis of FIFO

4.1 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.

We give a loop accessing $K$ blocks and a $K$-way cache. Since the whole loop can 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 [6]. As we mentioned the FIFO policy in Chapter 2, a block access is a hit, it will not change the cache state, and a block access is a miss, it will increase the age of the blocks in the cache set. Because there are only $K$ blocks in the loop, no matter what is the initial states of the cache set, it happens at most four misses in the cache set regarding this loop; it cannot evict the blocks out of the cache set. 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 interested 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 [6] cannot answer these questions.

Now we formally introduce the new metric miss distance:
**Definition 4.1** (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.

For example, assume the initial state of the cache set are empty, an access sequence of the same block are \( \delta \rightarrow a \rightarrow b \rightarrow \delta \rightarrow a \rightarrow d \rightarrow \delta \), the miss distance is 4, the blocks cause consecutive cache misses are \{ \delta, a, b, d \}.

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

**Lemma 4.1** 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. According to the rule of FIFO policy, the miss distance is \( K \) for a \( K \)-way FIFO cache, and is the same as \( K \) for LRU, MRU, but its is \( \log_2 K + 1 \) for a \( K \)-way PLRU cache (proof omitted). LRU, MRU and FIFO are optimal regarding this metric:

**Lemma 4.2** The miss distance of a \( K \)-way cache with every 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 4.3** Given a cache with miss distance \( X \), and a loop \( L \) in which the number of different blocks is no larger than \( X \). Any block in \( L \) encounters at most one miss (the cold miss) every time when the program executes inside \( L \).

**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 needed for the next miss on \( \delta \) to happen. However, this is impossible when the program executes inside the loop \( L \) 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 ac-
cessed in loop L is no larger than $K$, then all the nodes are $FM$ regarding L. Although the first access might be a hit regarding the cache initial state; those nodes can also be approximate looked as $FM$. 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.

4.2 Quantitative FIFO analysis

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

$$\delta \rightarrow a \rightarrow b \rightarrow \delta \rightarrow c \rightarrow d \rightarrow \delta \rightarrow e \rightarrow f \rightarrow g \rightarrow \delta \rightarrow h \rightarrow i \rightarrow \delta$$

Suppose the underlying FIFO cache has 4 ways, then by Lemma 4.3 and 4.1 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, and 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 infinitely 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 4.2** (Maximal Stack Distance). Let $n_i$ and $n_j$ be nodes in loop L 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 L, denoted by $\Pi_L(n_i, n_j)$, is defined as:

$$\Pi_L(n_i, n_j) = \begin{cases} \max \{ \pi(p) \mid p \in P(n_i, n_j) \} & \text{if } P(n_i, n_j) \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \quad (4.1)$$

where $P_L(n_i, n_j)$ is the set of paths satisfying:

- All nodes along the path are included in loop 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 $\delta$.

Figure 4.1: A CFG example. The letter inside each circle denotes the block accessed by this node.

Fig 4.1 illustrates the maximal stack related to block $\delta$ with an inner loop $L_{in}$ and an outer loop $L_{out}$ respectively. For example, we have $\Pi_{L_{in}}(n_6, n_6) = 3$ since the “longest” path from $n_6$ back to $n_6$ in the scope of $L_{in}$ accesses 3 different blocks $(n_6 \rightarrow n_3 \rightarrow n_5 \rightarrow n_6)$, while $\Pi_{L_{in}}(n_6, n_6) = 6$ since the “longest” path in the scope of $L_{out}$ 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 4.4** We give a cache with miss distance $K$. Let $\Delta$ be the set of nodes in a loop $L$ accessing block $\delta$, and it holds

$$\forall n_i, n_j \in \Delta : \Pi_L(n_i, n_j) \leq \ell$$

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

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

where $\gamma = 1/(1 + \lfloor (K-1)/(\ell - 1) \rfloor)$, $x$ is the total number of executions of nodes in $\Delta$ and $y$ is the total number of times the program enters loop $L$ 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 $\Delta$ 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 different 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 $L$, 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 at least $\lfloor (K - 1)/(\ell - 1) \rfloor$ times before it is evicted from the cache.

The program enters loop $L$ 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 $L$. Above, we have proved there are at least $\lfloor (K - 1)/(\ell - 1) \rfloor$ hits by nodes in $\triangle$ between any pair of consecutive misses by nodes in $\triangle$, so the total number of misses among these $x_m$ accesses to $\delta$ can be bounded by:

$$1 + \lfloor x_m/(1 + \lfloor (K - 1)/(\ell - 1) \rfloor) \rfloor \quad (4.4)$$

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

$$y + \sum_{i=1}^{y} \lfloor x_m/(1 + \lfloor (K - 1)/(\ell - 1) \rfloor) \rfloor \quad (4.5)$$

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

Intuitively speaking, Lemma 4.4 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 $L$. Note that a node may be included by several $\gamma$-set regarding different loops and different $\gamma$ values. For example, suppose the CFG in Fig 4.1 is executed with a cache of miss distance 8, then $n_6$ is included in a singleton $\frac{1}{4}$-set regarding $L_{in}(\gamma = 1/(1 + \lfloor (8 - 1)/(3 - 1) \rfloor) = 1/4)$, as well as a $\frac{1}{2}$-set $\{n_6, n_8\}$ regarding $L_{out}(\gamma = 1/(1 + \lfloor (8 - 1)/(6 - 1) \rfloor) = 1/2)$.

To use Lemma 4.4, one needs to compute the maximal stack distance $\Pi_L()$. In general, the time complexity of computing $\Pi_L()$ is at least exponential regarding the number of cache ways so we need efficient approximation to handle real-life-size problems. Actually, computing $\Pi_L()$ 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 $\Pi_L()$.

\footnote{This can be shown by a reduction from the well-know 3-SAT problem}
Lemma 4.5 Given a $\ell$-way LRU cache. Let $\Delta$ be the set of nodes in a loop $L$ accessing block $\delta$. If all the nodes in $\Delta$ are classified as AH or FM regarding $L$ by any safe analysis, then it must hold:

$$\forall n_i, n_j \in \Delta : \Pi_L(n_i, n_j) \leq \ell \quad (4.6)$$

Proof: Prove by contradiction. Assume two nodes $n_i$ and $n_j$ in $S$ have $\Pi_L(n_i, n_j) > \ell$. Then by the definition of $\Pi_L(n_i, n_j)$ there is at least one path from $n_i$ to $n_j$ inside $L$ 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 4.1, Lemma 4.4 and 4.5, we obtain the main result of this section:

Theorem 4.1 Let $\Delta$ be the set of nodes in a loop $L$ accessing block $\delta$. If all the nodes in $\Delta$ are classified as AH or FM regarding $L$ by a safe analysis on a $\ell$-way LRU cache, then the total number of misses caused by nodes in $\Delta$ on a $K$-way FIFO cache is bounded by:

$$[\gamma \cdot x] + y \quad (4.7)$$

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 $L$ during the whole program execution. The association $K$ has been configured by the processor, so $x$, $\ell$ will affect the “ratio” $\gamma$. According to our “ratio” $\gamma$, the prediction will be pessimistic, when the Maximal Stack Distance of the loop $L$ is bigger and $x$ is smaller. But it will be better, when the loop bound is larger (the value of $x$ will be larger), and the maximal stack distance is smaller. The results of the prediction depend on the structure of the loop $L$ and the configuration of the cache.

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 $L$ under LRU in order to minimize the “miss ratio”. To do this, we only need to conduct the LRU cache analysis once with a $K$-way cache actually. 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 $d$ has maximal age of 4 before the execution of a node accessing $d$, 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 [40], [41], [42].

4.3 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 4.3 (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 [43], 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 4.4 (Loop).** A loop in the CFG is a tuple $L_l = (\text{entr}_l, head_l, body_l, Ipb_l)$ with:

- $\text{entr}_l$: 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;
- $Ipb_l$: the loop bound.

The overall FIFO cache analysis results can be summarized as follows: $AH$ nodes decided by the Must analysis in [3], [7], $FM$ nodes (regarding some loop) decided according to Lemma 4.3 and $\gamma$-sets (regarding some loop) determined by Theorem 4.1. Lemma 4.3 by comparing the miss distances of the nodes to the association of the cache to decide the $FM$ classification. 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 \( L \), 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 \( \gamma \)-sets with \( \gamma = 0 \) (the bound (2) becomes \( y + \lfloor 0 \cdot x \rfloor = y \)). For simplicity of presentation, in the following we use term \( \gamma \)-set to include both the original ones derived by Theorem 4.1 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 \( b_a \), \( c_a \) 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:

\[
\text{Maximize } \left\{ \sum_{all b_a} c_a \right\}.
\]  

(4.8)

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:

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

(4.9)

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 \(L_l\) 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 \(L_l\). So we can bound the number of misses incurred by a \(\gamma\)-set:

\[
\forall (S, L_l) \text{ s.t. } S \text{ is a } \gamma\text{-set regarding } L_l
\]

\[
\sum_{n_i \in S} z_i \leq \sum_{e_j \in \text{entr}_l} y_j + \sum_{n_i \in S} [x_a \cdot \gamma] \tag{4.10}
\]

Where \(\text{entr}_l\) is the set of entry edges of \(L_l\) and \(y_j\) 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:

\[
\forall b_a : x_a = \sum_{e_j \in \text{input}(b_a)} y_j = \sum_{e_j \in \text{output}(b_a)} y_j \tag{4.11}
\]

Where \(\text{input}(b_a)\) and \(\text{output}(b_a)\) 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 \(Ipb_l\) times, so we have:

\[
\forall L_l, \forall b_a \in \text{body}_l : x_a \leq Ipbl \cdot \sum_{e_j \in \text{entr}_l} y_j \tag{4.12}
\]

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 \(x_a \leq (Ipb_l + 1) \cdot \sum_{e_j \in \text{entr}_l} y_j\).
Chapter 5

Experimental Evaluation

In order to evaluate our FIFO policy, we should consider the following two questions:

- What is the precision of our proposed FIFO analysis?
- How about the predictability comparison between Grund and Reineke’s FIFO and our FIFO analysis?

To evaluate the precision of our FIFO analysis, we compare the estimated WCET obtained by our FIFO analysis and the measured WCET obtained by simulation with FIFO caches. To evaluate the predictability comparison between Grund and Reineke’s FIFO analysis and our FIFO analysis, we compare the estimated WCET obtained by our FIFO analysis and Grund and Reineke’s FIFO must analysis, if the estimated WCET by our FIFO analysis is better or close to Grund and Reineke’s FIFO must analysis, then it is fair to claim that our FIFO analysis is also a good candidate for FIFO cache analysis.

5.1 Experiment Setup

In our experiment, 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 Malardalen Real-Time Benchmark [41]. 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

30
tens to about 8000 assembly instructions compiled by a gcc compiler re-targeted to the SimpleScalar architecture [33] 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. Two policies are used for our in-house simulator, policy one, if a basic block has more than one branches, and each branch executes the branches in an inter-leaving manner. Policy two, if a loop head is encountered, we have two choices, either enter the loop or exit the loop, we can take the exit edge only when the basic block in the loop is exhausted. The cache configuration is the same as our FIFO analysis, it ensures the same instructions to access the cache in our in-house simulator. 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.

5.2 Results

Three tables present the test results for the 0.5k cache; all the results contain the simulation and estimation WCET. Table I shows the results with 4-way caches. The simulation of the programs are the measured execution time(column “SIM WCET”). The estimation WCET (Grund and Reineke’s and our new FIFO analysis) of the programs are present in column “EST WCET”. We calculate the over-estimation ratio of our and Grund and Reineke’s FIFO analysis are presented in column “Over.Est.”. For example, the “SIM. WCET” and “EST. WCET” of program bs under Our FIFO is 3863 and 3953 respectively, then the over-estimation ratio is (3953-3863)/3863 = 2.33%. Finally, we calculate the excess ratio of our FIFO analysis over Grund and Reineke’s FIFO analysis (column “exc. FIFO”). For example, the estimated WCET of program bs under our FIFO and Grund and Reineke’s FIFO is 3953 and 12278 respectively, then the excess ratio is (12278−3953)/12278 = 67.80%.

From Table I we can see that the WCET estimation with our FIFO analysis has very good precision: the over-estimation comparing with the simulation WCET is on average 7.39%, but Grund and Reineke’s over-estimation ratio is on average 183.27%. We can also see that the exc.FIFO WCETs
with our FIFO and Grund and Reineke’s FIFO caches are improved drastically, we improved on average about 51.60% WCET bounds for the test benchmarks.

Then we conduct experiments with 8-way and 16-way caches (with the same total cache size but different number of cache sets). Table II and III show the results with 8-way and 16-way caches respectively. The overestimation by our FIFO analysis is 6.88% and 4.28% for 8-way and 16-way caches respectively, and the difference between our FIFO and Grund and Reineke’s FIFO analysis is 50.16% and 52.29% which improved the WCET drastically.

We also found that the over-estimation ratios (7.39%, 6.88%, 4.28%) of the WCET by our FIFO analysis scales linearly with respect to the number of ways, this is due to the calculation of miss ratio depend on two factors (the cache k way and max miss distance). As we mentioned in Chapter 3, the number of misses caused by nodes can bounded by: $\lfloor \gamma \cdot x \rfloor + y$, where $\gamma = 1/(1 + [(K - 1)/(\ell - 1)])$. The control flow information are consistence after the compile the same program, so the factor y is a constants in the formula, only $\gamma$ affects the number of miss, but only K and $\ell$ affect $\gamma$, if the size cache set is enough to install the tags in some loops in program, the $\ell$ also be a constancy in those loops. So K affects the number of miss at this condition only. We can explain the phenomenon that why our FIFO analysis scales linearly with respect to the number of ways at this condition. Due to this property, assume that we can increase the K infinitely, the over-estimation ratios will close to zero.

We also conducted experiments with various configurations: the cache size is 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. 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 \[45\] on a laptop with an Intel Core i7 CPU (2.7GHz). The computation for each program takes at most several seconds.
<table>
<thead>
<tr>
<th>Policy</th>
<th>SIM WCET</th>
<th>EST WCET</th>
<th>Over . Est</th>
<th>exc. FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>3070527</td>
<td>3379452</td>
<td>10.06%</td>
<td>76.74%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>3070527</td>
<td>14526900</td>
<td>373.11%</td>
<td></td>
</tr>
<tr>
<td>bs</td>
<td>3863</td>
<td>3953</td>
<td>2.33%</td>
<td>67.80%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>3863</td>
<td>12278</td>
<td>217.84%</td>
<td></td>
</tr>
<tr>
<td>bsort</td>
<td>309792</td>
<td>312069</td>
<td>0.74%</td>
<td>84.40%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>309792</td>
<td>2000810</td>
<td>545.86%</td>
<td></td>
</tr>
<tr>
<td>crc</td>
<td>243633</td>
<td>250959</td>
<td>3.01%</td>
<td>66.76%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>243633</td>
<td>754923</td>
<td>209.86%</td>
<td></td>
</tr>
<tr>
<td>edn</td>
<td>27480285</td>
<td>43420833</td>
<td>58.01%</td>
<td>26.53%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>27480285</td>
<td>59096800</td>
<td>115.05%</td>
<td></td>
</tr>
<tr>
<td>expint2</td>
<td>106989</td>
<td>108465</td>
<td>1.38%</td>
<td>64.85%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>106989</td>
<td>308598</td>
<td>188.44%</td>
<td></td>
</tr>
<tr>
<td>fdct</td>
<td>200169</td>
<td>200178</td>
<td>&lt; 0.01%</td>
<td>0</td>
</tr>
<tr>
<td>Grund’s</td>
<td>200169</td>
<td>12278</td>
<td>217.84%</td>
<td></td>
</tr>
<tr>
<td>fibcall</td>
<td>2315</td>
<td>2351</td>
<td>1.56%</td>
<td>45.80%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>2315</td>
<td>441341</td>
<td>261.80%</td>
<td></td>
</tr>
<tr>
<td>fir</td>
<td>134960</td>
<td>135887</td>
<td>0.69%</td>
<td>66.18%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>134960</td>
<td>401819</td>
<td>197.73%</td>
<td></td>
</tr>
<tr>
<td>insertsort</td>
<td>258271</td>
<td>258307</td>
<td>0.01%</td>
<td>70.40%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>258271</td>
<td>872719</td>
<td>237.91%</td>
<td></td>
</tr>
<tr>
<td>janne</td>
<td>121985</td>
<td>122057</td>
<td>0.06%</td>
<td>72.34%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>121985</td>
<td>441341</td>
<td>261.80%</td>
<td></td>
</tr>
<tr>
<td>jfdctint</td>
<td>205544</td>
<td>205580</td>
<td>0.02%</td>
<td>4.32%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>205544</td>
<td>214859</td>
<td>4.53%</td>
<td></td>
</tr>
<tr>
<td>matmult</td>
<td>13016215</td>
<td>13823074</td>
<td>6.20%</td>
<td>67.41%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>13016215</td>
<td>42419600</td>
<td>225.90%</td>
<td></td>
</tr>
<tr>
<td>ndes</td>
<td>4369317</td>
<td>4804548</td>
<td>9.96%</td>
<td>29.26%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>4369317</td>
<td>6791360</td>
<td>55.43%</td>
<td></td>
</tr>
<tr>
<td>ns</td>
<td>490834000</td>
<td>490834000</td>
<td>0.00%</td>
<td>71.30%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>490834000</td>
<td>1715580000</td>
<td>249.52%</td>
<td></td>
</tr>
<tr>
<td>prime</td>
<td>8152</td>
<td>8575</td>
<td>5.19%</td>
<td>66.59%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>8152</td>
<td>25665</td>
<td>214.83%</td>
<td></td>
</tr>
<tr>
<td>qsort</td>
<td>8442052</td>
<td>8738430</td>
<td>3.51%</td>
<td>70.71%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>8442052</td>
<td>29830100</td>
<td>253.35%</td>
<td></td>
</tr>
<tr>
<td>qurt</td>
<td>33336</td>
<td>48402</td>
<td>45.19%</td>
<td>22.82%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>33336</td>
<td>62712</td>
<td>88.12%</td>
<td></td>
</tr>
<tr>
<td>select</td>
<td>8854719</td>
<td>8991978</td>
<td>1.55%</td>
<td>69.11%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>8854719</td>
<td>29107000</td>
<td>228.72%</td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td>8780</td>
<td>16088</td>
<td>83.23%</td>
<td>25.56%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>8780</td>
<td>21613</td>
<td>146.16%</td>
<td></td>
</tr>
<tr>
<td>statemate</td>
<td>228002</td>
<td>232601</td>
<td>2.02%</td>
<td>0.37%</td>
</tr>
<tr>
<td>Grund’s</td>
<td>228002</td>
<td>233465</td>
<td>2.40%</td>
<td></td>
</tr>
<tr>
<td>ud</td>
<td>26649810</td>
<td>26732997</td>
<td>0.31%</td>
<td></td>
</tr>
<tr>
<td>Grund’s</td>
<td>26649810</td>
<td>78425700</td>
<td>194.28%</td>
<td>65.91%</td>
</tr>
<tr>
<td>average:</td>
<td>7.39%</td>
<td>51.60%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Grund's average: 183.27%
## Table II. Experiment results with 8-way caches

<table>
<thead>
<tr>
<th>Policy</th>
<th>SIM WCET</th>
<th>EST WCET</th>
<th>Over . Est</th>
<th>exc. FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>3158700</td>
<td>3432318</td>
<td>7.97%</td>
<td>76.38%</td>
</tr>
<tr>
<td>bs</td>
<td>3863</td>
<td>3953</td>
<td>2.28%</td>
<td>67.80%</td>
</tr>
<tr>
<td>bsort</td>
<td>311556</td>
<td>312141</td>
<td>0.19%</td>
<td>84.40%</td>
</tr>
<tr>
<td>crc</td>
<td>248529</td>
<td>251823</td>
<td>1.31%</td>
<td>66.64%</td>
</tr>
<tr>
<td>edn</td>
<td>32156649</td>
<td>45711297</td>
<td>29.65%</td>
<td>22.65%</td>
</tr>
<tr>
<td>expint2</td>
<td>106989</td>
<td>108249</td>
<td>1.16%</td>
<td>64.92%</td>
</tr>
<tr>
<td>fdct</td>
<td>200169</td>
<td>200178</td>
<td>&lt; 0.01%</td>
<td>0</td>
</tr>
<tr>
<td>fibcall</td>
<td>2315</td>
<td>2351</td>
<td>1.53%</td>
<td>45.80%</td>
</tr>
<tr>
<td>fir</td>
<td>134960</td>
<td>136103</td>
<td>0.84%</td>
<td>66.13%</td>
</tr>
<tr>
<td>insertsort</td>
<td>258271</td>
<td>258307</td>
<td>&lt; 0.01%</td>
<td>70.40%</td>
</tr>
<tr>
<td>janne</td>
<td>121985</td>
<td>122057</td>
<td>0.06%</td>
<td>72.34%</td>
</tr>
<tr>
<td>jfdctint</td>
<td>205544</td>
<td>205580</td>
<td>0.02%</td>
<td>4.32%</td>
</tr>
<tr>
<td>matmult</td>
<td>12896236</td>
<td>13400965</td>
<td>3.77%</td>
<td>68.41%</td>
</tr>
<tr>
<td>ndes</td>
<td>4471620</td>
<td>4827633</td>
<td>7.37%</td>
<td>28.92%</td>
</tr>
<tr>
<td>ns</td>
<td>490834000</td>
<td>490834000</td>
<td>0.00%</td>
<td></td>
</tr>
<tr>
<td>prime</td>
<td>8260</td>
<td>8575</td>
<td>3.67%</td>
<td>66.59%</td>
</tr>
<tr>
<td>qsort</td>
<td>8738430</td>
<td>879490</td>
<td>0.59%</td>
<td>70.53%</td>
</tr>
<tr>
<td>sqrt</td>
<td>25992</td>
<td>47781</td>
<td>45.60%</td>
<td>23.81%</td>
</tr>
<tr>
<td>select</td>
<td>8854719</td>
<td>8999178</td>
<td>1.53%</td>
<td>69.11%</td>
</tr>
<tr>
<td>sqrt</td>
<td>10552</td>
<td>17860</td>
<td>40.92%</td>
<td>17.36%</td>
</tr>
<tr>
<td>statemate</td>
<td>228002</td>
<td>232601</td>
<td>1.98%</td>
<td>0.37%</td>
</tr>
<tr>
<td>ud</td>
<td>26649810</td>
<td>26732997</td>
<td>0.31%</td>
<td>65.91%</td>
</tr>
<tr>
<td>average: Our's</td>
<td>6.88%</td>
<td>50.16%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grund's</td>
<td>169.00%</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table III. Experiment results with 16-way caches

<table>
<thead>
<tr>
<th>Policy</th>
<th>SIM</th>
<th>EST</th>
<th>WCET</th>
<th>WCET</th>
<th>Over . Est</th>
<th>exc. FIFO</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>Our's</td>
<td>2959386</td>
<td>2987277</td>
<td>0.94%</td>
<td>79.44%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>2959386</td>
<td>14529600</td>
<td>390.97%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bs</td>
<td>Our's</td>
<td>3863</td>
<td>3953</td>
<td>2.33%</td>
<td>67.80%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>3863</td>
<td>12278</td>
<td>217.84%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bsort</td>
<td>Our's</td>
<td>311556</td>
<td>312222</td>
<td>0.21%</td>
<td>84.40%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>311556</td>
<td>2000810</td>
<td>542.20%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>crc</td>
<td>Our's</td>
<td>248529</td>
<td>252255</td>
<td>1.50%</td>
<td>66.59%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>248529</td>
<td>754923</td>
<td>203.76%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edn</td>
<td>Our's</td>
<td>41002677</td>
<td>50044320</td>
<td>22.05%</td>
<td>15.32%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>41002677</td>
<td>59096800</td>
<td>44.13%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>expint2</td>
<td>Our's</td>
<td>106989</td>
<td>108249</td>
<td>1.18%</td>
<td>64.92%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>106989</td>
<td>308598</td>
<td>217.84%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fdct</td>
<td>Our's</td>
<td>200169</td>
<td>200178</td>
<td>&lt; 0.01%</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>200169</td>
<td>200178</td>
<td>&lt; 0.01%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fibcall</td>
<td>Our's</td>
<td>2315</td>
<td>2351</td>
<td>1.56%</td>
<td>45.80%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>2315</td>
<td>4338</td>
<td>87.39%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fir</td>
<td>Our's</td>
<td>134960</td>
<td>136103</td>
<td>0.85%</td>
<td>66.13%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>134960</td>
<td>401819</td>
<td>197.73%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insersort</td>
<td>Our's</td>
<td>258271</td>
<td>258307</td>
<td>0.01%</td>
<td>70.40%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>258271</td>
<td>872719</td>
<td>237.91%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>janne</td>
<td>Our's</td>
<td>121985</td>
<td>122057</td>
<td>0.06%</td>
<td>72.34%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>121985</td>
<td>441341</td>
<td>261.80%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jfdctint</td>
<td>Our's</td>
<td>205544</td>
<td>205580</td>
<td>0.02%</td>
<td>4.32%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>205544</td>
<td>214859</td>
<td>4.53%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>matmult</td>
<td>Our's</td>
<td>13086775</td>
<td>13426516</td>
<td>2.60%</td>
<td>68.35%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>13086775</td>
<td>42419600</td>
<td>224.14%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ndes</td>
<td>Our's</td>
<td>4606620</td>
<td>4848585</td>
<td>5.25%</td>
<td>28.61%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>4606620</td>
<td>6791360</td>
<td>47.43%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ns</td>
<td>Our's</td>
<td>490834000</td>
<td>490834000</td>
<td>0</td>
<td>71.39%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>490834000</td>
<td>1715580000</td>
<td>249.52%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>prime</td>
<td>Our's</td>
<td>8260</td>
<td>8575</td>
<td>3.81%</td>
<td>66.59%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>8260</td>
<td>25665</td>
<td>210.71%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>qsort</td>
<td>Our's</td>
<td>8790490</td>
<td>8790490</td>
<td>0</td>
<td>70.53%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>8790490</td>
<td>29830100</td>
<td>239.35%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>qurt</td>
<td>Our's</td>
<td>20106</td>
<td>20376</td>
<td>1.34%</td>
<td>67.51%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>20106</td>
<td>62712</td>
<td>211.91%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>select</td>
<td>Our's</td>
<td>8854719</td>
<td>8991978</td>
<td>1.55%</td>
<td>69.11%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>8854719</td>
<td>29107000</td>
<td>228.72%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td>Our's</td>
<td>14080</td>
<td>20623</td>
<td>46.47%</td>
<td>4.58%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>14080</td>
<td>21613</td>
<td>53.50%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>statemate</td>
<td>Our's</td>
<td>228002</td>
<td>232601</td>
<td>2.02%</td>
<td>0.37%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>228002</td>
<td>233465</td>
<td>2.40%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ud</td>
<td>Our's</td>
<td>26649810</td>
<td>26733033</td>
<td>0.31%</td>
<td>65.91%</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Grund's</td>
<td>26649810</td>
<td>78425700</td>
<td>194.28%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**average:**
- Our's: 4.28%
- Grund's: 183.58%
- Grund's: 35.00%
Chapter 6

Conclusion and Future works

This thesis presented a quantitative approach for FIFO cache analysis. Unlike the previous standard cache analysis based on qualitative AH/AM classification, this new approach quantitatively bounds 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 maintain good efficiency. An important future work is to study how to integrate the quantitative cache analysis with the analysis of other components 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.
Bibliography


