On the sensitivity of WCET estimates to the variability of basic blocks execution times

Hugues Cassé[a], Christine Rochange[a], Pascal Sainrat[a,b]
[a]Institut de Recherche en Informatique de Toulouse, [b]HiPEAC NoE
Université Toulouse III – Paul Sabatier
31062 Toulouse cedex 9, France
{casse, rochange, sainrat}@irit.fr

Abstract

The Implicit Path Enumeration Technique (IPET) is a very popular approach to evaluate the Worst-Case Execution Time (WCET) of hard real-time applications. It computes the execution time of an execution path as the sum of the execution times of the basic blocks weighted by their respective execution counts. Several techniques to estimate the execution time of a block taking into account every possible prefix path have been proposed: the maximum value of the block execution time is then used for IPET calculation. The first purpose of this paper is to analyze the sensitivity of block execution times to prefix paths. Then we show how expanding the IPET model to consider the execution times related to different contexts for each basic block improves the accuracy of WCET estimates.

1. Introduction

In hard real-time systems, the Worst-Case Execution Times (WCETs) of critical tasks have to be estimated as accurately as possible either to analyze the schedulability or to determine a static schedule that makes it possible for every task to meet its deadline. Several approaches to WCET estimation have been investigated these last fifteen years and the problem has been shown to be more and more complex as the architecture of processors is enhanced to provide better performance. Researchers recommend the use of simple hardware but even so the tendency is to consider high-performance processors in order to fit increasing performance requirements: the tasks to be executed are not necessarily more complex but, in the context of some approaches like IMA (Integrated Modular Avionics) or AUTOSAR (Automotive Open System ARchitecture), a single processing node should support several tasks.

Static methods for WCET analysis are usually preferred to measurements because they get round the need of examining every possible execution path. They roughly consist in adding the execution times of the basic blocks along execution paths. To be safe, they must ensure that the execution times of basic blocks are not under-estimated. When considering modern processors, this requires to include the impact of the context (i.e. the execution history) when analyzing the pipelined execution of a block. The result is the maximum execution time (or cost) of the block and is used to compute the WCET of the whole program.

Our purpose, in this paper, is to show that the execution time of a block depends sharply on the context (prefix path) and that the whole WCET would be estimated more tightly if different execution times (for different contexts) were considered for each basic block. In a second step, we introduce context-sensible data into the IPET model. Experimental results show that this helps in getting more accurate WCET estimates. The sensitivity of these results to architectural parameters (scalar/superscalar pipeline, static/dynamic instruction scheduling, size of the instruction window) is also investigated.

The paper is organized as follows. Section 2 provides some background information on timing analysis and pipeline modeling. In Section 3, we discuss experimental results that show the sensitivity to the context of block timings (experimental conditions are detailed in the Appendix). The benefits (in terms of tightness of WCET estimates) of extending the IPET model to include context-related block execution times are shown in Section 4. Section 5 concludes the paper.

2. Background: pipeline modeling for WCET estimation

Modern processors cut the processing of an instruction into several steps that are handled in a pipelined manner: in the absence of stalls, instruction *i* processes through step s in the same time as instruction *i-1* processes through step *s-1* (Figure 1 shows a typical processor pipeline). This means that, while the execution time of a single instruction is the sum of the latencies of all steps, the execution time of a sequence of instructions

is shorter than the sum of their individual execution times due to the overlap in the pipeline. In theory, the execution time of a sequence of *N* instructions in a *P*-stage pipeline (each stage having a single-cycle latency) should be *P+N-1*. In practice, the effective execution time would be longer due to stalls that result from resource conflicts and inter-instruction data dependencies.

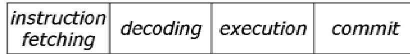

Figure 1. A typical 4-stage processor pipeline.

As said before, static WCET evaluation techniques compute the execution time of a program from the individual execution times of its basic blocks. To estimate the execution time of a basic block tightly, one must take into account the overlap of instructions in the pipeline. When a basic block is fetched in a pipelined processor, the pipeline is generally not empty and still contains instructions from the previous block(s). Then, two kind of effects should be taken into account in order to get an accurate WCET estimate. First, the instructions of the previous blocks use some resources and might be responsible for delaying the processing of the evaluated basic block. This is what we call the sensitivity to the context. Second, the blocks that are simultaneously present in the pipeline overlap, which reduces the execution time of the sequence.

In this section, we will review the various approaches that have been proposed to estimate the execution times of basic blocks in a pipeline. We will first show how the possible interferences between basic blocks can be accounted for while computing the WCET of a program. Then, we will explain how the behavior of a pipelined processor can be modeled and how the interferences between basic blocks can be evaluated.

2.1. Execution times and interferences between basic blocks

In this section, we will show how it is possible to evaluate the contribution of a basic block to the execution time of an execution path it belongs to.

A conservative approach to get an upper bound of this contribution consists in ignoring the overlap of successive blocks in the pipeline and in considering the full execution time of the block in an empty pipeline (i.e. the time between the first instruction is fetched and the last instruction is committed). Then the execution time of the path is computed as the sum of the individual execution times of its blocks. This is illustrated in Figure 2. While being safe, this approach clearly overestimates the execution time.

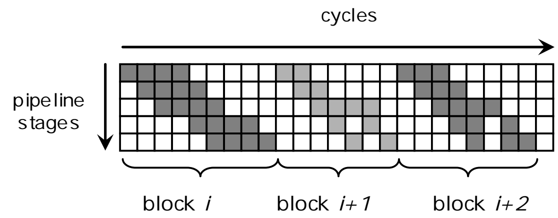

Figure 2. Conservative model to evaluate the execution time of a sequence of blocks.

To take into account the overlap of successive blocks in the pipeline, every possible initial context should be considered to derive time estimates for a given basic block. However, not only the overlap but also the possible block interferences must be analyzed. According to the pipeline characteristics, these interferences might lengthen or shorten the block execution time (in a dynamically-scheduled processor, timing anomalies can reduce or increase the final execution time [13]).

Assuming that the interferences can be properly analyzed (possible approaches will be reviewed in the next section), Figure 3 shows how they can be expressed. In this Figure, t_B is the execution time of block *B* when it is executed in an empty pipeline while $t_{B/A}$ is the execution time of *B* when it is executed after *A*. The cost $c_{B/A}$ is the contribution of block *B* to the execution time of the sequence [*A*-*B*]. The execution time of sequence [*A*-*B*] can be written as:

$$t_{[A-B]} = t_A + c_{B/A}$$

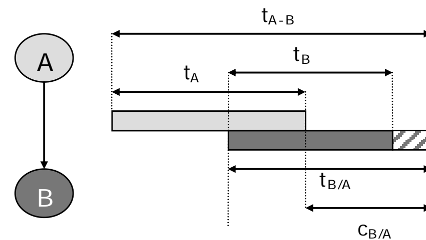

Figure 3. Expressing block interferences and overlapping.

Initially, users of the IPET method [11] to estimate the Worst-Case Execution Time of a program in a pipelined processor did not refer to the cost of blocks but to pipeline gains. The execution time of sequence [*A*-*B*] was expressed as:

$$t_{[A-B]} = t_A + t_B + \delta_{[A-B]}$$

which is the same ($c_{B/A}$ stands for $t_B + \delta_{[A-B]}$).

Early implementations of the IPET method did only consider 2-block sequences to determine the costs of blocks. In 2002, Engblom showed that this approach was no longer valid when considering modern processors [4].

To improve performance, modern processors often implement some mechanisms that might be a source of interferences between distant basic blocks (not only adjacent ones). Such features include superscalar execution, dynamic instruction scheduling, long-latency functional units, etc. Figure 4 illustrates how interferences between distant blocks may impact the execution time of a sequence. The execution time of sequence [*A*-*B*-*C*] should be computed as:

$$t_{[A-B-C]} = t_A + c_{B/A} + c_{C/[A-B]}$$

where $c_{C/[A-B]}$ might be shorter than, equal to or longer than $c_{C/B}$ due to the possible interferences from block A.

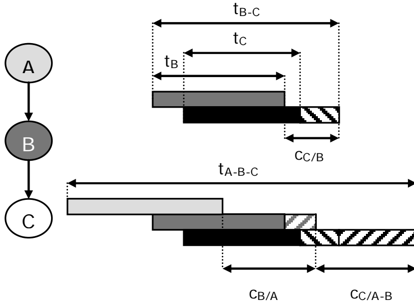

Figure 4. Long timing effects.

The main difficulty comes from the fact that the span of block interferences cannot be bounded, which means that the cost of a basic block on an execution path might be impacted by any other block on the path. In practice, the cost of a block only depends on the few preceding blocks but, for the sake of safety, a possible effect from very distant blocks must be imagined. Existing approaches to take all possible contexts into account to evaluate the cost of a block will be presented in the next section.

2.2. Pipeline model

Early contributions to pipeline modeling made use of reservation tables to find out how every instruction of a basic block would process through the pipeline [10][8]. A reservation table is a simple means to represent the use of the processor internal resources (pipeline stages, functional units, etc.) but its limited semantics is not sufficient to express the behavior of superscalar and dynamically-scheduled processors.

Cycle-level simulators (e.g. SimpleScalar [1]) make it possible to accurately determine the execution time of a sequence of blocks and the cost of each block in the sequence. Unfortunately, the number of possible prefix paths for a basic block in a real-life application is generally huge and it is not possible to simulate each of them. Then it is possible to determine the cost of a block for a set of short possible prefix paths, but not the *maximum* cost since all the possible prefixes cannot be considered.

When all the possible prefix paths cannot be considered one by one, the solution comes from static analysis techniques. The aiT tool of the AbsInt company [1] uses abstract interpretation to define an abstract pipeline state at the beginning of each basic block that includes every possible concrete state [15][16]. It then simulates the execution of the block on this input abstract pipeline state to determine an output abstract state which is propagated to the successors of the block. The tool iterates until a fix point has been found. Then execution times of basic blocks can be derived and included in the ILP (IPET) model to estimate the WCET of the whole program.

Recently, Li *et al.* [12] introduced execution graphs as a means for estimating the worst-case costs of basic blocks. An execution graph expresses precedence constraints between the processing steps of the instructions of a block and computes earliest and latest values of their respective finish times. Some preceding instructions (prologue) can be included in the graph to take the context into account and very conservative hypotheses are made on earlier instructions so that the computed cost is guaranteed to be an upper bound of the real possible costs, whatever the prefix path is. Again, this worst-case cost is used to compute the whole program WCET.

The differences between these two methods reside in the accuracy of the contexts considered for each basic block. With abstract interpretation, the input abstract pipeline state of a block results from the evaluation of effective prefix paths. On the contrary, the execution-graph method examines real prefixes until a given depth (which equals the instruction-window size) and considers that the previous instructions can be any. As a consequence, the overestimation of the costs is higher, which is the price of shorter analysis times.

3. On the sensitivity of block timings to the context

The experimental results reported in this paper were collected in the conditions detailed in the Appendix.

As explained in the previous section, static methods for pipeline modelling derive the worst-case cost of basic blocks. Our purpose here is to analyze the variability of the cost when different contexts (i.e. the prefix paths) are considered.

3.1. Context-related costs

For each basic block, we have examined the different costs when d-block deep contexts are considered, with $0 \leq d \leq 4$. In the rest of this paper, we will refer to the cost of a block evaluated by considering a d-block long prefix of the block as a d-cost.

At each depth level, the cost of a block would be estimated conservatively by a static analysis approach as the ones described in section 2.2. This estimate would be greater than or equal to the maximum of the costs evaluated at the next deeper level. In this work, we have evaluated all the possible costs of each block considering all its possible 8-block long prefixes (see the Appendix for details about the methodology). Then *i*-cost ($0 \leq i \leq 4$) of the block was computed as the maximum value of the corresponding ($i+1$)-costs.

To illustrate this, Table 1 lists the different costs of block A in the example CFG given in Figure 5 (in this example, the length of contexts is limited to three blocks). Block A has four possible 3-block contexts ($[D_1\text{-}C_1\text{-}B_1]$, $[D_2\text{-}C_2\text{-}B_1]$, $[D_3\text{-}C_3\text{-}B_2]$, $[D_4\text{-}C_3\text{-}B_2]$). In the table, the third column shows the number of different paths represented by each cost. For example, $c_{A/[D_1-C_1-B_1]}$ is the maximum value of the costs of A observed in the two possible paths $[F_1\text{-}E_1\text{-}D_1\text{-}C_1\text{-}B_1\text{-}A]$ and $[F_1\text{-}D_1\text{-}C_1\text{-}B_1\text{-}A]$. On a mean, each 3-cost represents 1.5 real cost values. We assume that paths $[F_1\text{-}D_3\text{-}C_3\text{-}B_2\text{-}A]$ and $[F_1\text{-}D_4\text{-}C_3\text{-}B_2\text{-}A]$ have the same cost value for block A. Then $c_{A/[C_3-B_2]}$ represents two paths but a single cost value. At the end, c_A represents five different values and is assigned the highest of them. Using c_A to compute the program WCET (instead of distinguishing between c_{A/B_1} and c_{A/B_2}) is a source of overestimation.

Context depth	Costs	# represented paths/values	mean # represented values
3	$c_{A/[D_1-C_1-B_1]}$	2 / 2	1.5
	$c_{A/[D_2-C_1-B_1]}$	2 / 2	
	$c_{A/[D_3-C_3-B_2]}$	1 / 1	
	$c_{A/[D_4-C_3-B_2]}$	1 / 1	
2	$c_{A/[C_1-B_1]}$	2 / 2	1.67
	$c_{A/[C_2-B_1]}$	2 / 2	
	$c_{A/[C_3-B_2]}$	2 / 1	
1	c_{A/B_1}	4 / 4	2.5
	c_{A/B_2}	2 / 1	
0	c_A	6 / 5	5

Table 1. Context depth and number of represented cost values (see Figure 5)

3.2. Variability of block execution times

For each context depth *d*, we have recorded the mean number of different represented cost values over the set *B* of basic blocks in the program (computed with the formula below, where $P_{b(d)}$ is the set of possible *d*-block long prefixes of block *b* and A_p is the set of possible paths before prefix *p*).

$$\frac{1}{|B|}\sum_{b\in B}\left(\frac{1}{|P|}\sum_{p\in P_{b(d)}}\left(\frac{1}{|A_p|}\sum_{a\in A_p} c_{b/[a-p]}\right)\right)$$

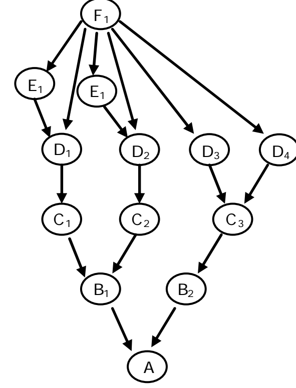

Figure 5. Example CFG.

In this section, we consider the 2-way superscalar processor configuration given in the Appendix. Results for each benchmark are given in Figure 6. As expected, the mean number of represented cost values decreases when deeper contexts are considered. For most of the applications, the variability of costs is noticeable. The case of `jfdctint` is interesting because each of its blocks has a constant cost value whatever the context depth is. This program contains three loops executed in sequence, and each loop has a long (up to 89 instructions) single-block body. These three blocks are the only ones to have two possible contexts and, since they are long, the impact of these contexts is not visible at the end of the blocks.

Considering the maximum of a set of possible costs values instead of each cost value individually might be detrimental to the accuracy of WCET estimates if the values differ noticeably. Table 2 gives additional information about the maximum number of represented cost values (observed over the set of blocks) and about the mean and maximum gap between the different cost values of a block. We observe that several benchmarks (`fft1`, `fir`, `lms`, `minver`, `qurt`) have at least one block for which the 0-cost represents more than 7 values. Moreover, the gap between cost values can be as long as 10 cycles (`minver`). This is likely to impact the whole WCET since a block that has many different timings probably belongs to a loop and might be executed many times on the worst-case path. Note that the maximum number of represented values and the maximum gap between them remains large for 4-costs (even if their means is lower). This indicates that some blocks are impacted by long timing effects.

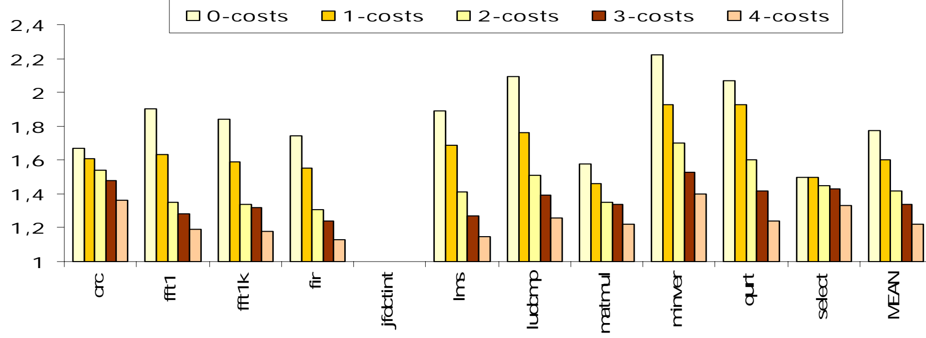

Figure 6. Variability of the costs of blocks (2-way superscalar processor)

	0-costs			4-costs		
benchmarks	**max. # values**	**mean gap**	**max. gap**	**max. # values**	**mean gap**	**max. gap**
`crc`	3	0.71	3	3	0.43	3
`fft1`	8	1.16	8	6	0.23	5
`fft1k`	5	1.02	5	4	0.22	4
`fir`	7	0.88	7	6	0.16	8
`jfdctint`	2	0.22	1	1	0	0
`lms`	7	1.09	8	6	0.2	7
`ludcmp`	6	1.3	7	4	0.3	4
`matmul`	3	0.67	3	2	0.22	1
`minver`	11	1.35	10	6	0.39	10
`qurt`	7	1.16	8	6	0.25	7
`select`	4	0.64	4	3	0.41	3

Table 2. Variation of the cost values represented by each 0-cost and 4-cost (2-way processor)

4. Context-sensitivity of blocks timings and WCET estimates accuracy

4.1. Including context-related timings in the IPET model

As said before, the execution time of an execution path is usually computed as:

$$T = \sum_{b \in B} x_b . c_b$$

where B is the set of blocks along the paths, xb is the execution count of block b in the path and c_b is the maximum contribution of block b to the execution time (c_b is evaluated taking any possible context into account).

This estimation can be refined by using deeper costs. With depth d, $x_b.c_b$ can be expanded into:

$$x_b . c_b = \sum_{p \in P_{b(d)}} x_{b/p} . c_{b/p}$$

where $P_{b(d)}$ is the set of d-block prefixes of block b.

As shown in earlier work by Ermedahl [7], a set of constraints must be added to the IPET model to bound the execution count of each prefix path. For each block, and for each path p in the set P of the possible prefixes of the block, with p being the sequence of blocks $[p_0\text{-}p_1\text{-}\dots\text{-}p_{n-1}]$, the constraints to be added are:

$$x_b = \sum_{p \in P_{b(d)}} x_{b/p}$$

$$\forall i, 0 \le i < n-1, \; x_{b/p} \le x_{[p_i - p_{i+1}]}$$

where $x_{[p_i - p_{i+1}]}$ is the execution count of sequence $[p_i\text{-}p_{i+1}]$.

4.2. Experimental results: improvement of WCET estimates

Figure 7 shows how the WCET can be improved when d-costs are used (the gain is computed against the WCET estimated with 0-costs). On a mean, the WCET estimation is improved by 5.5% with 1-costs, by 7% with 2-costs, by 10% with 3-costs and by 11% with 4-costs. Some of the benchmarks (`jfdctint`, `crc`, `fft1`, `select`) do not exhibit noticeable improvements. This was expected for `jfdctint` since the number of contexts is very limited. Other benchmarks (`fftk1`, `ludcmp`, `matmul`, `minver`, `qurt`) benefit from higher gains.

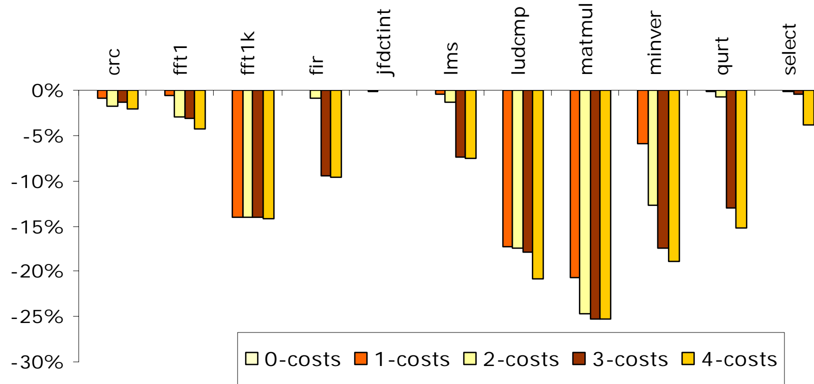

Figure 7. Improvement of the estimated WCET with deeper analysis (2-way superscalar).

4.3. Impact of the architecture parameters on the WCET improvement

In this section, we analyze the impact of the processor parameters on the WCET improvement obtained by using 4-costs. Figure 8 shows how the gains (against a WCET estimated with 0-costs) vary with the pipeline width (with out-of-order execution). It can be observed that very small gains are to be expected from considering deeper contexts for a scalar processor (only 2.3% on a mean). But the gains increase rapidly with the pipeline width. For several benchmarks, the improvement is higher than 20% for a 4-way processor (up to 43% for `ludcmp`) which is considerable. For these benchmarks, we have recorded up to 15-cycle gaps between the cost values represented by some 0-costs. This means that the execution time of a block is very dependent on the execution history. This is not surprising since a large pipeline with dynamic instruction scheduling can produce a large number of instructions interleaves, which generates inter-block effects.

In Figure 9, we observe the impact of the window size on the results. Measurements were made for a 4-way out-of-order processor, with different reorder buffer sizes, and the WCET was computed with 0-costs and 4-costs (the diagram plots the improvement). For most of the benchmarks, better gains are obtained when the instruction window is deeper. This is not true for `matmul` and select: they include some repeated small loops that do not fit well in a small reorder buffer (which generates many possible contexts for some blocks). This is why they exhibit high gains with a small instruction window.

Finally, we have considered in-order pipelines. The results given in Figure 10 were obtained for a 4-way processor with a 32-instruction window. As it could be expected, it appears that taking deeper contexts into account does only slightly improve WCET estimates for a statically-scheduled processor. This is due to the fact that the number of possible instruction interleaves in the pipeline is smaller, which limits the number of possible cost values for a basic block.

5. Conclusion

Estimating the WCET of a program using a static approach requires evaluating the individual execution times of basic blocks. When executing on a high-performance processor, the execution time of a basic block is likely to depend on the execution history (also referred to as the context). Some techniques to take into account all the possible contexts have been proposed and they provide the maximum execution time (or cost) of each basic block.

In this paper, we have presented some experimental results that show how much block execution times are sensitive to the context. For most of the benchmarks, some blocks have many different cost values related to different possible prefix paths (as many as 11 values) and the gap between the minimum and maximum value is large (up to 10 cycles). Considering the maximum value instead of distinguishing each of them is likely to lead to a large over-estimation of the WCET.

We have shown how taking context-related execution times could improve the tightness of WCET estimates. Experimental results prove that considering 4-block contexts tightens the WCET estimates by up to 25% for a 2-way superscalar out-of-order processor (11% on average) and up to 43% (18% on average) for a 4-way processor. As far as we know, most of the existing tools (e.g. aiT) mainly consider 1-costs (even if they might use loop unrolling techniques to take one part of the context-sensitivity into account. Our results show that considering deeper contexts is beneficial for most of the applications. Additional results show that the gain is slightly lower when the instruction window (reorder buffer) is less deep (4-way out-of-order processor) and that is negligible when instructions are scheduled in order.

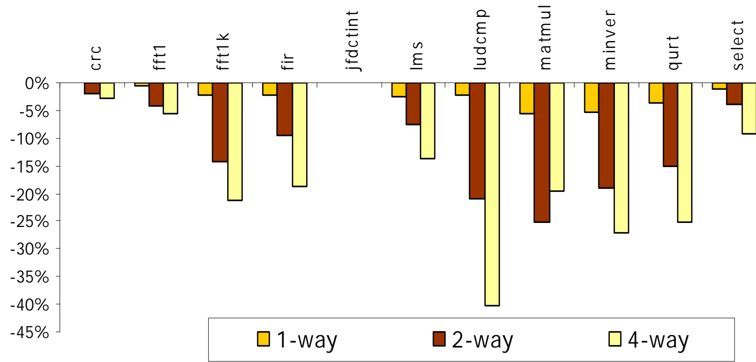

Figure 8. Improvement of the estimated WCET as a function of the pipeline width.

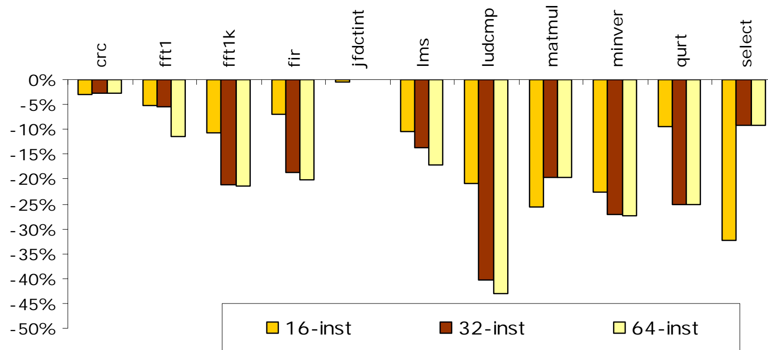

Figure 9. Impact of the ROB size on the WCET improvement with 4-costs (4-way out-of-order).

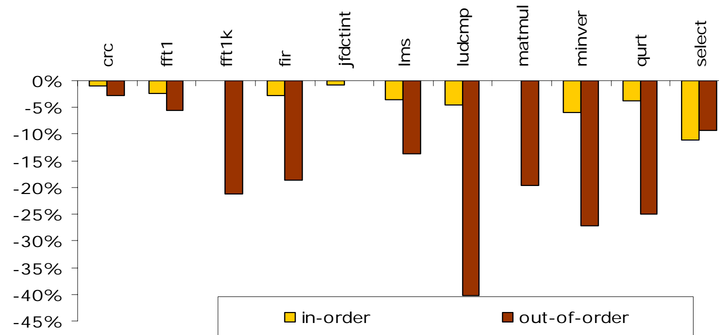

Figure 10. Impact of the scheduling policy on the WCET improvement with 4-costs (4-way superscalar, 32-inst. ROB)

References

[1] `http://www.absint.com`

[2] T. Austin, E. Larson, D. Ernst, *SimpleScalar: An Infrastructure for Computer System Modeling*, IEEE Computer, 35(2), 2002.

[3] H. Cassé, P. Sainrat, *OTAWA, a framework for experimenting WCET computations*, 3rd European Cong. on Embedded Real-Time Software, 2006.

[4] J. Engblom, *Processor Pipelines and and Static Worst-Case Execution Time Analysis*, Ph.D. thesis, University of Uppsala, 2002.

[5] J. Engblom, A. Ermedahl, *Pipeline Timing Analysis using a Trace-Driven Simulator*, 6th Int'l Conference on Real-Time Computing Systems and Applications (RTCSA), 1999.

[6] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, H. Hansson, *Towards Industry-Strength Worst Case Execution Time Analysis*, ASTEC 99/02 Report, 1999.

[7] A. Ermedahl, *A Modular Tool Architecture for Worst-Case Execution Time Analysis*, Ph.D. thesis, Uppsala University, 2003.

[8] C. Healy, R. Arnold, F. Mueller, D. Whalley, M. Harmon, *Bounding pipeline and instruction*

cache performance, IEEE Transactions on Computers 48(1), 1999.
[9] Y. Hur, Y.H. Bae, S.-S. Lim, S.-K. Kim, B.-D. Rhee, S.-L. Min, C.Y. Park, H. Shin, C.S. Kim, *Worst case timing analysis of RISC processors: R3000/R3010 case study*, IEEE Real-Time Systems Symposium, 1995.
[10] S.-S. Lim, Y. H. Bae, G. T. Jang, B.-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, C. S. Kim, *An accurate worst case timing analysis technique for RISC processors,* Real-Time Systems Symposium, 1994.
[11] Y.-T. S. Li, S. Malik, *Performance Analysis of Embedded Software using Implicit Path Enumeration,* Workshop on Languages, Compilers, and Tools for Real-time Systems, 1995.
[12] X. Li, A. Roychoudhury, T. Mitra, *Modeling out-of-order processors for WCET analysis*, Real-Time Systems, 34(3), 2006
[13] T. Lundqvist, P. Stenström, *Timing Anomalies in Dynamically Scheduled Microprocessors*, IEEE Real-Time Systems Symposium (RTSS), 1999.
[14] `http://archi.snu.ac.kr/realtime/benchmark/`
[15] S. Thesing, *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*, PhD thesis, Universität des Saarlandes, 2004.
[16] H. Theiling, C. Ferdinand, *Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis*, 19th IEEE Real-Time Systems Symposium (RTSS), 1998.

Appendix

The experiments reported in this paper were carried out using the OTAWA framework [3]. OTAWA[1] implements an infrastructure to support several kinds of analyses that can be combined to estimate the WCET of an application. In this work, we used a basic flow analyzer (it builds the CFG from the object code and retrieves user-specified flow facts), a cycle-level simulator (it estimates the execution times of sequences of blocks) and a module that generates the constraints for WCET estimation with IPET and calls an ILP solver.

Evaluating the costs of basic blocks from the simulation of sequences of limited length is questionable since some long timing effects might not be captured this way. However, as mentioned in Section 2, block interferences do not span over more than a few blocks in practice. Then, the costs considered in this work cannot be guaranteed as 100%-safe but can be thought as very close to the real costs.

The simulator models a pipelined processor and accepts different parameters: pipeline width, instruction-scheduling policy (in-order/out-of-order), fetch queue and reorder buffer sizes, functional units parameters (width, latency, pipeline). It includes perfect caches (every access is a hit) and a perfect branch predictor. The configurations used in this work are summarized in Table 3. The simulator accepts PowerPC object code as input.

		CONFIGURATIONS		
		1-way	**2-way**	**4-way**
instruction scheduling		out-of-order		
fetch queue size		4	8	8
reorder buffer size		4	8	32
functional units	**latency**			
integer ALU	1	1	2	2
memory unit	2	1	1	1
fp ALU	6	1	1	1
multiply unit	3	1	1	1
divide unit	15	1	1	1

Table 3. Processor configurations

The benchmarks come from the SNU-suite [14] and are listed in Table 4. They were compiled to PowerPC code using gcc with the -O0 optimization level option.

	# blocks	**mean block length**	**Function**
`crc`	52	5	CRC (Cyclic Redundancy Check)
`fft1`	151	7	FFT (Fast Fourier Transform) using Cooly-Turkey algorithm
`fft1k`	48	8	FFT (Fast Fourier Transform) for 1K array of complex numbers
`fir`	94	7	FIR filter with Gaussian number generation
`jfdctint`	10	22	JPEG slow-but-accurate integer implementation of the forward DCT
`lms`	85	7	LMS adaptive signal enhancement
`ludcmp`	47	6	LU decomposition
`matmul`	14	4	Matrix product
`minver`	75	5	Matrix inversion
`qurt`	77	7	Root computation of quadratic equations
`select`	30	4	N-th largest number selection

Table 4. Benchmarks

[1] OTAWA is supported by the French Agence Nationale pour la Recherche (MasCoTte project)