A structural model for WCET estimation of Simple Out-of-Order Superscalar Processor

Robin Schmutz, Karine Brifault, François Bodin

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A structural model for WCET estimation of Simple Out-of-Order Superscalar Processor

Robin Schmutz and Karine Brifault and François Bodin

N°????

Septembre 2006

Systèmes numériques

ISSN 0249-6399

A structural model for WCET estimation of Simple Out-of-Order Superscalar Processor

Robin Schmutz * and Karine Brifault † and François Bodin ‡

Systèmes numériques
Projet Projet Caps

Rapport de recherche n■???? — Septembre 2006 — 22 pages

Abstract: In the field of hard real time systems, there are two existing techniques to determine the worst-case execution time (WCET). The first techniques are based on measurements, and produce timing estimates, while the second ones consists in static analysis. However, both techniques are often inadequate to produce tight WCET estimates. As the predictions obtained through measurement techniques are usually not guaranteed, the measurement based estimates can be wrong and, for static models, there is a lack of correspondence between those models and the given architectures. As a consequence, WCET estimates are often grossly overestimated.
In this article, we provide a structural model for RISC superscalar microprocessors with out-of-order execution. Our goal is to establish this closer correspondence between the model and the architecture for the WCET calculation. To that effect, we have elaborated a validation methodology for the structural model which allows us to identify and quantify phenomena that produce deviations in order to refine the model with constraints.

Key-words: WCET, static analysis, processor architecture, structural model

(Résumé : tsvp)

* robin.schmutz@irisa.fr
† karine.brifault@irisa.fr
‡ francois.bodin@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Un modèle structurel pour une estimation WCET sur un processeur superscalaire avec exécution Out-of-Order

Résumé : Dans le cadre des systèmes Temps-Réel, il existe deux techniques pour déterminer l'exécution pire-cas (WCET : worst-case execution time). La première, méthode dynamique, est basée sur les mesures et produit des estimations de temps, tandis que la seconde, méthode statique, consiste en une analyse statique. Cependant, les deux techniques sont souvent inadaptées pour produire une estimation WCET judicieuse. En effet, les prédictions obtenues à travers ces deux techniques ne sont généralement pas garanties. En effet, les estimations dynamiques dépendent des jeux d'entrées dont on ne peut être totalement certain qu'ils conduisent au temps d'exécution le plus long. Et, pour les méthodes statiques, il existe souvent un manque de correspondance entre les modèles et les architectures cibles. Comme conséquence, les estimations WCET sont souvent surestimées.
Dans cet article, nous fournissons un modèle structurel pour des microprocesseurs superscalaires RISC avec une exécution dans le désordre. Notre but est d'établir une forte correspondance entre le modèle et l'architecture pour le calcul du WCET. A cet effet, nous avons élaboré une méthodologie de validation pour le modèle structurel qui nous permet d'identifier et de quantifier les phénomènes qui produisent les déviations dans le but d'affiner le modèle avec des contraintes.

Mots-clé : WCET, analyse statique, architecture des processeurs, modèle structurel

1 Introduction

Measuring has traditionally been used to validate timing properties of real-time systems. However, deriving safe upper bounds of response times by measurements, or any other dynamic method, is usually not possible, as the input data is not always known and the number of possible execution paths can be huge. Besides, there is no proof that the maximal observed execution time is the Worse-Case Execution Time (WCET).

For this reason, researches have proposed static analysis for WCET estimate on architectures without caches, complex pipelines, nor branch mechanisms [6, 3, 18], on superscalar pipelined processors with an in-order [14, 13, 8, 19] or out-of-order [5, 4, 22] execution. Those methods do not rely on executing code on real hardware. They rather analyze the set of possible control flow paths for a task code, combining control flow with some abstract model of the hardware architecture, and obtaining upper bounds for this combination.

However, the current state of the art fails to take into account many interacting dynamic features that can be found in modern micro-architectures, such as cache hits/misses, pipeline stalls due to hazards, aliasing, or branch prediction. As a consequence, the obtained WCET on pipelined processors with out-of-order execution is often overestimated to account for this kind of factors.

Recent researches have recognized this lack of correspondence between the static models and the architectures, and have taken into account the timing effects of those processors by simulating the execution of the task in an abstract model of the given processor [17, 23, 11] using static methods. But, those approaches [17, 23] make the model highly dependent on a given architecture, and some of those effects are only due to the specific hardware used. Besides, the simulation does not reveal the causes to the timing effects but implements one execution configuration. As a consequence, uncertain events cannot be taken into account. For instance, if a memory access can be a hit or a miss the two alternatives have to be explored as this has been shown by the work on timing anomalies.

Complex processors are problematic when dealing with real time constraints. On one hand, there is no complete formal description of an existing superscalar processor that would allow to prove that a WCET estimation is safe. On the other hand, the need for computing power in embedded systems involves it difficult to avoid this kind of technology.

Considering this, the work presented in this paper is twofold. First, we want to construct a processor model, accurate and simple enough to be easily implemented from the processor documentation and whose behavior are easily understood. As it is usually not possible, the second objective is to find out the cases where this kind of model can be defined. Besides, since it is not able to completely model the processor, the input code must be restricted.

The proposal model is not a simulator and does not need to run the full code in order to get a timing estimate for an sequence of instructions. It is a partitioned model for WCET and the long term goal would be to base the static analysis on testing and measurements to improve the actual WCET techniques.

Many previous researches such as cache locking [20], allow to restrict the behavior so that no feature not handled by the model do not appear in the input code. In our case, to properly defined the model and its limits, we propose a methodology that establish a tradeoff between the model extension and limits. When a feature cannot be taken into account it is then expressed as a limit of the model. And,

we show that this approach can handle non trivial input code and that it helps to define guidelines which conduct to generate predictable codes.
This paper is organized as follows. In Section 2 and 3, we survey the background and related works. Section 4 presents the structural model and describes an implementation of the proposed approach. In Section 5, we report the validation methodology. Section 6 gives a new exploration technique to find and explain existing timing effects for an out-of-order superscalar processor. Finally, Section 7 gives our conclusions.

2 Background

The main difficulty in studying a superscalar processor with an out-of-order execution resides in capturing the pipeline behavior, because of the instruction dependence on the functional units, and the possibilities of instruction schedulings which increase the complexity of WCET calculation. Before quantifying the interactions that have a real impact in the WCET determination, we describe relevant concepts in the next paragraphs.

2.1 Estimation contexts

A context, in the framework of this article, is a set of information about the states of the architectural features which affects the code execution. For instance, by default, the proposal model uses an estimation context which assumes the accesses of the D-cache, I-cache and Memory Management Unit (MMU) as hits[1], and the branch processing unit as never mispredicted. It is to be noted that the context definition and its use allow to have a predictable code.
The interest of contexts is double-barreled: a context modification allows to define a different instruction scheduling with a same code sequence, and several contexts make possible the exploration of instructions without any risk of an exponential increase of execution paths.

2.2 Timing effects

Given an estimation context, a symbolic execution of a program or a code sequence gives a timing execution T. A timing effect is the result of a behavior within the architecture which reflects on this execution time T. It is characterized by an execution time delay, positive or negative. The positive delay results in the improvement of execution time, and the negative one in its reduction.
This behavior only appears when several conditions are satisfied, and may never occur in the course of one, or even multiple executions. For this reason, the influence of timing effects are difficult to apprehend.
The timing effects make difficult the estimate of an upper bound field because of the lack of exact state information. In the remainder of the paper, we show some specific timing effects and their impacts on WCET.

[1] This can be achieved by cache locking

2.3 Out-of-order pipeline

The idea behind pipelining is to exploit the instruction level parallelism by overlapping the execution of instructions, e.g. by dividing them into several steps. In the ideal case, every pipeline stage processes one instruction step, and the pipeline has to be stalled occasionally because of data dependences, resource conflicts or control flow changes. To reduce the pipeline bubbles, the pipeline can have an out-of-order execution which consists in scheduling instructions for execution in an order different from the original program order. In such a processor, an instruction step can be executed if its operands are ready and the needed functional unit is available. The processor performance is significantly improved by out-of-order execution as the pipeline stalls can be replaced by useful computations. However, this kind of execution exhibits many phenomena we discuss below.

2.4 Pipeline overlap

In the last decade, static solutions have been proposed to analyze modern processors in the presence of caches and simple in-order pipelines [8], in which several processes may run concurrently and alterate the cache state predictability. Some researches have suggested to use a single linear program to predict this kind of behavior [14]. This method, extended IPET [10], highlighted the interferences between two successive basic blocks and the consequences of pipeline overlap in the WCET estimate. To understand these interactions, Figure 1 presents an example of pairwise timing effect found in article [5].

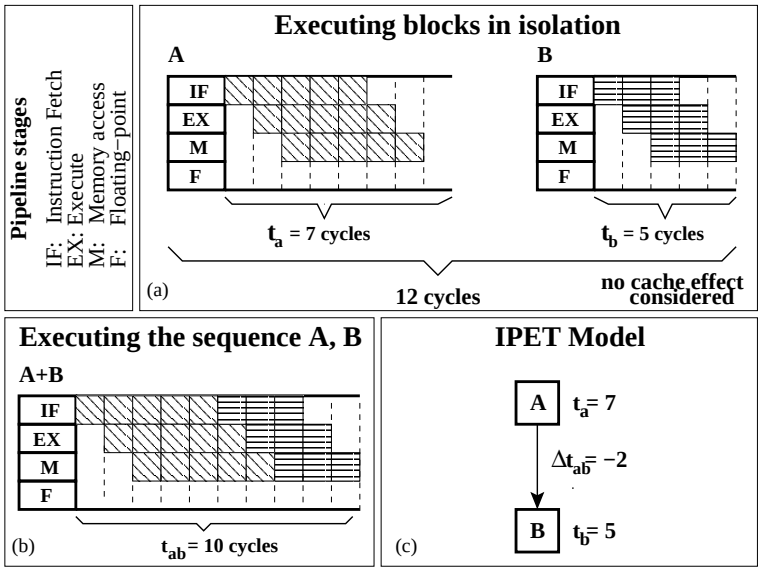

Figure 1: **Negative pairwise timing effect**

In this case, a safe (e.g. no underestimation of the execution time), but not tight, estimation of WCET would result from calculating each block independently and adding their execution times. For instance, the sum of execution time for instructions A and B is $t_a + t_b = 12$ cycles (Figure 1(a)).
With this technique, the WCET could be over-estimated depending on the pipeline overlap. In the same example, the pipeline stages can process B at the fifth cycle instead of waiting the end of A. Hence, the adequate WCET estimation would be $t_{ab} = t_{a\_not\_finished} + t_b = 10$ cycles (Figure 1(b)). The difference of $|\Delta t_{ab}| = 2$ cycles between the WCET estimation and the real execution time is a negative timing effect which decreases the WCET estimate (Figure 1(c)).
However, this method is not good enough to determine a safe WCET estimate for all interferences on pipelined processors. Indeed, J. Engblom [5] has highlighted that some temporal interactions may also exist between distant basic blocks. Besides, he has shown that those effects could occur in sequences of any length and that they could be either negative, null, or positive. He has devised a new analysis in [4] to take into account those temporal interactions named "long timing effects" (Figure 2). Hence, to model long timing effects when applying the IPET method (Figure 2(b)), we should add some weighted edge between non-adjacent blocks (e.g. Δt_{abc}).

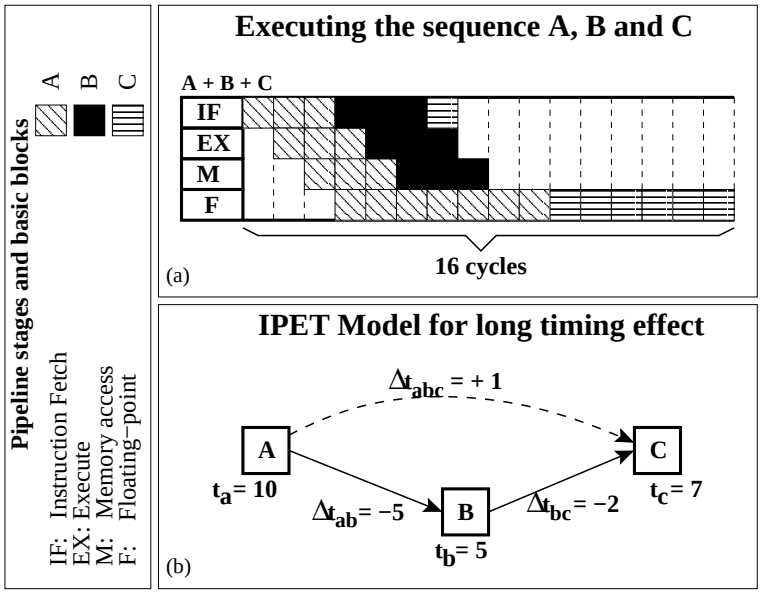

Figure 2: **Positive long timing effect**

The two examples above have shown the role of the pipeline in the accuracy of the WCET determination and the prevention of resource wasting from a defined scheduling. But, we should also wonder if there are timing effects dependent of out-of-order schedulings.

2.5 Timing anomalies

Lundqvist and Stenström [15] were the first to present an approach for obtaining WCET bounds through simulation of the pipeline, and to highlight a problematic timing behavior, a timing anomaly [17], in modern processor hardware with an out-of-order execution. A timing anomaly is characterized as a situation where a positive (negative) change of the latency of an instruction by i cycles results in a global decrease (increase) of the execution time of a sequence of instructions [21]. Indeed, in an instruction sequence where each instruction has a corresponding latency for its associated functional unit, this latency can change (increase or decrease) depending on the dynamic instruction schedule. For instance, in an instruction sequence, a cache miss is usually considered as allowing a conservative estimate of the WCET, and if the outcome of a cache access is unknown, a cache miss is then assumed. But this assumption could be wrong. To understand the consequences of the execution scheduling on the execution time in the case of cache hit/miss, Figure 3 presents an example of timing anomaly found in article [17].

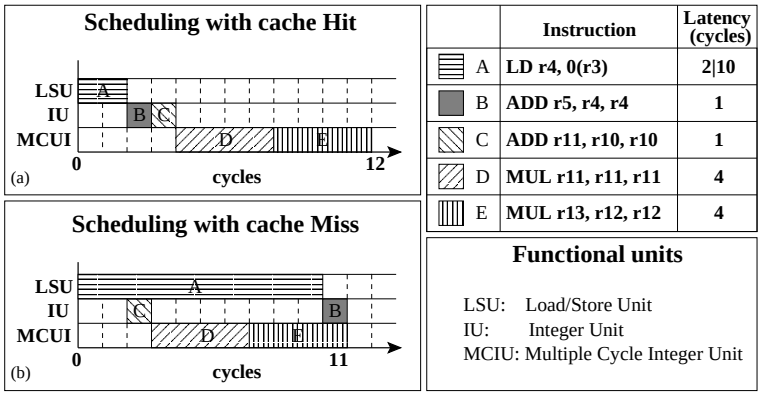

Figure 3: **Timing anomaly**

In this example, we can see that if the load address hits in the cache (Figure 3(a)), the sequential execution scheduling is used and the B instruction is processed before instructions C and D. On the contrary, if the load address misses (Figure 3(b)), as the B instruction is dependent on the A result, it must be postponed until A is finished, and C can be executed one cycle earlier than in the cache hit case. In the end, the instruction sequence is faster in the case of a cache miss.
We highlight, in Figure 3, a key problem of the timing anomalies: those timing effects depend on instruction scheduling, they are highly architecture-dependent [23] and they are counter-intuitive.
So, in order to choose an instruction scheduling allowing a safe and accurate WCET estimate, we have to analyze the effect of all possible schedules (those exploring the different hit/miss scenarios). Unfortunately, this technique could prove to be too time consuming.
The basic idea behind our approach consists in defining an instruction scheduling for basic blocks, in a frozen context, which could be considered as the standard scheduling for the abstract model.

2.6 Timing uncertainties

We have explained two kind of timing effects above - long timing effects and timing anomalies - in order to show that uncertain factors, such as resource conflicts, or data dependences, generate those effects by affecting delays in execution.
In the remainder of this article, we focus on a structural model of out-of-order pipelined processor which gives a symbolic timing execution independent on the data. However, some resources have variable-latencies dependent on the input data, such as cache hit/miss, branch prediction, or data misalignment, which are responsible of deviations in WCET estimate by the model. For this reason, we assume an execution context on the resources and we define a timing uncertainty as a kind of timing effects due to instruction which latency depends on runtime data. It emerges at execution-time when some data-dependent factors appear. It is to be noted that timing uncertainties can be sources of timing anomalies and long timing effects.
In a code sequence, if there is timing uncertainties many scheduling need to be explored to capture the WCET.
One of our goals is to investigate the common principle behind those uncertainties in order to determine their causes and occurrences.

2.7 Canonical execution

The canonical process of an architecture represents the foreseeable behavior of this architecture, e.g. the behavior when no perturbation appears.
In this article, the canonical execution is defined when no timing uncertainty can occur. To that effect, we have to give a default context which fixes all data-dependent factors such as hit/miss cache, and a standard scheduling which avoids all dynamic-latencies.

3 Related work

Recent researches have modeled and analyzed several micro-architectural features such as pipeline in modern processors [2, 5, 7].
Lundqvist and Stenström [16] have combined instruction level simulation with path analysis by allowing execution of instructions. In order to delete unfeasible paths they have introduced *unknown values*, and the simulator is able to handle programs even if the input values are not known. But, several problems have appeared such as the unknown values for memory addresses which lead to non-termination of the simulation. Schneider and Ferdinand [24] have applied semantics based approach to model superscalar processors, and are presented an implementation for SuperSPARC I processor. Heckmann and al. [9] have employed an abstract interpretation to cache and pipeline analysis on an out-of-order processor, the PowerPC 755.
Li and al. [12] have presented a structural model of out-of-order pipeline close to our conception. Their goal is to study the pipeline in modern processors and different possible schedules for a WCET analysis without performing an exhaustive enumeration of pipeline schedules. To that effect, they have created contexts allowing to define several feasible schedules.

However, there are some divergences between their proposition and ours. First, their context is defined in term of instructions which are preceding and succeeding a basic block B. Its effect is to determine the pipeline stalls due to data dependences or resource contentions. In our proposition, a context is a set of information about the states of the architectural features which affects an instruction sequence such as B. For instance, in B, the state of a Load can be defined as a hit or a miss. By this means, we take into account data dependences, resource contentions and timing uncertainties.
Second, the implementation of their model has been made in the SimpleScalar simulator and they have developed an integer linear programming to estimate the WCET of a basic block. In our case, we have chosen to implement the structural model in a real architecture, the Freescale PowerPC 7450. The WCET estimate is determined by the symbolic execution in absence of any knowledge of input data, by augmentation of the instruction semantics to also handle timing uncertainties instead of deriving it via a path analysis. The verification of the results have made by comparison to the real hardware. Besides, we have a methodology to validate the model and to find the occurrences and the causes of timing uncertainties.

4 Architectural model

This section presents a new approach to the exploration of instruction schedulings and timing uncertainties, using the design of a cycle-accurate structural model for out-of-order superscalar architectures. The first part of the section presents the main principles of the abstract model, the second part gives its working rules, and the third part introduces an implementation of this model on the Freescale PowerPC 7450.

4.1 Main principles

The key idea of our approach is to establish a closer correspondence between static WCET analysis and complex hardware. This bond is modelized through a structural cycle-accurate model[2] for RISC superscalar pipelined microprocessors with out-of-order execution.
This model is both a temporal simulation framework and an instruction set module for processors, dissociating the hardware components from the model, allowing us to reuse or replace them at will. Besides, as it is dedicated to perform the architecture behavior of a superscalar pipelined processors with out-of-order execution, it can model all other processors which structure is similar or less complex. However, at this date, it performs the architecture behavior analysis as part of a partitioned WCET analysis.
This model has three main purposes. The first goal is to take into account most of the current architectural features found in pipelines, thus allowing to give a temporal simulation of instruction sequences. However, since it is not feasible to exhaustively catch every architectural properties, we must consider a canonical behavior of the architecture. During this stage a frozen context is used to highlight every deviation between the model and the architecture, the timing uncertainties.

[2] Fig. 4 presents the model for PowerPC 7450.

The second objective is the possibility to generate different schedulings from an instruction sequence. And the third one is the possibility to give a context, such as defining non blocking cache accesses either as hits or misses, allowing us, for example, to highlight timing anomalies.
In opposition to architecture simulators like SimpleScalar, which are data dependent, this structural model symbolically executes instructions (with worst-case latencies). It offers the possibility of forcing the model in any desired architectural state to explore various instruction schedulings on various contexts, and to study the impact of several timing uncertainties on the overall execution time.
The next part is dedicated to the working rules of the model.

4.2 Instruction flow

The structural model is composed of several pipeline stages, each with associated attributes: current capacity, maximum capacity and issue rate per cycle. This information is directly deduced from the architectural description of technical manuals [1].
Instructions also have associated attributes corresponding to information about the authorized execution units, the instruction worst-case latency, the current position in the pipeline and the state at the last "executed" cycle. They are iteratively processed from a stage to another until all of them have crossed the pipeline. These translations depend on the pipeline flow and the availability of functional units. The instructions are stalled or moved to the next stage at each cycle step. The simulation stops when the last instruction goes out of the last pipeline stage.
This model examines, by a bottom-up traversal strategy, the pipeline stages in order to know the states of instruction properties. The goal is to catch inter-stage latch synchronization. and its symbolic execution consists in focusing on the input flow (basic blocks or instructions lists) and its execution speed, in cycles, for performance analysis. To that effect, a "scheduler" unit, the scoreboard, has been added in the wide-execution core of the model to operate the dynamic scheduling.
When a target architecture is given, all the information about its instruction set, its pipeline degree, its Load/Store Queue design and the latencies are available, thus benchmark results can be precisely traced.
In order to give an accurate WCET estimate, we compare executions on the target processor with the same contexts used by the structural model to identify and locate timing uncertainties. Besides, we use hardware counters to determine their causes.
More details on the implementation of our model for a given architecture are exposed in the next section.

4.3 Case study: the PowerPC 7450

The structural model implements a 32-Bit RISC PowerPC architecture, the PowerPC 7450 (also known as 'G4e'), chosen for its highly modular architectural model and its quite complete documentation. This model is composed of a Floating-Point Unit (FPU), a complex Integer Unit (IU2), three simple integer units (UI1) and a Load/Store Unit (LSU). The vector units are not implemented in the model yet.
Each execution unit has a particular number of stages required to complete an instruction and, in most cases, each stage uses a single cycle. Therefore, the number of cycles and the number of

stages are equal, but some exceptions exist such as the integer divide instruction. Of course, if the execution latency is greater than the unit pipeline degree, the unit is frozen during the adequate time. Figure 4 gives an implementation of the Freescale PPC7450 pipeline in the model, where each block represents a pipeline stage and its max capacity and the arrows are the outflow capacities of each stage.
This PowerPC 7450 pipelined model is separated in two parts: a front-end and a wide multiple-issue execution core where three instructions can be dispatched and completed per cycle.
The execution core contains a “scheduler” unit, the scoreboard, where the dynamic scheduling is operated, depending on data dependences (RAW only) and structural hazards. This process operates at no timing cost and allows to define several scheduling policies, as we see below.
After the instructions have been processed through the scheduler, they are stacked in one of the reservation stations where they are waiting to be symbolically executed by one of the functional execution units.
In the structural model, the Finished Store Queue is implemented for the LSU, but the Completed Store Queue (CSQ) and the Load Miss Queue are not. Similarly, the Branch Prediction Unit (BPU) and reorder buffer (ROB) are not modelized, neither is the Memory Management Unit (MMU), and its accesses are always hits. Besides, the memory hierarchy is not modelized but hit/miss events are taken into account thanks to a definition of contexts (see below for more information). Finally, for the registers sets, we assume that the processor has infinite register resources.
After the symbolic execution, the instructions are placed in the Completion Queue to wait for their in-order completion and decrease the execution window size.
The structural model does not define all the features of a microprocessor. Indeed, to modelize and simulate perfectly a given architecture is quite difficult and is not needed here. We have hence chosen to model a subset of the architecture features and, to reduce the interferences of the not-implemented features we have defined the necessary conditions, with constraints, allowing us to create a predictable code according to the model. We call these constraints “model limits”.
The next part shows how we can refine the model limits by discovering the hidden ones, and how we can guarantee the efficiency of the model under these limits.

5 Model Validation

A structural model does not reproduce all the features of the architecture (register ports implementation, result bus...) and can only work under given limits of correct operation (e.g. no exception appears), in conformity with the hardware results. Since those features are implicitly captured by the model, we compare it with the target architecture to discover hidden constraints we have made. From this point, either the model is tighten with additional flow rules either the model limits are augmented. Those limits lead to generate a predictable code. For unpredictable events that cannot be caught by a code modification, e.g. the timing uncertainties, a study case must be performed to quantify their impact on execution time.
The first part introduces the methodology used to quantify the deviation between the model and the real architecture, the second one gives the results of the model calibration, e.g. defines new model limits, and the third is the verification of the model when using predictable code.

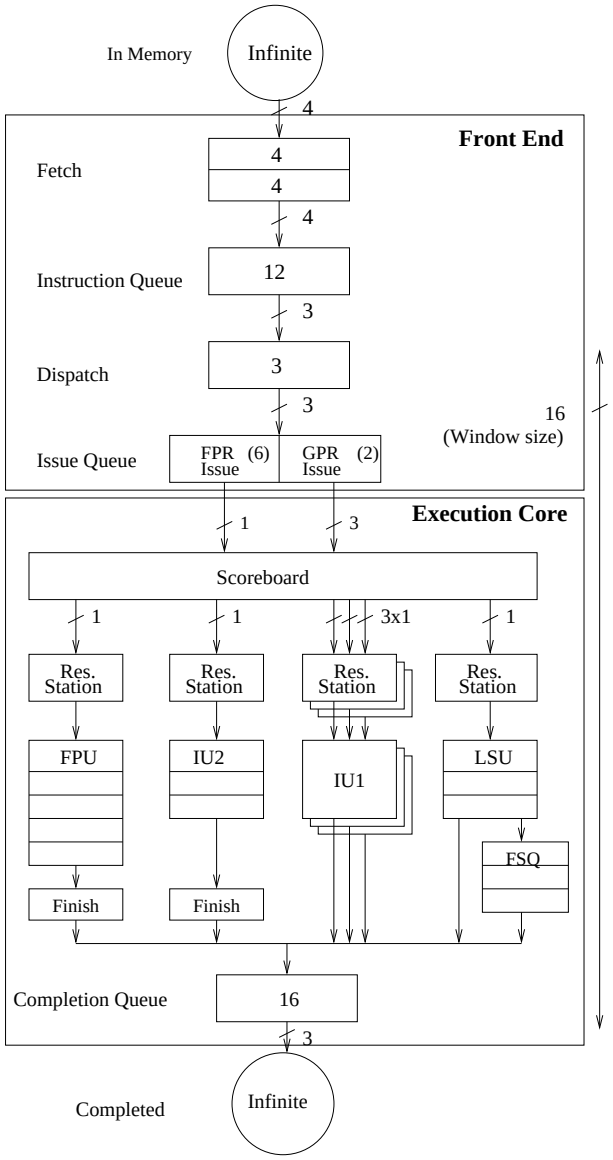

Figure 4: **Our PPC7450 pipelined model**

5.1 Methodology

The benchmark programs, specific basic blocks of a thousand instructions, oppose the WCET estimate given by the structural model defined in Section 4.3 to the execution time measured by dynamic evaluation on a PowerPC 7450 microprocessor.

These tests are intended to compare this structural model with the intrinsic pipeline architecture, hence some uncertainty factors, such as caches and branch predictor behaviors, must be turned predictable. To that effect, contexts have to be defined.
By default, we insure a frozen execution context for dynamic evaluation of benchmarks: the accesses of the D-cache, I-cache and MMU are forced as hits and the branch processing unit is never mispredicted. Besides, to initialize cache memory and branch predictor states, a cold start execution is made before each test.
The hardware measures are based on 100 000 iterations of the tested basic blocks. We used `isync` instructions to initialize the pipeline state, and a synchronization is executed before each basic block. It ensures the entire completion of the previous basic block to prevent them from overlapping in the pipeline. In our experiments, the synchronization does not incur any penalty as we only consider the results for each single basic block.
For each code sequence, we compare the Instruction Per Cycle (IPC) estimated by the model with those given by the microprocessor. We also calculate the ratio of the cycles needed to execute the same code sequence by the two parts. The formula of the ratio is:

$$ratio = \frac{Cycles_{model}}{CPI_{hardware} * instructions_{model}}$$

where the definition of CPI is Cycles per Instruction. We do not use directly the cycles given by the hardware to absorb the instruction overhead due to the control of the tested function (which adds no more than 10 instructions, e.g. less than one percent of the tested instructions).
The hardware measures, the number of executed instructions and elapsed cycles, are given by the performance counters located inside the chip in order to precisely profile the hardware behavior. With this profiling, we insure that the execution context is respected. For instance, we verify that cache accesses never misses or that no rename register contention occurs. We also enable to detect and identify uncertainty factors when a deviation between a model estimate and execution is observed, such as significant differences in schedulings.
The next parts describe the whole benchmarking process.

5.2 Model Calibration

The model calibration is dedicated to construct a list of each perturbing event that could involve a difference between model and architecture behaviors in order to refine the model with new model limits and flow rules.
To that effect, we have made series of tests to stress the functional units. Those code sequences are composed of a unique kind of instruction (such as `add` or `div` for FPU) which are, in one case, in sequential code with data dependences (RAW) and, in the other, in parallel execution. The IPCs of the model and the architecture are compared to detect a deviation in order to find the perturbing events. Indeed, the deviation between the model and the given architecture implies that the responsible perturbing event is not catchable by the model. Hence, its cause has to be avoided by defining more limits which guarantee the accuracy of the model.

The first model limits are linked to the implementation defects in the model. For instance, using consecutive store instructions must be avoided since the CSQ (Completed Store Queue) is not implemented.
The other limits or rules are deduced from the tests. With this benchmark, on the PowerPC 7450 architecture, some events have been highlighted:

- *Dimensioning the model:* the focus on the functional unit allows to find some differences between the documentation and the target architecture manual. For instance, the floating-point instructions take five cycles to complete. But, one rarely discussed weakness of the G4e's FPU is the fact that it is not fully pipelined, which means that it cannot have five different instructions in five different stages of execution simultaneously. Indeed, the FPU's peak theoretical instruction throughput is four instructions every five cycles. Another example has shown that the address determination in the LSU uses the IU resources for the calculation on the PowerPC 7450 architecture.
- *Identifying uncertainties:* some address aliasings can occur between a load instruction close to a store instruction pointing at the same address increasing the latency estimate of the structural model. But, for the structural model, the rules give that the load must wait the store to complete before fetching the pointed data. Some dead-codes can also be removed at execution (a useless write in a condition register) leading a high divergence between executed and estimated time. Lastly, a load with a cache hit can have its latency increased due to a misaligned address.

This structural model has allowed us to show this kind of architectural limitations or specificities, and to determine adequate rules to take them into account.
The next part shows the verification of the model when using predictable code.

5.3 Model Verification

The goal of this part is to verify that the model matches up the target architecture, with the model limits defined with the first series of tests, by examining every functional unit on various situations.
We focus on the model verification to assume a canonical behavior of the architecture. To that effect, the second series of tests are C code sequences randomly generated in order to use every C operators on integer and float data with possibilities of arrays. A frozen execution context prevents timing uncertainties due to cache and branch prediction unit to occur. The hardware counters are used for the verification of the accuracy of the model limits.

$$
\begin{array}{rcl}
tabf_c[i_d] & = & ((f_h - f_f) + (tabf_b[i_a] * f_h)); \\
f_f & = & f_c + ((tabf_e[i_a + 2] + tabf_a[i_a + 2])/(f_h - f_f)); \\
f_d & = & f_a + ((tabf_f[i_a] * f_d) - tabf_e[i_a + 2] + tabf_a[i_a + 2])); \\
tabf_g[i_h] & = & ((f_d - f_b + (tabf_f[i_a] * f_d));
\end{array}
$$

Figure 5: An example of a C code sequence

Figure 5 presents an example of a randomly generated C code sequence using float arrays and data (such as $tabf_d$ and f_h), and integer data to index arrays (i_a). We have tested 200 different code

sequences (of about 1000 assembly instructions) classified in four categories (so each is tested with 50 sequences):

- *Int Simple*: Sequences using only integer data with simple C operators $(" + ", " - ", "|", "\&")$.
- *Int Complex*: Sequences using only integer data with complex C operators $(" + ", " - ", " * ", "/")$.
- *Float Simple*: Sequences using float data and integer data (for array and address accesses only) with simple C operators $(" + ", " - ")$.
- *Float Complex*: Sequences using float and integer data with simple C operators $(" + ", " - ", " * ", "/")$.

Each code sequence has been tested with the "Late" scheduling policy which gives priority to the latest instruction, and the "Early" policy which gives priority to the earliest instruction. The values below are given by the IPC estimated by the model with those given by the microprocessor (as defined in Section 4.1).

Code Sequences	"Late" rule	"Early" rule
Int Simple	1.030	1.114
Int Complex	1.303	1.358
Float Simple	1.136	1.147
Float Complex	1.352	1.363

First, the results show that it is possible to give very tight estimates of the WCET while using simple operators. Indeed, the use of complex instructions with variable latencies avoided. With complex operators, the estimated WCET is far from the average-case execution time, but it highly depends on the proportion of complex instructions (such as `fdivs`).
Besides, when the code respects the previously listed limits with a scheduling policy giving the priority to the oldest instructions, we obtain a cycle ratio contained between 1.10 and 1.30. It shows that the structural model has the potential to create safe and tight WCET estimations.
Moreover, when the scoreboard policy gives the priority to the newest instructions, we obtain a cycle ratio contained between 1.25 and 1.50. It shows that this model has the potential to create efficient cycle accurate scheduling estimations.
To conclude, those benchmarks reveals the potential of this model to give safe and tight WCET estimates with accurate schedulings for out-of-order superscalar architectures when using a predictable code. But the impact of the various timing uncertainties has not yet been taken into account and is analyzed in the next section.
In the previous part, we have shown how this structural model could capture complex pipelines behaviors. Depending on the required level of accuracy, every architectural feature could be taken into account from pipelines to register files, thanks to flow constraints. From this level of accuracy of the architecture modeling, explicit assumptions of good working are deduced and lead to create a predictable code. Using this constrained code, and placed in a safe execution context, i.e. no timing uncertainty is allowed from outside the pipeline, we have shown that this model enables to estimate efficient worst-case timings and accurate schedulings.

6 Toward a safe WCET

The model focuses on the processor's pipeline itself and does not model other performance architectural features like the branch predictor and the cache levels. However, in the previous part, we have shown how the structural model captures complex pipelines behaviors. Depending on the required level of accuracy, explicit limits of correct operations are deduced and lead to create, by contexts, a predictable code, e.g. no timing uncertainty is allowed from outside the pipeline. And the model placed in this safe execution context enables to estimate efficient worst-case timing and accurate schedulings.
The long term objective is to generate a fully predictable code associated with a worst-case reference scheduling. Indeed, rather than trying to estimate WCETs by taking timing effects into account, we prefer giving a safe WCET estimate and try to disable any perturbing effect, e.g. a positive timing effect. This safe WCET estimate is given by the worst-case reference scheduling previously computed by disabling every timing uncertainty factor. For now, the goal is to detect every negative timing effect in a code by studying every feasible scheduling resulting of a timing uncertainty. Then, methods to prevent the occurrence of those timing effects could be discussed.
Two major problems in current WCET analysis have been studied: first, the timing uncertainties caused by the dynamic resource allocation [17, 25] and second, the impact of a scheduling modification on the long term, due to a branch flush or a timing effect, which can cause "long timing effects" [5].

6.1 Catching timing effects

Out-of-order execution is the best context to generate timing uncertainties because the scheduler which performs the dynamic resource allocation is free to modify the instructions scheduling at any time, depending on resources or results availability.
Since the model is very cheap in computational power, it is possible to spend time in a recursive search of various possible pipeline states at a given time. This analysis can be performed at any stage of the structural model and exhaustively study the possible schedulings given an instruction interval. The produced schedulings are then compared to the reference scheduling.
The model has allowed to identify two different kind of dynamic timing uncertainties: first, the load latency uncertainties which depend on a L1 cache hit or miss (indeed a L2 miss stalls the pipeline long enough to be considered as a synchronization barrier) or, on an aligned or misaligned address on natural boundaries, and then, the computation latency uncertainties which dynamically depend on data.
We will see how this feature can be used to catch the dynamic timing uncertainties variations and point the timing effects out.

6.1.1 Handling load latency uncertainties

Current static cache analysis can only predict a subset of the load instructions to do a sure hit or miss. Other accesses are categorized as "maybes". On the contrary, the structural model proposes to fork the "simulation" at the LSU stage to study the scheduling resulting of either a L1-D Cache

hit or miss. It is to be noted that, as the current microprocessor designed for performance supports non-blocking caches accesses, the study every situation like “hit-under-miss” is possible.
To understand the interest of the model part, Figure 6 presents an example of a timing effect due to a load latency uncertainty captured by this model.

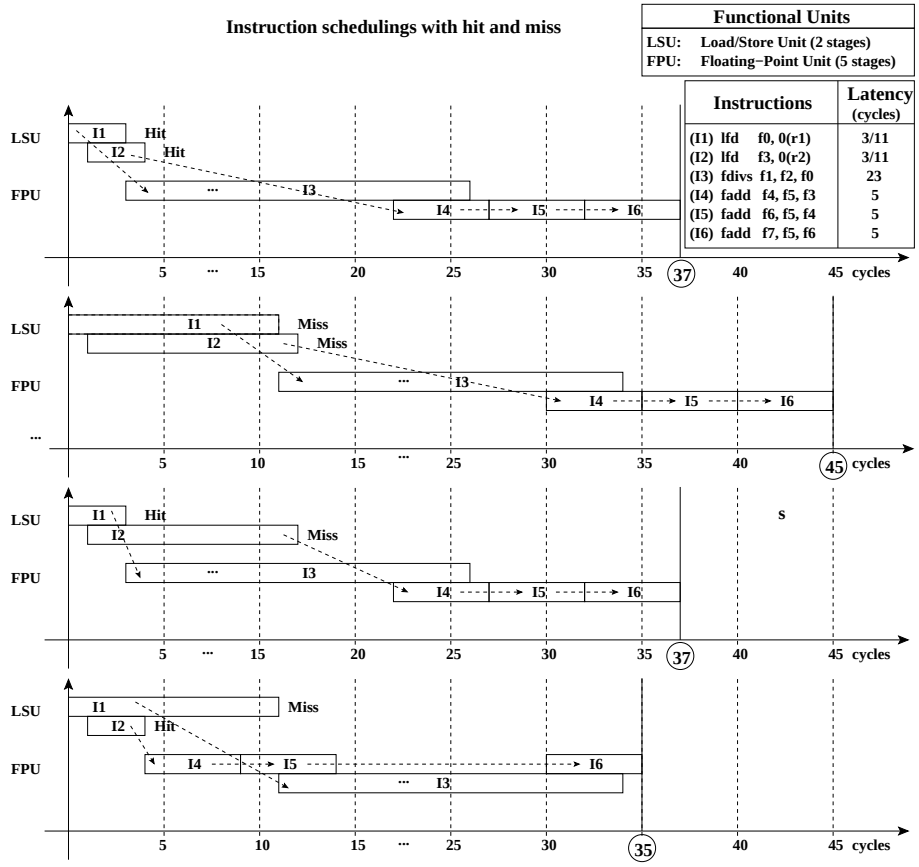

Figure 6: **A cache access leading to a timing anomaly**

In the three first examples, (I4) cannot overlap (I2) until the FPU pipeline is released. On the contrary, the last example shows that (I3) is waiting for the result of (I1) and can not be executed. For that reason, (I4) is launched. But, the scheduling modification resulting gives a timing anomaly [25].
Similarly, the timing uncertainties due to misaligned accesses by considering them as cache misses with shorter latency could be analyzed.

6.1.2 Handling computation latency uncertainties

The execution latency in a multi-cycle unit may depend on the computed data. If the computation is finished before the last stage of the unit pipeline, the result is forwarded to the unit's end. This mechanism also works for the complex computations such as the float division on the PowerPC 7450 architecture. This kind of situation can lead to timing uncertainties, because of the dynamic resource allocation.
The structural model can fork the "simulation" at each decision of both the scheduler and the reservation station stages to study all possible schedulings. At this date, this scheduling analysis has been programmed for every instruction, whether they are source of timing uncertainty or not. However, we plan to trigger this recursive scheduling search only on the presence of variable latency instructions, using the evaluation of the instructions minimal latencies performed in Section 4.2.
Figure 7 presents an example of a timing effect due to a computation latency uncertainty captured by the model.

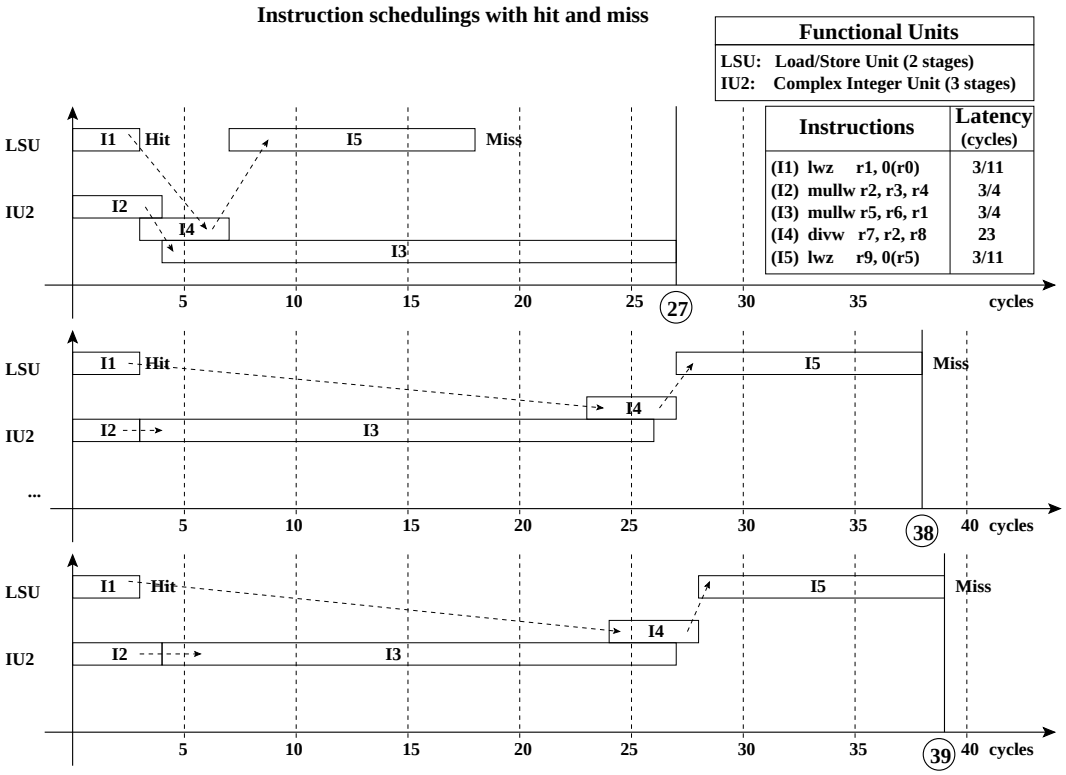

Figure 7: **An execution latency leading to a timing anomaly**

The first example shows the total execution time C of the reference worst-case scheduling. The second example shows how a decrease of (I2) latency lead to a strong impact timing effect. And the last example shows one of the computed scheduling, giving an execution time C', which reveals a timing effect. Indeed, a timing effect is detected since $C \neq C'$.
This part of the analysis will require a deepened study in our future works, taking into account the instructions minimal latency estimated in Section 5.2 to check for the feasibility of the scheduling.

7 Discussion and Future work

In this paper, we have presented a new approach to produce accurate timing estimates and schedulings. It consists in designing a cycle-accurate structural model for RISC superscalar architectures with out-of-order execution.
After that, we have defined a methodology to validate this model by refining it with constraints and rules. The goal of those model limits is to establish a closer correspondence to a given architecture, here the PowerPC 7450. But, it is to be noted that the same methodology can be used with other architectural features.
Finally, we have tested on several codes to estimate efficient worst-case timings and accurate schedulings. We have shown that the model could fit to detect several timing anomalies, which are a major problem in current static WCET analysis, even if it only models a subset of the real features.
The experiments have highlighted that even with basic blocks, the WCET estimates are not always tight and that data misalignments could strongly play against the upper bound by underestimate the real latency. The results have pinpointed that the static methods which do not verify on a target architecture are often not very accurate, because they can forget timing uncertainties such as aliasing, or data misalignments.
Our future works will be to model another architecture with a different design. This new architecture could have some new timing effects and we could refine the validation methodology. Besides, another purpose will be to precisely implement other features, mainly register files and a real register renaming mechanism.

References

[1] *MPC7450 RISC Microprocessor Family Reference Manual*. Freescale Semiconductor, 2005.
[2] U. Chandra and M. G. Harmon. Predictability of program execution times on superscalar pipelined architectures. In *Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems*, pages 104–112, Los Alamitos, CA, USA, April 1995. IEEE Computer Society.
[3] J.-Y. Choi, I. Lee, and I. Kang. Timing analysis of superscalar processor programs using acsr. In *Proceedings of 11th IEEE Workshop on the Real-Time Operating Systems and Software*, pages 63–67, Los Alamitos, CA, USA, May 1994. IEEE Computer Society.
[4] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Dept. of Information Technology, Uppsala University, April 2002.
[5] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications*, page 88, Washington, DC, USA, December 1999. IEEE Computer Society.

[6] M. G. Harmon, T. P. Baker, and D. B. Whalley. A retargetable technique for predicting execution time. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 68–77, Los Alamitos, CA, USA, December 1992. IEEE Computer Society.

[7] C. A. Healy, R. D. Arnold, F. Mueller, M. G. Harmon, and D. B. Walley. Bounding pipeline and instruction cache performance. *IEEE Trans. Comput.*, 48(1):53–70, 1999.

[8] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 288–297, Los Alamitos, CA, USA, December 1995. IEEE Computer Society.

[9] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *IEEE Trans. Comput.*, 91(7):1038– 1054, July 2003.

[10] Y. Hur, Y. H. Bae, S.-S. Lim, S.-K. Kim, B.-D. Rhee, S. L. Min, C. Y. Park, M. Lee, H. Shin, and C. S. Kim. Worst case timing analysis of risc processors: R3000/r3010 case study. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, page 308, Washington, DC, USA, 1995. IEEE Computer Society.

[11] X. Li. *Microarchitecture Modeling for Timing Analysis of Embedded Software*. PhD thesis, Dept. of Computer Science, National University of Singapore, 2005.

[12] X. Li, A. Roychoudhury, and T. Mitra. Modeling out-of-order processors for software timing analysis. In *Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 92–103, Los Alamitos, CA, USA, December 2004. IEEE Computer Society.

[13] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, New York, NY, USA, 1995. ACM Press.

[14] 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. *IEEE Trans. Softw. Eng.*, 21(7):pp. 593–604, 1995.

[15] T. Lundqvist and P. Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 1–15, London, UK, 1998. Springer-Verlag.

[16] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based oncycle-level symbolic execution. *Real-Time Syst.*, 17(2-3):183–207, 1999.

[17] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 12–21, Los Alamitos, CA, USA, December 1999. IEEE Computer Society.

[18] K. D. Nilsen and B. Rygg. Worst-case execution time analysis on modern processors. In *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 20–30, New York, NY, USA, 1995. ACM Press.

[19] G. Ottosson and M. Sjödin. Worst-case execution time analysis for modern hardware architectures. In *Proceedings of ACM SIGPLAN 1997 Workshop on Languages, Compilers & Tools for Real-Time Systems*, New York, NY, USA, June 1997. ACM Press.

[20] I. Puaut. Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems. In *Proceedings of the 2nd Workshop on Worst-Case Execution Time Analysis*, pages 62 – 67, Vienna, Austria, June 2002.

[21] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dresden, Germany, July 2006.

[22] C. Rochange and P. Sainrat. Towards designing wcet-predictable processors. In *Proceedings of the 3rd Workshop on Worst-Case Execution Time Analysis*, pages 87 – 90, Porto, Portugal, July 2003.

[23] J. Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Compiler Research Group, University of Saarlandes, Saarbrücken, Germany, March 2003.

[24] J. Schneider and C. Ferdinand. Pipeline behavior prediction for superscalar processors by abstract interpretation. In *Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 35–44, New York, NY, USA, 1999. ACM Press.

[25] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proceedings of the 5th International Conference on Quality Software*, pages 295–306, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

Contents

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifi que,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399