Towards designing WCET-predictable processors

Christine Rochange and Pascal Sainrat
Institut de Recherche en Informatique de Toulouse
118, route de Narbonne
31062 Toulouse cedex 4, France
{rochange, sainrat}@irit.fr

Abstract

Several methods based on a static analysis of the executable code have been proposed in the past to estimate the worst-case execution time of programs. Their main advantage is to limit measurements to small parts of code (e.g. basic blocks). However these methods have been designed for basic processor architectures and recent work by Engblom has shown that they would not be safe for more advanced designs. Actually, the execution of a basic block could have an impact on the execution of a distant subsequent basic block. Ignoring this impact could result in an under-estimated WCET, which could be dramatic in a hard real-time context. In this paper, we suggest that advanced architectures could include specific hardware that would eliminate all possible long timing effects. We show how this idea could permit to make superscalar pipelines analyzable for the WCET.

1. Introduction

1.1 Evaluating the Worst-Case Execution Time

For the calculation of the Worst-Case Execution Time of a program, the ideal thing would be to measure (or simulate) all the possible execution paths. This is generally not affordable because it would be very expensive in time. Moreover, while measuring all the complete paths, parts of code that belong to several paths would be evaluated several times. Thus, the objective is to limit measurements, as much as possible, to small parts of code.

The behaviour of some components of the processor architecture (like the cache memories or the branch predictor) is very dependent on the execution history and the timing analysis has to consider complete execution paths. To fasten the analysis, techniques like *static simulation* examine several paths in parallel, which might require a large storage capacity.

Some other parts of the processor, like the execution pipeline, are expected to exhibit a more «local» behaviour. So, as far as they are concerned, it is possible to measure parts of code, instead of complete paths, which limits the redundancy of measurements. These parts of code can be basic blocks, or bodies of algorithmic structures. The calculation of the WCET then consists in combining the individual execution times of the parts to obtain the execution time of the longest possible path.

Some WCET computation methods are based on a bottom-up traversal of the program syntax tree, while others are based on the control flow graph. In this paper, we focus on a method of this second category: the Implict Path Enumeration Technique or IPET [LiMa95], which consists in representing the control flow graph and the results of the flow analysis by a set of constraints on the numbers of executions of each part of code (basic block), and then in maximizing the total execution time (which is the sum of the products of the number of executions by the execution time of each basic block).

1.2 Modeling pipelined processors

To be able to model both correctly and precisely pipelined processors, the above method has to be adapted to take into account the pipeline effect that makes the execution of a sequence of two basic blocks shorter than the addition of the two individual execution times.

In section 2, we show how this effect can be included in the model. We also outline Engblom's analysis of long timing effects associated to sequences of more than two basic blocks and we argue that the IPET method cannot easily take into account these effects. Instead of restricting the choice for a real-time system to very simple architectures that cannot generate long timing effects, we think that high-performance processors could include a hardware mechanism to eliminate them, as defended in section 3. In section 4, we apply this principle to a perfect superscalar processor and show that the performance loss is very small. Section 5 concludes the paper.

2. Inter-block timing effects

Processor pipelines generate two kinds of timing effects:

- *pairwise effects* due to the overlapping of two adjacent basic blocks in the pipeline
- *long timing effects* that represent the impact of this overlap when sequences of three and more blocks are considered.

In this section, we examine how these effects could be modeled in the IPET method.

2.1 Pairwise timing effects

Timing effects between two adjacent blocks can be modeled by defining an execution time for the edge that connects them in the control flow graph (as illustrated in Figure 1). The execution time of an edge represents the gain due to the overlap of the two blocks in the pipeline. It is always negative. Then, the set of constraints that

express the structure of the control flow graph is rewritten to link the execution times of blocks and edges.

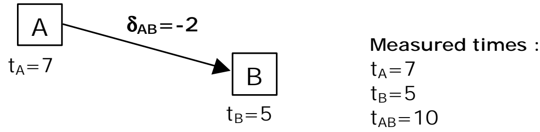

Figure 1. *Pairwise timing effects*

2.2 Long timing effects: Engblom's timing model

Engblom has shown in his PhD thesis [Engb02] that there could exist timing effects between distant basic blocks: «*A long timing effect for a sequence of instructions $I_1...I_m$, $m\geq3$, occurs whenever I_1 has the effect of disturbing the execution in such a way that the execution of the instructions $I_2...I_m$ is different compared to if I_1 had not been present*». An example of a long timing effect is given in Figure 2. Among the sources of long timing effects, Engblom mentions parallel pipelines, long latency instructions, dynamic scheduling, … Engblom has highlighted that long timing effects could occur for sequences of any length (potentially infinite), and that they could be either negative, null or positive.

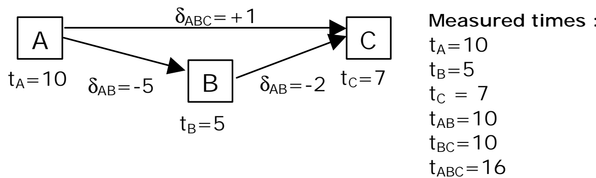

Figure 2. *Long timing effects*

Engblom proposed a timing model where a timing effect $\delta_{i...j}$ is associated to each sequence of basic blocks B_i … B_j. The execution time of a sequence of basic blocks B_1 … B_n is then given by :

$$t_{1...n} = \sum_{i=1}^{n} t_i + \sum_{1\leq j\leq k\leq n} \delta_{j...k}$$

Whenever a long timing effect is negative, it can be ignored, which might lead to WCET over-estimation. But when it is positive, it has to be taken into account for a safe WCET analysis. To model long timing effects when applying the IPET method, one should add some weighted edges between non-adjacent blocks. The main difficulty is then to obtain the weight of these edges because it requires to measure the execution time of the corresponding sequences. Since a long timing effect can exist between two blocks that are very far from each other, all the possible sequences of blocks have to be measured, which could be even longer than measuring all the possible execution paths and obviously goes against the principle of the IPET method (that is to limit measures to small parts of code). Expressing constraints on the number of executions of long edges might be difficult too.

3. Towards a high-performance processor without long timing effects

Pipelined processors can be safely analysed using the IPET method if they are guaranteed not to generate positive long timing effects.

What has been proposed until now is to restrict the choice of a processor architecture to one that does not exhibit any possibility of positive long term effects. Engblom has shown that a single in-order pipeline has this property.

However, this kind of architecture might not meet higher and higher performance requirements. This is why we suggest a new approach that consists in adding to the processor hardware the ability to prevent the appearance of positive long timing effects, as illustrated in Figure 3. This mechanism analyses the instruction flow to detect conditions that could engender long timing effects (LTEs): occupation of a resource during several clock cycles, access to the memory hierarchy with default in the first-level cache memory, … Then, the mechanism controls the pipeline to prevent the next basic block to enter the pipeline until the end of the risk of long timing effects.

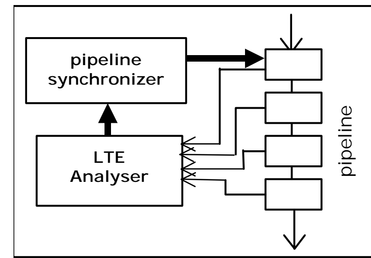

Figure 3. *A mechanism to eliminate long timing effects*

We feel that this approach could be implemented in high-performance processors, and the challenge is to limit the induced performance degradation.

In the next section, we show how this principle could be applied to a processor with a superscalar in-order pipeline (that was shown by Engblom to possibly generate infinite positive long term effects).

4. Case study: superscalar pipeline

4.1 Evaluation methodology

In order to focus on the impact of parallel pipelines, we make the following assumptions: perfect multiple branch prediction, perfect cache memories, no interlocks due to data dependencies, all instructions executed in a single cycle, … This is what we call a «perfect» pipeline.

Performance results that we will give were obtained using a simulator of a 6-stage perfect pipelined processor that we developped within the MicroLib project [MIC] based on SystemC.

The benchmark codes we used are listed in Table 1. They were taken from the SNU benchmark suite [SNU], and were slightly modified to inline function calls. We

only excuted the main function (i.e. not the starting and ending code).

		# insts
`crc`	*CRC computation*	54 790
`fft1`	*FFT using Cooly-Turkey algorithm*	3163
`jfdctint`	*JPEG slow-but-accurate integer implementation of the forward DCT*	5828
`lms`	*LMS adaptive signal enhancement*	535 985
`ludcmp`	*LU decomposition*	7 797

Table 1. *Benchmark applications (from the SNU suite)*

4.2 Long timing effects in a superscalar pipeline

Figure 3 shows how a sequence of 4 basic blocks (A-B-C-D) would be executed in a perfect 4-stage 2-way superscalar pipeline.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
IF	A_1 A_2	A_3 B_1	B_2 C_1	C_2 C_3	C_4 C_5	D_1			
D		A_1 A_2	A_3 B_1	B_2 C_1	C_2 C_3	C_4 C_5	D_1		
EX			A_1 A_2	A_3 B_1	B_2 C_1	C_2 C_3	C_4 C_5	D_1	
WB				A_1 A_2	A_3 B_1	B_2 C_1	C_2 C_3	C_4 C_5	D_1

starting with block A

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
IF	B_1 B_2	C_1 C_2	C_3 C_4	C_5 D_1					
D		B_1 B_2	C_1 C_2	C_3 C_4	C_5 D_1				
EX			B_1 B_2	C_1 C_2	C_3 C_4	C_5 D_1			
WB				B_1 B_2	C_1 C_2	C_3 C_4	C_5 D_1		

starting with block B

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
IF	C_1 C_2	C_3 C_4	C_5 D_1						
D		C_1 C_2	C_3 C_4	C_5 D_1					
EX			C_1 C_2	C_3 C_4	C_5 D_1				
WB				C_1 C_2	C_3 C_4	C_5 D_1			

starting with block C

Figure 3. *Executing a sequence of 4 basic blocks in a 2-way superscalar pipeline*

From this figure, we can compute the execution times and the timing effects of each sub-sequence of A-B-C-D, as shown in Table 2.

More generally, we can compute that, whatever the number of parallel pipelines is, all the long timing effects (even the timing effects for infinite-length basic block sequences) equal -1, 0 or +1. Proof is given in Appendix. As mentioned before, a positive timing effect constitutes a difficulty for computing the WCET using the IPET method.

sub-sequence	***execution time***	***timing effect***
A	5	-
B	4	-
C	6	-
D	4	-
A-B	6	-3
B-C	7	-3
C-D	6	-4
A-B-C	8	-1
B-C-D	7	0
A-B-C-D	9	**+1**

Table 2. *Computing inter-block timing effects*

We have measured the frequency of positive timing effects. Results are given in Table 3. They show that positive timing effects are frequent: on a mean, 14.39% of the 6-block sequences exhibit a positive effect (+1). This confirms the importance of the problem, even with a quite simple pipeline.

seq. length	**3**	**4**	**5**	**6**
`crc`	21.28%	13.18%	22.03%	18.33%
`fft1`	7.85%	10.83%	10.91%	11.02%
`jfdctint`	38.10%	24.00%	28.13%	23.08%
`lms`	19.05%	19.10%	18.45%	9.09%
`ludcmp`	7.07%	11.72%	10.18%	10.41%
`MEAN`	**18.67%**	**15.76%**	**17.94%**	**14.39%**

Table 3. *Percentage of positive timing effects, as a function of the sequence length.*

4.3 Synchronizing the pipeline to eliminate long timing effects

We propose to include in the processor a mechanism that resynchronizes the pipeline whenever a basic block enters in the fetch stage (see Appendix 2) while the first slot of this stage is occupied by one instruction of the previous block. In our example, this would produce the scheduling shown in Figure 4. Instructions that belong to the same basic block can then be processed in parallel but no timing effect can occur between basic blocks.

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
IF	A_1 A_2	A_3	B_1 B_2	C_1 C_2	C_3 C_4	C_5	D_1			
D		A_1 A_2	A_3	B_1 B_2	C_1 C_2	C_3 C_4	C_5	D_1		
EX			A_1 A_2	A_3	B_1 B_2	C_1 C_2	C_3 C_4	C_5	D_1	
WB				A_1 A_2	A_3	B_1 B_2	C_1 C_2	C_3 C_4	C_5	D_1

Figure 4. *Executing a sequence of 4 blocks in a 2-way superscalar pipeline with a resynchonizing mechanism*

In the previous example, resynchronizing increments the execution time of the sequence by one clock cycle. Table 4 gives measures of the performance degradation in terms of instruction throughput (number of committed instructions per cycle) over a non-synchronized pipeline.

pipeline width	*2-way*	*4-way*	*8-way*
`crc`	-8.55%	-15.98%	-39.67%
`fft1`	-3.11%	-10.76%	-22.44%
`jfdctint`	-1.52%	-4.07%	-10.15%
`lms`	-5.76%	-8.32%	-25.13%
`ludcmp`	-2.72%	-7.65%	-23.32%
MEAN	**-4.33%**	**-9.36%**	**-24.14%**

Table 4. *Instruction throughput degradation due to the synchronization of the pipeline*

The reduction of the instruction throughput appears not to be so high and synchronizing the pipeline does not give up the benefit of parallel pipelines. For example, the speedup of a 2-way synchronized superscalar processor over a scalar processor is, on a mean over the five benchmarks, 1.91.

5. Conclusion

This paper comes after Englom's thesis which has highlighted that the execution of a given basic block can have an influence on the execution of a distant subsequent block. This influence can be expressed as a value, called *long timing effect*, that can either be positive, negative or null. A long timing effect has to be taken into account in the computation of the execution time of any sequence that includes those two blocks.

Now, several methods for WCET computation, like IPET, aim for restricting measurements to small parts of code (basic blocks). This does not permit to take long timing effects into account, and then this class of methods can only be applied to processors that cannot generate long timing effects. Engblom has shown that some simple pipelined processors have this property.

In this paper, we proposed another approach that consists in including in the processor a mechanism that dynamically detects conditions that might engender a long timing effect and synchronizes the pipeline in such a way that this effect cannot appear. To illustrate this approach, we applied it to a perfect superscalar pipeline.

Simulation results show that the performance degradation is limited: the *synchronized* superscalar processor is competitive compared to a scalar processor.

References

[Engb02] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. Uppsala Dissertations from the Faculty of Science and Technology 36, April 2002.

[LM95] Y.-T. S. Li, S. Malik. *Performance Analysis of Embedded Software Using Implicit Path Enumeration*. 32nd Design Automation Conference (DAC), 1995.

[MIC] `http://www.microlib.org`

[SNU] `http://archi.snu.ac.kr/realtime/benchmark/`

Appendix 1. Computing long time effects in a superscalar pipeline.

Let us consider an n_i-instruction basic block. Its execution time in a perfect (without any stall) s-stage scalar pipeline is given by $t_i = s + n_i - 1$.

To express its execution time in a perfect w-way s-stage superscalar pipeline, we write $n_i = x_i.w + y_i$. Then, we obtain:

$$t_i = s + \left\lceil \frac{n_i}{w} \right\rceil - 1 \quad \text{or} \quad t_i = s + x_i + \left\lceil \frac{y_i}{w} \right\rceil - 1$$

where $\lceil q \rceil$ is the upper integer value of q.

Now, the execution time of a sequence of basic blocks $B_i \ldots B_j$ can be expressed as:

$$t_{i \ldots j} = s + \left\lceil \frac{\sum_{k=i}^{j} n_k}{w} \right\rceil - 1 = s + \sum_{k=i}^{j} x_k + \left\lceil \frac{\sum_{k=i}^{j} y_k}{w} \right\rceil - 1$$

We can then compute the long timing effect $\delta_{1 \ldots m}$ associated to the sequence of basic blocks $B_1 \ldots B_m$. In Engblom's thesis, it is defined as:

$$\delta_{1 \ldots m} = t_{1 \ldots m} - t_{2 \ldots m} - t_{1 \ldots m-1} + t_{2 \ldots m-1}$$

So we can write:

$$\delta_{1 \ldots m} = \sum_1^m x_i - \sum_2^m x_i - \sum_1^{m-1} x_i + \sum_2^{m-1} x_i + \left\lceil \frac{\sum_1^m y_i}{w} \right\rceil - \left\lceil \frac{\sum_2^m y_i}{w} \right\rceil - \left\lceil \frac{\sum_1^{m-1} y_i}{w} \right\rceil + \left\lceil \frac{\sum_2^{m-1} y_i}{w} \right\rceil$$

Since $0 \le y_i < d, \forall i$, we have:

$$\left\lceil \frac{\sum_1^m y_i}{w} \right\rceil - \left\lceil \frac{\sum_2^m y_i}{w} \right\rceil = \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} \quad (0 \; or \; 1)$$

and then $\delta_{1 \ldots m} = \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} - \begin{Bmatrix} 0 \\ 1 \end{Bmatrix} = \begin{Bmatrix} -1 \\ 0 \\ 1 \end{Bmatrix}$

Appendix 2. Detecting basic blocks.

Hardware designers usually define a basic block as a sequence of code preceded by a branch and ending with the next branch. But our mechanism has to detect compiler basic blocks, defined as sequences of code that can only be entered at their first instruction and exited at their last instruction. Their detection requires to have a full knowledge of the static code, which is not the case of the hardware.

Then we suggest that the compiler could help the hardware by marking the beginning of basic blocks. It could exploit unused bits of the instruction codes, if such bits are available in the target instruction set. Otherwise, the compiler could make sure that every basic block ends with a control flow instruction, by adding a branch to the next instruction whenever it should not be the case.