# A GENERIC FRAMEWORK FOR BLACKBOX COMPONENTS IN WCET COMPUTATION

C. Ballabriga, H. Cassé, M. De Michiel[1]

***Abstract***

*Validation of embedded hard real-time systems requires the computation of the Worst Case Execution Time (WCET). Although these systems make more and more use of Components Off The Shelf (COTS), the current WCET computation methods are usually applied to whole programs: these analysis methods require access to the whole system code, that is incompatible with the use of COTS. In this paper, after discussing the specific cases of the loop bounds estimation and the instruction cache analysis, we show in a generic way how static analysis involved in WCET computation can be pre-computed on COTS in order to obtain component partial results. These partial results can be distributed with the COTS, in order to compute the WCET in the context of a full application. We describe also the information items to include in the partial result, and we propose an XML exchange format to represent these data. Additionally, we show that the partial analysis enables us to reduce the analysis time while introducing very little pessimism.*

## 1. Introduction

Critical hard real-time systems are composed of tasks which must imperatively finish before their deadline. To guarantee this, a task scheduling analysis, that requires the knowledge of each task WCET, is performed. To compute the WCET, we must take into account (1) the control flow of the task (flow analysis) and (2) the host hardware (timing analysis). The current 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 the program size. Second, when the program to analyze depends on external components (e.g. COTS or libraries) whose sources are not available, the lack of information about the components prevents the WCET computation if information from the user is requested.

This paper focuses on the last problem, but it is shown that, as a side-effect, the computation time is also decreased. This article presents a generic method to partially analyze a black-box component, producing a component partial result that may be instantiated at each component call point to produce actual analysis results. We also define the information items that must be included in the component partial result, and propose an XML exchange file format.

As WCET computation involves a lot of analyses, the first section presents two of them as examples, the instruction cache analysis and the loop bound estimation, and show how to adapt them to be able to do partial analysis. Then, in the next section, we generalize our approach based on the previously described examples. We present our implementation (XML exchange format definition, and experimental results with OTAWA) in the third section and, after presenting related works in section 5,

[1]IRIT, Université de Toulouse, CNRS - UPS - INP - UT1 - UTM, 118 rte de Narbonne, 31062 Toulouse cedex 9, email: {ballabri,casse}@irit.fr

ECRTS 2009
9th International Workshop on Worst-Case Execution Time (WCET) Analysis
http://drops.dagstuhl.de/opus/volltexte/2009/2290

we conclude in the last section.

## 2. Component analysis

To compute the WCET of a program, one must previously perform a lot of analyses, some related to the flow control analysis (infeasible paths detection, loop bounds evaluation, etc), other related to the architecture effects analysis (cache analysis, branch predictor analysis, etc). The actual WCET computation can be done in several ways, but a widely used approach is IPET [14] (*Implicit Path Enumeration Technique)*. This technique assigns a variable to each *Basic Block* (BB) and edge, representing the number of executions on the path producing the WCET and builds a linear constraint system with these variables to represent the program structure and the flow facts (such as loop bounds). The WCET is then the maximization of an objective function depending on these variables and on the individual execution time of each BB computed by an ILP (Integer Linear Programming) solver.

To explain the general principles of component analysis, we study a simple scenario where a program `main()` is calling a function `callee()` in a component. Well known analyses involved in WCET are used: the instruction cache behavior analysis and the loop bounds analysis. For each one, we show how to partially analyse the `callee()` function to produce a partial result $PR_{callee}$. Then we show how to analyse the program `main()` while composing informations from $PR_{callee}$ (but without any access to the `callee()` code), to obtain analysis results for the whole program.

### 2.1. Instruction Cache Analysis

The partial cache analysis method presented in this section is based on the whole-program instruction cache[1] analysis presented by [12, 1] and improved in [2], 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). From these categories, new ILP constraints are added to the ILP system.

The categories are computed by abstract interpretation [8, 9] on a *Control Flow Graph* (CFG), composed of BB. *Abstract Cache States* (ACS) are computed before and after each BB which are in turn used to compute the categories. Three different analyses are done, each one using its own kind of ACS: the *May*, the *Must*, and *Persistence* analyses. While the *May* (resp. *Must*) analyses computes the cache blocks that may (resp. must) be in the cache, the *Persistence* finds the persistent blocks.

As previously stated in [4], three issues exist when dealing with partial instruction cache analysis. First, when executed, the `callee()` function ages and/or evicts cache blocks from the main program and so, it affects the categorization of the main program blocks. This is modeled by a cache *transfer* function (based on previous works by A. Rakib et al. in [18]) for each cache analysis. This *transfer* function is defined by $ACS^{after}_{type} = transfer(ACS^{before}_{type})$ (where *type* is one of *may*, *must*, or *pers).* In other words, it gives, for each cache analysis, the ACS after the function call based on the ACS before. Second, the `callee()` function contains blocks that need to be categorized. Unfortunately, the categorization of these blocks is not constant, but depends on the ACS at the function calling site.

[1] In this paper, we suppose that we have a set-associative instruction cache with *Least Recently Used* (LRU) replacement policy.

So we have introduced in [4] a *summary* function (only one is necessary to handle the three analyses) that is defined by $cat = summary(ACS^{before}_{must}, ACS^{before}_{may}, ACS^{before}_{pers})$, where *cat* is a Conditional Category (CC), i.e. a function such that $cat : Block \rightarrow \{ AH \mid AM \mid PS \mid NC \}$. In other words, it takes the various ACS before the call, and returns a category for each BB of `callee()`.

The last problem is that, when doing the partial analysis on `callee()`, the component base address is not known while it dictates which cache lines are affected by ageing, and which blocks are loaded. To take this into account, we force the alignment of cache blocks boundaries on `callee()` to be the same at partial analysis time and at instantiation time by enforcing the actual base address to be aligned on cache block size. Then, the matching between line results at partial analysis time and instantiation time is just a matter of shift and roll on the whole set of lines. This constraint may be easily achieved with most linkers.

Once we have the `callee()` partial result, the composition with the main program is done in two steps (illustrated in figure 1), without any further access to the `callee()` source code or executable:

1. We perform the instruction cache analysis on the main program in the same way as we would have done on a traditional (monolithic) analysis, except that we model the calls to the `callee()` function by the *transfer* function. At the end of this step, we obtain the ACS before and after each BB of `main()`.
2. Next, as we know the ACS values for `main()`, we have the ACS at the calling site of `callee()` and, therefore we can apply the *summary* function to obtain the categorization of `callee()` blocks.

At the end, we have got the categorization for all BB of the program (including component BB).

### 2.2. Loop Bound Analysis

The loop bound partial analysis presented in this paper is based on works of [10]. It tries to estimates the *max* and *total* iteration count for each loop in the program (the *max* is the maximum number of iterations for one entry into the loop, while the *total* is the total number of iterations of the loop during the execution of the outermost loop). The loop bound estimation is performed in three steps:

1. First, a context tree representing the program is built (a context tree is a tree where nodes are loops or function calls, and where children nodes are inner loops or function calls), and each loop is represented by a *normal form.* This normal form enables us (if possible) to get an expression representing the maximum iteration number of the loop, parametrized by the context of the loop. The computation uses abstract interpretation, in which the domain is an *Abstract Store (AS)* that maps each variable in the program to a symbolic expression describing its value.
2. The second step performs a bottom-up traversal of the context tree and instantiates each parametrized expression of *max* and *total* (computed in first step) to get absolute, context-dependant expressions for each loop.
3. Although the second step computes *max* and *total* expressions for each loops, it remains complex expressions that are evaluated in this last step to get a numerical value. Finally, a tree annotated with the results for each loop is returned by the analysis.

To perform partial loop bound analysis, we need (1) to determine the effect of the call to `callee()` on the main program (side effects, modified global variables, reference-passed arguments), and (2) to supply parametrized loop bound expressions depending on the calling context.

In fact, the analysis proposed in [10] is well suited to perform partial analysis. For the sub-problem (1), the *Abstract Stores* represents values of the program variables as symbolic expressions, that is, they support free variables representing data provided in the caller context. Therefore, performing the transfer function is just replacing the context variables by their actual values in the *Abstract Store* at the exit of the component. This allows us to satisfy the sub-problem (1), but it can be refined: to avoid wasting memory and computation time, we can trim the component *Abstract Store* by eliminating the variables that cannot have an effect on the main program. Only the remainder is part of the partial result of `callee()`.

To handle the sub-problem (2), we have to analyze the `callee()` function on its own (as if it was the main entry point). We need to perform the analysis up to (and including) step (2) that provides the symbolic expressions of the *max* and *total* for all loops in `callee()`. Then, it is easy to apply a context *Abstract Store* to this symbolic values at composition time by replacing free variables by their actual values. These symbolic expressions are stored in the partial result computed for `callee()`.

To sum up: the partial result for `callee()` contains (1) the (trimmed) component *Abstract Store* playing role of the *transfer* function and (2) the symbolic expressions of the *max* and *total*, used as a *summary* of the component. Likewise to the cache partial analysis, the obtained partial results are composed in two steps:

1. First, loop bound analysis is performed on the main program, while modeling `callee()` by the *Abstract Store*. This enables us to know the *max* and *total* bounds for the loops in `main()`.
2. Second, as we have got the calling context of `callee()`, we can apply it to the symbolic expressions of its loop bounds.

We can see a lot of similarities between the partial cache analysis and the partial loop bound analysis approaches. The next section proposes a generalization to other analyses.

## 3. Generic Principles of Component Analysis

A component is a block of code with one or more component entry points. When the analysis is performed, there is a partial analysis for each entry point, each one giving a partial result, and their aggregation is the partial result of the entry point. While in the last section, we have surveyed some specific examples of partial analyses, we give here a more general approach of component analysis.

### 3.1. Distribution Model

Let's consider an example where a black-box component (a COTS) is developed by a so-called *Producer*. The black-box component is delivered to a so-called *Assembler* that develops the main program using, among other things, the COTS and assembles them into the final executable. The main program contains hard real-time tasks, and so their WCET must be known. Even if the *Assembler* has no access to the COTS code, components must be analyzed and the *Producer* must compute the

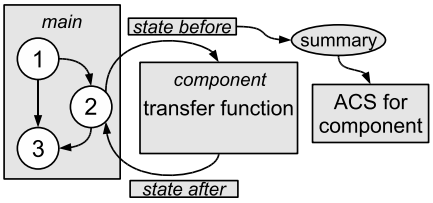



**Figure 1. Partial cache analysis composition.**

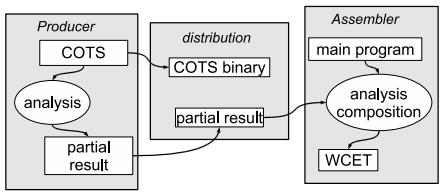
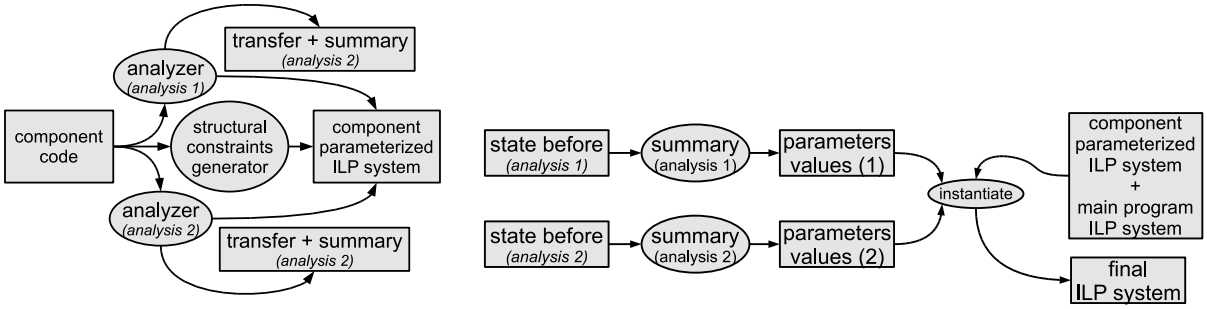



**Figure 2. Distribution model.**




(a) creating partial result

(b) applying summary function(s)

**Figure 3. partial result creation and use**

component partial result of its COTS, and delivers it with the COTS binaries, to let the *Assembler* compute the WCET of its application (figure 2).

In the examples of the previous section, we have seen that there is three sub-problems: (1) the fact that the called COTS function has an effect on the main program, (2) the call site dependencies of the COTS function analysis results, and (3) the variability of the COTS base address.

### 3.2. Transfer and Summary Functions

A lot of analyses involved in WCET computation produces abstract *states* at specific program points and then derives properties useful to the WCET computation. For example, we have several cache analysis methods (see [12] and [15] for two different techniques), loop bound analysis [10], branch prediction analysis [7], etc.

For this kind of analyses, the sub-problem (1) can always be handled by a *transfer* function, associated with each component entry point function. This *transfer* function is a generalization of the cache *transfer* function and is defined such that $state^{after} = transfer(state^{before})$, giving state after the call as a function of the state before the call. The terms $state^{after}$ and $state^{before}$ are of the same type, that is, dependent on the analysis and corresponding to the analysis domain. If the domain of the analysis is not the same for the COTS and for the main program, then the *transfer* function must be extensible (adaptable) so that, when we try to use it at composition time, its signature matches the domain used for the main program (the same is true for the generalized *summary* function).

The sub-problem (2) can be handled by a *summary* function that is a generalization of the *summary* function presented in previous section, with some extensions to model the contribution to the ILP system used for IPET.

While the generalization of the *transfer* function was quite straightforward, the *summary* function needs to consider the fact that a static analysis does essentially two things: (1) it computes states at some program points, and (2) it interprets the resulting states to deduce useful information for WCET computation. The state type, the deduction method and the resulting WCET-related informations are analysis-specific, but all analysis results contribute to the ILP system: some generalization may be possible for the *summary* function.

To this end, we define a parametrized system, and describe the results of the *summary* function as its instantiation in function of actual parameter values. The parameters are integer values with symbolic names. In the parametrized ILP system, each term in a constraint can optionally contain one parameter factor per term (ex: $2x_1.par_1 + 3.x_2 = 5.par_2$). We may also refine our parametrized ILP system by adding conditional rules with condition depending on the parameters that may activates some constraints. For example if a category of block 1 is always hit, we add a constraint specifying that number of miss is 0: $if(par_1 = AH)\,\{x_1^{mis} = 0\}$.

For each component entry point and for each analysis, the partial result includes a set of parameters describing the WCET-related properties depending on the calling context. In turn, the parameters are applied to a parametrized ILP sub-system describing the contribution of the component to the WCET. Each possible calling context can be represented by a valuation of the parameters. The *summary* function takes $state^{before}$ as parameters, and return a list of $(name, value)$ pairs for each parameter: $< (n_1, p_1), (n_2, p_2), ..., (n_k, p_k) >= summary(state^{before})$, where the $p_i$ are the parameter values and $n_i$ are the parameter names. The parametrized ILP sub-system contains common parts like the structural constraints, and analysis-specific parts (i.e. the contribution of each specific analyzer).

To sum it up, the component partial result (whose creation is outlined in figure 3 (a)) contains, for each component entry point function, (1) for each analysis, the *transfer* function taking the analysis state before the call, and giving the state after, (2) for each analysis, the *summary* function, which takes the analysis state before the call to the function, and gives back the values of the parameters, and (3) the partial parametrized ILP system describing the effect of the entry point on the WCET.

To instantiate this partial result (i.e. to find the WCET contribution associated with this component entry point function given the context), we need (1) to analyse the main program while modeling the component function by the *transfer* function, (2) apply the *summary* function to find the values of the parameters, that allows us (3) to instantiate the parametrized ILP sub-system representing the component. This ILP sub-system is then merged with the ILP system of the main program, allowing us to compute the WCET (see figure 3 (b)).

### 3.3. Relocation

A last problem arises when dealing with components: because the component code is relocatable, the actual base address of the component is different according to the application it is linked in. For some cases, like the instruction cache or the branch prediction analysis, the base address has an effect that must be taken into account. For example, in the instruction cache, the program addresses have an influence on the occupied cache line and blocks.

To prevent any problem, two mechanisms (possibly combined) can be used. First, the partial results may contain information that restricts the acceptable base address of the component. These restrictions ensures that the base address is chosen such that the computed partial results remains usable to

describe the WCET contribution of the component at composition time. This constraint is illustrated with the instruction cache analysis, where we impose restrictions on the component base address based on cache block alignment.

Second, if the analysis is concerned with the component base address, the *transfer* and *summary* functions must take the base address as arguments, that is, the *transfer* is defined such that $state^{after} = transfer(base,\ state^{before})$. This can be seen as a general case of the relative *transfer* and *summary* functions discussed in 2.1

### 3.4. Contribution to the ILP System: Minimization

In previous paragraphs, we have seen that the *summary* function takes a calling context and returns a configuration of the parameters used to instantiate the parametrized ILP system. Then, a parametrized ILP system must be returned as a part of the component partial result. Obviously, this component ILP system can be computed as if we were processing a standalone program: build structural constraints, and then build constraints related to various effects. The only difference is that we have to take care of context-dependent parts and introduce parameters accordingly. For example, the cache-related constraint could be built traditionally for all known categories, while leaving conditional rules for BB having context-dependant categories. However, this approach is not very efficient: the resulting parametrized sub-system, once instantiated and merged with the main system, gives an ILP system as large as the system of the monolithic analysis.

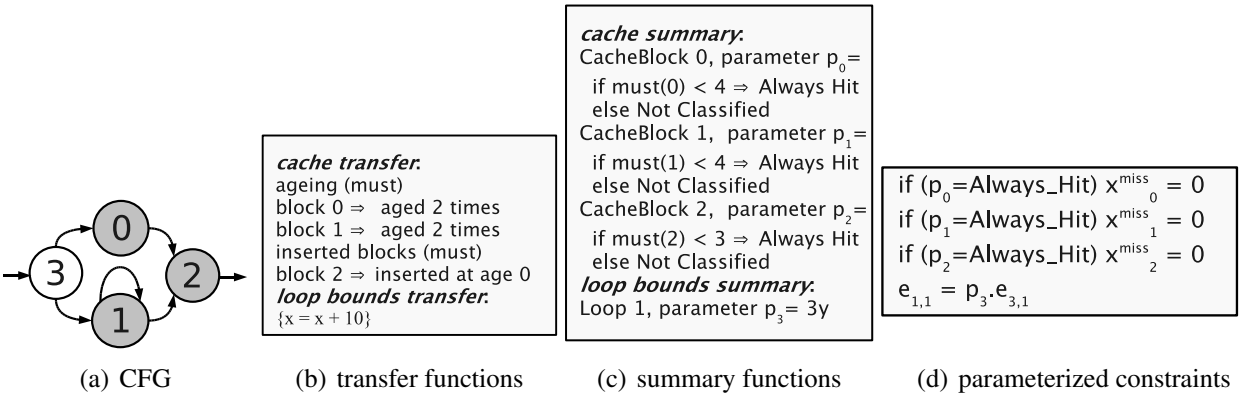
Yet, one may observe that some parts of the ILP system of a component entry point does not depend on the call context. Pre-computing these constant parts of the ILP would allow to minimize the component parametrized ILP sub-system and to speed up the WCET computation at assembly time. We have proposed a method to achieve this in [3] by CFG partitioning. It uses the concept of *Program Structure Tree* (PST) and *Single-Entry Single-Exit (*SESE*) regions* defined in [13]. We have shown that, if a SESE region is context-independent (*feasible region*), its WCET can be computed separately, and the region can be replaced by a single edge whose cost is the WCET of the region. This can be used to minimize the component ILP sub-system by pre-computing context-independent SESE regions in the component and replacing them by constant-cost edges.

To pre-compute a SESE region, two conditions must be verified: (1) the region must be a *feasible region* (that is, the ILP system restricted to the region must not depend on variables defined out of the region), and (2) the region WCET must not depend on the calling context of the component entry point function.

### 3.5. Example

Figure 4(a) shows an example of a component CFG where gray blocks are in the currently-analysed cache line (line 0), while the white block is in a different line. In addition, the bound of the loop at block 1 depends on an input argument named $y$, and there is a global variable $x$ which is incremented by 10 due to the call of the component. The partial analysis of this component produces the transfer functions, the summary functions, and the parametrized ILP system in 4(b), 4(c) and 4(d).

The effect of the component on the cache for the Must ACS is detailed (with the ageing of each block and the list of inserted blocks with their associated ages) in transfer functions of 4(b). The effect of the component on the global variable $x$ is also expressed. The summary function in figure 4(c) defines




(a) CFG (b) transfer functions (c) summary functions (d) parameterized constraints

**Figure 4. Example**

4 parameters, from $p_0$ to $p_3$. The parameters $p_0$, $p_1$and $p_2$ represents the possible categorizations for blocks 0, 1, and 2. The parameter $p_3$ represents the bounds of the loop, depending on the input argument $y$. The figure 4(d) shows the parametrized part of the ILP system (constants parts are not shown for the sake of brevity). There is a conditional constraint for each cache parameter: for each Always-Hit block, we generate a constraints setting the $x_{miss}$ to 0 and the last constraint express the loop bounds in a parametrized way, depending on $p_3$.

# 4. Implementation

In this section, we present the implementation of our component approach with OTAWA [6].

## 4.1. An Exchange Format for Partial Results

A common format is required so that each component-aware analyzer can communicate with each other. Wehave chosen XML as it maximizes extensibility and flexibility. The format must sum up the various data, presented in the previous sections, that must be included in the partial result as a tree of XML elements and attributes. While some information items are generic and can be specified precisely in the format, other ones are analysis-specific and are handled as an opaque XML content.

For each component entry point, the partial result needs to include two main items: (1) for each analysis, the *transfer* and *summary* functions data and (2) for all analyses, the parametrized ILP sub-system representing the component WCET contribution (this system includes contributions from various analyzers, but the result is merged).

The *component* tag contains a collection of entry points represented by the *function* element. Each one matches an entry point, corresponding to an actual function in the binary file of the component and contains the associated partial result. The functions that are not entry points should not appear in this list: they are taken into account in the partial results of entry points calling them. The *transfer* and *summary* functions data should be represented respectively by *transfer* and *summary* XML elements, children of *function* elements. although their content is analysis-specific, XML allows us to process them easily as opaque data.

The ILP system, is stored in *function* as a *system* element. The *entry* attribute specifies the variable which represents the execution count of the entry BB. It is used to merge the component ILP system

```
<component name="comp"> <function name="funct"> <analysis type="instcache">
    <transfer>
      <damage block="0" aging="2" /> <damage block="1" aging="2" >
      <insert block="2" age="0" />
    </transfer>
    <summary> <lblock cacheblockid="1" cond_category="..." /> ... </summary>
  </analysis> <system entry="x1">
    <objective type="max" const="0">
      <term var="x1" coef="12" /> <term var="xmiss_1" coef="32" /> ...
    </objective>
    <constraint op="EQ" const="1"> <term var="x1" coef="1"/> </constraint>
    <switch param="p0">
      <case value="always_hit">
        <constraint op="EQ" const="0" > <term var="xmiss_0" coef="1"/> </constraint>
      </case>
    </switch> <constraint op="LE" const="0" >
       <term var="e_1_1" coef="1"/> <term var="e_3_1" param="p3" coef="-1"/>
    </constraint> ... </system> </function> </component>
```

**Figure 5. XML format example**

with the main ILP system since the WCET contribution of the component is proportional to this variable. Inside the system, the objective function part is represented by an *objective* element with *const* attribute representing the constant part. Constraints are represented by *constraint* elements and contains *op* and *const* attributes to determine the comparator type and constant.

Both the objective function and the constraints contains terms represented as *term* elements with *coef*, *var* and *param* (in case of parameter dependency) attributes. If the *var* attribute is omitted, the term is considered as a constant. Notice that *var* and *param* can not be both omitted, otherwise the term is constant and must be added to *const* attribute. The final coefficient of a term is obtained by multiplying, if any, the *coef* and the *param* actual value providing a constant ensuring that the objective function or the constraint remains affine.

While parameters in terms allows to modify coefficients, they are also used to conditionally define some constraints with *if* and *switch* elements. The *if* condition is expressed using *param, const,* and *op* attributes and embeds a *then* element and (optionally) an *else* element. The *switch* uses the tested parameter in the *param* attribute, and conditional constraints are put in *case* children elements. Each *case* elements corresponds to a parameter value (specified by *value* attribute). As an example, a part of the XML exchange file that would be generated for the component example discussed in section 3.5 is shown in figure 5. It is not complete, but contains the transfer function informations for the cache, and the parametrized part of the ILP system.

### 4.2. Experimentation

To validate our approach, we have first tested it with the instruction cache partial analysis presented in 2.1. The analysis were done by OTAWA [6], our WCET computation framework. The target architecture features a simple pipelined processor (handled by contextual execution graph) and a 4-way set-associative instruction cache. The measurements have been done on a small subset of Mälardalen benchmarks with the *lp_solve* ILP solver. The small number of the selected benchmarks is due to the fact that (1) some benchmarks do not contain any function (except *main*) and (2) the provided functions are too small to make them interesting component entry points.

While the component partial analysis is meant to allow COTS usage, a nice side-effect is that it makes the analysis faster because (1) the sections of the partial result are reused when the entry point is

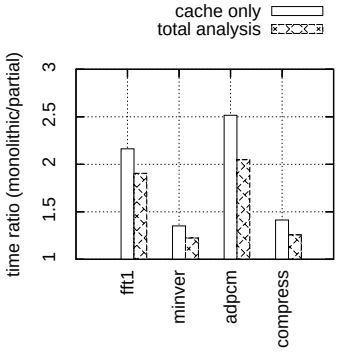



**Figure 6. time gain**

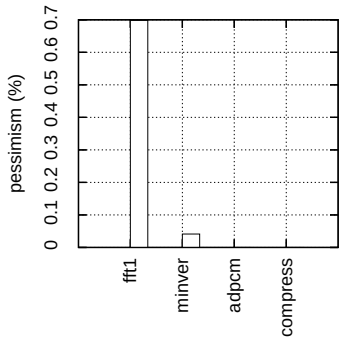



**Figure 7. pessimism measurement**

called more than once (due to regions and ILP system minimization) and (2) the WCET computation is usually of exponential complexity, so it is faster to analyse components separately than to analyse the whole program. We have measured this speed increase in figure 6: for each benchmark we display the ratio between the time with partial analysis and without. There is a measurement for the whole WCET computation (including the ILP solving), and one for the cache analysis time only. On average, the partial analysis is 1.6 times faster (even though our component is called few times in our small benchmarks). When accounting only for the cache analysis time, the average time gain ratio is of 1.86 (applying transfer and summary for the cache is immediate, while the parametrized ILP system instantiation requires handling regions that could not be pre-computed).

A drawback of our method is that the *transfer* and *summary* are not guaranteed to be exact. They must at least be conservative (for example, for the cache transfer function, the real $ACS_{must}^{after}$ that would have been computed by a real analysis of the component must be *less or equal* than the one computed by $transfer(ACS_{must}^{before})$). This may lead to pessimism, which we have measured: in the figure 7, for each benchmark we display the ratio between WCET with partial analysis and without. Fortunately, we detect that, on average, the WCET increase is only 0.18 % (the pessimism is caused by the *transfer* and *summary* functions because the region optimization method is exact).

In addition to this experiments, we have tested our loop bound partial analysis with oRange [10] on the following Mälardalen benchmarks providing enough interesting components with loops: *adpcm*, *cnt, crc, duff, expint, jfdctint, ludcmp*. On all the performed tests, the loop bound estimation showed the same results with and without the partial analysis. Furthermore, the few big-enough test (*adpcm*) exhibited a 1.5 times faster partial analysis (other tests were too small and therefore too quickly processed by oRange to deduce any valuable performance information).

## 5. Related work

In [5], S. Bygde and B. Lisper proposes an approach for parametric WCET computation. This method addresses the fact that the WCET of a program depends on the input variables and arguments. Some parameters, introduced in the IPET system, are expressed in terms of the input variables. Then, the method for parametric ILP resolution of [11] is applied to get a parametric, tree-based WCET, with nodes representing tests on parameters and leaves representing numerical WCET values.

In [17], F. Mueller extends the method defined in [15, 16] 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. Mueller's approach is focused on the performance gain resulting from the partial analysis, but does not appear to be designed to handle COTS, since the compositional step needs to visit each component CFG. Moreover, his approach needs more analyses passes than ours, and is bound to direct-mapped instruction caches.

In [18], a method is proposed to perform component-wise instruction-cache behavior prediction. It adapts [1] 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: (1) 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; and (2) 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 performs the same role as the cache transfer function described in 2.1 We have extended this approach by adding the *Persistence* analysis.

One may observe that the two last papers provide a working approach for components but they are bound to the instruction cache analysis. In the opposite, our approach proposes a generic framework that supports several types of analyses.

## 6. Conclusion

In this paper we have proposed a generic approach to handle blackbox component partial analysis, which enables us to compute partial WCET for components, and then to reuse this result in the WCET computation of a main application. By surveying some existing IPET-based partial analysis (the instruction cache, and the loop bound estimation), we have generalized this approach to a generic framework integrating partial analyses and contribution to the WCET ILP system, and we have defined a common partial result exchange format in XML, extensible and adaptable to other WCET-related analyses.

While this approach is primarily designed to handle COTS, tests with OTAWA and oRange have shown that (1) due to exponential complexity of WCET computation and (2) to the partial result reuse, we also gain a significant analysis time without sacrificing precision. Adaptation of others partial analysis in OTAWA (like branch prediction) is underway. It would be also interesting to continue experimentations on bigger benchmarks.

In addition, we would like to fix one drawback of our approach: it needs ad-hoc derivation of partial analyses from existing, non-partial ones. In the future, for some analysis based on abstract interpretation, we want to survey methods to help to or to derive automatically *transfer* and *summary* functions from a description of the analysis. At least, it may be valuable to provide a framework to help writing partial analyses by automating the more used parts.

## References

[1] ALT, M., FERDINAND, C., MARTIN, F., AND WILHELM, R. Cache behavior prediction by abstract interpretation. In *SAS'96* (1996), Springer-Verlag.

[2] BALLABRIGA, C., AND CASSÉ, H. Improving the First-Miss Computation in Set-Associative Instruction Caches. In *ECRTS'08* (2008).

[3] BALLABRIGA, C., AND CASSÉ, H. Improving the WCET computation time by IPET using control flow graph partitioning. In *International Workshop on Worst-Case Execution Time Analysis (WCET), Prague* (juillet 2008).

[4] BALLABRIGA, C., CASSÉ, H., AND SAINRAT, P. An improved approach for set-associative instruction cache partial analysis. In *SAC'08* (March 2008).

[5] BYGDE, S., AND LISPER, B. Towards an automatic parametric wcet analysis. In *WCET'08*.

[6] CASSÉ, H., AND SAINRAT, P. OTAWA, a framework for experimenting WCET computations. In *ERTS'05* (2005).

[7] COLIN, A., AND PUAUT, I. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems 18*, 2-3 (2000).

[8] COUSOT, P., AND COUSOT, R. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming* (1976), Dunod.

[9] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN-SIGACT* (1977).

[10] DE MICHIEL, M., BONENFANT, A., CASSÉ, H., AND SAINRAT, P. Static loop bound analysis of C programs based on flow analysis and abstract interpretation. In *IEEE RTCSA'08* (August 2008).

[11] FEAUTRIER, P. Parametric integer programming. *RAIRO Recherche Opérationnelle 22*, 3 (1988).

[12] FERDINAND, C., MARTIN, F., AND WILHELM, R. Applying compiler techniques to cache behavior prediction. In *ACM SIGPLAN workshop on language, compiler and tool support for real-time systems* (1997).

[13] JOHNSON, R., PEARSON, D., AND PINGALI, K. The program structure tree: Computing control regions in linear time. In *SIGPLAN PLDI'94* (1994).

[14] LI, Y.-T. S., AND MALIK, S. Performance analysis of embedded software using implicit path enumeration. In *Workshop on Languages, Compilers, and Tools for Real-Time Systems* (1995).

[15] MUELLER, F. Timing analysis for instruction caches. *Real-Time Systems* (May 2000).

[16] MUELLER, F., AND WHALLEY, D. Fast instruction cache analysis via static cache simulation. *TR 94042, Dept. of CS, Florida State University* (April 1994).

[17] PATIL, K., SETH, K., AND MUELLER, F. Compositional static instruction cache simulation. *SIGPLAN Not. 39*, 7 (2004).

[18] RAKIB, A., PARSHIN, O., THESING, S., AND WILHELM, R. Component-wise instruction-cache behavior prediction. In *Automated Technology for Verification and Analysis* (2004).