An improved approach for set-associative instruction cache partial analysis

C. Ballabriga
ballabri@irit.fr

H. Casse
casse@irit.fr

P. Sainrat[*]
sainrat@irit.fr

Université de Toulouse
Institut de Recherche en Informatique de Toulouse (IRIT)
CNRS - UPS - INP - UT1 - UTM
118 route de Narbonne 31062 Toulouse cedex 9

ABSTRACT

The current Worst Case Execution Time (WCET) computation methods are usually applied to whole programs, this may drive to scalability limitations as the program size becomes bigger. A solution could be to split programs into components that could support separated partial analyses to decrease the computation time. The componentization is also consistent with the more and more frequent use of Component Off The Shelf (COTS). Consequently, we need algorithms to perform analyses on component-wise applications. In this paper, we focus on the partial analysis of set-associative instruction caches, based on the categorization method described by M. Alt et al. We have first evaluated A. Rakib et al.'s approach to this problem and we have shown that, while correct, this approach can be greatly improved by a better estimation of the component effect on the cache. The version we have developed addresses the identified shortcomings and the experimentation results have been evaluated according to two criteria: (1) overestimation of the WCET and (2) computation time gain against the whole program analysis approach.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*WCET computation*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*; D.4.7 [**Operating Systems**]: Real-Time and Embedded systems

General Terms

partial cache analysis, WCET computation

[*]European Network of Excellence on High-Performance Embedded Architecture and Compilation

Keywords

partial static analysis, instruction cache, abstract interpretation, COTS

1. INTRODUCTION

All hard real-time systems are composed of tasks which 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)

To perform this timing analysis, we need to take into account miscellaneous hardware effects like the instruction cache, the main concern in this paper. Currently, the WCET computation methods are designed to be used on a whole program.

However, there is some drawbacks to this approach:

1. The analyses used for WCET computation usually run in exponential time with respect to the program size,
2. When the program to analyze depends on external components (e.g. libraries) whose sources are not available, the lack of information on the components may prevent WCET computation.

To address these problems, it is introduced in [2] a method to partially analyze a component, producing a partial result that may be instantiated at each component call point to produce actual analysis results. The approach presented in [2] is focused on the direct-mapped instruction cache analysis.

In the present paper, we extend this approach to the set-associative instruction cache. We evaluate its results by comparing its computation time to a traditional cache analysis, and by comparing its precision to a partial analysis approach presented by A. Rakib et al. in [13].

Section 2 shows related works. In section 3, we define the problem: we first present the whole cache analysis by M. Alt et al., and the partial analysis approach by A. Rakib et al. We then evaluate A. Rakib et al.'s approach. In section 4, we present our partial analysis approach, describing each improvement made. In section 5, we test our approach, and show the obtained results. We conclude in section 6.

2. RELATED WORKS

The WCET computation approach used in this paper, probably the most used one, is the Implicit Path Enumeration Technique (IPET) [9]. This technique assigns a variable to each basic block, representing the number of executions of this basic block on the path producing the WCET and builds a linear constraint system with theses variables to represent the program structure and the flow facts (such as loop bounds). The WCET is then expressed by an objective function depending on the variables and on the individual execution time of the basic blocks. An Integer Linear Programming solver is used to compute the result. Among studies on the instruction cache effects on WCET computation, the nearest to this work are given below.

In [11], F. Mueller defines a method to categorize instructions into three categories (*Always Hit*, *Always Miss*, and *First Miss*). This method is bound to direct-mapped caches, but is later generalized to set-associative caches in [10].

M. Alt et al. [1, 8, 7, 14] have proposed a method to compute the cache behaviour categories using abstract interpretation. They use three distinct analyses, the *Must*, *May*, and *Persistence* analyses. From the results of these analyses, they categorize the instructions as *Always Hit*, *Always Miss, Persistent,* or *Not Classified.* The *Persistent* category is equivalent to F. Mueller's *First Miss* category.

In [12], K. Patil et al. extend the method defined in [10, 11] to a program made up of multiple modules. First, a module-level analysis, consisting of four analyses for different possible contexts (scopes), is done independently for each module, resulting in temporary categories. Next, a compositional analysis is performed on the whole program to adjust the categories according to the call context. This approach is focused on the performance gain resulting from the partial analysis, but doesn't appear to be designed to handle COTS, since the compositional step needs to visit each component CFG. Moreover, this approach requires also more passes than the analyses presented in this paper, and is bound to direct-mapped instruction caches.

In [13], A. Rakib et al. describes a method to perform component-wise instruction-cache behavior prediction. It adapts M. Alt et al.'s analysis to an executable linked from multiple components by analyzing independently the object files (producing partial results), and composing the partial results.

Two main aspects of the problem are addressed:

- The absolute base address of the component is not known during the analysis, so the linker is instructed to put the component into a memory location that is equivalent (from the cache-behavior point of view) to the memory location that was used in the analysis.
- When a component calls a function in another component, the called function has an influence on the cache state, and on the ACS of the caller. To handle this problem, the paper defines, for each called component, a cache damage function which is discussed in detail in section 3.2.

3. PROBLEM DEFINITION

In this section, we first present the cache analysis proposed by M. Alt et al. [1, 8]. Then we present the concepts of partial analysis, and describe the approach proposed by A. Rakib et al. [13]. At last, we evaluate A. Rakib et al.'s method results and show the improvement that can be made.

3.1 Whole Cache Analysis

The cache is a fast and small memory used to speed up access to the main memory. Copies of main memory blocks are stored in the cache to allow a faster access by the CPU. 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 set-associative cache, each line contains A blocks (in this paper, A refers to the cache associativity level).

Consequently, a *replacement policy* is needed to determine where to put a newly loaded cache block. 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, then 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 then, all the blocks that were younger than the referenced cache block get their age increased. No block is kicked out in the latter case.

The partial cache analysis method presented in this paper is based on M. Alt et al.'s analysis [1, 8], which computes, by static analysis, a category describing a particular cache behavior for each instruction. The instructions whose execution always result in a cache hit (resp. miss) are categorized as *Always Hit* (AH) (resp. *Always Miss* - AM). A third category, *Persistent* (PS), exists for the instructions whose first reference may result in a cache miss, while the subsequent references result in cache hits. Other cases are set to *Not Classified* (NC).

The categories are determined 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 BB which are in turn used to compute the categories. The ACS represents a set of concrete cache states that are possible during the execution, and are computed using two functions:

- the *Update* function computes the output ACS of a basic block from its input ACS,
- the *Join* function merges the input ACS of a basic block that has several predecessors in the CFG, that is, there are several paths leading to this block.

The Update and Join functions are applied repeatedly until the algorithm reaches a fix point.

Three different analyses are done, each one using its own kind of ACS: the *May*, *Must*, and *Persistence* analyses. Each kind of ACS associates a set s of cache blocks (each one labeled with an age 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 $a \in [0..A[$, represents their youngest (resp. oldest) possible age. In the Persistence analysis, 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. The additional (oldest) virtual age l_T represents the blocks which may have been put into the cache, and subsequently wiped out. A cache block is persistent if it is in the Persistence ACS, and its age is not $l_\top$ (meaning that once the block is loaded, it can never be wiped out).

The set of all blocks of line l and age a that are in the abstract state at program point p is noted $ACS^p_{type}(l, a)$ (where *type* is one of *must*, *may*, or *pers*). We also use the notation $ACS^p_{type}(l) = [x, y]$, where x represents a block of age 0, and y a block of age 1, and $blocks(ACS^p_{type}(l))$ to refer to the set of all blocks in the ACS for line *l,* regardless of their age.

The ACS are then used to compute the categories, like described in table 1. For the sake of simplicity, we consider a CFG projected on the cache blocks (i.e. each basic block is split according to cache block boundaries). Now, since only the first instruction of each basic block matters (all the others results in hits), we can assign a single category to each basic block.

3.2 Partial Cache Analysis

[2] shows a method to perform partial analysis on the direct-mapped instruction cache. This partial analysis produces a partial result which is composed of a *transfer* and a *summary* function, which computes, respectively, the ACS at component exit (for Must, May, and Persistence), and the categories of the component basic blocks, given the component entry ACS.

These results can then be used for the composition, in two steps:

1. While doing the analysis of the main (caller) component, the transfer function is used to represent the effects of the callee component.

2. Once we have got the ACS before each call to the component, they are used with the summary function to build categories for the component basic blocks.

When considering the analysis of a program made up of multiple components, the notation ACS^p_{type} defined in the last section can become ambiguous. That's why we provide here some specific notation: If a basic block B belongs to a sub-component, ACS^B_{type} refers to the ACS before B computed by the analysis of the sub-component alone, regardless of the calling context (by taking an empty entry ACS). Likewise, ACS^{entry}_{type} and ACS^{exit}_{type} refer to the ACS at the sub-component entry and exit, resulting from an independent analysis. On the other hand, ACS^{before}_{type} (resp. ACS^{after}_{type}) refers to the ACS at the sub-component entry (resp. exit) resulting from the complete analysis, including the caller.

Table 1: Categorization rules for basic block B

Condition	Category
$block_B \in ACS^B_{must}(l,a)$	*Always Hit*
$block_B \in ACS^B_{pers}(l,a) \wedge a \neq l_\top$	*Persistent*
$block_B \notin ACS^B_{may}(l,a)$	*Always Miss*
otherwise	*Not Classified*

Figure 1: Update for ageing information (must analysis)

$$\forall l\, DMG' = Update_{must}(DMG, B)/DMG'(l) =$$
$$\begin{cases} DMG(l) & if\, line_B \neq l \\ DMG(l) & if\, line_B = l \wedge block_B \in ACS^B_{must}(l) \\ DMG(l) \oplus 1 & otherwise \end{cases}$$

Figure 2: Join for ageing information (must analysis)

$$\forall l\, DMG' = Join_{must}(DMG1, DMG2)\, /$$
$$DMG'(l) = MAX(DMG1(l), DMG2(l))$$

The method presented in the beginning of this subsection is bound to the direct mapped cache, but is based on an approach presented in [13] by A. Rakib et al., which works with the set-associative cache. This approach adapts M. Alt et al.'s cache analysis to a program made up of multiple components. In A. Rakib et al.'s analysis, the partial analysis of a component produces a cache damage function, which represents the effect of the component on the ACS. This function has the same purpose as the transfer function of [2]. There is not any mechanism in A. Rakib et al.'s analysis that fulfills the same role as the summary function of [2]. Additionally, A. Rakib et al.'s method does not include the persistence analysis.

The cache damage function depends on two items:

- The *cache ageing information* for each line *l*, that we note $DMG^{exit}_{may/must}(l)$, which is an integer in $[0..A]$ representing the upper (must) or lower (may) bound of the ageing number for line *l.* Its purpose is to determine which blocks that are present in the ACS before the call are preserved.

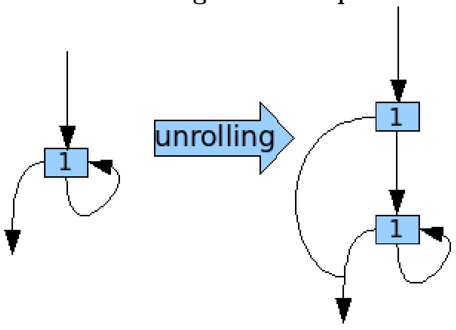
- The *inserted cache blocks,* that we note $ACS^{exit}_{may/must}(l)$, which corresponds to the cache blocks inserted into the ACS by the component, regardless of the calling context.

While the latter is computed like the traditional May and Must ACS, the former is computed with cache ageing Update and Join functions, described in figure 1, and 2. The operator $\oplus$ is defined, such that $x \oplus y = MAX(A, x + y)$. For the sake of simplicity, even though A. Rakib et al.'s

Figure 3: Application of cache damage (must analysis)

$$\forall l,\, \forall a \in [0..A[\, ACS^{after}_{must} =$$
$$Dam(ACS^{before}_{must}, ACS^{exit}_{must}, DMG^{exit}_{must})\, /ACS^{after}_{must}(l,a) =$$
$$\begin{cases} ACS^{exit}_{must}(l,a) & \forall a \in [0, j-1] \\ ACS^{before}_{must}(l, a-j) \backslash blocks(ACS^{exit}_{must}(l)) & \forall a \in [j, A-1] \\ where\, j = DMG^{exit}_{must}(l) \end{cases}$$

Figure 4: Loop Unrolling



Must damage analysis processes conjointly the cache ageing information and the Must ACS, in our paper, we primarily focus on the cache ageing analysis, and so, we assume that the Must ACS are already computed for the callee component. Also, we do not discuss the May analysis, since the presented approach is straight-forwardly transposed to the May analysis. For the rest of this paper, unless otherwise specified, it can be assumed that we are in the context of the Must analysis.

The ageing Update function implicitly takes as entry argument ACS^{B}_{must}, not shown in figure 1. Using both items described above, the cache damage function is applied to the Must ACS as in figure 3 (the ageing information composition is not shown and can be found in A. Rakib et al.'s paper). The general idea of this damage function is that we age each cache block of ACS^{before}_{must} by the integer damage, while we insert the cache blocks from ACS^{exit}_{must}.

3.3 Evaluation of the Partial Analysis

We have implemented our own version of A. Rakib et al.'s approach, and tested it on the SNU-RT benchmark programs, with OTAWA [3], a framework designed for WCET computation. The pipeline effects were implemented with J. Engblom et al.'s *delta* method [6]. Unless specified, we consider that we have a 2-way set-associative instruction cache for the rest of this paper.

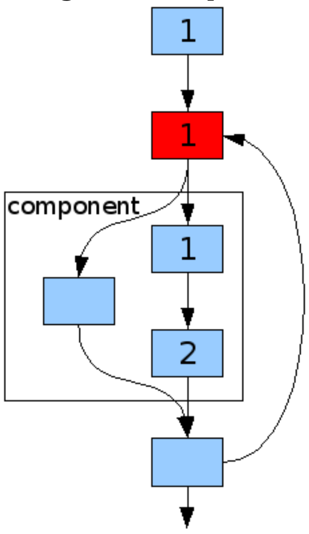
Table 2: A. Rakib et al.'s partial analysis results

Test	WCET	WCETpartial	pessimism(%)
crc	241660	262200	8.499
fft1	35715	40295	12.82
fft1k	525809745	526983875	0.2232
fir	147830	167240	13.12
lms	1970195	2169875	10.13
qurt	10440	11840	13.40

The results are presented in table 2. For each test, we have recorded the analysis time and the WCET for the partial and non-partial analysis (as done by M. Alt et al).

We have unrolled the first iteration of each loop in the case of the partial analysis because it compensates for the absence of the persistence analysis. As shown for the basic block 1 of figure 4, this basic block is Persistent (it cannot be wiped by any other block). This Persistent behavior is captured by the Loop Unrolling, where the basic block is duplicated into two contexts: the First-Iteration context, and the Other-Iterations context. The First-Iteration context

Figure 5: Damage function pessimism example



results in a miss as the block 1 is not already in the cache. Yet, after the execution of the First-Iteration context, the block 1 is stored in the cache and present in the MUST set. Therefore, the block 1 accesses in the Other-Iterations context result in hits. This mechanism mimics the behavior of the Persistence and makes the measures more fair but one may notice that both analyses are not fully equivalent.

From the statistics, we see that the partial analysis induces a significant (9.69% on average for all tested programs) additional overestimation of the WCET. In an attempt to reduce this overestimation, we have examined the tests that caused the worst pessimism, and explored the range of possible pessimism causes. So we have analyzed the cache ageing information effect on cache block *1* from the CFG of Figure 5: for the sake of simplicity, we ensured that, in these examples, $1 \notin ACS^{must}_{exit}(l)$, to make the status of block in $ACS^{after}_{exit}(l)$ dependent only on the cache ageing information.

The most obvious cause is that A. Rakib et al.'s damage does not differentiate between the different cache blocks in a cache line. The problem is illustrated in figure 5, where only blocks of the currently processed line are numbered. We see that the ageing of the considered line is 2. Since the ageing information is computed on a per line basis, all the blocks of the line are aged by 2 (and thus, wiped) without distinction of the individual block. This leads to pessimism. If we take the example of block 1, according to the computed cache ageing information, it is aged at most twice within the component. However, this is pessimistic: either the control flow passes through the left path, and the block 1 is not aged at all (since the block on the left path does not belong to the current cache line), or it passes through the right path, and the block 1 is aged once (by passing through the block 2). The worst case ageing for block 1 is really 1, but this can be detected only if we compute cache ageing information on a per cache block basis.

Other causes of pessimism in A. Rakib et al.'s analysis includes:

- the lack of persistence information,
- the absence of any equivalent to the summary func-

Figure 6: Update for cache ageing

$$\forall l, cb\, DMG' = Update(DMG, B)\, / DMG'(l, cb) =$$
$$\begin{cases} 0 & if\, (line_B, block_B) = (l, cb) \\ DMG(l, cb) \oplus 1 & if\, line_B = l \wedge block_B \notin ACS^B_{must}(l) \\ DMG(l, cb) & otherwise \end{cases}$$

Figure 7: Join for cache ageing

$$\forall l, cb\, DMG' = Join(DMG1, DMG2)\, /$$
$$DMG'(l, cb) = MAX(DMG1(l, cb), DMG2(l, cb))$$

tion (making the categories in the component context-insensitive).

4. OUR PARTIAL ANALYSIS APPROACH

The approach introduced in [2] uses a cache damage analysis based upon cache ageing and inserted blocks information. While generalizing this approach to set-associative caches, we have improved A. Rakib et al.'s cache ageing computation methods to reduce pessimism, while keeping the same principle for the cache damage function (figure 3). The *transfer function* defined in [2] is equivalent to the cache damage function, and for the sake of consistency we use the former term in this paper to talk about our approach.

We have also generalized the summary function presented in [2] to the set-associative cache, and completed the analysis by including persistence information. Since our new approach takes into account the calling-context of the components (thanks to the summary function), it will be referred to as Context-Sensitive Partial Cache Analysis (CSPCA) approach for the rest of this paper.

4.1 Improved Cache Damage

As stated in the previous section, a first cause of overestimation is a lack of precision in the cache ageing information. To reduce this pessimism, an integer is associated, in the cache ageing information, to each cache block instead of each line (we note $[x \rightarrow y, ...]$ the representation of an ageing information which associates integer y to cache block x). To take this modification into account, the corresponding Update and Join functions are modified as shown in figures 6 and 7.

This improvement solves the problem presented in figure 5, but figure 8 shows another case that still causes pessimism. We see that $ACS^{before}_{must}(l)$ = [3, 1] (l is the currently analyzed line), and $DMG^{exit}(l) = [1 \rightarrow 1, 2 \rightarrow 0]$ since cache block 1 is aged at most once (by the loading of the cache block 2). Therefore, the ACS^{after}_{must} found by ageing the $ACS^{before}_{must}(l)$ = [3, 1] is [3,]: 1 is wiped. This is pessimistic because, if block 1 is in the cache for sure at the component entry, one can deduce that block 1 is surely in the cache at the component exit. Indeed, if the left path is taken, block 1 is loaded, then aged only once, while on the right path it is not aged at all. In both cases, if we consider the real ACS^{after}_{must} instead of the one estimated by our partial result, we get $\exists a/1 \in ACS^{after}_{must}(l, a)$, a fact that is not detected by our current cache ageing analysis.

To overcome this limitation, we split the ageing information into two parts, noted DMG_{in} and DMG_{out}. Conse-

Figure 8: Damage function pessimism example 2

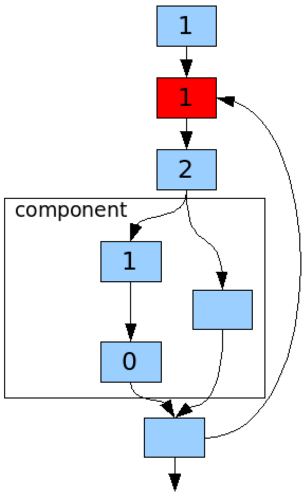


Figure 10: Join for both DMG_{in} and DMG_{out}

$$\forall l, cb\, DMG' = Join(DMG1, DMG2)\, /$$
$$DMG'_{in}(l, cb) = MAX_N(DMG1_{in}(l, cb), DMG2_{in}(l, cb))$$
$$DMG'_{out}(l, cb) = MAX_N(DMG2_{out}(l, cb), DMG2_{out}(l, cb))$$
$$with\, MAX_N = \lambda x \lambda y. \begin{cases} y & if\, x = N \\ x & if\, y = N \\ MAX(x, y) & otherwise \end{cases}$$

Figure 11: Improved transfer function

$$\forall l, a' \in [0..A[\, ACS' =$$
$$Transfer(ACS, DMG)\, / ACS'(l, a') =$$
$$\{cb/a' = MAX_N(a \oplus DMG_{out}(l, cb), DMG_{in}(l, cb)\}$$
$$with\, N \oplus x = N\, \forall x$$

Figure 9: Update for improved cache ageing

$$\forall l, cb\, DMG' = Update_{out}(DMG, B)\, /DMG'(l, cb) = \begin{cases} N & if\, (line_B, block_B) = (l, cb) \lor \\ & \quad DMG(l, cb) = N \\ DMG(l, cb) \oplus 1 & if\, line_B = l \land block_B \notin ACS^B_{must}(l) \\ DMG(l, cb) & otherwise \end{cases}$$

(a) Update for DMG_{out}

$$\forall l, cb\, DMG' = Update_{in}(DMG, B)\, /DMG'(l, cb) = \begin{cases} 0 & if\, (line_B, block_B) = (l, cb) \\ N & if\, DMG(l, cb) = N \\ DMG(l, cb) \oplus 1 & if\, line_B = l \land block_B \notin ACS^B_{must}(l) \\ DMG(l, cb) & otherwise \end{cases}$$

(b) Update for DMG_{in}

quently, for any cache block *cb*, there is now two integers ($DMG_{in}(l, cb)$ and $DMG_{out}(l, cb)$) representing the ageing of *cb* while passing through the component. $DMG_{in}(l, cb)$ and $DMG_{out}(l, cb)$ are produced respectively by the analysis of all paths containing a reference to a cache block *cb*, and by the analysis of all paths containing no reference to *cb*. For example in figure 8, $DMG^{exit}_{in}(l)(1) = 1$ and $DMG^{exit}_{out}(l)(1) = 0$, represent, respectively, the ageing of cache block 1 on the left and right paths. We can note that the information contained in DMG_{in} can be computed easily from the persistence, inducing a gain in computation time.

Figures 9, and 10 show the improved Update and Join functions. With this improved version of the analysis, the entry is such that $\forall l, cb : DMG_{in}(l, cb) = N \land DMG_{out}(l, cb) = 0$. The special value N represents the absence of any analysable path (for example, for the DMG_{out}, when all paths reference *cb*). Now it can be noted that to get ACS^{after} given ACS^{before}, only the cache ageing information has to be taken into account. It is no longer necessary to use the ACS^{exit}_{must} as described in 3: indeed, if the component always puts a block *cb* in the cache (regardless of the context), all paths in the component go through the block. In that case, $DMG^{exit}_{out}(cb) = N$, a condition that is checked by the new transfer function given in figure 11. Consequently, the transfer function depends only upon DMG^{exit}.

4.2 Partial Persistence Analysis

Our partial persistence analysis uses our improved must cache ageing information. There is no need for a Persistence-specific ageing analysis, because in the Persistence analysis, the blocks are aged similarly as in the must analysis. Therefore, to apply the cache damage function for the Persistence analysis, we proceed similarly as in the Must analysis. This way, we can add the persistence to our partial analysis without any other analysis.

4.3 Summary function

Finally, we have extended the summary function introduced in [2] to the set-associative cache.

The goal of the summary function is to determine categories for the component basic blocks from the component entry ACS. A summary function can be decomposed in a collection of Partial Categories, one for each basic block of the component. Each Partial Category is a function of signature $ACS_{must} \times ACS_{pers} \rightarrow CATEGORY$, that gives the basic block category in terms of the Must and Persistence entry ACS.

There is four kinds of partial categories:

- *Always Hit* (1) and *Always Miss* (2) partial categories are used whenever it is asserted that the basic block is, respectively, an always hit, or always miss, regardless of the component calling context.
- *Conditional Persistent* (3) is used if the basic block is at worst an Always Miss, but can be derived to Persistent at composition time, depending on the calling context. Although we are sure that it cannot be an Always Hit, this partial category depends on the calling context.
- *Conditional Always Hit* (4) category is used if we can not say anything about the basic block category at the time of the partial analysis. The category can be instantiated to Always Miss, Persistent, or Always Hit at composition time, depending on the calling context.

Table 3 gives the partial category construction process for each of the component basic blocks. B represents the examined basic block, and *cb* represents the cache block containing B. The left column contains the case number and the second one gives a condition to apply to the basic block. If the condition is true, the Partial Category of the basic block is given by the λ-expression given on the right.

We can note that the third condition includes the second one: thus, it could have been sufficient to have only three kinds of partial categories. However, this additional distinction enables us to detect the cases where an *Always Hit* is impossible, and so, decreases the cost of the summary function application.

5. EXPERIMENTATION

Table 4: WCET comparison

Test	WCET	WCETpartial	pessimism(%)
crc	241660	241700	0.0165
fft1	35715	36405	1.931
fft1k	525809745	525983905	0.0331
fir	147830	148440	0.4126
lms	1970195	1972205	0.1020
qurt	10440	10440	0

We have experimented our CSPCA approach (with the fully improved cache ageing analysis) in the same conditions as in section 3. As shown in [2], the partial analysis computation time is greatly reduced as long as the partially-analyzed function is called many times (including calls made in loops). These cases are well suited for any method of partial analysis (A. Rakib et al.'s, or CSPCA) and get the most benefits from the partial analysis. For each test, we compare the computation time and the precision with M. Alt et al.'s

Table 3: Summary function creation

Case	Condition	Summary: $\lambda ACS_{must}^{before} \lambda ACS_{pers}^{before}$.
(1)	$cb \in ACS_{must}^{B}(l)$	AH
(4)	$DMG_{out}(l, cb) < A$ $DMG_{in}(l, cb) < A$	$\begin{cases} AH\, if\, \exists a : cb \in ACS_{must}^{before}(l,a) \wedge (DMG_{out}(l,cb) + a < A) \\ FM\, if\, \exists a : cb \in ACS_{pers}^{before}(l,a) \wedge (DMG_{out}(l,cb) + a < A) \\ \quad \vee cb \notin ACS_{pers}^{before}(l) \\ AM\, otherwise \end{cases}$
(3)	$DMG_{in}(l, cb) < A$	$\begin{cases} FM\, if\, cb \notin ACS_{pers}^{before}(l) \\ AM\, otherwise \end{cases}$
(2)	*otherwise*	AM

Figure 12: Pessimism comparison

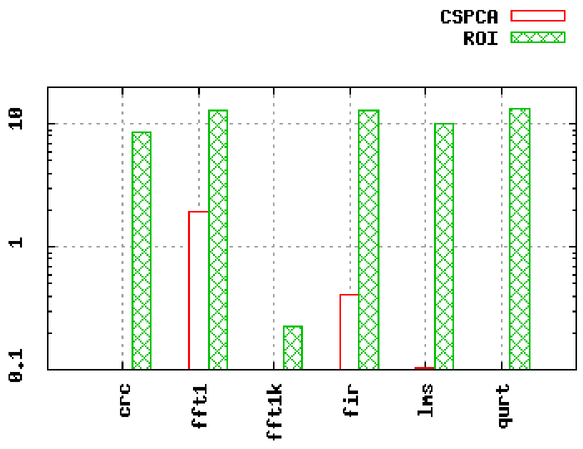
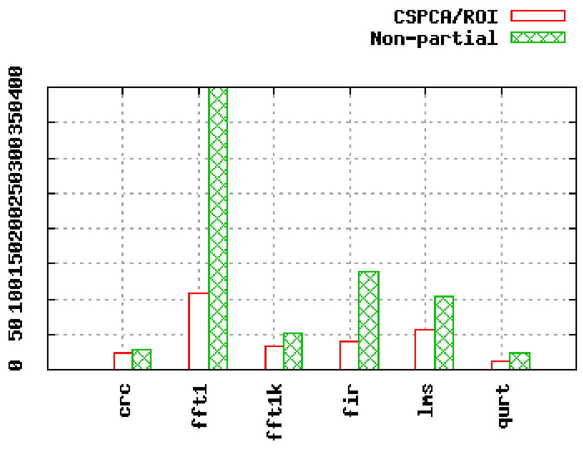


Figure 13: Analysis time comparison



non-partial analysis, A. Rakib et al.'s method (called ROI – Rakib et al.'s method with Our Implementation), and the CSPCA method.

Table 4 displays, for each test, the WCET found with a traditional analysis and with the CSPCA approach, along with the added pessimism (represented as a percentage of the non-partial analysis WCET). This table shows an average pessimism of 0.42% for the CSPCA approach, while the pessimism for the ROI approach is 9.7%.

Graph 12 represents graphically on the logarithmic Y-axis the added pessimism both CSPCA and ROI methods. This comparison shows that, for all tested programs, the CSPCA approach causes on average 23 times less pessimism. Graph 13 represents graphically on the linear Y-axis the computation time for the CSPCA approach, and for a traditional analysis (the computation time for CSPCA and ROI are similar). This graph shows a big speedup in computation time in favor of the partial analysis. The mean speedup ratio over our tested programs is 2.33. The speedup is greater when the components partially analysed are large enough and often called, as in fft1, which is quite frequent in real embedded applications.

6. CONCLUSION

In this paper, we have presented a complete method to perform partial analysis and composition of components for set-associative instruction caches. This approach provides two improvements: (1) the pessimism induced by the componentization is minimized and (2) the analysis time is reduced compared to a traditional analysis.

Our method enables us to partially analyse a component, producing a partial result which contains two items: a transfer function which returns the ACS after the component given the ACS at the input of the component, and a summary function which provides the categories of the component blocks given the ACS before the component. While analysing the calling program, whenever a call to the component is encountered we can use the partial results instead of fully analysing the component. Since the component is typically called a lot of time (taking into account loops) and because the time needed to use the partial result is negligible, analysis time is gained by having to analyse the component only once.

We have experimented our method, and compared it to a traditional, non-partial analysis on several benchmarks. The results show that, for the tested cases, the WCET produced using our partial analysis was correct, and the analysis time was greatly reduced. In addition, the comparison with A. Rakib et al.'s approach shows that our method introduces less pessimism. This is due to our improved cache ageing analysis, our support of the persistence, and our summary functions which allows us to have context-sensitive categories that can be instantiated differently for distinct component call places. Concerning this comparison, it can be noted that while A. Rakib et al.'s goal in [13] is to partially analyse whole object files, we are focused on a more fine-grained partial analysis (lots of smaller components of a few functions each). That's why we had to provide a method to greatly reduce pessimism.

In our future work, we will address the problem of embedded applications using COTS, where the lack of information about the component prevents WCET computation. To cope with COTS, a solution for maintainers would be to provide partial results that may be instantiated at WCET computation time.

To make this possible, one needs partial analyses for other hardware devices, such as the pipeline, branch prediction and for input-dependent flow fact information. Our work is now oriented to refine the concepts described in this paper and to find more general methods for partial analyses of components in the whole WCET computation.

7. REFERENCES

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

[2] Ballabriga, C., Cassé, H., and Sainrat, P. WCET computation on software components by partial static analysis. In *Junior Researcher Workshop on Real-Time Computing, Nancy, 29/03/2007-30/03/2007* (http://www.loria.fr, March 2007), LORIA, pp. 15–18.

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

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

[5] Cousot, P., and Cousot, R. 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* (Los Angeles, California, 1977), pp. 238–252.

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

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

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

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

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

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

[12] Patil, K., Seth, K., and Mueller, F. Compositional static instruction cache simulation. *SIGPLAN Not. 39*, 7 (2004), 136–145.

[13] Rakib, A., Parshin, O., Thesing, S., and Wilhelm, R. Component-wise instruction-cache behavior prediction. In *Automated Technology for Verification and Analysis* (2004), pp. 211–229.

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