WCET Computation on Software Components by Partial Static Analysis

*C. Ballabriga, H. Cassé, P. Sainrat**

{ballabri, casse, sainrat}@irit.fr

IRIT - Université de Toulouse – France

* HiPEAC European Network of Excellence

Abstract: *The current Worst Case Execution Time (WCET) computation methods are designed to be used on a whole program. This approach has scalability limitations and can not be applied to programs made up of multiple components: a method to perform partial analysis on components is needed. In this paper, we present a general method to perform partial analysis on components and to compose these partial results to compute the overall WCET. To check the approach, we have implemented and experimented the partial analysis on the behavior prediction of direct-mapped instruction cache. The experimentation shows big speedup in analysis computation time with an almost negligible growth of pessimism.*

Keywords: Real-time, instruction cache analysis, WCET, partial static analysis, components, abstract interpretation.

1. Introduction

Hard real-time systems are composed of tasks that must imperatively meet deadlines constraints. To check this, scheduling analysis, based on tasks WCET, is used. Computation of the WCET by static analysis provides proven WCET estimation. This approach may be decomposed in two phases:

- the control flow of the task (program path analysis),
- the hardware which the task will run on (architecture model)

Each phase requires a bunch of computation to handle accurately the execution flow of the program and the hardware model. As real-time programs becomes bigger and bigger, two problems arises: (1) WCET computation time will increase exponentially, (2) the use of software components will make ineffective the current techniques.

In this paper, we propose to perform partial analyses on the program (1) to reduce the computation time by factorizing the performed analyses and (2) to provide functions summarizing the content of components for the WCET computation. We apply the approach to the case of direct-mapped instructions caches.

The next section explores in details the partial analysis problem that is applied in the third section to direct-mapped instruction cache. In the fourth section, we show experimentation of our method. Fifth section presents the related works and we conclude in the last section.

2. Problem definition

Currently, the WCET computation methods are designed to be used on a whole program. However, there is some drawbacks to this approach. First, the analyses used for WCET computation usually run in exponential time with respect to program size, and embedded real-time programs are getting bigger and bigger. Second, there is a problem when the program to analyze depends on external components developed by third-parties (for example, programs using libraries) whose sources are not available: it is not possible to compute the WCET of the whole program because of lack of information on the components. This problem will become more and more important as embedded and real-time industry will use more and more Component Off The Shelf (COTS).

Both problems suggest to change the current analysis practice: it is no longer possible to do the analysis all over the program. We need to find a way to do partial WCET analysis on parts of the program, and to compose the partial results into a global WCET for the whole program.

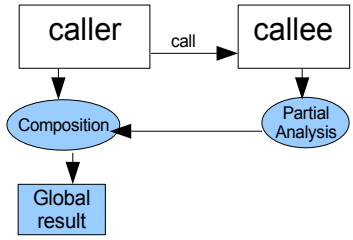

Figure 1: overview of the partial analysis

Figure 1 represents the partial analysis and composition of a program consisting of a function calling another function located in an external component. The partial analyzer takes the code of `callee` and produces a partial result. The composer takes the partial result, and the code of `caller`, to produce a global result, without accessing `callee` code.

In this paper we will consider the case described by the figure: a main program using an external component consisting of only one function. In the case of the instruction cache analysis, more complex cases can be processed similarly, and will be addressed in future work.

We need to define, for each analysis participating in the WCET computation, information needed in the partial results, the method to produce this data, and the method for integrating the partial results into the analysis of the whole program.

There are two main issues when considering the partial analysis:

(1) how to influence the analysis of the main program with the content of the component?
(2) how to make the analysis result of the component dependent on its call context?

To address (1), we describe a *transfer* function, which represents the effect of the component code on the analysis of the whole program. To address (2), we describe a *summary* function, which represents the analysis results for the component, according to the calling context (state before the function call). The method to compute these functions is specific to the particular analysis we want to do.

Once the transfer and summary functions associated with the component are available, we integrate them into the analysis of the whole program. This is done in two steps:

- Step 1: with the transfer function available, we can do the analysis on the main program without having to access to the component code. During the analysis, the component is represented by the transfer function.
- Step 2: once the analysis on the main program is done, we have access to the calling context of the component.

We can use this context, together with the summary function, to get the actual analysis results for the component.

Although we have not experimented nested function call, this approach may be easily adapted. First, the transfer functions may composed in a straight forward way. Yet, some additional computations need to be performed for the summary function to get a sound context before each nested function call.

3. Instruction cache analysis

This section shows how to apply the general approach described previously to the instruction cache analysis. The cache is a fast and small memory used to store a partial copy of the main memory, which speeds up data accesses. 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* of the same size than cache lines. Each cache block from memory matches a single line. In case of miss, the whole cache block containing the target location is loaded into the matching line. In the direct-mapped cache, each line contains only one block, while in the A-way associative cache, each line contains A blocks.

3.1. Presentation of the analysis of a single component

To predict instruction cache behavior, we have used the 3 analyses (*Must*, *May*, and *Persistence*) from [2, 8], restricted to direct-mapped cache, by considering that a direct-mapped cache is a A-way associative cache where A = 1.

These analyses are based on the techniques of abstract interpretation [5], and are performed on a Control Flow Graph (CFG), composed of Basic Blocks (bb). They work by computing *abstract cache states* (ACS) before and after each basic block using two functions:

- the Update function computes the output ACS from the input ACS of a basic block, that is, the effect of the basic block on the ACS;
- the Join function merges the input ACS of a basic block that has several predecessors in the CFG.

We call $ACS^{\text{in}}_{may}(bb)$, $ACS^{\text{in}}_{must}(bb)$ and $ACS^{\text{in}}_{pers}(bb)$ the input ACS for, respectively, the *May*, *Must* and *Persistence* analyses. The syntax $ACS(bb)(l)$ allows to get the content of a specific line of the ACS.

The *May* ACS gives the set of blocks which may be in the cache. The *Must* ACS gives the set of blocks which must definitely be in the cache. The *Persistence* ACS maps two sets ($ACS^{\text{in}}_{pers,0}(bb)(l)$ and $ACS^{\text{in}}_{pers,1}(bb)(l)$) to each cache line. Both sets contains blocks that may have been loaded in the cache. While the blocks of the latter set may have been wiped out, the blocks of the former must be still in the cache.

The ACS resulting from the analyses will be used to determine basic block categories. 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 a basic block of the projected CFG cannot span multiple cache blocks, we can assign a single category to each basic block.

Let *cb* be the cache block containing the basic block *bb*, and *l* its cache line. The following array shows the process to assign a category to each basic block:

Condition to test	*Category*
$cb \in ACS^{\text{in}}_{must}(bb)(l)$	Always-Hit (AH)
$cb \in ACS^{\text{in}}_{may}(bb)(l) \wedge cb \notin ACS^{\text{in}}_{pers,1}(bb)(l)$	First-Miss (FM)
$cb \notin ACS^{\text{in}}_{may}(bb)(l)$	Always-Miss (AM)
else	Non-Classified (NC)[1]

Table 1: Computation of the category

3.2. The transfer function

The transfer function describes the effect of `callee` on the analysis of the main program. In the case of the cache analysis, the `callee` transfer function takes the ACS from the caller, and computes the ACS at `callee` exit. We call ACS^{entry}_{xxx} and ACS^{exit}_{xxx} the ACS at `callee` entry and exit (where *xxx* stands for one of *may*, *must*, or *pers*).

Independently of the particular cache analysis, there are two effects that must be taken into account to build the transfer function. First, cache blocks of `callee` may appear in the ACS at `callee` exit, independently of the ACS at `callee` entry. This effect is handled by executing the analysis on `callee` with an empty entry ACS, and the result is retrieved from ACS^{exit}_{xxx}.

Second, some cache blocks that were present in the ACS at `callee` entry may be wiped out by `callee`. This effect is handled by the *damage update* function. This function, of type $ACS \rightarrow ACS$, takes an ACS, and returns the *damaged* ACS, according to the cache blocks which may have been replaced by `callee` cache blocks. It is different from the transfer function: the damage update function does not take into account the cache blocks belonging to `callee` which appear on `callee` exit ACS.

Whatever the performed analysis, the transfer function can be expressed by the following formula:

$$transfer(ACS) = Join(DamageUpdate(ACS), ACS^{exit}_{xxx})$$

In the following, we will see how to apply this general method to the transfer functions of the *May*, *Must*, and *Persistence* analyses.

In the case of the *May* analysis, the damage update is deduced from a cache damage analysis. This analysis builds damage information that represents, for each line, the list of cache blocks which would definitely be replaced if they were present in the cache at the beginning of `callee`. For example, if $cb \in damage^{\text{in}}_{may}(bb)$, all the paths from `callee` entry to basic block *bb* remove *cb* from the cache.

We call $line_{cb}$ the cache line of the cache block *cb*. The Update function of this analysis is defined as follows:

$$\forall l \neq line_{cb}, Update^{damage}_{may}(damage, cb)(l) = damage(l)$$
$$Update^{damage}_{may}(damage, cb)(line_{cb}) = damage(line_{cb}) \cup \{cb' / line_{cb} = l\} - \{cb\}$$

The Join function is defined like this:

$$Join^{damage}_{may}(damage1, damage2) = damage' \;/\; \forall l, damage'(l) = damage1(l) \cap damage2(l)$$

Let call $damage^{\text{exit}}_{may}$ the damage information at `callee` exit. Using the result of the analysis, the following damage update is defined:

$$\forall l, DamageUpdate_{may}(ACS)(l) = ACS(l) - damage^{exit}_{may}(l)$$

The transfer function is then created as explained previously.

The *Must* transfer function works in the same way as the

1 We have conservatively considered NC category as an Always-Miss.

May transfer function, by doing a *Must* version of the cache damage analysis, and of the damage update function. The only difference is that, in the $Join_{must}^{damage}$ function, a union of damages is done, instead of the intersection.

The transfer function for the *Persistence* analysis is built a bit differently. It uses the results from the *Must* and *May* damage analysis, and is defined below:

$$DamageUpdate_{pers}(ACS_{pers}) = ACS'_{pers} \;/\; \forall l$$
$$ACS'_{pers,0}(l) = ACS_{pers,0}(l) - damage_{must}^{exit}(l);$$
$$ACS'_{pers,1}(l) = ACS_{pers,1}(l) \cup (ACS_{pers,0}(l) \cap damage_{must}^{exit}(l))$$

3.3. The summary function

The summary function will give a category to each basic block, according to the entry states of `callee`. This function is defined on $ACS_{must} \times ACS_{pers} \rightarrow CATEGORY$ and is associated to each basic block of `callee`.

The results from the *Must* and *Persistence* analysis are required to create the summary function.

Let *cb* be the container cache block of the basic block *bb*, and *l* its cache line. The following table represents the different possible types of summary functions of the basic block *bb*:

Condition to test	summary function
$cb \in ACS_{must}^{\text{in}}(bb)(l)$	$\lambda\, ACS_{must}^{entry}\, ACS_{pers}^{entry}\, .\, AH$
$cb \notin damage_{must}^{\text{in}}(bb)(l)$	$\lambda\, ACS_{must}^{entry}\, ACS_{pers}^{entry}\, .\, if\ (cb \in ACS_{must}^{entry}(bb)(l))$ $then\ AH\ else\ if\ (cb \notin ACS_{pers,1}^{entry}(l))$ $then\ FM\ else\ AM$
$cb \notin ACS_{pers,1}^{\text{in}}(bb)(l)$	$\lambda\, ACS_{must}^{entry}\, ACS_{pers}^{entry}\, .\, if$ $(cb \notin ACS_{pers,0..1}^{entry}(bb)(l))\, then\ FM\ else\ AM$
else	$\lambda\, acs_{must}^{ENTRY}\, acs_{pers}^{ENTRY}\, .\, AM$

Table 2: Computation of the summary function

3.4. Composition of the partial analysis

We explain now how to use the partial analysis results (the transfer functions, and the summary function) into the analysis of the whole program.

Since the base address of the component file containing `callee` is unknown when the partial analysis is performed, we assume that the base address is 0. If we force the linker to allocate the component object file at a base address multiple of the cache block size, we keep the same structure for the mapping of the component cache blocks into lines: if we know the matching line of a location in the original mapping, we can find the new cache line with a simple addition, modulo the number of cache lines.

Therefore, using this simple transformation, we can adapt the transfer and summary functions for `callee`, and use them for the composition in the two steps described in section 3:

1. When the *May*, *Must*, and *Persistence* analyses on `caller` are performed, `callee` CFG is replaced by a *virtual node,* whose Update function is the Transfer function.
2. With the analyses from step 1 performed, we have the ACS at `callee` entry, which the summary function can be applied to, in order to determine `callee` basic block categories.

4. Experimentation

This section describes the environment of experimentation and presents the obtained results.

4.1. Mode of operation

Our method has been implemented using OTAWA [7], a framework dedicated to the development of static analyses for WCET computation. We have performed experimentation on the SNU-RT[2] benchmarks, a set of small programs widely used to test WCET computation and have been focused on the benchmark program containing function calls.

The partial analysis of each function induces a computation overhead due to the summary and transfer functions computation. Yet, this drawback may be balanced by the time gain at composition time if the program performs enough partially-analyzed function calls. To evaluate this issue, we have done 7 tests with different numbers of calls of the partially analyzed function, and different loop nesting depths of these calls.

For each test, we have compared the computation time and the obtained WCET between the partial and the non-partial analyses.

The architecture that the WCET computation is applied to is a simple non-pipelined processor with each instruction taking 5 clock cycles (since the goal of this paper was to show results for the cache analysis, we did not include other effects, like the pipeline), and with a direct-mapped instruction cache of 8 lines with 8-bytes blocks. Such a small cache allows exhibiting more block conflicts with the small programs of the SNU-RT benchmark.

4.2. Results

There are two interesting aspects to examine in the results:

- The comparison between the computed WCET on both analyses, to show that the partial analysis does not add too much pessimism.
- The comparison of the analysis time for both analyses, to show in which cases the partial analysis is faster.

The following tables show the relevant information (P.A. and N.A. stands, respectively, for Partial and Non-partial Analyses):

Bench	Function	N.A. (cycles)	P.A. (cycles)	P.A. / N.A.
fibcall	fib	1100	1100	1
matmul	matmul	18010	18010	1
fft1 (1)	fft1	33950	34420	1,01
fft1 (2)	sin	33950	33950	1
qurt	qurt	19105	19645	1,03
fir (1)	sin	219020	219020	1
fir (2)	gaussian	219020	219020	1

Table 3: WCET comparison

Bench	Function	P.A. / N.A.	Call sites	Max loop depth
fibcall	fib	2	1	0
matmul	matmul	2,25	1	0
fft1 (1)	fft1	1,01	2	0
fft1 (2)	sin	0,41	2	2
qurt	qurt	0,67	3	0
fir (1)	sin	0,75	2	1
fir (2)	gaussian	0,66	2	1

Table 4: call context sensitivity

The table 3 compares the computed WCET, while the table 4 shows the computation time ratio for each test in function of the call context.

The WCET results are coherent: the WCET found by the partial analysis is always equal or slightly greater than the non-partial analysis WCET. Furthermore, only few pessimism is added: the partial analysis adds on average a pessimism of 1% (P.A. / S.A. column)

2 Singapour National University - Real-Time

For the *fibcall* and *matmul* tests, the diagram shows that the partial analysis is slower. This can be explained because, for these tests, the function which was partially analyzed is called only once, so the partial result is only used once: the overhead for computing the partial result is not compensated.

For the *fft1 (1)* test, the function fft1 is called exactly two times, and the overhead is almost exactly compensated.

For the *qurt, fft1 (2)*, *fir (1)*, and *fir (2)* tests with a high number of calls to the function, the partial analysis is faster than the non-partial analysis. As shown by the *fft1 (2)* test, the loop nesting level of the call site has a great impact.

To summarize, although the computation of the transfer and the summary functions creates an overhead compared to a single non-partial analysis, the transfer function is much more fast to apply and the summary function is used only once for each call site. As shown in the experimentation, this overhead is compensated as soon as the partial analysis is applied twice. Such a situation arises very often in programs as (1) functions are usually defined to be called several times, and (2) in case of loops, the fix-point computation of static analyses requires at least two iterations.

5. Related work

The effect of the instruction caches on WCET computation has been extensively studied.

A widely used method for WCET computation is the Implicit Path Enumeration Technique (IPET) [6]. 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.

In [1], F. Mueller defines a method to categorize instructions into three categories (Always-Hit, Always-Miss, and First-Miss). These categories describe the worst-case instruction cache behavior, and are included in the WCET computation. While Always-Miss and Always-Hit are self-explaining, First-Miss means that the first execution of an instruction in a loop may result in a miss, but subsequent executions will result in a hit.

C. Ferdinand [2, 8] describes a method to compute the categories using abstract interpretation. He uses three analyses that consist in computing an Abstract Cache State (ACS) for each basic block as explained in the section 3.1. The ACS resulting from this analysis are used to build the categories for each instruction memory access.

In [4], F. Mueller extends the method defined in [1] 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 results in a merged context-insensitive category causing a lot of pessimism. Moreover, this approach requires also more analyses passes than our analysis that is performed in one pass followed by the fast instantiation of our functional categories.

In [3], a method is proposed to perform component-wise instruction cache behavior prediction. It adapts Ferdinand's cache analysis [2] (bound to *May* and *Must* analyses) to an executable linked from multiple components. This method addresses two main aspects of the problem.

First, the absolute base address of the component is unknown during the analysis, so it works with relative addresses (i.e. assuming that the base address is 0). To overcome this issue, it is proposed to force the linker to put the component into a memory location that is multiple of the cache size. This method ensures that the cache mapping is the same, whether the base address is absolute or relative. As shown in 3.4, we alleviate this constraint as we only require a cache block size alignment.

Next, 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 function, a "cache damage update" function that tells which cache lines are replaced (and how many times for associative caches) by the function. In this paper, we have augmented this analysis, as we also compute the persistence information, but we have also improved the accuracy of the results thanks to a context-sensitive definition of the categories.

6. Conclusion

We have presented a method to perform partial analysis on the instruction cache of a function in a component, and to compose the obtained partial result to compute the WCET of the whole program. Then, we have compared our method to the non-partial analysis on several benchmarks. The results show that, for the tested cases, the WCET produced using our method induced a very small pessimism.

Also, it seems that if the function being partially analyzed is called at least twice, or is called in a loop, the time needed to do the partial analysis and the composition is lower than the time needed to do the non-partial analysis. This means that the partial analysis is useful to speed up the analysis of large programs containing functions called many times.

Our future work will include the adaptation of this method to the A-way associative cache, and to other types of analyses for WCET computation (pipeline effects, etc...). Our goal is to apply partial analysis to the overall WCET computation.

7. References

[1] F. Mueller, *Fast instruction cache analysis by static cache simulation*, Journal of Real-Time Systems, 2000.

[2] M. Alt, C. Ferdinand, F. Martin and R. Wilhelm, *Cache behavior prediction by abstract interpretation*, Proceedings of Static Analysis Symposium, 1996.

[3] A. Rakib, O. Parshin, S. Thesing and R. Wilhelm, *Component-wise instruction-cache behavior prediction*, Proceedings of 2nd International Symposium on Automated Technology for Verification and Analysis, 2004.

[4] K. Patil, K. Seth and F. Mueller, *Compositional static instruction cache simulation*, ACM SIGPLAN Notices, 2004.

[5] P. Cousot and R. Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, Proceedings of 4th ACM Symposium on Principles of Programming Languages, 1977.

[6] Y. T. Li and S. Malik, *Performance analysis of embedded software using implicit path enumeration*, Proceedings of the 32nd ACM/IEEE conference on Design automation, 1995.

[7] H. Cassé and P. Sainrat, *OTAWA, a Framework for Experimenting WCET Computations*, 3rd European Congress on Embedded Real-Time Software, 2006.

[8] H. Theiling, C. Ferdinand and R. Wilhelm, *Fast and precise WCET prediction by separate cache and path analyses*, Journal of Real-Time Systems, 2000.