Improving the First-Miss Computation in Set-Associative Instruction Caches

Clément Ballabriga, Hugues Cassé
Université de Toulouse
Institut de Recherche en Informatique de Toulouse (IRIT)
CNRS - UPS - INP - UT1 - UTM
118 route de Narbonne 31062 Toulouse cedex 9
{ballabri,casse}@irit.fr

Abstract

The methods for Worst Case Execution Time (WCET) computation need to analyse both the control flow of the task, and the architecture effects involved by the hosting architecture. An important architectural effect that needs to be predicted is the instruction cache behavior. This prediction is commonly performed by assigning to each program instruction a category that describes its behavior. One of these categories, First Miss, means that the first reference is a cache miss, while the subsequent references give hits. Yet, there is variations in the meanings of this category according to the used methods, capturing overlapping but not equivalents sets of cache behaviors. In this paper, we have analysed the shortcomings of the First-Miss computation methods, and we have deduced an improved First Miss computation approach which captures a maximum of cache behaviors while eliminating some of the most time-consuming processing. We have implemented our method in the frame of C. Ferdinand's categorization method, enhancing his approach for First Miss handling, and compared it with the non-enhanced versions. The results shows a tighter WCET, and a greatly reduced computation time.

1. Introduction

Hard real-time systems are usually composed of tasks that must imperatively finish before their deadlines. In order to guarantee this termination, a method often used is the scheduling analysis, that requires the knowledge of each task WCET. To compute the WCET of a task, several aspects must be taken into account:

- the control flow of the task (flow analysis),
- the hosting hardware (timing analysis)

Each phase requires a bunch of computation to handle accurately the execution flow of the program and the hardware model. An important aspect of the hardware model is the instruction cache. A widely used approach to cope with the cache behaviour is the categorization that assigns to each cache block of the program a category. Usual categories include Always Hit and Always Miss meaning that the access to the cache block results ever, respectively, in a cache hit or in a cache miss. More accurate behaviour can be modeled with the First Miss category that means, intuitively, that the first block access is a miss and the next accesses give hits: once the block is loaded into the cache, it stays inside as long as it is used.

In this paper, we show that the First Miss category has several semantics according to the used methods. We explore the range of possibilities for the First Miss computation and we derive a new method to handle the First Miss range of categories. Implemented in the frame of C. Ferdinand's cache analysis [14, 8], it provides better performances in WCET tightness and in analysis computation time.

In the first section, we introduce the categorization method, and discuss several analyses based on it. The next section proposes several improvements to overcome the limitations of these analyses. Using OTAWA [3] (our framework for WCET analysis), we then evaluate the derived solutions against each others and against the Ferdinand's analysis [8, 14], by comparing them on the criteria of WCET precision and computation time. Based on this results, we propose further improvements to our method. In the fourth section, we experiment our fully improved method to finally conclude in the last section.

2. Related Work and Problem definition

In this paper, we have performed our WCET computation with a largely used approach called the Implicit Path Enumeration Technique (IPET) [10]. In IPET, the program structure and flow facts (such as loop bounds) are modeled by linear constraints. The WCET is then expressed by an objective function to maximize, and an Integer Linear Programming solver is used to compute the result. When the cache is modeled using the categorization approach, the categories are derived into a set of new constraints that represents the cache behaviour in the ILP system [11, 12].

In this section, after a brief introduction to cache memories, we describe the general principle of the categorization method (common to all the analyses discussed in this paper). Next we present several cache behavior analyses using the First Miss categorization and show their advantages and limitations.

2.1. Categorization Approach

The cache is a fast and small memory that stores copies of main memory blocks whose accesses are sped up. When a memory access is performed, either the data is present in the cache, resulting in a fast access called a hit, or it must be retrieved from memory, resulting in a slow access called a miss. The cache is divided into fixed-size lines, and the main memory is divided into cache blocks. Each cache block from memory matches a specific line. In case of miss, the whole cache block containing the target location is loaded into the matching line.

In the A-way associative cache, each line contains A blocks. Consequently, a replacement policy is needed to determine where to put a newly loaded cache block and which already loaded block to wipe out. A widely used policy is the so-called Least Recently Used (LRU) policy, which associates an age $a \in [0..A[$ to each block in the cache. When a new cache block is referenced, it overwrites the oldest one, its age is set to 0, and the age of other cache blocks in the line is increased. If the block was already in the cache, its age is set to 0 (refreshed), and all blocks younger than the referenced cache block get their age increased. No block is kicked out in the latter case. For the rest of this paper and for the sake of readability, the examples are given with a 2-way associative instruction cache ($A = 2$) although our method may be applied to any number of ways.

To handle the cache effects in the WCET computation, [13] has proposed an approach, called *Static Cache Simulation*, that assigns a category describing the cache behavior to each instruction. First applied to the direct-mapped caches, it is then generalized to set-associative caches in [12] and reused by various surveys ([9, 1, 14]). The instructions whose execution always result in a cache hit are categorized as *Always Hit*, while the instructions whose execution always result in a cache miss are categorized as *Always Miss*. There exists other categories like *Not Classified*, to describe the case where we can not predict the cache behavior for this instructions.

Always Hit, Always Miss and Not Classified categories are insufficient for computing a tight WCET estimation. As an example, one can figure an instruction in a loop whose cache block does not conflict with any other block of the loop. The first access to the instruction may result in a miss, but we can be sure that subsequent accesses will result in a hit. Various surveys have previously referred this behaviour as First Miss ([13], [12], [7]). There are several analyses based on the categorization which takes advantage of this behavior, we present them in the next few paragraphs.

2.2. The First-Iteration Loop Unrolling

The analysis proposed by Christian Ferdinand in [9] computes the categories by performing an abstract interpretation [4, 5] on a *Control Flow Graph* (CFG), composed of *Basic Blocks* (BB). *Abstract Cache States* (ACS) are computed before and after each basic block, which are in turn used to compute the categories. The ACS represent a set of concrete cache states that are possible during the execution, and are computed using two functions:

- the *Update* function computes the effects of a basic block on an input ACS,
- the *Join* function merges the input ACS of a basic block that has several predecessors in the CFG.

The Update and Join functions are applied repeatedly until the algorithm reaches a fix point (that is, until the application of any Update or Join function anywhere on the CFG does not cause any more changes).

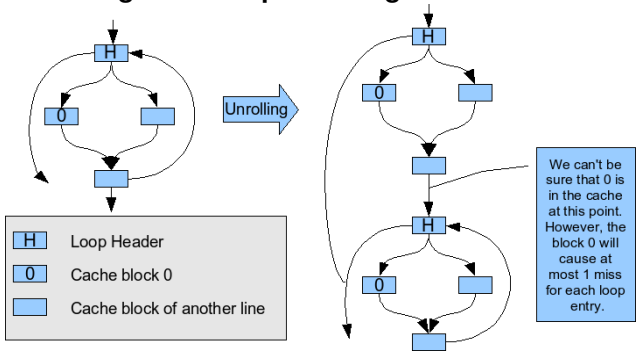
Two different analyses are done, each one using its own kind of ACS: the *May* and *Must* analyses. Each kind of ACS associates a set *s* of cache blocks (each one labeled by an age $a \in [0..A[$) to each cache line. In the May (resp. Must) analysis, the set *s* contains the blocks that may (resp. must) be in the cache, and the age *a* represent their youngest (resp. oldest) possible age. The set of all blocks of line l and age a that are in the abstract state before basic block B is noted $ACS^B_{may/must}(l,\, a)$. We also use the notation $ACS^B_{may/must}(l) = [x,\, y]$, where x represents the blocks of age 0, and y the blocks of age 1. The Update and Join functions for May and Must analysis are described in [9], and are not detailed here.

The ACS are then used to compute the categories as described in Table 1. For the sake of simplicity, we consider a Control Flow Graph (CFG) projected on the cache blocks (i.e. each basic block is split according to cache block

Table 1. Categorization rules for basic block B

Condition	Category
$\exists a / block_B \in ACS^B_{must}(line_B, a)$	*AH*
$\forall a / block_B \notin ACS^B_{may}(line_B, a)$	*AM*
otherwise	NC

Figure 1. Loop Unrolling Pessimism



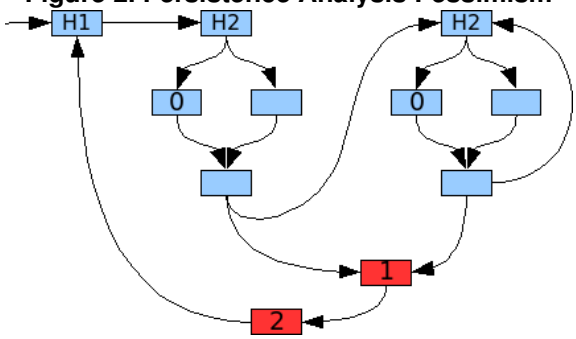
boundaries). Now, since a basic block of the projected CFG can not span multiple cache blocks, we can assign a single category to each basic block (only the first instruction of each block matters, because all the others instructions have been loaded because of the miss).

To take into account the behavior of blocks whose category depends on the loop iteration, this analysis unrolls the first iteration of all loops[1]. The May and Must analyses are performed for the unrolled version of the loops. Therefore, this allows a block in a loop to have its category computed in two contexts, the *"first iteration"* context (F context) and the *"other iterations"* context (O context).

This method has several drawbacks. One of them is illustrated in Figure 1, where only the blocks belonging to the currently analyzed line are numbered (as the analyses presented in this paper can be run independently line by line, we adopt this convention for the following figures). The block 0 cannot be identified as Always Hit in the O context because it belongs to a conditional path: we can assert that all the accesses to block 0 except the first one result in a hit. Yet, as the first access does not necessarily happen on the first iteration, the access to this block is categorized as Not Classified: the analysis misses a lot of hits, inducing an overestimation of the WCET.

[1]Although C. Ferdinand handles inter-procedural analysis by virtual inlining, for the sake of simplicity we consider that all the functions are already inlined, and that there is no recursivity. Also, while C. Ferdinand can handle the unrolling of any number of iterations, we will focus only on the "first iteration" case, relevant for the First Miss.

Figure 2. Persistence Analysis Pessimism



Another problem with this approach arises when the program contains a large number of nested loops. When many loops are nested, the unrolling has for effect to duplicate a basic block at depth n in 2^n contexts. As the ILP system in IPET requires several constraints and variable for each block, the unrolling can quickly lead to a very large number of constraints and variables which can lead to high computation time caused by the ILP solving process. The actual cache analysis (computation of categories) is quite fast, but the large number of generated constraints and variables make the overall analysis slow.

2.3. The Persistence Analysis

We describe in this paragraph an improved version of the cache behavior analysis proposed by C. Ferdinand in [8], addressing some of the limitations presented the previous paragraph. This method introduces a new analysis, the so-called *Persistence* analysis that is based on abstract interpretation and uses its own kind of ACS. The goal of this analysis is to find which cache blocks are persistent, that is, the cache blocks that can not be wiped out of the cache once they have been loaded. These blocks are guaranteed to cause at most one miss for the whole program. We give below a brief description of the Persistence analysis, but a more precise formulation can be found in [8].

Similarly to the May and Must ACS, the Persistence ACS associates a set s of cache blocks to each cache line, each block being labeled with an age a. The set s contains the blocks that may be in the cache, and the age $a \in [0..A[\cup l_\top$represents their oldest possible age (for the purpose of this comparison, $l_\top$ is considered to be older than A). The additional age $l_\top$ represents the blocks which may have been put in the cache, and subsequently wiped out. Similarly to the May and Must ACS, we use the notation $ACS^B_{pers}(l,\ a)$ to represent the set of all blocks in line l at age a. A new category *Persistent* is defined: a basic block B is categorized as Persistent if it is not an Always Hit, and if $\exists a \,/\, block_B \in ACS^B_{pers}(l,\ a) \ \wedge \ a \neq l_\top$.

The main drawback with this approach arises with nested

loops, and is illustrated in Figure 2.

This example shows two nested loops (with headers H1 and H2). Only the first iteration of the inner loop is unrolled. We see that the cache block 0 is persistent for the inner loop, but in conflict with the two darker blocks from the outer loop body so that the analysis does not find it persistent. In addition, the block 0 is in a conditional path, so it is not categorized Always Hit in the "other iterations" context of the loop $H2$. In this case, both the persistence analysis and the loop unrolling fails to take advantage of the fact that the block can not be wiped out within the inner loop.

This improved analysis has better (and, at least equivalent) precision than the previous one but suffers from the same problem: the large number of constraints due to the loop unrolling causing long computation time.

2.4. Ambiguous *First Miss* Terminology

In his paper [7], J. Engblom talks about the *First Miss* category, representing the fact that a cache block in a loop may cause a cache miss on the first iteration, and cache hits in the following iterations. He explains next that this approach may introduce some pessimism if we can not be sure that the cache block is referenced during the first iteration of the loop, and states that it would be nice to have a First Reference Miss category which means that the cache block may cause a miss for the first reference of the block (regardless of the iteration in which the reference occurs).

Actually, Engblom's First Miss handles the same problem as the loop unrolling, while the First Reference Miss is equivalent to the Persistent category. To avoid such ambiguity, we avoid using the term First Miss, but instead, categorize a block as Persistent / First Reference Miss or as First Iteration Miss handled, respectively, by the Persistence analysis and the loop unrolling.

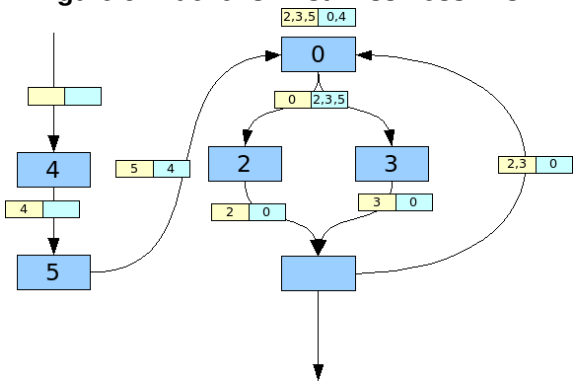
Additionally, the First Miss can be also interpreted in two ways:

1. The first reference / iteration is guaranteed to be a miss, and the other references / iterations are guaranteed to be hits,

2. We do not know anything about the first reference / iteration (Not Classified), and the other references / iterations are guaranteed to be hits.

The Persistence analyses discussed in this paper are corresponding to the latter case, that is, they cannot predict the behavior of the first reference of a First Miss block. Indeed, the conditions for categorizing a block as Persistent (the block is not Always Hit, and $\exists a \,/\, block_B \in ACS^B_{pers}(l,\, a) \,\wedge\, a \,\neq\, l_\top$) do not imply that the first reference is always a Miss. Moreover, it can be noted that a cache miss can lead to a shorter execution time in some cases. Therefore, depending on the used pipeline analysis method, a cache miss is not necessarily the scenario leading to the WCET ([6]).

Figure 3. Mueller's First-Miss Pessimism



2.5. F. Mueller's Static Cache Simulation

F. Mueller's static cache simulation was first introduced in [13] and was including a method to find persistent blocks (which are called First Miss by F. Mueller). In the paper [12], the method was extended to set-associative cache, and the First Miss category was improved: it is now parametrized, each First Miss block is associated with a loop. If the block B is categorized as First Miss with loop L, it means that L is the outer-most loop such that B can not be replaced from the cache within L, that is, L and its inner loops does not wipe out B. Therefore, the cache block cause at most one miss each time the loop L is entered.

F. Mueller's analysis computes various informations on the CFG, and the ACS is one of them. Unlike to C. Ferdinand's analysis where there is a different ACS for Must and May, there is only one type of ACS, which is similar to the May ACS: it associates an A-tuple of cache block sets to each cache line, that represents the possible presence of blocks in the cache during the program execution.

F. Mueller's approach for the First Miss is very interesting because it allows the highest possible precision for the Persistence informations. Indeed, by handling loops and by being actually a *persistence* analysis, it solves both of the problems presented in sections 2.2 and 2.3. However, the actual computation method used to perform the analysis leads to some pessimism. Indeed, to differentiate between First Miss and Always Miss categories, one needs to count the number of conflicting blocks belonging to the loop that are present in the ACS before the processed block and to compare that number to the cache associativity level. If the number is greater than the associativity level, the block is always categorized as Always Miss.

Figure 3 illustrates the pessimism induced by F.

Mueller's approach. The ACS computed by the analysis are shown on each edge. The block to categorize is 0, we note $ACS^0(l)$ its input ACS for the currently analysed line l. We can see easily that for a 2-way associative cache, it is persistent. However 2, 3, and 0 are in $ACS^0(l)$ and are conflicting. Additionally, they are in the loop. Consequently, since the number of concerned blocks (2, 3, and 0) is greater than the associativity level, the analysis fails to categorize block 0 as persistent, causing pessimism. According to our measurements, the ratio of misclassified blocks differs depending on the program, but varies from 2% to 11%.

3. Persistence Analysis Improvements

We have adopted Ferdinand's improved cache analysis presented in Section 2.3 as a starting point for the improvements presented in this paper. Indeed, this analysis is quite precise, because most of the times the persistence analysis and the loop unrolling compensate for each other limitations. However, it can be noted that:

- the loop unrolling causes constraint explosion;
- there is some cases unhandled by both the persistence analysis and the loop unrolling (as showed in the previous section).

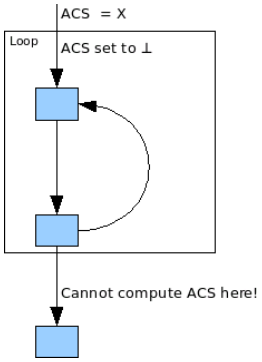
Actually, the loop unrolling is needed only because the persistence does not perform well with nested loops. This is why a better persistence analysis would solve both issues listed above. However, it should be noted that, because Ferdinand's improved analysis is quite precise, there is not much room for WCET tightness improvement: our main goal is to remove the loop unrolling in order to speed up the analysis.

In the next subsection, we present and evaluate improved versions of the persistence analysis in an attempt to solve these problems.

3.1. The Inner Persistence

To solve the problem of Figure 2, we need a persistence analysis that can focus on the inner loops, in case of loop nesting. We define the Inner Persistence to distinguish this new Persistence from the Ferdinand's persistence and we call the latter "Outer Persistence" from now on. A cache block is considered inner-persistent if once it has been loaded, it cannot be replaced within the body of the inner-most loop containing it. A way to compute this is to modify the analysis such that when it enters a loop, it sets the current Persistence ACS to $\perp$ (empty ACS). This modification does what we want, because usually, to determine the (outer) Persistence of a block B, we check that it exists no path going from B to B that replaces $block_B$. Setting

Figure 4. Inner Persistence Computation



Persistence ACS to $\perp$ when entering a loop enables us to forget replacements which occurs on paths not contained in the inner-most loop. However, none of the paths entirely contained in the inner-most loop are affected (because these paths do not pass through the loop entry, so the ACS is not set to $\perp$). Therefore, the analysis of the inner loop takes into account only the cache replacements within the inner loop. However, if we do that, there is a problem when the analysis exits the loop, as depicted in Figure 4: we "forgot" the ACS X during the process, and so we cannot update correctly the successors of the loop exit edges because we do not know what happened to the cache blocks of X during the loop analysis.

The problem can be overcomed by doing in parallel the analysis from X and $\perp$ while entering the loop (we can subsequently discard the latter results when exiting the loop). Thus, we can get the correct ACS at the loop exits. If we generalize this principle to n levels of nested loops, we need to process up to n analyses together. Furthermore, while this approach solves the problem presented in Figure 2, the inner-persistence property is weaker than the outer-persistence. Indeed, when a cache block is outer-persistent, it can cause at most one miss for the whole program. Yet, an inner-persistent cache block can cause a miss, at most, each time the inner loop is entered. Thus, this approach can cause added pessimism if a cache block is actually outer-persistent. However, because of our parallel analysis processing when we analyse a loop L, we also have the information concerning all loops containing L. As a result, we can know if a block is inner-persistent or outer-persistent. Actually, in the case of multiple nested loops, we can also pinpoint exactly the loop nesting level causing the conflict. We call this approach the Multi-Level Persistence analysis that is detailed in the next paragraph.

3.2. The Multi-Level Persistence

In this approach, the Persistent category is parametrized, that is, it is associated with a loop (this category is simi-

lar to the First Miss from [12], but our computation method is performed by abstract interpretation avoiding drawbacks showed in 2.5). We call a cache block *Persistent(L1)* if once the cache block has been loaded in the cache, it can not be replaced within the body of L1 and any loop contained in L1. A Persistent(L1) cache block causes at most one miss each time L1 is entered. This approach is more costly to compute but (1) this is balanced by getting rid of the loop unrolling and (2) it allows to get the best precision available. Actually, it can be noted that the Outer and Inner Persistence are a subset of the Multi-Level Persistence: if a cache block cb is Outer-Persistent (resp. Inner-Persistent), then it is Persistent(L), where L is the outer-most (resp. inner-most) loop containing cb. This ensures that the Multi-Level Persistence is never worse than the Outer and Inner Persistences (but it is usually better).

To compute the Multi-Level Persistence, we use an ACS which is a stack of standard Persistence ACS. For each basic block B, the Multi-Level Persistence ACS associates a stack item to each loop L containing B, that we note $ACS^B_{pers}[L]$. The stack item $ACS^B_{pers}[L]$ represents the Persistence information before the basic block B, taking into account only the cache block replacements that occurs within loop L.

The Update and Join functions are adapted to operate on the stack by updating / joining corresponding elements.

The new Update and Join functions are described by:

$$Update(ACS_{pers}, B) = ACS'_{pers} \,/\forall L$$
$$ACS'_{pers}[L] = Update_{outer}(ACS^B_{pers}[L])$$

$$Join(ACS1_{pers}, ACS2_{pers}) = ACS'_{pers} \,/\forall L$$
$$ACS'_{pers}[L] = Join_{outer}\begin{pmatrix} ACS1_{pers}[L], \\ ACS2_{pers}[L] \end{pmatrix}$$

The functions $Update_{outer}$ and $Join_{outer}$ represents, respectively, the Update and Join function for the Ferdinand's (Outer) Persistence.

The stack is initialized as the empty stack at the program entry point. When a loop is entered during the analysis (when we encounter a loop-entry edge), we create a new Persistence ACS (empty), and push it onto the stack. When we leave a loop, we pop the ACS from the stack and discard it. So, in other words, $ACS^B_{pers}[L]$ is the ACS before B computed while setting the ACS to $\perp$ when entering the loop L. This ensures that the stack item $ACS^B_{pers}[L]$ takes into account only the paths contained in L, for the same reason that was explained in the previous section about Inner Persistence.

To derive the Persistent category from the stacked ACS, we first define the function $IsPers$, such that $IsPers(B, L) = \exists a/block_B \in ACS^B_{pers}[L](l, a) \wedge a \neq l_\top$. This function tests if the basic block B is Persistent with respect to the loop L, and returns a boolean value.

Table 2. ILP system size comparison

Test	Unrolling	No Unrolling
adpcm	6587	3021
qurt	1187	920
statemate	6532	3423
compress	2056	1115
ns	964	192
bs	182	129
fft1	2450	1399
fft1k	1054	463
lms	1535	718
ludcmp	1156	481
minver	1854	702
select	744	282
matmult	973	388

The actual category computation is then done with the following rule:

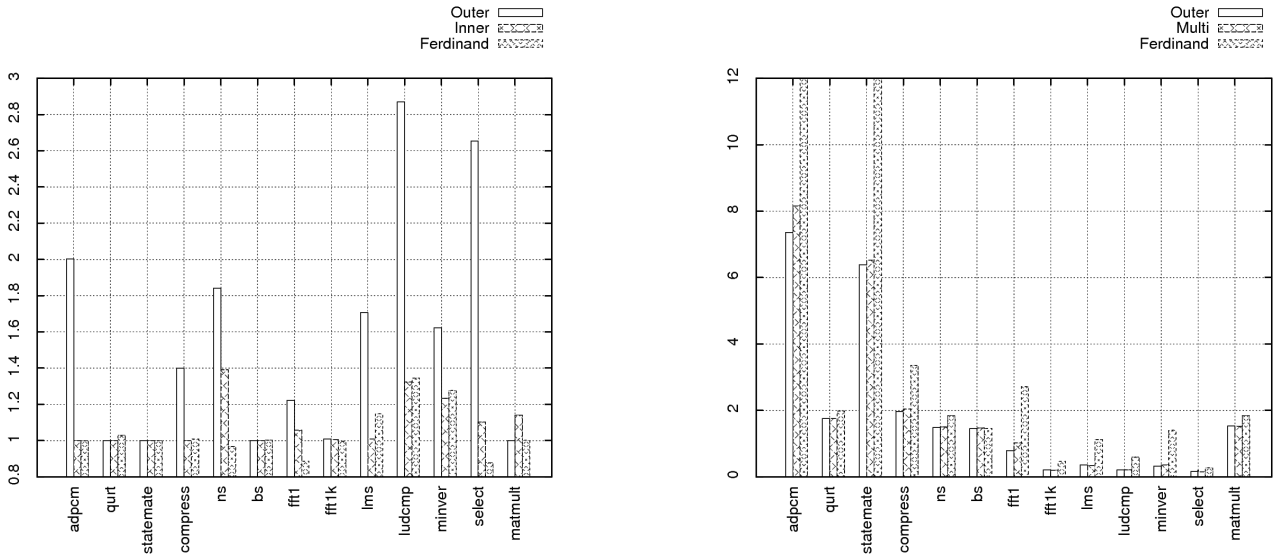
$$category_B = \begin{cases} Persistent(L) & if\ IsPers(B,\ L) \wedge \forall L'/ \\ & L \subset L' \rightarrow \neg IsPers(B, L') \\ Always\ Miss & otherwise \end{cases}$$

3.3. Comparison of the different Persistence versions

Even if the ideal candidate is surely the Multi-Level Persistence, for the sake of completeness, we have compared all of them on the criteria of WCET precision and computation time analysis to assert that its computation effort is worth enough. The three persistence methods were compared without loop unrolling, because we want to determine the intrinsic qualities of each persistence method, without the loop unrolling interfering with the results. The measurements have been done with OTAWA (our framework for WCET computation [3]) and with a subset of the big-enough Mälardelen benchmarks, programs widely used for WCET computation. The target architecture which the WCET computation was done for was a pipelined processor (the pipelining effects were taken into account by the *delta* method of [6]) with a 4-way associative instruction cache of 1024 bytes. This cache may seem a bit small but is adapted to the used benchmark in order to get enough cache usage conflicts.

The measurements are shown on Figure 5, and also include comparison with C. Ferdinand's Improved Analysis (Outer Persistence + Loop Unrolling). Figure 5(a) compares the WCET tightness of each approach. To do this, the Multi-Level Persistence is taken as a reference: all the measures represents the ratio $\frac{WCET}{WCET_{MultiLevel}}$. Figure 5(b)

Figure 5. Persistence Comparison



(a) WCET ratio comparison

(b) Time comparison

represents the analysis time of each method, including the ILP system solving time (the analysis time for the Inner persistence are not included, because we obtained them by a restriction of the Multi-Level analysis to the inner-most loop).

We first compare the three Persistence methods. As you can see on the diagram, the respective precision of the Outer and Inner variants depends on the particular test. Since the respective precision of the Outer and Inner variants involves the loop nesting level of the conflicts on the First Misses, the outer variant is better when the $\frac{cachesize}{programsize}$ ratio is higher : indeed, a bigger cache size means less conflicts, and so, more chances to have outer-persistent basic blocks, thus the difference in precision between the outer and multi-level variants decrease, and the pessimism of the inner variant increases. However, as it should have been expected, the result of the Multi-Level persistence approach is always equivalent or better than the Outer and Inner variants, because it always take into account the exact loop nesting level of the conflict. This is why we choose the Multi-Level Persistence as our new improved Persistence analysis.

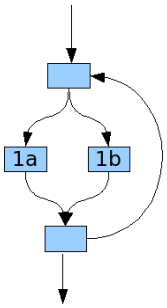
Now, we describe the comparison between the Multi-Level Persistence analysis (without Loop Unrolling) and C. Ferdinand's improved analysis. The analysis time comparison shows that the multi-level persistence analysis takes more time to compute (because there is more information to process), but is nonetheless faster than C. Ferdinand's improved analysis, because it does not suffer from the ILP constraints explosion problem. To evaluate this improvement, Table 2 shows the number of ILP constraints generated for each test and each method: the loop unrolling increases the number of constraints by 130.84 % on average.

On the other hand, the WCET comparison shows that the Multi-Level Persistence analysis is not as precise as C. Ferdinand's improved analysis. First, it may sound surprising, because the Multi-Level Persistence is supposed to take into account all the effects handled by the Loop Unrolling (nested loops with non-outer persistent blocks) and more (conditional paths within loops), and thereby making the Loop Unrolling analysis unnecessary. However, as shown by the results, this is not the case. Indeed, for some benchmarks, the WCET ratio for Ferdinand's analysis is less than 1, meaning a tighter WCET when using his approach. Therefore, this point need to be examined in order to fully benefit from the computation time gain.

4. Improvements of the Multi-Level Persistence

Since the Multi-Level Persistence analysis is more precise (or at worst as precise, for some cases) than the Persistence used by C. Ferdinand's improved analysis, it is clear that the inferiority of the Multi-Level Persistence discussed previously can only be explained by effects that are taken into account by the Loop Unrolling (but undetected by the Multi-Level Persistence). We have identified at least two main effects that are handled by the Loop Unrolling and not by the Multi-level Persistence that are discussed here.

Figure 6. Multi-Level persistence lack of precision



4.1. Non-Conflicting Blocks Analysis

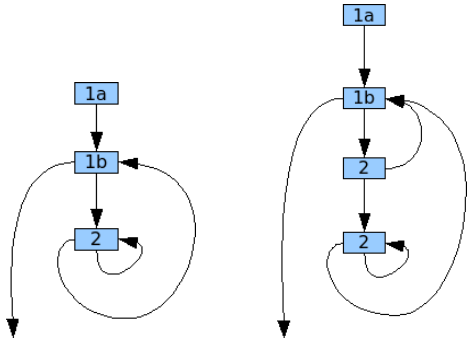
The first cause of pessimism is illustrated in Figure 6. In this schema, the basic blocks $1a$ and $1b$ are both in the same cache block 1.

As expected, thanks to the Loop Unrolling, basic blocks $1a$ and $1b$ are categorized as Not Classified in the F context (first iteration) and Always Hit in the O context (other iterations). Likewise, if no unrolling is performed but instead the Persistence analysis is used, we categorize basic blocks $1a$ and $1b$ as Persistent. So far, there are not any differences between the two approaches.

The difference lies in the fact that, in the case of the Loop Unrolling, we know that we get only one miss each time the loop is entered (because block $1a$ and block $1b$ belongs to the same cache block). However, the Persistence analysis lose this information, and so, 2 misses are predicted (one for basic block $1a$, and one for basic block $1b$), even though that is impossible because they are contained in the same cache block.

To solve this problem, we introduce a new Non-Conflicting Blocks Analysis, whose purpose is to identify sets of basic blocks that are categorized as Persistent with respect to the same loop, and which share the same cache block. This analysis is pretty straightforward: once the Persistence analysis is done, we only need to iterate over all Persistent basic blocks and regroup the blocks whose associated loop and cache block are the same. This improvement enables us to generates ILP constraints taking into account the effect described in Figure 6. For this particular example, an analysis with the original Multi-Level Persistence causes the generation of the constraints $x^1_{miss} \leq 1$ and $x^2_{miss} \leq 1$ (meaning that the basic block 1 does not cause more than one miss, and that the basic block 2 does not cause more than one miss but both may cause two misses). On the other hand, if the Non-Conflicting Blocks Analysis is taken into account, the only constraint generated is $x^1_{miss} + x^2_{miss} \leq 1$, meaning that basic blocks 1 and 2 together does not cause more than one miss.

Figure 7. Input edge merging problem



(a) Non-unrolled Version (b) Unrolled version

4.2. Pseudo Loop-Unrolling

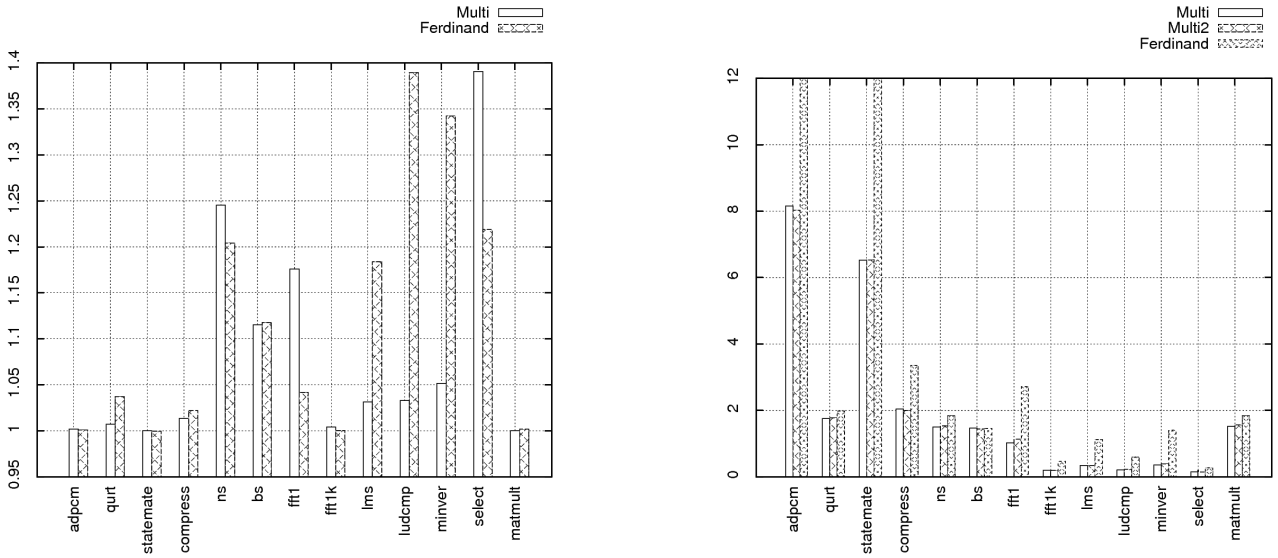
The second cause of pessimism can be explained by the figures 7(a) and 7(b). Both figures show the non-unrolled and unrolled versions of the same CFG. Again, basic blocks $1a$ and $1b$ share the same cache block. This particular example is not directly related to the Persistence because only the Must analysis is involved. However, its purpose is to show an example of pessimism introduced by removing the Loop Unrolling, and therefore, explains the pessimism that was measured in Section 3.3.

In this example, it is possible to determine intuitively, by looking at the schema, that block $1b$ is Always Hit. Indeed, there are two paths leading to $1b$, the path coming from $1a$ and the path passing by the loop back edge. On the first path $1b$ causes a hit because cache block 1 is already loaded while passing through $1a$. On the second path, since there is only one cache block (2) referenced in the loop and the cache is 2-way set-associative, the cache block 1 stays in the cache.

However, when doing the Must analysis without Loop Unrolling, $1b$ is not classified as Always Hit. When the analysis first passes through $1b$, it computes $ACS^{1b\rightarrow 2}_{must}(l) = [1, \emptyset]$. The update for block 2 produces $ACS^{2\rightarrow 2}_{must}(l) = [2, 1]$. Then, the Join between all the ACS before 2 computes $ACS^{2}_{must}(l) = [\emptyset, 1]$. The update of block 2 produces $ACS^{2\rightarrow 1b}_{must}(l) = ACS^{2\rightarrow 2}_{must}(l) = [2, \emptyset]$, which the cache block 1 is absent from and consequently 1 is absent from $ACS^{1b}_{must}(l) = Join(ACS^{2\rightarrow 1b}_{must}(l), ACS^{1a\rightarrow 1b}_{must}(l)) = [\emptyset, \emptyset]$. This prevents $1b$ from being classified as Always Hit.

We see that the problem comes from the merging of the two edges going to 2 (entry edge $1b \rightarrow 2$ and back edge $2 \rightarrow 2$). Since the result is $ACS^{2}_{must}(l) = Join([1, \emptyset], [2, 1]) = [\emptyset, 1]$, we determine that it is possible for the cache block 1 to be wiped while passing through block 2. However, if we consider the ACS before the Join, we see that it is not

Figure 8. Evaluation of the improved Multi-Level Persistence



(a) WCET ratio comparison

(b) Time comparison

possible (either cache block 1 is in the cache at age 0 and its age is simply increased, or cache block is at age 1 but is not aged because the referenced cache block, 2, is already in the cache). Yet, this information is lost while merging. On the other hand, if we unroll the inner loop, the entry edge and back edges goes to distinct instances of basic block 2, and thus, the ACS merging problem is avoided.

It would be nice to benefit from the Loop Unrolling while computing the ACS and avoiding the ILP constraint explosion problem caused by this approach. So, in order to achieve this goal, we perform loop unrolling (just like with Ferdinand's improved analysis) while computing the ACS for the various cache analyses. This enables us to work around the problem described above. Once the ACS has been computed, we return to the normal (non-unrolled) CFG and merge into a single ACS each set of ACS that was computed for a single basic blocks in different contexts. In our example, if we compute the ACS on the unrolled version shown in Figure 7(b), we get $ACS^{1a}_{must}(l) = [\emptyset, \emptyset]$, $ACS^{1b}_{must}(l) = [\emptyset, 1]$, $ACS^{2first}_{must}(l) = [1, \emptyset]$, and $ACS^{2others}_{must}(l) = [2, 1]$. Reverting to the non-unrolled version, we have to merge ACS^{2first}_{must} and $ACS^{2others}_{must}$, giving $ACS^{2}_{must}(l) = [\emptyset, 1]$. Since we have $ACS^{1b}_{must}(l) = [\emptyset, 1]$, the block $1b$ is categorized as Always Hit, showing that we have solved the problem successfully. It should be noted that unlike the Loop Unrolling, the Pseudo Unrolling does not suffer from the ILP constraint explosion problem. Indeed, although in both cases we compute the ACS on the unrolled version of the loops, in the Pseudo Unrolling method we merge ACS back together before categorization, and so the structural constraints created for IPET correspond to the non-unrolled version of the loops.

4.3. Evaluation

In this paragraph, we compare the performance and the precision of our improved version with our implementation of the C. Ferdinand's analysis including Persistence and Loop Unrolling analysis. Figure 8 shows the results of this comparison. The tests have been performed in the same conditions as in Paragraph 2.3.

The Figure 8(a) measures the WCET tightness of each method. For each test of the Mälardelen benchmarks, the diagram shows the WCET relative to the new improved Multi-Level Persistence, computed by the ratio $\frac{WCET}{WCET_{Multi2}}$ where $WCET_{Multi2}$ is the WCET obtained by the analysis descibed in this section. The Figure 8(b) measures the analysis time of each method. For each test, the total analysis time is shown including ILP system solving time.

In terms of precision, we see that the improved Multi-Level Persistence is better than both the older Multi-Level Persistence and the C. Ferdinand's analysis including persistence. This shows that we have successfully identified the problems that made the previous multi-level persistence analysis not precise enough, and thus, eliminated the need for full Loop Unrolling. On average, the WCET is improved by 6.68 % compared to the C. Ferdinand's analysis including Persistence and Loop Unrolling, and by 9.57 % compared to our older Multi-Level analysis.

For the computation time criteria, the improved multi-level persistence analysis is slightly slower than the previ-

ous one. The additional analyses (Non-Conflicting Blocks Analysis and the Pseudo unrolling) takes a little additional time (only 1.01 times slower in average). On few cases, the improved Multi-Level analysis is slightly faster than the old one, this can be attributed to measurement errors due to the host hardware architecture and operating system which were used to run the tests. Yet, it is still far faster than C. Ferdinand's improved analysis, because the time lost during our analysis is more than balanced by the fact that we do not have to resolve a huge ILP system. On average, the improved Multi-Level analysis is 2.70 times faster than C. Ferdinand's analysis including Persistence and Loop Unrolling.

5. Conclusion

This paper gives an overview of the different existing approaches for handling Persistent cache blocks, and clarifies the ambiguous First Miss naming. We have compared the existing approaches, identifying theirs drawbacks, and we have addressed the identified shortcomings of the cache analysis proposed by C. Ferdinand in [8] (lack of precision in the Persistence analysis, and ILP constraint number explosion caused by the Loop Unrolling) by developping our own improved version of the Persistence analysis (Multi-Level Persistence). As our approach computes Persistence information relative to each loop, the resulting loop-dependent Persistent categories eliminates the need for the Loop Unrolling. We have then refined our approach by developping the Non-Conflicting Blocks Analysis and the Pseudo Unrolling, that increase the tightness of the computed WCET, while keeping the analysis time reasonable. The experimentation results with the OTAWA tool shows that our method computes a tighter WCET than our implementation of C. Ferdinand's analysis. Moreover, the ILP system complexity is reduced, thus reducing greatly the analysis time (the ILP computation being the most time-consuming operation in the WCET computation).

Even if a lot of time is gained, the computation of the Multi-Level Persistence is still a bit costly. Therefore, in our future work, we will attempt to address this problem. Since the cost of the Multi-Level Persistence computation is caused by the fact that we may have to process many (as many times as the number of nested loops) ACS in parallel, a solution to reduce this cost may involve computing for each loop informations that describe the ageing of cache blocks within that loop. In case of loop nesting, it would enable us to compute easily the ACS associated with the outer loop, and thus, decrease greatly the cost of the Multi-Level Persistence computation. We are currently surveying methods to perform cache analyses on components to get partial cache behaviour, that are then instantiated when the component is called [2]. This approach gives interesting results in term of computation time and our First-Miss analysis fits well with this approach.

References

[1] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 52–66, London, UK, 1996. Springer-Verlag.

[2] C. Ballabriga, H. Cassé, and P. Sainrat. An improved approach for set-associative instruction cache partial analysis. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing*, March 2008.

[3] H. Cassé and P. Sainrat. OTAWA, a framework for experimenting WCET computations. In *3rd European Congress on Embedded Real-Time Software, Toulouse*, 25-27 December 2005.

[4] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

[6] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In IEEE Computer Society Press, editor, *Proc. of 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Dec. 1999.

[7] J. Engblom, A. Ermedahl, M. Sjodin, J. Gustafsson, and H. Hansson. Towards industry-strength worst case execution time analysis. In *Technical Report ASTEC 99/02, Advanced Software Technology Center (ASTEC)*, April 1999.

[8] C. Ferdinand. A fast and efficient cache persistence analysis. *Technical report, Universitat des Saarlandes*, Sept. 1997.

[9] C. Ferdinand, F. Martin, and R. Wilhelm. Applying compiler techniques to cache behavior prediction. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, pages 37–46, 1997.

[10] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems*, pages 88–98, 1995.

[11] Y.-T. S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modelling and path analysis for real-time software. *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 254–263, Dec. 1995.

[12] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, pages 209–239, May 2000.

[13] F. Mueller and D. Whalley. Fast instruction cache analysis via static cache simulation. *TR 94042, Dept. of CS, Florida State University*, April 1994.

[14] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179, 2000.