Code padding to Improve the WCET Calculability

Christine Rochange
Université Paul Sabatier
IRIT
31 062 Toulouse cedex 9
rochange@irit.fr

Pascal Sainrat
Université Paul Sabatier
IRIT
31 062 Toulouse cedex 9
sainrat@irit.fr

Abstract

The Worst-Case Execution Time of tasks with strict deadlines must be predictable: it must be possible to estimate this time both safely and tightly at an acceptable computing cost. Static WCET analysis is facilitated if parts of code can be analyzed more or less independently of one another. This is why it is desirable to prevent timing interferences between blocks. In this paper, we show how it is possible to transform the code to prevent timing effects between distant basic blocks on an execution path. Our approach consists in padding the code to space out basic blocks. Performance results show that the code size is sensibly increased but that the cost in terms of WCET degradation is moderate.

1. Introduction

Being able to estimate the Worst-Case Execution Time (WCET) of tasks is absolutely necessary for hard real-time systems. Measurement is generally inadequate because it cannot be guaranteed that all the possible execution paths have been tested. This is why academic research has focused on static WCET analysis.

The WCET estimated by static methods should obviously be safe since missing deadlines can have dramatic consequences in some critical systems. However, it should also be as tight as possible: WCET overestimation can have undesirable effects like the impossibility to schedule the tasks. It might also lead to oversized hardware.

Static methods compute an upper bound of the real WCET by combining information about the possible execution paths (produced by a preliminary analysis of the code) and the execution times of the basic blocks. These times can be determined by a cycle-level simulator of the target processor.

However, critical applications follow the general evolution towards more and more computing requirements. This is why advanced processor architectures tend to be used in critical systems [16]. Unfortunately, in a high-performance processor, a basic block does generally not execute the same way in the application code as it would do if it was executed alone. This is due to interferences (data dependencies, precedence constraints or resource conflicts) with other blocks on the execution path. To get the worst-case execution time of a block, these possible interferences should be taken into account. This is often a very complex task, as it will be explained in Section 2.

To keep the WCET analysis simple, we have recently proposed to modify the processor architecture to eliminate any possible timing effect between basic blocks [13]. The idea was to space out successive basic blocks in the pipeline in such a way that they cannot interfere. The proposed scheme obviously degrades the performance (in the order of 42% for an 4-way superscalar out-of-order processor) but the loss could be acceptable in the name of timing safety. However, the main problem is that such a processor does not exist for the moment. This is the reason why we suggest that the distance between blocks could be enforced by the compiler instead of the hardware.

Our approach consists in padding the code by inserting neutral filler-instructions, *i.e.* instructions that will not be executed but only fetched and decoded before being removed from the pipeline (like a true NOP). The lengths of the padding blocks are computed so that they eliminate all the possible interferences between basic blocks.

Note that this work focuses on timing interferences related to the use of the pipeline and of the internal processor resources. We do not address here the question of modeling caches, branch predictors, etc. This is why we will consider these components as perfect (*i.e.* with a very predictable behaviour) in the evaluation part.

The paper is organized as follows. Section 2 gives some background information on static WCET analysis and on the possible timing interferences between blocks in high-performance processors. It also overviews related work. We introduce our approach in Section 3. Performance results are analyzed in Section 4 and concluding remarks are given in Section 5.

2. Background

2.1. Static WCET estimation

Static analysis techniques add the execution times of basic blocks on the possible execution paths extracted either from the syntax tree [8] or from the control flow graph [6]. For example, the *Implicit Path Enumeration Technique* handles the search of the WCET as an optimization problem where:

- the objective function is the program execution time expressed as the sum of the basic block execution times weighted by their respective numbers of execution. As we will explain it in Section 2.3, it should also include the possible timing interferences between basic blocks.
- the constraints are the relations between the unit execution times. Some of them can be extracted from the control flow graph, others come from a preliminary flow analysis and express loop bounds, infeasible paths, etc.

Evaluating the WCET of the program comes to determining the numbers of execution of the basic blocks that maximize the objective function while meeting the constraints.

2.2. Timing interferences

As mentioned above, the expression of the program execution time should include inter-block timing interferences.

For very simple processors, such interferences are limited to adjacent blocks which overlap in the pipeline: the execution time of a two-block sequence is shorter than the sum of their respective execution times. In that case, all of the timing effects can be captured by measuring the execution times of blocks alone and of sequences of two blocks.

However, more advanced architectures make interferences between distant blocks possible, as it was shown by Engblom [3]. He has found that a block can interfere with a distant one, and this kind of interference is referred to as a *long timing effect* (LTE).

The execution time of a path can be computed as:

$$T = \sum_{i \in \mathcal{B}} t_i + \sum_{0 \le j < \ldots < k \le n} \delta_{j..k}$$

where $\mathcal{B}$ is the set of blocks (which are numbered from 0 to n), t_i is the execution time of block i and $\delta_{j..k}$ is the timing effect associated to the sequence of blocks $B_j \ldots B_k$. This is illustrated in Figure 1.

Sources of long timing effects include block alignment (*i.e.* the relation between the number of instructions in the block and the width of the pipeline) [12], long latency instructions, data dependencies, out-of-order execution, limited-capacity queues, etc.

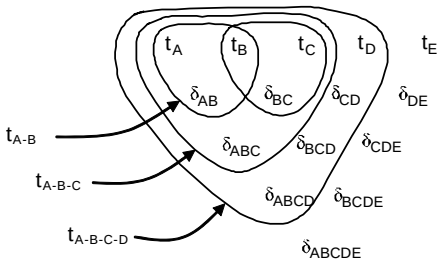


Figure 1. Engblom's timing model

Engblom has shown that long timing effects would span over unlimited block sequences: at the very worst, the first block of the program can affect the execution of the last block. Moreover, a long timing effect value ($\delta_{i...j}$) can be negative as well as null or positive. A negative value should be taken into account to get a tight WCET estimation, but a positive value *must* be accounted for to compute a *safe* estimation.

2.3. Including timing interferences in WCET analysis

The original IPET method was developed considering very simple processor architectures where only adjacent basic blocks could interfere by overlapping in the pipeline. The corresponding gain was seen as the (negative) execution time of the edge linking the two blocks. Then, the edge execution time (weighted by the number of executions of the edge) was taken into account in the expression of the program execution time. For example, the WCET model for the control flow graph given in Figure 2 would have been:

$$\max T = x_A t_A + x_B t_B + x_C t_C + x_D t_D + x_E t_E + x_{AB}\delta_{AB} + x_{BC}\delta_{BC} + x_{CD}\delta_{CD} + x_{BE}\delta_{BE} + x_{ED}\delta_{ED}$$

$$1 = x_A = x_{AB} = x_B \qquad x_B = x_{BC} + x_{BE}$$
$$x_{BC} = x_C = x_{CD} \qquad x_{BE} = x_E = x_{ED}$$
$$x_D = x_{CD} + x_{ED} = 1$$

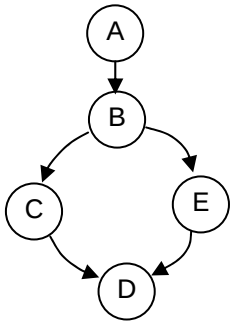


Figure 2. Example Control Flow Graph

Now, advanced processor architectures are often used for real-time systems and long timing

interferences should also be taken into account. We have found two different approaches in the literature.

The first one, described in [4], extends the original IPET model to include the long timing effects. For the example given in Figure 2, the model comes to:

$$\max T = x_A t_A + x_B t_B + x_C t_C + x_D t_D + x_E t_E + x_{AB}\delta_{AB} + x_{BC}\delta_{BC} + x_{CD}\delta_{CD} + x_{BE}\delta_{BE} + x_{ED}\delta_{ED} + x_{ABC}\delta_{ABC} + x_{BCD}\delta_{BCD} + x_{ABE}\delta_{ABE} + x_{BED}\delta_{BED} + x_{ABCD}\delta_{ABCD} + x_{ABED}\delta_{ABED}$$

$$1 = x_A = x_{AB} = x_B \qquad x_B = x_{BC} + x_{BE}$$
$$x_{BC} = x_C = x_{CD} \qquad x_{BE} = x_E = x_{ED}$$
$$x_D = x_{CD} + x_{ED} = 1$$
$$x_{ABC} \leq x_{AB} \qquad x_{ABC} \leq x_{BC} \qquad x_{ABC} \bullet x_{AB} - x_{BE}$$
$$x_{ABE} \leq x_{AB} \qquad x_{ABE} \leq x_{BE} \qquad x_{ABE} \bullet x_{AB} - x_{BC}$$
$$x_{BCD} = x_{BC} \qquad x_{BED} = x_{BE}$$
$$x_{ABCD} = x_{ABC} \qquad x_{ABED} = x_{ABE}$$

This solution weighs the expression of the objective function down and adds several constraints for each possible sequence in the execution path. Since LTEs can be as long as complete execution paths, the number of sequences to consider is potentially very high. Then the optimization problem might be very difficult to solve. Moreover, a value must be assigned to the LTE associated to each sequence of blocks: it must be computed from the execution times of all the sub-sequences. At the end, evaluating the LTE values comes to measuring every possible sequence of blocks, which is very time consuming in the general case.

Another approach consists in including the possible timing interferences in the execution times of blocks. When the target architecture can generate long timing effects, the execution time of a basic block should be evaluated by considering all the possible prefix paths and by keeping the highest value. While simulating numerous prefix paths could be unfeasible, the use of the abstract interpretation theory can make things more tractable [15]. The IPET model is then transformed as follows, where τ_A is the execution time of block A including the possible impact of other basic blocks:

$$\max T = x_A\tau_A + x_B\tau_B + x_C\tau_C + x_D\tau_D + x_E\tau_E$$

$$1 = x_A = x_B = x_C + x_E$$
$$x_D = x_C + x_E = 1$$

The algorithm for obtaining the adjusted unit execution times by abstract interpretation is not much detailed in papers, but it seems that it necessitates high computing power [14].

Moreover, including the effects of any possible prefix path in the execution time of a block leads to WCET overestimations since: (a) the flow analysis can find out that some prefix paths are infeasible, and (b) some prefix paths might not belong to the longest path and then should not be accounted for in the WCET. In the preceding example, τ_D includes the impact of block B on block D within the sequence BED (while δ_{BCD} might be null). If the flow analysis determines that block E is never executed and if δ_{BED} is positive, the WCET will be overestimated. Similarly, if the execution time of block C is far longer than that of block E, the longest path is along the path BCD and the impact of B on D in sequence BED should be ignored.

To sum up, the evaluation of unit execution times is costly in time for both approaches. The first solution also makes the IPET model more complex while the second one introduces some pessimism. These are the reasons why we are investigating solutions to limit timing interferences.

2.4. Related work

Li et al. [7] define a model based on dependence graphs to evaluate the execution time of a basic block in an out-of-order processor. However, they do not model superscalar execution and their experiments consider a very small core. Whether their model would scale to more realistic processors still has to be further investigated.

Heckmann et al. [5] use abstract interpretation to estimate the impact of previously executed blocks on the execution time of each basic block. This approach has been implemented in the aiT tool by the AbsInt company. While their method is an interesting alternative to exhaustive measurement (which is generally not affordable), each unit execution time includes the effects of all the possible prefix paths, which might result in WCET overestimations as shown in Section 2.3. Moreover, it seems that some pessimistic assumptions have sometimes to be taken to reduce the number of states. They might also lead to WCET overestimation.

In a recent work [13] we defined a processor pipeline where non-adjacent blocks cannot have timing interferences thanks to a fetch gating mechanism that enforces some distance between basic blocks in the pipeline. While this architecture makes the WCET easily computable by adding the execution times of the basic blocks among the possible execution paths, this solution does not solve today's problems since such an architecture does not exist yet.

As far as other parts of the processor are concerned (cache memories, branch predictor), guidelines to make their behaviour more predictable have also been proposed as an alternative to build too much complex models [2][11].

3. Code padding

3.1. General principle

The basic idea of the scheme proposed in this paper is close to the one that was behind our previous work [13]. To avoid long timing effects, basic blocks should not enter the pipeline one after the other: a certain distance should be enforced between them in such a way that the execution of a block cannot be disturbed by a previous block still in the pipeline. We suggest here that this distance could be enforced by the way of code padding, using neutral filler-instructions like NOPs. A *filler*-instruction is not executed and is removed from the pipeline after decoding. It does not require any other hardware resource than a slot in the fetch and decode stages. Some examples of filler-instructions in real processors will be given in Section 3.2.

The lengths of the code padding blocks have to be calculated by analysing the instructions belonging to basic blocks that might be executed consecutively and by determining their respective resource requirements. This analysis can be done by the compiler, and an algorithm is proposed in Section 3.3.

3.2. Neutral filler-instructions

To implement code padding, we need some instructions that use the fetch and decode stages to space out basic blocks, but are not executed (they should not consume computing resources) and not processed to the completion stage (otherwise, they might impact the execution time of the basic blocks). In this section, our purpose is to show that most instruction sets feature instructions that meet these constraints.

Most architectures have a NOP instruction that does not produce any result. In modern pipelines, NOP instructions are quashed from the pipeline after decoding in order to save the occupation of the functional units and the pipeline bandwidth.

Some processors have other instructions that do not go to the end of the pipeline. For example, on the PowerPC 750, fall-through branch instructions are removed from the instruction stream at dispatch. Then, an unconditional branch targetting the next instruction can be considered as a neutral instruction and used as a filler.

3.3. Code padding

The role of code padding is to avoid any possible interaction between distant blocks on an execution path. In the case where no long timing effect can occur, only the interferences between successive blocks are to be accounted for. Then the execution time of a sequence of *n* blocks can be computed as:

$$T = \sum_{i \in B} t_i + \sum_{0 \le j < n} \left(\sum_{s \in \mathcal{S}_j} \delta_{j,s} \right)$$

where $\mathcal{S}_j$ is the set of possible successors of block B_j.

A sufficient condition for this formula to be correct is that every possible sequence of two blocks executes exactly as if it was not preceded by other blocks in the pipeline. In this case, the LTE term — that normally stands for the distortion of the execution trace of the sequence by previous instructions — is null. This can be illustrated by the following example.

Let us consider three blocks A, B and C processing through a 3-stage pipeline with two non pipelined functional units, FU1 and FU2, that have a 3-cycle latency. Blocks A and C use FU1 while B uses FU2. The execution patterns are shown in Figure 3. The execution times of the blocks and the timing effects can be computed from these tables:

$t_A = t_B = t_C = 5$

$t_{AB} = 6 \quad \Rightarrow \quad \delta_{AB} = t_{AB} - t_A - t_B = 6 - 5 - 5 = -4$

$t_{BC} = 6 \quad \Rightarrow \quad \delta_{BC} = t_{BC} - t_B - t_C = 6 - 5 - 5 = -4$

$t_{ABC} = 8 \quad \Rightarrow \quad \delta_{ABC} = t_{ABC} - t_A - t_B - t_C - \delta_{AB} - \delta_{BC}$

$= 8 - 5\times3 - (-4)\times2 = +1$

	1	2	3	4	5
FETCH	A				
FU1		A	A	A	
FU2					
COMPLETE					A

	1	2	3	4	5
FETCH	B				
FU1					
FU2		B	B	B	
COMPLETE					B

	1	2	3	4	5
FETCH	C				
FU1		C	C	C	
FU2					
COMPLETE					C

	1	2	3	4	5	6
FETCH	A	B				
FU1		A	A	A		
FU2			B	B	B	
COMPLETE					A	B

	1	2	3	4	5	6
FETCH	B	C				
FU1			C	C	C	
FU2		B	B	B		
COMPLETE					B	C

	1	2	3	4	5	6	7	8
FETCH	A	B	C					
FU1		A	A	A	C	C	C	
FU2			B	B	B			
COMPLETE					A	B		C

Figure 3. Execution of a 3-block sequence (example)

The *positive* LTE δ_{ABC} expresses that the execution pattern of the sequence B-C is distorted when it is preceded by block A, due to A and C conflicting for the use of FU1. This resource is free before the end of the fetch of B when it is executed alone, but it remains busy until two cycles after the end of the fetch of B when it is preceded by A.

Our purpose is to prevent this distortion and to make the sequence execute as shown in Figure 4. The approach consists in filling the black cell with a neutral instruction. Now, δ_{AB} = -3 and δ_{ABC} = 0.

	1	2	3	4	5	6	7	8
FETCH	A		B	C				
FU1		A	A	A	C	C	C	
FU2				B	B	B		
COMPLETE					A		B	C

Figure 4. Safe execution of a 3-block sequence (example cont'd)

Filler-instructions are added before a basic block to absorb any resource conflict that might occur with a previous block. As we will see in Section 4, the fetching of some basic blocks has to be delayed by several cycles if we want to prevent long timing effects. This means that these blocks should be preceded by a large number of neutral instructions, since a one-cycle delay is enforced by as many filler-instructions as the pipeline width.

To keep the code size acceptable, it is possible to group all the required filler-instructions into a common *padding block* that has multiple entry points and is terminated by a return branch (`blr`). Then every sequence of filler-instructions required to delay the fetch of a basic block can be implemented as a linked-branch (`bl`) to the appropriate entry point of the padding block. This is illustrated in Figure 5 where a 2-way pipeline is assumed. Note that the linked branch and the return branch each enforce a one-cycle delay.

4. Algorithm for code padding

To compute the padding lengths, the compiler first needs to collect timing information about the execution of sequences of basic blocks in the pipeline. Such information can be profiled by a cycle-level simulator of the processor that simulates blocks and up-to-*n*-block sequences (the simulation time is generally acceptable if *n* is small). Cycle-level simulation is required because precise dynamic information is needed to generate safe results. The simulation can be done within the compiler (provided it has an exact knowledge of the hardware) or by calling external software. Figure 6 gives an algorithm that analyses the resource needs of blocks and sequences: for each block B and for each resource R, it computes the time at which R is needed after B starts to be fetched (`n[R,B]`) and the time at which R is released by B after B has been completely fetched (`r[R,B]`). Release times are also derived for sequences: `r[R,S]` stands for the time at which resource R is available after sequence S has been entirely fetched.

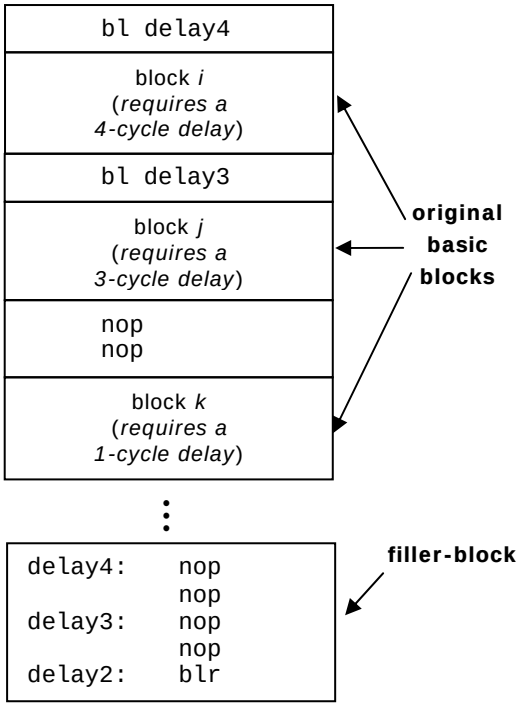


Figure 5. Padded basic blocks

```
foreach block B do {
   ff[B] ← first fetch cycle of B;
   lf[B] ← (last fetch cycle of B) + 1;
   foreach resource R do {
      n[R] ← cycle at which R is needed;
      r[R] ← cycle at which R is released;
               // 0 if R not used by B
      n[R,B] ← n[R] - ff[B];
      r[R,B] ← r[R] - lf[B];
   }
   d[B] ← 0;
}
foreach sequence B1-…-Bx (x < n) do {
   lf[Bx] ← (last fetch cycle of Bx) + 1;
   foreach resource R do {
      r[R] ← cycle at which R is released;
               // 0 if R not used by any Bi
      r[R,B1-…-Bx] ← r[R] - lf[Bx];
   }
}
```

Figure 6. Algorithm to analyse the resource requirements of blocks and sequences

4.1. *Depth-1* approach

As stated before, a long timing effect δ_{ABC} is not null if block A has an influence on how sequence B-C executes. On the contrary, δ_{ABC} is null if C executes after A-B exactly as after B. A *sufficient* — but not necessary — condition for this is that every resource (register, pipeline stage, functional unit, etc.) is released after A-B exactly at the same time as after B.

This assertion leads to the algorithm given in Figure 7. It analyses each two-block sequence to find out whether the first block has an impact on the availability of resources after the sequence. If so, the algorithm calls the `StrictDelay()` function that computes the distance *d* to put between the two blocks so that every resource is available after the sequence as soon as after the second block executed alone. Note that this distance is not always equal to the difference between `r[R,A-B]` and `r[R,B]`: it can be smaller but also larger due to timing anomalies, a phenomenon identified by Lundqvist [10]. For the moment, the `StrictDelay()` function computes the right distance by successive trials (the distance is upper-bounded by the size of the instruction window (fetch queue plus reoder buffer). A more clever algorithm based on execution graphs [7] is under development.

In the rest of this paper, this first algorithm will be referred to as the *depth-1* strategy.

```
foreach sequence A-B do {
   foreach resource R do {
      if r[R,A-B] > r[R,B] then {
         d ← StrictDelay(R,A-B);
         if d > d[B] then
            d[B] ← d;
      }
   }
}
```

Figure 7. *Depth-1* algorithm for computing the padding lengths

4.2. *Depth-n* strategy

The algorithm proposed in the previous section guarantees that every resource is available after sequence A-B exactly at the same time as after block B executed alone. This caution can be considered as excessive since the blocks executed after A-B might not require the resources delayed by A.

A more aggressive approach consists in examining the requirements of the possible successors of sequence A-B to determine whether a delay on the availability of a given resource induced by block A is likely to generate a long timing effect or not.

In the *depth-n* algorithm, the effective requirements of each basic block in every *n*-block sequence (B_0-B_1-…-B_{n-1}) are analyzed. Two kinds of situations necessitate that a distance is put between B_0 and B_1. The first case is when B_i uses a resource that is (a) not ready at the time B_i needs it, and (b) available later after B_0-…-B_{i-1} than after B_1-…-B_{i-1}. The second case is when the resource is ready on time for any block within the sequence but is released later after B_0-…-B_{n-1} than after B_1-…-B_{n-1}.

The analysis of the possible conflicts can be further refined for resources that can handle several instructions in parallel: they do not necessarily have to be completely free for blocks that use them but they should provide enough free slots to fulfil the needs.

In the case where a distance is to be enforced, the padding length is computed by the `MinimumDelay()` function (which implements the same approach as the `StrictDelay()` function). Figure 8 details the *depth-4* algorithm that analyses 5-block sequences and considers the exact requirements of the three last blocks to determine whether the second one has to be delayed after the first one.

```
foreach sequence A-B-C-D-E do {
   foreach resource R do {
      if n[R,C] > 0
         && r[R,A-B] > n[R,C]
         && r[R,A-B] > r[R,B] then {
         d ← MinimumDelay(R,A-B-C);
         if d > d[B] then
            d[B] ← d;
      }
      elsif n[R,D] > 0
            && r[R,A-B-C] > n[R,D]
            && r[R,A-B-C] > r[R,B-C] then {
         d ← MinimumDelay(R,A-B-C-D);
         if d > d[B] then
            d[B] ← d;
      }
      elsif n[R,E] > 0
            && r[R,A-B-C-D] > n[R,E]
            && r[R,A-B-C-D]> r[R,B-C-D] then {
         d ← MinimumDelay (R,A-B-C-D-E);
         if d > d[B] then
            d[B] ← d;
      }
      elsif
         r[R,A-B-C-D-E] > r[R,B-C-D-E] then {
         d ← StrictDelay(R,A-B-C-D-E);
         if d > d[B] then
            d[B] ← d;
      }
   }
}
```

Figure 8. *Depth-4* algorithm for computing the padding lengths

5. Performance results and discussion

5.1. Evaluation methodology

We have developed in SystemC a cycle-level simulator that models a generic processor architecture with parameterized features. The configuration we used for our tests is shown in Figure 9. The cache and the branch predictor are considered as perfect since modeling them is outside the scope of this work. The simulator is able to execute PowerPC code.

Pipeline width	**2-way**	**4-way**
fetch queue size	16	32
instruction cache	perfect (100% hit rate)	
branch predictor	perfect	
re-order buffer size	16	64
# of functional units (*latency*)		
integer add (*1 cycle*)	2	4
integer mul/div (*6 cycles*)	1	1
floating-point add (*3 cycles*)	1	1
fp mul (*6 cycles*)	1	1
fp div (*15 cycles*)	1	1
load/store (*2 cycles*)	2	2
data cache	perfect (100% hit rate)	

Figure 9. Simulated processor architecture

The results presented below were measured for several benchmarks commonly used in research on WCET analysis and presented in Figure 10 They implement standard algorithms: matrix arithmetic, signal processing, sorts.

`matmul`	matrix multiplication
`ludcmp`	LU decomposition
`jfdctint`	JPEG integer implementation of the forward Discrete Cosine Transform
`bsort`	bubble sort
`heapsort`	heap sort
`insertsort`	insert sort

Figure 10. Benchmarks

Figure 11 shows our framework for code padding. The object code is produced with the standard `gcc` compiler, targeted for the PowerPC instruction set. We have developed a utility that extracts the Control Flow Graph from the object code. The list of the basic blocks is used to drive the processor simulator which produces the execution trace of each block and of each possible sequence of up-to-5 blocks. The main tool of the chain is the *Interference Analysis* script that computes the padding lengths to eliminate any possible resource conflict between distant basic blocks. This script gets timing information from the simulator to compute the required delays down to the last cycle. Finally, the *Code Padding* script inserts the filler-instructions in the original assembly code.

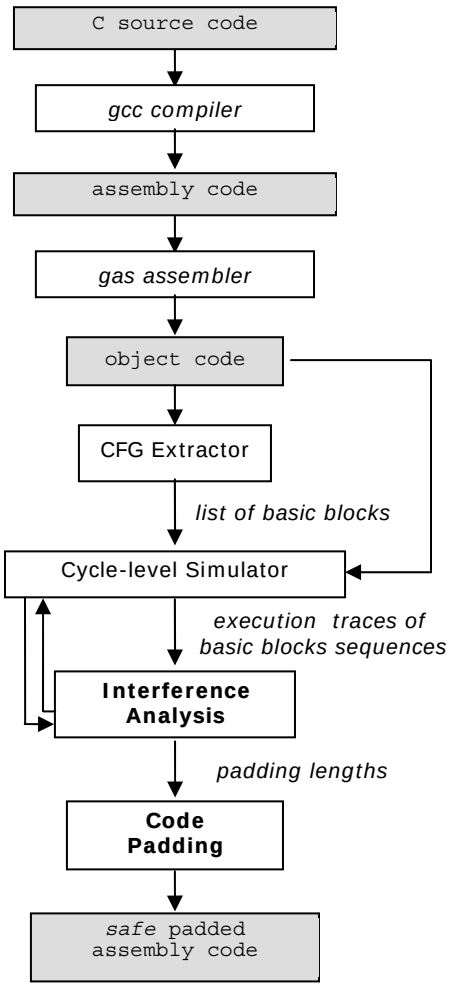


Figure 11. Code transformation framework

5.2. Impact of code padding on code size

Figure 12 shows the increase of the code size (number of static instructions) due to the filler-instructions added by the *depth-1* algorithm. The cost is undeniably sensible, especially for a 4-way target pipeline.

As shown in Figure 13, the increase is smaller when the analysis is done more in depth, *i.e.* when it takes into account the real requirements of the basic blocks. With the *depth-4* strategy, the mean increase is 18.28% for a 2-way pipeline, and 57.01% for a 4-way pipeline. We acknowledge that the increase is still noticeable but as we will discuss it in Section 9, we argue that it is the price of predictability.

	2-way	4-way
matmul	35.24%	76.19%
ludcmp	16.51%	28.20%
jfdctint	11.37%	126.97%
bsort	31.25%	76.25%
heapsort	25.00%	51.47%
insertsort	23.81%	59.52%
MEAN	***23.86%***	***69.77%***

Figure 12. Code size increase for the *depth-1* strategy

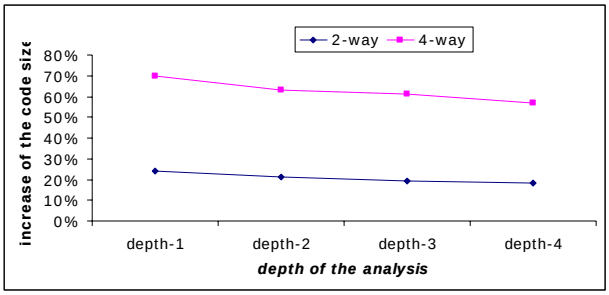


Figure 13. Code size increase as a function of the analysis depth

The cost in code size is higher for larger pipelines because (a) a single-cycle distance is enforced by as many NOPs as the pipeline width, and (b) a larger pipeline augments the overlapping of blocks and then augments the risks of resource conflicts. Results per benchmark are given in Figure 15.

Figure 14 gives further insight in how the increase is broken down into the length of the common padding block, the number of calls to this block and the number of NOPs added to implement the 1-cycle delays. The most severe increases in code size are due to the length of the padding block. For example, `jfdctint` requires a 649-instruction-long padding block while the original code has 519 instructions. This padding length is due to a 225-instruction-long basic block that seriously delays the availability of some resources.

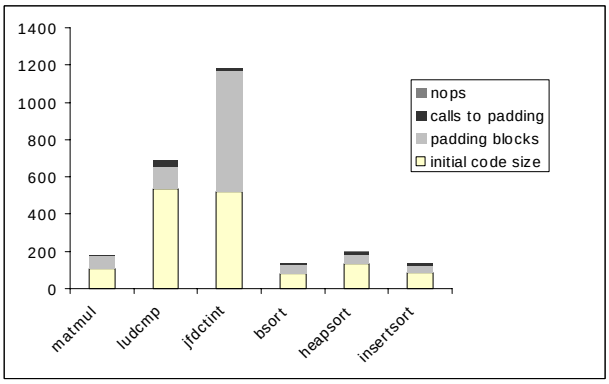
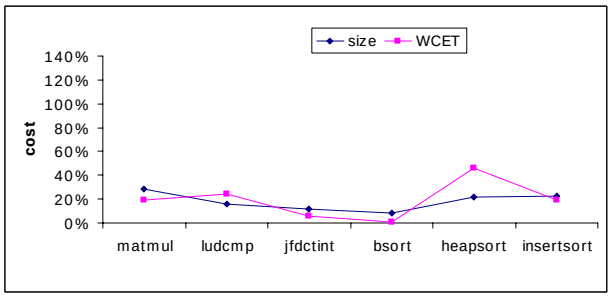
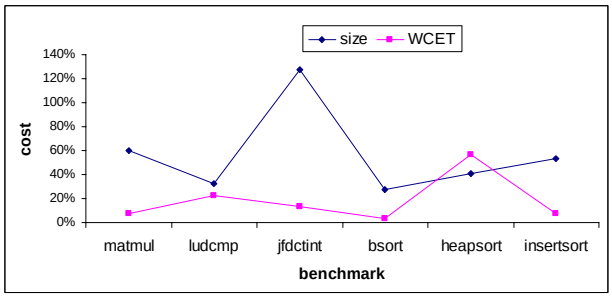


Figure 14. Breaking down of the code size increase (4-way pipeline, depth-4 policy)



(a) 2-way pipeline



(a) 4-way pipeline

Figure 15. Code size and WCET increase with the *depth-4* analysis

5.3. Impact on the real WCET

As said before, long timing effects can occur for long sequences of blocks (*i.e.* the execution of a basic block can have an impact on the execution of a very distant block). Measuring them involves analysing the execution traces of all the possible block sequences of any length which is very costly both in computing time and in memory requirements. This cost is generally unaffordable. However, we have analyzed up-to-6-block sequences and, for each of the benchmarks, we have observed some positive long timing effects (some of them spanning over 6-block sequences). This justifies the need for a solution to make the execution time predictable.

We have evaluated the real WCET of each benchmark code, without and with code padding. We used the symbolic execution method [9] that simulates every possible path. To make it possible, we have limited the size of the data so as to keep the number of possible paths reasonable. Figure 16 shows the results obtained with different analysis depths.

As expected, code padding, that enforces some delay between the execution of successive basic blocks and then limits the instruction parallelism, is responsible for an increase of the execution time. Note that the plotted time is the *real* WCET, not the estimated one (since we cannot make WCET estimations by static analysis when the target architecture generates long timing effects). Augmenting the depth of the analysis helps greatly in

limiting the cost in performance which comes to about 19% on average for the *depth-4* algorithm. This cost can be considered as moderate if we keep in mind that the WCET of the padded code can be estimated quickly, easily, tightly and, above all, safely.

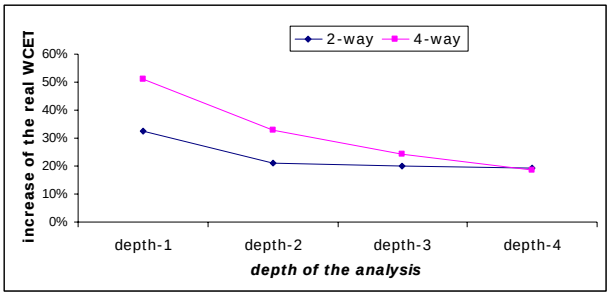


Figure 16. WCET increase as a function of the analysis depth

5.4. Discussion

When having to evaluate the WCET for a program that is to be run on a high-performance architecture, two strategies might be considered. The first one consists in using a method that takes into account any possible long timing effect (of any length and of any value). As far as we know, the only method doing that is the one implemented in the `aiT` tool by the AbsInt company. Its drawbacks include high computation times, complexity of the task of modeling the processor architecture (in the case where a new processor is targeted) and the use of pessimistic assumptions that might produce inaccurate WCET estimates.

The second possible strategy, which is the one we incline towards, aims to make the hardware/ software pair predictable. In [13], we proposed some modifications to the processor architecture to eliminate the possible interferences between distant blocks along the execution path. These modifications included two components: the first one prescheduled the instructions as they enter the reorder buffer; the second one acted as a gate that delays the fetch of a new basic block until it cannot be impacted by another block under execution in the pipeline. This scheme increased the mean execution time by 21% (2-way) to 42% (4-way).

The approach proposed here clearly has a lower cost in terms of execution time: it is smaller by one third for a 2-way processor (19.4% against 21%) and by more than one half for a 4-way processor (18.5% against 42%). This is because we compute the distance required between successive basic blocks off-line. Then we know exactly which instructions belong to the blocks and we exploit profiling information to identify the data and resource dependencies that result in timing interferences. On the contrary, the runtime mechanism proposed in [9] does not know anything about a block that is to be fetched. Then, it has to make pessimistic assumptions and it enforces unnecessarily long distances between the basic blocks.

Moreover, our solution does not need any particular hardware and only requires that a free instruction is available in the instruction set. As mentioned in Section 3.2, such an instruction exists in most processors. Then the code padding method can be used immediately (*i.e.* without waiting that a processor manufacturer decides to design a processor compatible with safe WCET evaluation). The required effort is moderate since the code transformation is done at the assembly level.

Code padding has a cost both in code size and in execution time. However, if we want to keep the evaluation of the WCET simple, the only alternative is to use simpler processors (scalar, with in-order instruction scheduling, etc.) that were proved to be LTE-free. However, they might not meet the performance requirements.

6. Conclusion

This paper deals with timing interferences that make the evaluation of the WCET of a task complicated, pessimistic and possibly unsafe. This problem has already been addressed in a previous work where a processor was designed to prevent timing interferences between basic blocks while keeping most part of the performance. However, the proposed solution has not yet been implemented in a real-life processor. Our purpose is to show how the prevention of timing interferences can be done by transforming the source code, which does not require any specific hardware.

Our approach consists in profiling the execution of basic blocks and of *n*-blocks sequences extracted from the Control Flow Graph of the application. This can be done using a cycle-level simulator and is much faster than simulating all the possible execution paths. Execution profiles are then analyzed to detect data dependencies and resource conflicts that could generate interferences between distant blocks. Filler-instructions, which are discarded from the pipeline as soon as they have been decoded, are inserted in the source code to enforce a distance between blocks so that the interferences are eliminated.

Performance analysis show that, even if the number of added padding instructions is significant, the impact on the worst-case execution time is moderate (a mean slowdown of 19% has been measured).

The increase in the code size and in the real WCET is sensible but this is the cost to pay for predictability, and thus for safety. The WCET of padded codes can be evaluated accurately using simple state-of-the-art methods.

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”, *30th Int. Symp. on Computer Architecture*, may 2003.

[2] F. Bodin, I. Puaut, “A WCET-oriented static branch prediction scheme for real-time systems”, *17th Euromicro Conf. on Real-Time Systems*, july 2005.

[3] J. Engblom, *Processor pipelines and static worst-case execution time analysis*, PhD thesis, Uppsala University, april 2002.

[4] A. Ermedahl, *A Modular Tool Architecture for Worst-Case Execution Time Analysis*, PhD thesis, Uppsala University, june 2003.

[5] R. Heckmann, M. Langenbach, S. Thesing, R. Wilhelm, “The influence of processor architecture on the design and the results of WCET tools”, *Proceedings of the IEEE*, vol. 91, n°7, july 2003.

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

[7] X. Li, A. Roychoudhury, T. Mitra, “Modeling Out-Of-Order Processors for Software Timing Analysis”, *IEEE Real-Time Systems Symposium*, december 2004.

[8] S.-S. Lim, S. Min, M. Lee, C. Park, H. Shin, C. S. Kim, “An Accurate Instruction Cache Analysis Technique for Real-Time Systems”, *Workshop on Architectures for Real-Time Applications*, 1994.

[9] T. Lundqvist, P. Stenström, “An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution”, *Real-Time Systems*, 17(2), 1999.

[10] T. Lundqvist, P. Stenström, “Timing Anomalies in Dynamically Scheduled Processors," *IEEE Real-Time System Symposium (RTSS'99)*, december 1999.

[11] I. Puaut, D. Decotigny, “Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems”, *23rd Int. Real-Time Systems Symp.*, december 2002.

[12] C. Rochange, P. Sainrat, “Towards Designing WCET-predictable Processors”, *3rd Workshop on Worst-Case Execution Time Analysis*, june 2003.

[13] C. Rochange, P. Sainrat, “A Time-Predictable Execution Mode for Superscalar Pipelines with Instruction Prescheduling”, *ACM International Conference on Computing Frontiers*, may 2005.

[14] J. Souyris, E. Le Pavec, G. Himbert, V. Jegu, G. Borios, R. Heckmann, “Computing the Worst-Case Execution Time of an Avionics Program by Abstract Interpretation”, *5th Workshop on WCET Analysis*, july 2005.

[15] H. Theiling, C. Ferdinand”, Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis”, *IEEE Real-Time Systems Symposium*, december 1998.

[16] R. Wilhelm, J. Engblom, S. Thesing, D. Whalley, “Industrial Requirements for WCET Tools”, *3rd Workshop on WCET Analysis*, june 2003.