Modeling Instruction-Level Parallelism for WCET Evaluation

Jonathan Barre, Cédric Landet, Christine Rochange, Pascal Sainrat
HIPEAC: European Network on High-Performance Embedded Architecture and Compilation
Institut de Recherche en Informatique de Toulouse
{barre, landet, rochange, sainrat}@irit.fr

Abstract

The estimation of the Worst-Case Execution Time of hard real-time applications becomes very hard as more and more complex processors are used in real-time systems. In modern architectures, estimating the execution time of a single basic block is not trivial due to possible timing anomalies linked to out-of-order execution. The influence of preceding basic blocks on the pipeline state also has to be accounted for. Recently, graphs have been used to model the execution of a block on a dynamically-scheduled pipelined processor [11]. In this paper we extend this model to express instruction-level parallelism so that superscalar processors with multiple functional units can be analyzed. Simulation results show how this extended model estimates WCETs tightly even when a realistic processor is considered. They also give an insight into the complexity of the model in terms of analysis time.

1. Introduction

Scheduling real-time tasks with strict deadlines requires having knowledge of their Worst-Case Execution Times (WCETs). Strategies to estimate WCETs have been the subject of many research works these ten last years. Methods based on measurements can be used to deal with a complex target architecture but it generally cannot be proved that their results do not underestimate the WCET. On the contrary, methods based on static program analysis intend to produce a true upper bound on the execution time.

Among static methods, the Implicit Path Enumeration Technique (IPET) [10] works on the program object code and seems to be the most widely used. This technique expresses the program execution time as the sum of the execution times of the basic blocks weighted by their respective executions counts on the execution path. The WCET is then obtained by maximizing the program execution time under some constraints that link the numbers of executions of the basic blocks (this problem is solved using Integer Linear Programming algorithms). The constraints come from the Control Flow Graph and from a preliminary flow analysis. The individual execution times of the basic blocks come from a low-level analysis that takes into account the features of the target processor. In this paper, we focus on the low-level analysis.

The model of the processor architecture used to estimate the worst-case execution time of a basic block must be accurate. While early pipelined processors could be analyzed through reservation tables [6][8], more recent architectures mean more complex models. The mechanisms that must be accounted for include pipelined, superscalar and dynamically-scheduled execution, as well as branch prediction and cache memories. The two latter have been addressed in several papers [2][3][6][7][12] but modeling the execution core still requires some research effort, as discussed in Section 2.3.

Li *et al.* [11] have proposed a very nice way of representing the behavior of a pipelined processor with out-of-order execution. Their solution seems to capture inter-block timing interferences that were identified by Engblom as Long Timing Effects [5]. However, their model does not express instruction-level parallelism: every pipeline stage is considered as scalar and processes a single instruction per cycle. Moreover, possible resource parallelism (i.e. multiple resources of the same type) is not represented. In this paper, we extend their model to make it possible to express parallelism and to take it into account in the evaluation of the WCET of a basic block.

The paper is organized as follows. Section 2 describes Li's model and lists the features that it cannot capture. In Section 3, we propose an extension to the model that makes it possible to consider (dynamically-scheduled) superscalar processors.

Performance results are given in Section 4 and we draw conclusions in Section 5.

2. Background

2.1. Li's model of a dynamically-scheduled pipelined processor

In [11], Li *et al.* propose to represent the execution of a basic block on an out-of-order pipelined processor by an execution graph that expresses the dependencies between instructions as well as possible contention for the use of shared resources. The instructions that can be executed before and after the block and their possible influence are also represented. The graph is then analyzed to compute lower and upper bounds on the times at which the different instructions are processed by the pipeline stage. The WCET of the basic block is the latest time at which the last instruction of the block can leave the pipeline. In this section, we give an overview of the model and of the algorithms used to determine execution times.

Execution graph. The execution of a basic block in the pipeline is modeled as a set of nodes, each of which stands for the processing of a given instruction by a given pipeline stage. The nodes are connected by edges that express inter-node dependencies. Solid edges express: (i) the program order (e.g. instructions are fetched in the program order – they are inserted in the limited-capacity reorder buffer in the program order); (ii) the pipeline structure (e.g. instructions are fetched before being decoded); (iii) data dependencies (e.g. an instruction has to wait for its operands to be ready before being executed). Besides to solid edges, undirected dashed edges express possible contention between nodes for using a shared resource (e.g. two instructions require the same functional unit and might be executed out-of-order). Figure 1 gives an example of execution graph for a processor with a 5-stage pipeline, a 2-entry instruction queue and a 4-entry reorder buffer. This example is taken from [9].

Latest and earliest times. The model is aimed at taking into account functional units with variable latencies expressed as intervals. From these latencies and from the inter-node dependencies expressed in the execution graph, an earliest and a latest value for the times at which a node is ready, starts and finishes are computed (the first instruction of the basic block – I_1 – is assumed to be fetched at time 0). The algorithm to estimate earliest and latest times has three main steps: (a) latest times are computed for every node considering every possible contender (a contender is a node that might compete for the same resource) and then are propagated to successor nodes; (b) earliest times are computed according to the same principle; (c) pairs of nodes that cannot be contemporary, i.e. with disjoint time intervals, are identified and declared as not-contending anymore. These three steps are repeated until stabilization or until the number of iterations has reached a limit. More details on the algorithm can be found in [9] and [11].

```
I1:   mult r6,r10,r4
I2:   mult r1,r10,r1
I3:   sub   r6,r6,r2
I4:   mult r4,r8,r4
I5:   add   r1,r1,r4
```

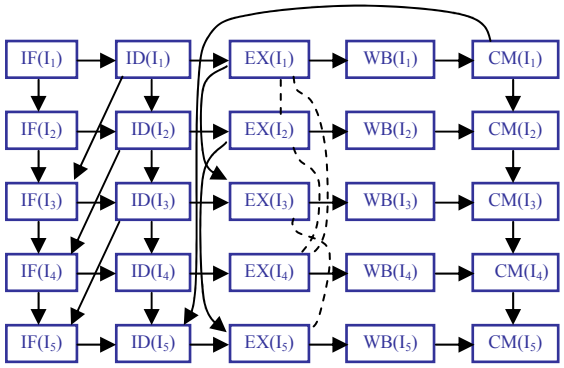

Figure 1. Example execution graph

Impact of earlier and later blocks. A prologue (resp. an epilogue) of a basic block is a sequence of instructions that can be executed just before (resp. after) the block. It contains as many instructions as can be active in the processor (i.e. the cumulative size of the instruction queue and the reorder buffer). Conservative hypotheses are taken about instructions that might precede a prologue. Obviously, a basic block can have many possible *(prologue, epilogue)* pairs: its WCET is the longest execution time when all these pairs have been considered. Prologue and epilogue nodes are included in the execution graph. Upper bounds on the ready/start/finish times for prologue nodes that have a path to node IF(I_1) (i.e. the fetch of the first instruction of the basic block) are derived. The earliest and latest times of the other prologue nodes are then computed as described above. Times for the epilogue nodes are evaluated the same way. Then the basic block is analyzed taking into account the information derived for the prologue and the epilogue. Again, more details are given in [9] and [11].

Block overlapping. Two basic blocks executed consecutively in a pipeline partially overlap. Then the execution time of a sequence of two blocks is generally lower than the sum of their individual

execution times. Li *et al.* define the cost of a basic block as the time between the end of the block (the time at which its last instruction leaves the commit stage) and the end of the previous block. They show how the worst-case cost can be safely computed from the earliest and latest times.

2.2. Instruction-level parallelism and super-scalar execution

The model proposed by Li *et al.* constitutes a firm foundation for the timing analysis of programs running on modern processors. It makes it possible to take into account pipelined and out-of-order execution, while capturing possible timing anomalies [14] when variable-latency functional units are considered. It also includes the effects of earlier and later basic blocks on the execution path. Finally, it can be combined with the analysis of the instruction cache and of the branch predictor [11].

However, this model misses a major feature of modern microprocessors: the capability of processing several instructions in parallel, known as superscalar execution. Actually, the execution graph can only express that some nodes have to be serialized either in a specified order (order of the program, order of the pipeline, data dependency) or in any order (access to a shared resource). This implies that every pipeline stage can process a single instruction per cycle.

The model also fails in taking into account multiple functional units of the same type: dashed edges prevent concurrent nodes to use a shared resource simultaneously. But modern processors often have several identical functional units and several operations of the same type (e.g. integer add) can be handled in parallel.

Because it fails to capture instruction-level parallel processing, this model cannot be used to analyze realistic modern processors. This is why we propose, in this paper, some extensions that would make it more useful.

2.3. Related work

As said before, evaluating the WCET of the basic blocks in a program comes up against two well-known difficulties: timing anomalies related to variable-latency functional units [14] and inter-block interferences, also known as “long timing effects” [5]. This has led some authors to recommend using scalar in-order processors when safe WCETs have to be estimated.

Several approaches have been proposed to get round the difficulty. In the VISA architecture, the code is annotated with time estimations [1]. The processor dynamically switches to a safe in-order execution mode whenever the actual measured time at some point in the program exceeds the estimation. In a recent work, we defined a processor pipeline that includes a fetch gating mechanism that enforces some distance between basic blocks in the pipeline [15]. This mechanism prevents non-adjacent blocks from having any timing interference.

As far as we know, there have been very few attempts to model superscalar, dynamically-scheduled processors. Besides the work by Li *et al.* which was described in Section 2.1, we are only aware of the *aiT* commercial tool by the AbsInt company. This tool uses abstract interpretation to determine all the possible pipeline states at the beginning of each basic block [16]. Unfortunately, its code source is not publicly distributed for research experimentation.

5. 3. Modeling instruction-level parallelism

To be applicable to modern processors, Li’s model has to be extended to support instruction parallelism within pipeline stages. It must be able to express two different situations: (i) two nodes have to be processed in a given order but possibly in parallel; (ii) several nodes compete for a given type of resource that is available in multiple copies. In this section, we show how these features can be included in Li’s model.

3.1 Modeling superscalar instruction processing

As said above, a solid directed edge between two nodes $S(I_a)$ and $S(I_b)$ means that I_a and I_b must be processed by stage S in the specified order. In other words, the start time of $S(I_b)$ is greater than or equal to the finish time of $S(I_a)$, that is the start time of $S(I_a)$ plus the latency of stage S.

To express that $S(I_a)$ and $S(I_b)$ must be processed in that order but possibly in parallel, we suggest to use slashed solid edges. A slashed edge between the two nodes then means that the start time of $S(I_b)$ is greater than or equal to the start time of $S(I_a)$. The parallelism degree of stage S is expressed with solid edges in the same way as the limited capacity of queues.

Figure 2 shows an example of the extended execution graph that models the processor front-end. In this example, an instruction cache line can be fetched every cycle: several instructions are fetched in parallel but the program order has to be respected. This

is modeled by slashed edges between the IF(I_x) nodes. We suppose that instructions I_3 and I_4 do not belong to the same cache line: this is why the edge between IF(I_3) and IF(I_4) is not slashed (a single cache line can be fetched each cycle). It is assumed that the decode stage has a capacity of two instructions per cycle. However, it is not possible to determine which pairs of instructions will be decoded together: it depends on the behavior of the rest of the pipeline (it can happen that a single instruction is decoded during a given cycle). This is why slashed edges indicate a possible parallel processing for every pair of adjacent ID nodes. The limited capacity of the decode stage (two instructions per cycle) is expressed by additional solid edges that link ID(I_x) to ID(I_{x+2}).

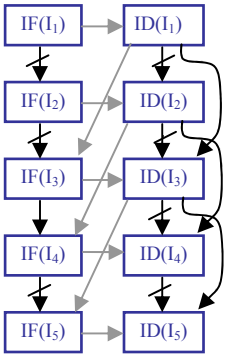

Figure 2. An execution graph expressing superscalar processing

Taking account slashed edges in the computation of the earliest and latest ready/start/finish times of nodes is somewhat trivial: it only modifies the way the times of a given node influences the times of its successors. In the computation of the latest times, the propagation of times from node *v* to its successor *w* becomes:

$$latest\left[t_w^{ready}\right] = \max\left(latest\left[t_w^{ready}\right], latest\left[t_v^{\boldsymbol{start}}\right]\right)$$

Modification for the estimation of the earliest times is similar:

$$earliest\left[t_w^{ready}\right] = \max\left(earliest\left[t_w^{ready}\right], earliest\left[t_v^{\boldsymbol{start}}\right]\right)$$

3.2 Modeling superscalar resources

Modern processors often include multiple functional units of the same type. The model proposed by Li *et al.* does not support this kind of parallelism and it assumes that instructions having the same resource requirements are automatically serialized. As said before, nodes competing for a shared resource are linked by a dashed edge which is not directed to allow out-of-order execution.

To take into account multiple resources, we propose to label dashed edges with the number of copies of the resource. Then the algorithms that compute the node times must be modified to allow several nodes to use a resource at the same time when this resource exists in multiple copies.

Figure 3 shows a graph with labeled dashed edges. Instructions I_1, I_2 and I_4 have to be processed by a multiply unit that can execute two instructions per cycle (i.e. the processor includes two multiply units), while instructions I_3 and I_5 must be processed one after the other one (in any order) by a unique add unit.

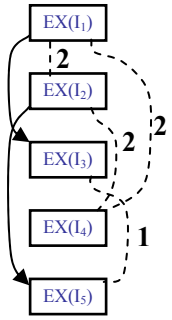

Figure 3. An execution graph modeling multiple resources

To compute the earliest and the latest times of the nodes, the algorithm proposed by Li *et al.* [11] first determines the sets of actual contenders. The actual contenders of node *v* are the nodes that might delay its start. This includes the nodes that require the same resource as *v* and: (i) either concern earlier instructions and are ready before (or at the same time as) *v*; or (ii) concern later instructions and start before *v* is ready. Actual contenders can delay node *v* only if their number exceeds the capacity of the resource. Figures 4 and 5 show how the delays induced by contending nodes can be determined when multiple resources are considered. In these algorithms, par_v stands for the degree of parallelism of the resource required by node *v* (example of parallel resources are functional units). We also note as $\max_{x\in S}^{[n]}\left(f\left(x\right)\right)$ the n^{th} largest value of *f(x)* for all values of $x \in S$.

As far as pipelined functional units are concerned, we have found that they can simply be modeled by splitting the corresponding EX nodes into as many nodes as pipeline stages linked by solid edges, as shown in Figure 6.

$latest\left[t_{IF(I_1)}^{ready}\right] = 0;$

foreach *node v in topologically sorted order* **do**

$latest\left[t_v^{start}\right] = latest\left[t_v^{ready}\right];$

$S_{late} = late\_contenders(v) \cap \left\{u \middle| \neg separated[u,v] \wedge earliest\left[t_u^{start}\right] < latest\left[t_v^{ready}\right]\right\};$

if $|S_{late}| \geq par_v$ **then**

$latest\left[t_v^{start}\right] = \min\left(\max_{u \in S_{late}}^{\lceil par_v \rceil}\left(latest\left[t_u^{finish}\right]\right), latest\left[t_v^{ready}\right] + max\_lat_v - 1\right);$

$S_{early} = early\_contenders(v) \cap \left\{u \middle| \neg separated[u,v]\right\};$

if $|S_{early}| \geq par_v$ **then**

$tmp = \min\left(\max_{u \in S_{late}}^{\lceil par_v \rceil}\left(latest\left[t_u^{finish}\right]\right), latest\left[t_v^{start}\right] + \left(|S_{early}| / par_v\right) \times max\_lat_v\right);$

$latest\left[t_v^{start}\right] = \max\left(tmp, latest\left[t_v^{start}\right]\right);$

$latest\left[t_v^{finish}\right] = latest\left[t_v^{start}\right] + max\_lat_v;$

foreach *immediate successor w of v* **do**

$latest\left[t_w^{ready}\right] = \max\left(latest\left[t_w^{ready}\right], latest\left[t_v^{finish}\right]\right);$

Figure 4. Modified LatestTimes() algorithm

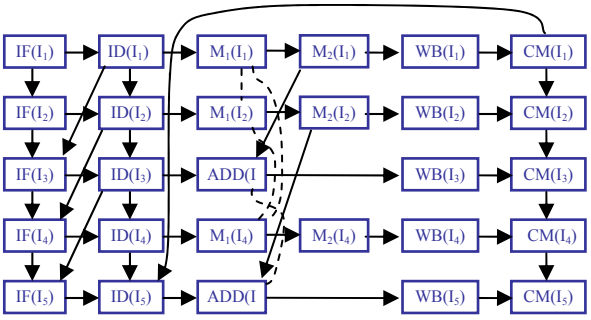

Figure 6. Example execution graph for a processor with pipelined functional units

4. Performance evaluation

In this section, our goal is to provide some performance results that should give insight into how the extended model helps in getting tight WCET estimations for the basic blocks. We also estimate the complexity of the model in order to determine whether it can be used to analyze realistic processors.

$earliest\left[t_{IF(I_1)}^{ready}\right] = 0;$

foreach *node v in topologically sorted order* **do**

$earliest\left[t_v^{start}\right] = earliest\left[t_v^{ready}\right];$

$S_{late} = late\_contenders(v) \cap \left\{u \middle| \neg separated[u,v] \wedge latest\left[t_u^{start}\right] < earliest\left[t_v^{ready}\right] < earliest\left[t_u^{finish}\right]\right\};$

$S_{early} = early\_contenders(v) \cap \left\{u \middle| \neg separated[u,v] \wedge latest\left[t_u^{start}\right] \leq earliest\left[t_v^{ready}\right] < earliest\left[t_u^{finish}\right]\right\};$

$S = S_{late} \cup S_{early};$

if $|S| \geq par_v$ **then**

$earliest\left[t_v^{start}\right] = \max\left(\max_{u \in S_{late}}^{\lceil par_v \rceil}\left(earliest\left[t_u^{finish}\right]\right), earliest\left[t_v^{ready}\right]\right);$

$earliest\left[t_v^{finish}\right] = earliest\left[t_v^{start}\right] + min\_lat_v;$

foreach *immediate successor w of v* **do**

$earliest\left[t_w^{ready}\right] = \max\left(earliest\left[t_w^{ready}\right], earliest\left[t_v^{finish}\right]\right);$

Figure 5. Modified EarliestTimes() algorithm

4.1 Methodology

The original model was implemented in the Chronos tool developed at the National University of Singapore [13]. This tool is able to estimate the WCET of a C program taking into account the characteristics of the target processor (pipelined, dynamically-scheduled but scalar, with an instruction cache and a branch predictor). It performs an analysis of the possible flows, estimates the individual WCET of the basic blocks (using the execution graph model) and then computes the WCET of the entire program using the IPET method.

We have implemented the extended pipeline model proposed in this paper in the Chronos tool to obtain the results given in this Section. Some of the benchmarks are distributed with the tool. Others come from the Mälardalen suite (`www.mrtc.mdh.se/projects/wcet/benchmarks.html`). Their WCETs have been computed considering a 5-stage superscalar out-of-

order processor with multiple functional units. Table 1 lists the main parameters used for the experiments.

pipeline width	2 or 4
instruction queue size	8 inst.
reorder buffer size	16 inst.
instruction cache	perfect (no miss)
branch predictor	perfect (no mispred.)
functional units	
integer add	2 units – 1 cycle latency
integer mul/div	1 unit – 3 cycle latency
float add	1 unit – 2 cycle latency
float mul	1 unit – 4 cycle latency
float div	1 unit – 12 cycle latency
load-store	1 unit – 1 cycle latency

Table 1. Processor specifications

4.2 Complexity of the model

Table 2 gives the computation times measured during the experiments. The first column gives the time needed to evaluate the WCETs of all the basic blocks, while the second one indicate the length of the estimation of the WCET for the whole program (using the IPET method). Even if the benchmarks are quite small, these results show that the computation requirements of the proposed model remain limited.

benchmark	analysis time (sec.)	ILP solving time (sec.)
matmul	0.059	0.007
matsum	0.052	0.007
isort	3.088	0.006
fdct	0.040	0.007
sqrt	5.022	0.006
cnt	2.018	0.006

Table 2. Analysis and ILP solving times.

4.3 Tightness of the estimated WCET

Table 3 gives the observed and estimated WCETs for the benchmarks. The observed WCET was obtained using the SimpleScalar cycle-accurate simulator (`sim_outorder`). Each benchmark was run with an input data set that is likely to show up the worst-case execution time. The observed WCET is not guaranteed to be the real WCET but it is expected to be close to it. The estimated WCET results from the timing analysis based on the extended model. The ratio between the estimated and the observed WCET shows that the overestimation is moderate: 20% on average for a 2-way superscalar processor, and 24% for a 4-way pipeline.

	2-way superscalar		
benchmark	**obs. WCET**	**est. WCET**	**ratio**
matmul	11 207	14 657	1.31
matsum	70 306	80 512	1.15
isort	49 910	50 242	1.01
fdct	2 779	3 239	1.17
sqrt	393	506	1.29
cnt	2 955	3 919	1.33
mean			**1.20**

	4-way superscalar		
benchmark	**obs. WCET**	**est. WCET**	**ratio**
matmul	9 752	13 657	1.40
matsum	60 305	80 511	1.34
isort	39 930	45 251	1.13
fdct	2 680	3 154	1.18
sqrt	379	432	1.14
cnt	2 678	3 512	1.31
mean			**1.24**

Table 3. Observed and estimated WCET

5. Conclusion

The ever growing performance requirements in real-time systems make the use of advanced processors inevitable. Many authors have pointed out that modeling these processors for WCET analysis is complex. Pipelines have been studied for several years but out-of-order execution was addressed only recently by Li *et al.* [12]. However, their model was limited to scalar processors. In this paper, we have extended it to superscalar architectures. We have shown how to model instruction-level parallelism in the pipeline stages and in the functional units (when there are several units of the same type). This implies some modifications in the execution graph (new types of edges) and extended algorithms.

Performance results show that the overestimation of the WCET is still moderate when instruction-level parallelism is taken into account. The tightness of the estimation is noteworthy when considering the very small analysis times.

Acknowledgements

We are grateful to Li *et al.* from the National University of Singapore for making the Chronos tool open source and downloadable from their web site. In turn, we plan to make the extended version of the code

available soon, as part of the OTAWA framework that is under development in our team [4].

References

[1] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, F. Mueller, "Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-time Systems", *IEEE Int'l Symp. on Computer Architecture*, 2003.

[2] I. Bate, R. Reutemann, "Efficient Integration of Bimodal Branch Prediction and Pipeline Analysis", *IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications*, 2005.

[3] C. Burguière, C. Rochange, "A Contribution to Branch Prediction Modeling in WCET Analysis", *Design, Automation and Test in Europe (DATE)*, 2005.

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

[5] J. Engblom, *Processor Pipelines and Static Worst-Case Execution Time Analysis*, PhD thesis, Uppsala University, 2002.

[6] C. Healy, D. Whalley, M. Harmon, "Integrating the Timing Analysis of Pipelining and Instruction Caching", *IEEE Real-Time Systems Symposium,* 1995.

[7] R. Heckmann, M. Langenbach, S. Thesing, R. Wilhelm, "The Influence of Processor Architecture on the Design and the Results of WCET Tool", *Proceedings of the IEEE*, vol. 91(7), 2003.

[8] S. Lim, Y.H. Bae, G.T. Jang, B. Rhee, S.L. Min, C.Y., Park, H. Shin, K. Park, S. Moon, C.S. Kim, "An Accurate Worst Case Timing Analysis for RISC Processors", *IEEE Transactions on Software Engineering*, vol. 27(7), 1995.

[9] X. Li, *Microarchitecture modeling for timing analysis of embedded software*, PhD thesis, National University of Singapore, 2005.

[10] Y.-T. Li, S. Malik, "Performance Analysis of Embedded Software using Implicit Path Enumeration", *ACM SIGPLAN Notices*, vol. 30, n°11, 1995.

[11] X. Li, T. Mitra, A. Roychoudhury, "Modeling Out-of-order Processors for Software Timing Analysis", *IEEE Real-Time Systems Symposium*, 2004.

[12] X. Li, T. Mitra, A. Roychoudhury, "Modeling Control Speculation for Timing Analysis", *Real-Time Systems Journal*, Kluwer Academic Publishers, 29(1), 2005

[13] X. Li, Y. Liang, T. Mitra, A. Roychoudhury, *Chronos: a timing analyzer for embedded software*, `www.comp.nus.edu.sg/~rpembed/chronos`.

[14] T. Lundqvist, P. Stenström, "Timing Anomalies in Dynamically Scheduled Processors", *IEEE Real-Time System Symposium*, 1999.

[15] C. Rochange, P. Sainrat, "A Time-Predictable Execution Mode for Superscalar Pipelines with Instruction Prescheduling", *ACM Int'l Conf. on Computing Frontiers*, 2005.

[16] S. Thesing, *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*, PhD thesis of the University of Saarlandes, 2004.