A Modular & Retargetable Framework for Tree-based WCET Analysis*

Antoine Colin, Isabelle Puaut
IRISA, Campus de beaulieu, 35042 Rennes Cédex, France, {acolin/puaut}@irisa.fr

Abstract

A fundamental requirement for hard real-time systems is the knowledge of tasks worst case execution times (WCET). Static worst-case execution time analysis (WCET analysis), thanks to the static analysis of a piece of source code, returns an upper bound of the time required to execute it on a given hardware. Taking into account modern architectural features makes it possible to determine tight WCET bounds. Several mechanisms that use modeling and simulate some architectural feature behaviors such as instruction cache, branch prediction mechanism and pipeline have been proposed in the literature. These methods have often been designed independently from each other, which leads to an integration issue. This paper proposes to formalize (through data structures) three techniques for static simulation of instruction cache, pipeline and branch prediction in order to gather them in an integrated static WCET analysis framework. Performance improvements due to the integrated approach are also given.

1. Introduction

Real-time systems differ from other systems by a stricter criterion of the correctness of their applications. Actually, the correctness of a real-time application does not only depend on the delivered result, but also on the time when it is produced. In *hard* real-time systems, missing task deadlines can be catastrophic. As a consequence, knowing task worst-case execution times (WCET) is of prime importance for the timing analysis of such systems.

The purpose of static WCET analysis [17] is to estimate *a priori* (before execution) the worst case execution time of a given program on a given processor thanks to the analysis of its source code. Static WCET analysis is complicated due to the presence of architectural features that improve the performances of the processor such as instruction cache, branch prediction and pipeline. A main issue in static WCET analysis is to avoid pessimism while achieving safe estimations. On a given architecture, reducing estimation pessimism is possible thanks to the modeling of modern architectural features. Such modelings allow to know the behavior of architectural features in the worst case. Usually, either these methods have been developed apart from one another in a dedicated timing analyzer, or they have been jointly designed and they cannot be used separately from one another. In both cases, the resulting static WCET analysis tools are very dependent on the chosen target architecture and the architectural features that have been taken into account. Thus it can be difficult to retarget them when the underlying architectural feature simulation methods have to be changed in order to improve the accuracy or to adapt them to a new target.

This paper is a proposal for formalizing the results obtained thanks to three architectural modeling techniques in order to have them cooperate in an integrated and easily retargetable timing analysis framework. The considered techniques aim at estimating the worst case behavior of the instruction cache, the branch prediction mechanism and the pipeline. These data structures make it possible to integrate the considered techniques in the framework of a timing analyzer in a modular way. We report our experience in using this formalism to design a static WCET analysis tool named HEPTANE that takes into account the three above-mentioned architectural features for a Pentium processor. As it will be shown, using these methods all together allows to tighten the WCET estimation.

The remainder of the paper is structured as follows. First, we present some related works in the domain of static WCET analysis (section 2). Then, section 3 defines some basic concepts related to static WCET analysis. Section 4 describes the common formalism we propose to express the results of micro-architecture simulation techniques and section 5 presents their integration and cooperation in an efficient WCET timing schema. An overview of HEPTANE and three Pentium specific micro-architecture simulation techniques are exposed in section 6. Section 7 gives quantitative results of our experiments performed using the *SNU real-time benchmarks suite*[1]. Finally, we conclude in section 8.

2. Related Work

Static WCET analysis has been the subject of many researches, and substantial progress has been made in the area

*Appeared in the 13th Euromicro conference on real-time systems, Delft, The Netherlands, June 2001

[1]Provided by Seoul National University RTOS Lab (*http://archi.snu.ac.kr*)

over the last decade (see [17] for an overview of the essential contributions to the state of the art in static WCET analysis achieved so far).

The WCET estimation is a twofold issue: (i) possible execution paths have to be determined to calculate the execution time along the worst of these paths. (ii) execution time for each atomic unit of a path has to be accurately estimated.

There are two ways to determine what sequence of instructions will be executed. On the one hand, it is possible to consider all paths implicitly by using integer linear programming [10, 19, 14]. On the other hand, the enumeration of program paths can be explicit as in tree-based methods [16, 12, 2]. Some information is either required or useful to tighten path description: how many times loops iterate, dependencies between **if**-statements, etc. This information can be user-provided (through annotations embedded in the source code [15], provided separately [10]), or automatically obtained using flow analysis [7, 8, 6].

Some techniques have been defined to take into account the effects of architectural features, as for instance F. Mueller's *static cache simulation* [13], which statically simulates all possible contents of the cache by considering possible execution paths all at once. Several works use *static simulation* techniques to take into account architectural features such as pipelines [5, 8, 20], instruction caches [13, 11, 8, 20], cache hierarchies [13], data caches [9], and branch prediction [2]. Most timing analysis approaches that integrate several of the above static simulation techniques (usually cache and pipeline) into a single static WCET analysis method lead to static WCET analysis tools that are difficult to retarget. As said in [20], *"a main issue for future work on WCET analysis is to make the approach [...] more easily adaptable to various hardware architectures"*. The proposal exposed in this paper addresses this issue. The framework exposed in [6] addresses the modularity and retargetability issues in the field of WCET analysis based on IPET (Implicit Path Enumeration Technique). It is very difficult to compare our *tree-based* static WCET analysis framework (explicit path enumeration) to the above IPET-based framework because underlying path enumeration techniques are very different.

3. Program representations

We present here the program representations and the terminology required to describe our static WCET analysis framework. Given a source code, two different representations of the program can be constructed: a syntax tree and a control flow graph (see § 3.1). Finally, the tight static WCET analysis can be achieved (among other things) thanks to the information regarding loop nesting provided by the syntax-tree (see § 3.2).

3.1. Program representations

The assembly code of the program is split into smaller parts : *basic blocks*. A basic block is a sequence of one or more instructions with a single entry point and a single exit point. Control transfer instructions[2] (*CT-instructions* for short) are present only at the end of basic blocks. Based on the basic blocks and on syntactic information gathered during compilation, we build two logical representations of the analyzed source code: a control flow graph and a syntax tree. The source language is restricted to ensure the equivalence between the control flow graph and the syntax tree (*i.e.* all execution paths that are expressed by the control flow graph are also expressed by the syntax tree, and vice versa).

♦ The **control-flow graph** describes the control flow for each function in the program. The graph nodes are the basic blocks and the edges are the possible sequencing between the basic blocks. An edge can be either an seq edge or a jmp edge. Following an seq edge means executing the following instruction in the assembly code whereas following a jmp edge means executing the instruction located at the target address of the CT-instruction.

♦ The **syntax tree** is a tree whose nodes represent the structure of programs in the high-level language and whose leaves represent basic blocks. Leaves in the syntax tree coincide with the nodes in the control flow graph. There are four kinds of tree nodes : leaves, "SEQUENCE", "FOR" and "IF".

3.2. Loop nesting

Taking loop nesting into account is of prime importance for obtaining a tight WCET estimation. As shown further, the behavior of an instruction within a basic block may depend on the surrounding loop which is executed. We define a hierarchical naming for nested loops that reflects the position of a loop relatively to its enclosing and enclosed loops.

Each loop is associated with a loop nesting level (*ln-level* for short). In the following, a loop whose *ln-level* is α will be referred as loop α. Top level loops are named [0], [1], etc. Loops enclosed by loop [0] are named [0.0], [0.1], and so on. We also define *ln-level* [] which is the level enclosing the whole syntax tree. For example, a loop whose *ln-level* is [2.3.1.0] is enclosed in loops [2.3.1], [2.3], [2], and lastly []. We add a dummy *ln-level* named $never$, upper than []. In the following, we will refer to the innermost loop that contains a basic block as the *ln-level*$_\perp$ of this basic block. Let $\succeq$ be a partial order on *ln-levels*:

$$\begin{cases} x \succeq y \iff x \text{ is a left prefix for } y \\ \forall x,\ never \succeq [] \succeq x \end{cases}$$

And let us define $\succ$ as $x \succ y \Leftrightarrow \big((x \succeq y) \wedge (x \neq$

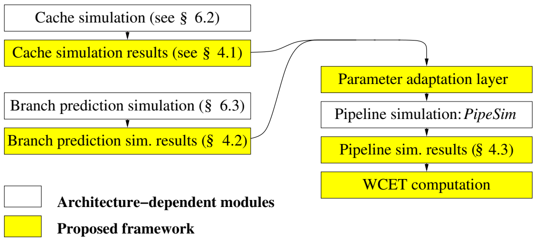
[2]Control transfer instructions are conditional and unconditional jumps, call and return instructions

$y)$). Then, *ln-level* $\alpha \succ$ *ln-level* β implies that loop β is enclosed in loop α. We also define a function (Ψ) giving the *ln-level* of the innermost enclosing loop of 2 other loops (*e.g.* $\Psi([2.3.1],[2.3.4]) = [2.3]$).

$$q = \Psi(x,y) \iff (q \succeq x \wedge q \succeq y) \wedge (\nexists p \mid q \succ p \wedge\ p \succeq x \wedge p \succeq y)$$

4. A formalism for the integration of micro-architecture simulation techniques

This section is devoted to the description of a formalism allowing to use static micro-architecture simulation techniques in a modular way in tree-based static WCET analysis. It provides a means to use various simulation methods (as long as they comply to that formalism) in order to obtain a fully-fledged timing analyzer. Such an analyzer can be retargeted by adapting or replacing some modules (white areas in the figure below) in this framework.



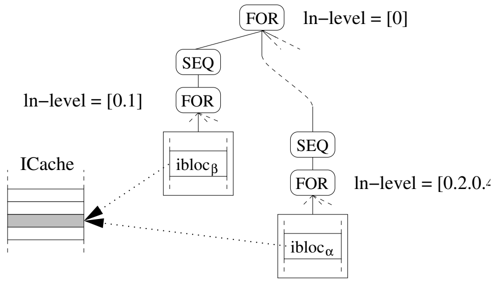
In this section, we define the data structures used to represent the micro-architecture simulation results of a basic block (shaded areas in the figure). Paragraph 4.1 and 4.2 deal respectively with instruction cache and branch prediction simulation. Then, the pipeline is represented as a “pipelining function” in 4.3.

4.1. Instruction cache simulation results

The goal of the caches is to improve the speed of the transfers from/to memory. Since the address of each instruction is statically known (unlike many data references) taking into account the instruction fetching speed improvement due to the instruction cache is possible. The presence (or absence) of an instruction in the instruction cache impacts on the WCET of a basic block.

Our formalism defines the cache worst case behavior at the level of instruction blocks. An instruction block ($iblock$ for short) is a portion of a basic block that fits exactly into a cache line. It may contain several instruction(s) and/or instruction fragment(s).

The use of $iblocks$ as an intermediate representation allows to adapt the instruction cache simulation to various instruction sets (with fixed and variable size) and cache line sizes. The proposed data structure provides for every $iblock$ information on its presence or absence in the cache according to the loop nesting hierarchy. The proposed formalism associates a *ln-level* with each $iblock$ to indicate which loop level execution will cause a conflict that may result in a cache-miss. The example below shows such a situation caused by two conflicting $iblocks$, *i.e.* two $iblocks$ competing for the same cache line.



Each time $iblock_\alpha$ is executed, it overwrites $iblock_\beta$ in the cache. This happens each time loop $\Psi([0.1],[0.2.0.4]) = [0]$ is iterated. So $iblock_\alpha$ is associated to *ln-level* $[0]$. The couple $< iblock_\alpha, [0] >$ indicates that $iblock_\alpha$ causes a cache-miss each time loop $[0]$ is iterated. To put it another way, $iblock_\alpha$ is estimated to be in the cache when loops $[0.2.0.4]$, $[0.2.0]$ and $[0.2]$ are considered, and is estimated to be a cache miss for loops $[0]$ and [].

When the *ln-level* associated with an $iblock$ is *ln-level*$_\perp$ (the innermost loop that contains the $iblock$) it means that the $iblock$ is never in the instruction cache at execution time. Such $iblocks$ cause a miss each time they are executed (*i.e.* when loop *ln-level*$_\perp$ is iterated). On the contrary, an $iblock$ which is associated to *ln-level* $never$ will never cause a miss.

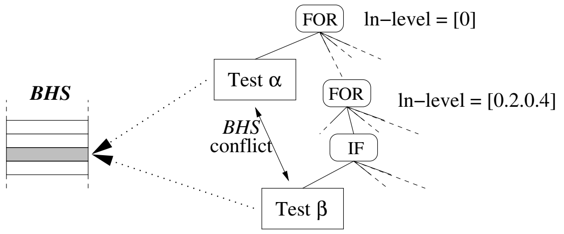
As a basic block is made up of several $iblocks$, the proposed data structure, called *IB-est* ($iblocks$ behavior estimation), is a set of $< iblock$, *ln-level* $>$ couples (one per $iblock$ in the considered basic block).

4.2. Branch prediction simulation results

The branch prediction mechanism aims at improving the pipeline efficiency in the presence of CT-instructions. When a CT-instruction is reached, a break in the flow of the instructions in the pipeline may occur, because the microprocessor does not know which instruction to execute until it has finished executing the CT-instruction. Modern processors incorporate a *branch prediction mechanism* which allows the processor to decode instructions beyond branches in order to keep the instruction pipeline full. This mechanism predicts whether a CT-instruction will be taken or not. When a CT-instruction is predicted taken, its target address is also predicted. We assume that predictions are based on the branching history of CT-instructions. The prediction of the result of a CT-instruction execution is computed using former execution outcomes of the predicted CT-instruction. There are various ways of storing branch histories, as for instance a dedicated cache (as for example Intel’s BTB), a dedicated set of bits in the instruction cache (UltraSparc architecture). In the following, this storage will be referred to as the *branch history storage* (BHS for short).

As said in § 3.1, a basic block contains at most one CT-instruction. The static branch prediction simulation aims at estimating which CT-instruction executions will be mispredicted. These estimations are *ln-level* dependent (a CT-instruction behavior depends on the considered *ln-level*). Considering that there are two possibilities when a CT-instruction is executed (jmp and seq), each CT-instruction (*i.e.* each basic block) is associated with two potentially different results. The chosen data structure is very similar to the previous one: two *ln-levels* are associated to every basic block, one per branching possibility. The *ln-level* indicates which loop level execution will cause a conflict that may result in a CT-instruction misprediction.

Whatever BHS implementation is chosen, it always has a bounded number of entries, so conflicts may appear when two CT-instructions compete for the same entry. The next figure gives an example of such a conflict. The history of CT-instruction of the basic block "$Test\ \alpha$" is never in the BHS at prediction time because it is overwritten by the history of CT-instruction "$Test\ \beta$" each time loop [0] iterates.



Conflicts in the BHS do not always lead to mispredictions. Here, we have to rely on the prediction policy of the considered architecture. Let us assume that, in this example, CT-instructions whose branching histories are unknown at prediction time are predicted to **not jump** by default (that is the Pentium case). With this assumption, we estimate that the branch prediction mechanism predicts that CT-instruction of $Test\ \alpha$ never jumps when loop [0] is iterated (*n.b.* *ln-level*$_\perp$ of $Test\ \alpha$ is [0]). Thus, the seq branch of basic block $Test\ \alpha$ is never mispredicted whereas its jmp branch is always mispredicted. So, *ln-level* [0] is associated with the jmp branch and *ln-level* $never$ is associated to the seq branch.

As for the instruction cache, the associated *ln-level* can take any value between *ln-level*$_\perp$ and $never$. To characterize both branching possibilities, static branch prediction simulation results are formalized by the following data structure (namely *CTI-est*): $\{< BB_n^{seq}, \textit{ln-level} >, < BB_n^{jmp}, \textit{ln-level} >\}$. These data will be further used in the next paragraph as an input data for the pipeline simulation.

4.3. Pipelined execution simulation

The goal of pipelined execution is to increase the instruction throughput, *i.e.* the number of completed instructions per clock cycle. Pipelining allows various execution steps of instructions to simultaneously overlap. We assume that a pipeline simulation function (called $PipeSim$) is provided as a module for the considered micro-architecture. Here, we define only the interface of the $PipeSim$ function. We do not detail here any implementation of the $PipeSim$ function (such a description is given in § 6.4). This paragraph focuses on: (i) data formalisms needed to achieve pipeline simulation (required inputs for pipeline simulation), (ii) the kind of result provided by $PipeSim$ function, (iii) the way we use pipeline simulation results to estimate basic block WCET (further used in § 5).

4.3.1. Pipeline simulation parameters

The estimation of the execution time of a basic block depends on the considered *ln-level*. Performing the estimation requires that several pieces of information is provided: (i) a description of the basic block (*i.e.* the list of its instructions), (ii) knowledge about the decomposition of each instruction into pipeline stages (resource needs, execution time, etc.) and (iii) the results of the static instruction cache simulation (*IB-est*, § 4.1) and the static branch prediction simulation (*CTI-est*, § 4.2) which must be converted, as described in § 4.3.2, to fit $PipeSim$ interface.

4.3.2. Interface of the $PipeSim$ function

The static pipelined execution simulation estimates the worst case time needed for a basic block to be executed assuming a known context of execution. This context is made of *IB-est* and *CTI-est* of the basic block and a *ln-level*. These data structures are transformed by an adaptation layer as described below.

Given a *ln-level* (L), an *IB-est* set is split into two subsets. $IBhit_L$ is the set of $iblocks$ which are in the instruction cache when loop L is iterated, and $IBmiss_L$ contains the rest of the $iblocks$ of the basic block. An $iblock$ is predicted to be in the cache if *ln-level* L is lower than its associated *ln-level* .

Note that in order to fit the pipeline simulation needs, $iblocks$ caching information is converted into instruction caching information.

Given a *ln-level* (L) and a taken branch x (x = jmp or seq), $CTIdelay_L^x$ is computed from *CTI-est*. It is a boolean indicating whether a misprediction has to be simulated or not: Note that $\succeq$ only defines a partial order on *ln-levels* , but L is chosen such as either $L \succeq$ *ln-level* or *ln-level* $\succeq L$ is true for all *ln-levels* in *IB-est* and *CTI-est*.

$$
\begin{array}{rcl}
iblock \in IBhit_L & \Leftrightarrow & (< iblock, \textit{ln-level} > \in \textit{IB-est}) \wedge (\textit{ln-level} \succ L) \\
iblock \in IBmiss_L & \Leftrightarrow & (< iblock, \textit{ln-level} > \in \textit{IB-est}) \wedge (L \succeq \textit{ln-level}) \\
CTIdelay_L^x \text{ is true} & \Leftrightarrow & (< BB^x, \textit{ln-level} > \in \textit{CTI-est}) \wedge (L \succeq \textit{ln-level}) \\
CTIdelay_L^x \text{ is false} & \Leftrightarrow & (< BB^x, \textit{ln-level} > \in \textit{CTI-est}) \wedge (\textit{ln-level} \succ L)
\end{array}
$$

The $PipeSim$ function provides the WCET of a basic block considering a *ln-level* L and an outgoing basic block branch x. These results associated with *ln-level* L are formalized by: $< time, ln\text{-}level >$.

$$PipeSim(BB^x, IBhit_L, IBmiss_L, CTIdelay_L^x) =< time, L >$$

4.3.3. Representing the WCET of a basic block

The WCET of a basic block depends on the considered *ln-level* and branch execution. $WCET\_BB^x$, defined below, represents the WCET of the basic block BB for all *ln-levels* assuming that branch x (x = seq or jmp) is executed. This WCET representation takes instruction cache and branch prediction effects into account by using the $PipeSim$ function. An optimistic result : $T_0 = PipeSim_{L_0}(BB^x, all-iblocks, \emptyset, false)$, is associated to *ln-level*$_\perp$ (the *ln-level* of the inmost loop containing BB^x). T_0 is the best possible timing result for BB^x, we use it as the execution time basis. Let $\Lambda = L_1 \dots L_n$ be the sorted list ($L_{i+1} \succ L_i$) of all *ln-levels* that have to be considered (*i.e.* *ln-levels* that appear in *IB-est* or *CTI-est*). A $PipeSim$ result (T_i) is computed for each needed *ln-level* ($L_i \in \Lambda$). The differences between T_i and its preceding result, T_{i-1}, are computed to provide the timing difference between executions of *ln-levels* L_{i-1} and L_i (*i.e.* $t_i = T_i - T_{i-1}$).

$$WCET\_BB^x = \{< T_0, \textit{ln-level}_\perp \textit{ of } BB >, < t_1, L_1 >, \dots, < t_n, L_n >\}$$

This formalism allows to express the WCET of a basic block for every particular *ln-level* context. *E.g.* the WCET of BB^x when loop L_k is iterated is $T_0 + \sum_{i=1}^{k} t_i$.

Finally, each basic block BB is associated to two WCET representations: WCET of BB^{jmp} and WCET of BB^{seq}. These WCET representations are equal when the basic block has only one branching possibility.

5. Tree-based WCET estimation

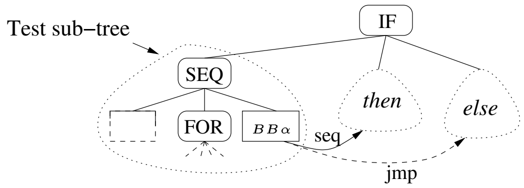
In tree-based methods, the final WCET is generated by a bottom-up traversal of the tree representing the program (syntax tree, § 3.1). The analysis results for the basic blocks are used to make timing estimations for larger parts of the program. The set of basic formulae given by Puschner and Koza in [18], originally proposed for a processor without caches, branch prediction, or pipelines, gives the maximum execution time of every control construct according to the syntax tree structure (*cf.* table 1.a). This timing schema uses a simple basic block WCET represented by a scalar. To use the basic block WCET results we introduced in the previous paragraph, we have to define a new timing schema suitable for more complex data.

First of all, when computing the WCET of a language construct its *ln-level* is of prime importance. Thus, an *ln-level* parameter is added to original WCET equations.

Data formalisms defined in § 4.3 require the definition of basic operators to build a new timing schema. As concluded in § 4.3, a $WCET\_BB^x$ is a set of $< wcet_i, lnlev_i >$. As these data are not scalars, we first have to redefine operators $+$ and $*$. Let $\overset{L}{\oplus}$ be the addition operator on WCET elements. Two $WCET\_BB^x$ are added by making the union of the two sets and summing together (i) elements whose *ln-levels* are equal, and (ii) elements which *ln-levels* are equal or lower than L. Let $\overset{L}{\otimes}$ be the operator that applies a multiplying factor only to elements which *ln-levels* are equal or lower than L and then sums elements which *ln-levels* are equal or lower than L.

These operators allow to define a new timing schema (see table 1.b).

When computing the WCET of conditional and loop structures, two kinds of executions (*i.e.* jmp and seq) of the *test* construct have to be considered. The *test* sub-tree of a conditional or a loop structure can either be a single basic block or a sequence of structures. In the second case (sequence of structures), the branching choice is made by the right-most basic block of the *test* sub-tree.



So, WCET$_{seq}$ and WCET$_{jmp}$ are only computed for structures that can be a loop or conditional test: *i.e.* sequences and single basic blocks.

The timing equations of the "FOR" and "IF" constructs shown in table 1.b depend on assumptions about the compilation schemes of these constructs. These timing equations may be adapted to other compilation schemes by swapping seq and jmp labels in timing formulae.

6 Adapting micro-architecture simulation techniques

Three micro-architecture simulation techniques have been adapted to the formalism and timing schema defined above. These techniques have been included in the framework of an experimental static WCET analysis tool targeted at the Pentium processor. This section provides a quick overview of our WCET analyzer prototype. Then, it briefly describes the "Pentium suited" static simulation techniques and the adaptation of their results to make them compliant with the data formalism defined in section 4.

6.1. The HEPTANE static worst-case timing analyzer

HEPTANE (*Hades Embedded Processor Timing ANalyzEr*) is a WCET analyzer for programs written in ANSI C. Figure 1 depicts the internal structure of HEPTANE. White areas in the figure are Pentium specific modules. It is made of four parts: (i) a front-end that generates the program syntax tree, (ii) the external retargetable assembly transformation tool Salto [1], used to obtain hardware-dependent in-

(a) Original timing schema [18]

Construct S	
$S_1; \ldots; S_n$	$WCET(S) = \sum_{i=1}^{n} WCET(S_i)$
if (tst) then S_1 else S_2	$WCET(S) = WCET(Test) + \max(WCET(S_1), WCET(S_2))$
for (;tst;inc) S_1	$WCET(S) = maxiter \times (WCET(Test) + WCET(S_1) + WCET(Incr))$ $+WCET(Test) + WCET(Exit)$
basic block B_x	$WCET(S) = MAXT(B_x)$

$MAXT(B_x)$ is the maximum execution time for a basic block B_x.
$maxiter$ is the maximum number of iterations of a loop.

(b) Modified timing schema

Construct S	
$S_1; \ldots; S_n$	$WCET(S,L) = WCET(S_1,L_1) \overset{L}{\oplus} \ldots \overset{L}{\oplus} WCET(S_n,L_n)$ $WCET^{seq}(S,L) = WCET(S_1,L_1) \overset{L}{\oplus} \ldots \overset{L}{\oplus} WCET(S_{n-1},L_{n-1}) \overset{L}{\oplus} WCET^{seq}(S_n,L_n)$ $WCET^{jmp}(S,L) = WCET(S_1,L_1) \overset{L}{\oplus} \ldots \overset{L}{\oplus} WCET(S_{n-1},L_{n-1}) \overset{L}{\oplus} WCET^{jmp}(S_n,L_n)$
if (tst) then S_1 else S_2	$WCET(S,L) = max(\ WCET^{seq}(Test,L) \overset{L}{\oplus} WCET(S_1,L)\ ,$ $WCET^{jmp}(Test,L) \overset{L}{\oplus} WCET(S_2,L)\ )$
for (;tst;inc) S_1	$WCET(S,L) = maxiter \overset{L}{\otimes} (WCET^{jmp}(Test,L) \overset{L}{\oplus} WCET(S_1) \overset{L}{\oplus} WCET(Incr,L))$ $\overset{L}{\oplus} WCET^{seq}(Test,L) \overset{L}{\oplus} WCET(Exit,L)$
basic block BB_x	$WCET(BB,L) = WCET\_BB^{?}$ $WCET^{seq}(BB,L) = WCET\_BB^{seq}$ $WCET^{jmp}(BB,L) = WCET\_BB^{jmp}$

$WCET\_BB^{?}$ is the WCET of basic block BB which has only one outgoing branch (seq **xor** jmp).
$maxiter$ is the maximum number of iterations of a loop.

Table 1. Original and modified timing schema

formation, (iii) the core of HEPTANE that includes modules for the management of the instruction cache, the pipeline and the branch prediction mechanism, (iv) the Maple tool which evaluates expressions required to compute the maximum number of iterations with symbolic annotations of loops [2].

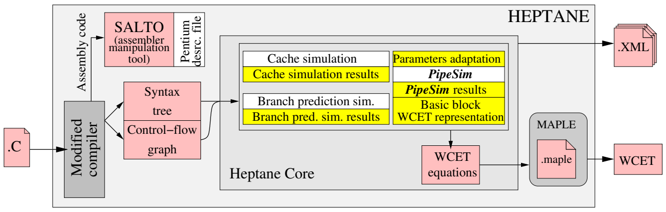


Figure 1. Internal structure of the HEPTANE static WCET analysis tool

6.2. Instruction cache

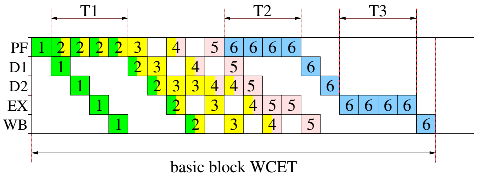
On the Pentium processor, the instruction cache is organized as a 2-way set associative cache. There are 128 sets and each set contains 2 lines (a cache line is 32 bytes wide). The static simulation technique we used in HEPTANE to estimate the instruction cache behavior is based on F. Mueller work, detailed in [13]. Since Mueller's technique deals with plain instructions instead of $iblocks$, a slight modification to Mueller's original work has been introduced. The method determines for each $iblock$ reference whether it **will** result in a cache hit or if it **may** cause a miss during program execution.

6.3. Branch prediction mechanism

In our context (Pentium processor) predictions are based on the *Branch Target Buffer* (BTB) which is an implementation of the BHS defined in § 4.2 (*i.e.* it stores the branch history of CT-instructions). The branch prediction mechanism makes predictions according to this history. When the prediction turns out to be wrong, the pipeline has to be flushed, which incurs a timing penalty. The technique we use to take into account branch prediction is the static BTB simulation, introduced in [2].

6.4. Pipelining effects

We describe here the implementation of the $PipeSim$ (see § 4.3.2) function for the simulation of the Pentium pipeline. The execution time of a sequence of instructions is estimated by modeling the five stages of the Pentium pipeline as a set of resources and representing each instruction as a process that acquires and consumes a subset of these resources. Information about the resource needs of the instructions is obtained through the retargetable assembly language manipulation tool SALTO [1]. The obtained reservation table is an intermediate timing information (see figure below).



Information about **cache-hits and misses** and **branch prediction** that is derived from the instruction cache and branch prediction mechanism analysis is used for the analysis of the pipelining behavior.

Firstly, the *IB-est* associated with a basic block determines the time needed to complete the Instruction-Fetch steps of instructions. As said in § 4.1, *IB-est* data deals with $iblocks$. So, we have to infer instruction presence (or absence) in the instruction cache using the *IB-est* provided by static cache simulation. If the instruction reference is a cache-hit, fetching it takes one clock-cycle. Otherwise, fetching it takes much longer (see $T1$ and $T2$ in the PF stage of the reservation table in the above figure.

Lastly, *CTI-est* are used to determine the execution time of CT-instructions. When a CT-instruction is **not** correctly predicted, its execution on a Pentium architecture takes four additional clock-cycles in order to take into account the induced pipeline flush (see $T3$ in the EX stage of the above figure).

Then, the length of the reservation table is the result of the $PipeSim$ function introduced in § 4.3 (*i.e.* WCET of the studied basic block).

The method we use to take inter basic block pipelining effects into account is detailled in [3].

7. Experimental results

7.1. Quantitative results

This paragraph gives quantitative results obtained through static analysis for a set of benchmarks. These results are compared with the experimental evaluations collected during actual executions of the benchmark executables. This experiment has been performed using the "SNU real-time benchmarks suite", provided by SNU RTOS Lab, that covers a wide algorithmic domain (see description below).

Name	Code size	Fct nbr	Description
bs	934	2	binary search program
crc	2223	3	CRC (Cyclic Redundancy Check) example program
fft1	4693	6	FFT (Fast Fourier Transform) Cooly-Turkey algorithm
fibcall	599	2	Fibonacci series function
inssort	753	1	Insertion sort
jfdctint	2918	2	JPEG integer implementation of the forward DCT
matmul	1663	2	Matrix multiplication
minver	6325	4	Matrix inversion algorithm
qurt	2445	4	Root computation of quadratic equations

Measurements were obtained on code generated for the Pentium processor by the Gnu C compiler, and executed on a Pentium 90MHz processor (P54C stepping B5). The experimental results were obtained on code compiled with no optimization. For each algorithm, we have reproduced its worst-case execution scenario. The Pentium on-chip monitoring features (Pentium event counter registers) allow us to count executed instructions, branch mispredictions and cache misses. Since our analysis does not take into account the data cache and the second pipeline of the Pentium processor, these features were disabled in the measurements. Before starting a benchmark application, the instruction cache and branch target buffer are flushed.

Table 2 gives the results. Columns labeled *"est."* are the results obtained by static analysis and columns labeled *"exec."* are the results obtained by executing the directives.

Column II gives the number of instruction executions. Column III gives the number of instruction-block reads that cause a cache miss, Column IV shows the number of mispredicted conditional branches, and the last column shows timing results as clock cycle numbers (knowing the speed of the processor, these kind of WCET can easily be converted into seconds). The ratio between estimated and actual WCET is never less than 1 (otherwise the analyzer would be incorrect), the average ratio is 1.68. Other results concerning the WCET analysis of operating system code can be found in [4].

8. Conclusion

This paper has presented a modular and retargetable timing analysis framework for the accurate prediction of the worst case execution time of programs on modern microprocessors. It addresses the retargetability issue of WCET analyzers using a modular approach based on well defined data formalisms. Thanks to the formalisms defined, static micro-architecture simulation techniques, which are designed as separate entities, can be embedded into an existing static WCET analysis framework as modules that interact with each other.

The chosen method is a tree-based analysis method that uses a modified timing schema to carry out WCET estimations. The analysis framework we focused on takes three modern architectural features into account: instruction cache, branch prediction and pipeline, to achieve the WCET estimation as accurately as possible. HEPTANE, a prototype built using the above concepts, as well as quantitative results that illustrate the accuracy of the computed WCET estimations have been exposed in this paper.

Concerning our current work, we are working on the use of *partial evaluation* before static WCET analysis (*i.e.* specialization of programs by exploiting partial knowledge of its inputs) to improve the static WCET analysis accuracy. This technique would make it possible to remove some infeasible paths in the syntax tree.

I	II			III			IV			V		
	Executed instr.			Inst. cache misses			Branch mispredictions			WCET		
benchmark	*est.*	*exec.*	ratio	*est.*	*exec.*	ratio	*est.*	*exec.*	ratio	*est.*	*exec.*	ratio
bs	136	134	1.01	7	6	1.17	20	16	1.25	1925	1840	1.05
crc	2906	2601	1.12	29	19	1.53	22	19	1.16	35440	31732	1.12
fft1	19759	14529	1.36	266	163	1.63	166	87	1.9	449852	254441	1.77
fibcall	349	348	1	5	4	1.25	10	9	1.11	8106	8008	1.01
inssort	1761	1760	1	11	8	1.38	53	41	1.29	21609	19307	1.12
jfdctint	17514	8340	2.1	174	69	2.52	44	15	2.94	237510	87000	2.73
matmul	7761	7761	1	19	13	1.46	155	154	1.01	110367	100017	1.1
minver	5681	4814	1.18	170	120	1.42	256	155	1.65	117214	76411	1.53
qurt	3645	1184	3.08	60	37	1.62	58	47	1.23	71892	19691	3.65

Table 2. Analytical and experimental results

Acknowledgements

This work is partially supported by the French Departement of Defense (DGA/DSP), #198C4980031303011. We are grateful to F. Bodin, E. Rohou and A. Seznec for providing the static assembly language analyzer SALTO [1] for this research. We also thank P. Chevochot, D. Decotigny from IRISA and the anonymous reviewers for their comments and helpful suggestions.

References

[1] F. Bodin, E. Rohou, and A. Seznec. Salto: System for assembly-language transformation and optimization. In *Proc. of the Sixth Workshop on Compilers for Parallel Computers*, Dec. 1996.

[2] A. Colin and I. Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2-3):249–274, May 2000.

[3] A. Colin and I. Puaut. A modular and retargetable framework for tree-based wcet analysis. Technical Report 1386, IRISA, March 2001.

[4] A. Colin and I. Puaut. Worst-case execution time analysis of the rtems real-time operating system. In *Proc of the 13th Euromicro Conference on Real-Time Systems*, June 2001.

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

[6] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson. Towards industry strength worst-case exection time analysis. Technical Report 99/02, Department of Computer Systems, Uppsala University, Uppsala, Sweden, 1999.

[7] A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of executing time. In *Proc. Euro-Par'97 Parallel Processing*, volume 986 of *Lecture Notes in Computer Sciences*, pages 1298–1307. Springer-Verlag, Aug. 1997.

[8] C. Healy, R. Arnold, F. Müller, D. Whalley, and M. Harmon. Bounding pipeline and instruction cache performance. *IEEE Transactions on Computers*, 48(1), Jan. 1999.

[9] S.-K. Kim, S. L. Min, and R. Ha. Efficient worst case timing analysis of data caching. In *Proceedings of the 1996 Real-Time technology and Applications Symposium*, pages 230–240. IEEE Computer Society Press, June 1996.

[10] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In R. Gerber and T. Marlowe, editors, *ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems*, volume 30 of *ACM SIGPLAN Notices*, pages 88–98, New York, NY, USA, Nov. 1995. ACM Press.

[11] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: Beyond direct mapped instruction cache. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS96)*, pages 254–263. IEEE, IEEE Computer Society Press, Dec. 1996.

[12] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S.-M. Moon, and C.-S. Kim. An accurate worst case timing analysis for RISC processors. In *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS94)*, pages 97–108, Dec. 1994.

[13] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems*, 18(2):217–247, May 2000.

[14] G. Ottosson and M. Sjödin. Worst-case execution time analysis for modern hardware architectures. In *ACM SIGPLAN Workshop on Languages, Compilers, and Tools Support for Real-Time Systems (LCTRTS'97)*, June 1997.

[15] C. Y. Park. Predicting program execution times by analyzing static and dynamic program paths. *Real-Time Systems*, 5(1):31–62, 1993.

[16] C. Y. Park and A. C. Shaw. Experiments with a program timing tool based on source-level timing schema. In *IEEE Computer*, pages 48–57, May 1991.

[17] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2-3):115–128, May 2000. Guest Editorial.

[18] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, Sept. 1989.

[19] P. Puschner and A. V. Schedl. Computing maximum task execution times – a graph based approach. In *Proc. of IEEE Real-Time Systems Symposium*, volume 13, pages 67–91. Kluwer Academic Publishers, 1997.

[20] F. Stappert and P. Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46(4):339–355, 2000.