Dissecting Execution Traces to Understand Long Timing Effects

Christine Rochange and Pascal Sainrat

February 2005

Rapport IRIT-2005-6-R

Contents

1. Introduction

Designing real-time systems requires being able to evaluate the worst-case execution time (WCET) of applications. This time is used to schedule the different tasks with the aim to guarantee that the deadlines will be met.

Determining the WCET by measurement would require exploring every possible execution path. This involves two kinds of difficulties. First, input data sets that insure a total coverage of all the possible paths must be available. Building these data sets is complex in the general case. Second, the number of paths to be explored often makes the measurement time prohibitive. For these reasons, methods to evaluate the WCET, which are based on static analysis of the code and limit measurements to small parts of code (basic blocks) are generally preferred.

The Implicit Path Enumeration Technique (IPET) [LiMa95] expresses the search of the WCET as an optimization problem. The objective function represents the execution time of the program, which is the sum of the individual execution times of the basic blocks weighted by their number of executions. This function is to be maximized under a set of constraints that link the numbers of executions of the basic blocks (as we will see later, the numbers of transitions from one block to another one are also taken account, and are weighted by the gain due to the overlapping of the two blocks in the pipeline). Most of these constraints are built from the Control Flow Graph (CFG) of the program, but some other ones are produced by a preliminary flow analysis and specify loop bounds and infeasible paths. An example of the problem specification using the IPET method will be given in section 4.1.2.
Unfortunately, it has been shown that static WCET analysis can under-estimate the execution time when a high-performance processor is considered. This is due to timing effects between distant blocks (*long timing effects*, or LTEs). This phenomenon will be described in section 2.

In this report, our goal is to examine the execution of some sequences of blocks in a pipeline to gain a better understanding of LTEs.

2. Long timing effects

When considering a pipelined processor, one generally thinks to the natural speedup achieved when two basic blocks executed in sequence overlap in the pipeline. Now, Engblom showed [Engb02] that some interactions between distant blocks can also exist. He proposed a general temporal model that relates the execution times of blocks and those of sequences of blocks. This model is expressed by the equations and diagram given in Figure 1.

$$t_{B_1\ldots B_n} = \sum_{j=1}^{n} t_{B_j} + \sum_{1\le i<k\le n} \delta_{B_i\ldots B_k}$$

$$\delta_{B_1\ldots B_n, n>2} = t_{B_1\ldots B_n} - t_{B_2\ldots B_n} - t_{B_1\ldots B_{n-1}} + t_{B_2\ldots B_{n-1}}$$

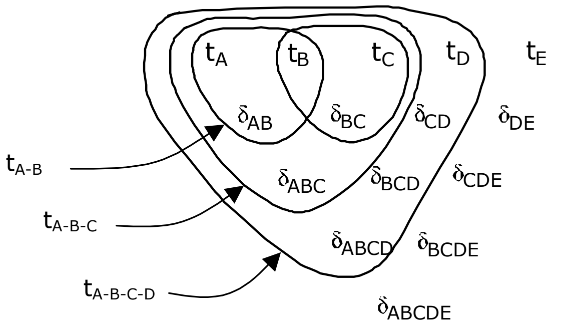


Figure 1. ***Engblom's timing model.***

A component of the execution time of the sequence of blocks $B_1 \ldots B_n$ relates to the impact on the execution of the sequence $B_{i+1} \ldots B_k$ of the preceding execution of the block B_i and is called a *long timing effect* (LTE). Engblom has shown that such an effect can only occur if B_i stalls the execution of a later instruction belonging to the sequence $B_{i+1} \ldots B_k$ or if B_i is finally parallel to $B_{i+1} \ldots B_k$, that is if B_i terminates after the sequence. Since we assume in our processor model that instructions complete in the program order (which is true in most of current processors), we will not consider the second possibility and we will only retain instruction stalling as a source of long timing effects.

Engblom has analysed many examples and has found that long timing effects can be unbounded, i.e. that they can occur for block sequences of any length (potentially full execution paths). This means that identifying all the possible long timing effects would require examining all the possible complete execution paths, which obviously goes against the basic principle of static WCET analysis.

Unfortunately, Engblom has found that long timing effects can either be positive, negative or null. Ignoring the negative effects can make the estimated WCET longer than the real WCET. WCET over-estimation is generally undesirable, first because it can lead to budget components far too much powerful compared to the real requirements (and then these components will be under-used) and second because it can lead to the wrong conclusion that the system cannot be scheduled to meet the deadlines. However, positive long timing effects are far more dangerous because ignoring them during the WCET analysis can engender WCET under-estimation which is not acceptable in a hard real-time context.

In the rest of this report, we will illustrate long timing effects with concrete examples.

3. Methodology

3.1 Example code to be dissected

To illustrate long timing effects, we will consider the very simple example code that implements the bubble-sort algorithm, shown in Figure 2. We have compiled this code with `gcc` targeted for

the PowerPC instruction set architecture. All optimisations were disabled, as it is usually the case for real-time systems. The assembly code extracted by `objdump` from the executable code is given in Figure 3, and the corresponding Control Flow Graph is shown in Figure 4.
The source code includes two nested loops with the same worst-case number of iterations: N-1, where N is the size of the array to be sorted. The external loop is controlled by two conditions and is implemented, at compile time, by two successive conditional branches. The internal loop body includes a conditional statement (*if … then …*), and thus two possible paths (the condition depends on the input data).

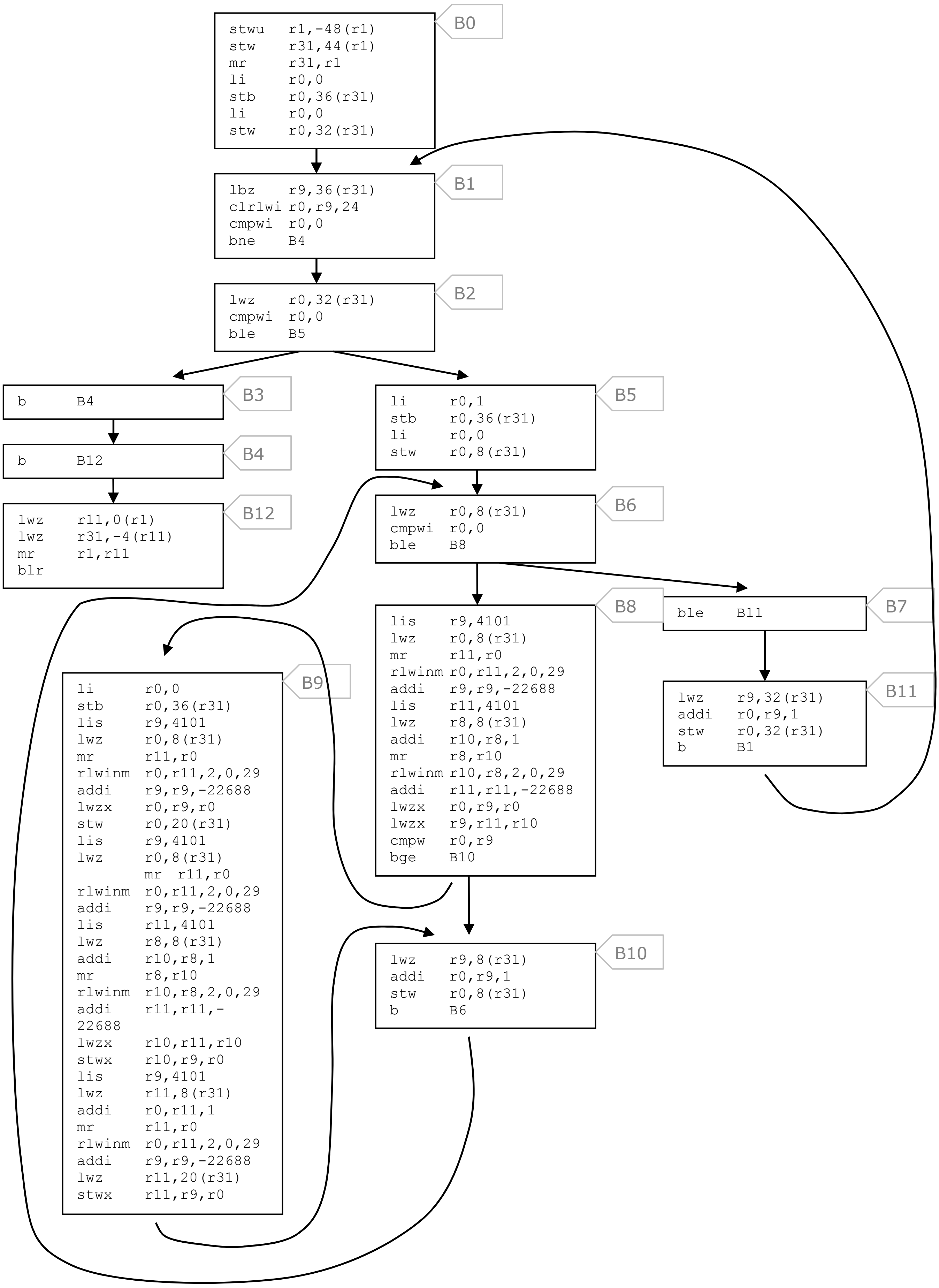
```
int array[20];
#define N 2
int main()
{
     int i, tmp, nb;
     char done;

     done = 0;
     nb = 0;
     while ( !done && (nb < N-1))
     {
          done = 1;
          for (i=0 ; i<N-1 ; i++)
               if (array[i] < array[i+1])
               {
                    done = 0;
                    tmp = array[i];
                    array[i] = array[i+1];
                    array[i+1] = tmp;
               }
          nb++;
     }
}
```

Figure 2. ***Example source code***

```
int main()
{
10000180:      stwu      r1,-48(r1)
10000184:      stw       r31,44(r1)
10000188:      mr        r31,r1
  done = 0;
1000018c:      li        r0,0
10000190:      stb       r0,36(r31)
  nb = 0;
10000194:      li        r0,0
10000198:      stw       r0,32(r31)
  while ( !done && (nb < N-1))
1000019c:      lbz       r9,36(r31)
100001a0:      clrlwi    r0,r9,24
100001a4:      cmpwi     r0,0
100001a8:      bne       100001bc <main+0x3c>
100001ac:      lwz       r0,32(r31)
100001b0:      cmpwi     r0,0
100001b4:      ble       100001c0 <main+0x40>
100001b8:      b         100001bc <main+0x3c>
100001bc:      b         100002b4 <main+0x134>
    {
      done = 1;
100001c0:      li        r0,1
100001c4:      stb       r0,36(r31)
      for (i=0 ; i<N-1 ; i++)
100001c8:      li        r0,0
100001cc:      stw       r0,8(r31)
100001d0:      lwz       r0,8(r31)
```

```
      100001d4:      cmpwi     r0,0
      100001d8:      ble       100001e0 <main+0x60>
      100001dc:      b         100002a4 <main+0x124>
          if (array[i] < array[i+1])
      100001e0:      lis       r9,4101
      100001e4:      lwz       r0,8(r31)
      100001e8:      mr        r11,r0
      100001ec:      rlwinm    r0,r11,2,0,29
      100001f0:      addi      r9,r9,-22688
      100001f4:      lis       r11,4101
      100001f8:      lwz       r8,8(r31)
      100001fc:      addi      r10,r8,1
      10000200:      mr        r8,r10
      10000204:      rlwinm    r10,r8,2,0,29
      10000208:      addi      r11,r11,-22688
      1000020c:      lwzx      r0,r9,r0
      10000210:      lwzx      r9,r11,r10
      10000214:      cmpw      r0,r9
      10000218:      bge       10000294 <main+0x114>
            {
              done = 0;
      1000021c:      li        r0,0
      10000220:      stb       r0,36(r31)
              tmp = array[i];
      10000224:      lis       r9,4101
      10000228:      lwz       r0,8(r31)
      1000022c:      mr        r11,r0
      10000230:      rlwinm    r0,r11,2,0,29
      10000234:      addi      r9,r9,-22688
      10000238:      lwzx      r0,r9,r0
      1000023c:      stw       r0,20(r31)
              array[i] = array[i+1];
      10000240:      lis       r9,4101
      10000244:      lwz       r0,8(r31)
      10000248:      mr        r11,r0
      1000024c:      rlwinm    r0,r11,2,0,29
      10000250:      addi      r9,r9,-22688
      10000254:      lis       r11,4101
      10000258:      lwz       r8,8(r31)
      1000025c:      addi      r10,r8,1
      10000260:      mr        r8,r10
      10000264:      rlwinm    r10,r8,2,0,29
      10000268:      addi      r11,r11,-22688
      1000026c:      lwzx      r10,r11,r10
      10000270:      stwx      r10,r9,r0
              array[i+1] = tmp;
      10000274:      lis       r9,4101
      10000278:      lwz       r11,8(r31)
      1000027c:      addi      r0,r11,1
      10000280:      mr        r11,r0
      10000284:      rlwinm    r0,r11,2,0,29
      10000288:      addi      r9,r9,-22688
      1000028c:      lwz       r11,20(r31)
      10000290:      stwx      r11,r9,r0
      10000294:      lwz       r9,8(r31)
      10000298:      addi      r0,r9,1
      1000029c:      stw       r0,8(r31)
      100002a0:      b         100001d0 <main+0x50>
            }
            nb++;
      100002a4:      lwz       r9,32(r31)
      100002a8:      addi      r0,r9,1
      100002ac:      stw       r0,32(r31)
          }
      100002b0:      b         1000019c <main+0x1c>
      }
      100002b4:      lwz       r11,0(r1)
      100002b8:      lwz       r31,-4(r11)
      100002bc:      mr        r1,r11
      100002c0:      blr
```

Figure 3. ***Example assembly code.***



```
B0
stwu   r1,-48(r1)
stw    r31,44(r1)
mr     r31,r1
li     r0,0
stb    r0,36(r31)
li     r0,0
stw    r0,32(r31)
```

```
B1
lbz    r9,36(r31)
clrlwi r0,r9,24
cmpwi  r0,0
bne    B4
```

```
B2
lwz    r0,32(r31)
cmpwi  r0,0
ble    B5
```

```
B3
b      B4
```

```
B4
b      B12
```

```
B12
lwz    r11,0(r1)
lwz    r31,-4(r11)
mr     r1,r11
blr
```

```
B5
li     r0,1
stb    r0,36(r31)
li     r0,0
stw    r0,8(r31)
```

```
B6
lwz    r0,8(r31)
cmpwi  r0,0
ble    B8
```

```
B7
ble    B11
```

```
B8
lis    r9,4101
lwz    r0,8(r31)
mr     r11,r0
rlwinm r0,r11,2,0,29
addi   r9,r9,-22688
lis    r11,4101
lwz    r8,8(r31)
addi   r10,r8,1
mr     r8,r10
rlwinm r10,r8,2,0,29
addi   r11,r11,-22688
lwzx   r0,r9,r0
lwzx   r9,r11,r10
cmpw   r0,r9
bge    B10
```

```
B11
lwz    r9,32(r31)
addi   r0,r9,1
stw    r0,32(r31)
b      B1
```

```
B9
li      r0,0
stb     r0,36(r31)
lis     r9,4101
lwz     r0,8(r31)
mr      r11,r0
rlwinm  r0,r11,2,0,29
addi    r9,r9,-22688
lwzx    r0,r9,r0
stw     r0,20(r31)
lis     r9,4101
lwz     r0,8(r31)
        mr  r11,r0
rlwinm  r0,r11,2,0,29
addi    r9,r9,-22688
lis     r11,4101
lwz     r8,8(r31)
addi    r10,r8,1
mr      r8,r10
rlwinm  r10,r8,2,0,29
addi    r11,r11,-
22688
lwzx    r10,r11,r10
stwx    r10,r9,r0
lis     r9,4101
lwz     r11,8(r31)
addi    r0,r11,1
mr      r11,r0
rlwinm  r0,r11,2,0,29
addi    r9,r9,-22688
lwz     r11,20(r31)
stwx    r11,r9,r0
```

```
B10
lwz    r9,8(r31)
addi   r0,r9,1
stw    r0,8(r31)
b      B6
```

Figure 4. ***Example Control Flow Graph***

3.2 Simulation framework

The measurements presented in this paper were done using a simulation framework that we developed and that consists of three modules:

- an **instruction-set simulator** able to fetch an executable code into the simulated memory, and to decode and execute instructions (`www.irit.fr/recherches/ARCHI/MARCH/` → *tools* → *GLISS*)
- a **timing simulator** that models the processor architecture and was developed on top of SystemC (`www.systemc.org`). The simulated architecture will be described in section 3.3.
- a **simulation controller** that controls the execution path and enables three simulation modes:
    - ***full-path simulation***: the program is executed from the beginning to the end, with a given input data set.
    - ***graph simulation***: the controller extracts the Control Flow Graph from the program binary code and builds the list of all the possible sequences of blocks shorter than a fixed limit. Then, it guides the simulation along all these sequences to measure their individual execution times, reinitializing the processor between two sequences. These times can then be analysed to compute inter-block timing effects.
    - ***symbolic simulation***: this mode implements a technique proposed by Lundqvist and Stenström [LuSta99] that consists in assuming that the input data is unknown and in propagating this "unknown" value through the computations. Whenever a conditional branch with an "unknown" condition is encountered, the simulation controller first enforces the exploration of one of the possible paths and later guides the execution onto the second possible path. This mode allows simulating all the possible paths in a program without having to determine input data sets that guarantee a total coverage of these paths.

3.3 Processor model

We have carried out a series of experiments to highlight the existence of long timing effects and to collect execution traces that can help in understanding their origin. These experiments involved the use of a cycle-level processor simulator that we developed on top of the SystemC environment.

The simulated architecture is a 2-way superscalar processor, with dynamic instruction scheduling. The 6-stage pipeline is shown in Figure 5. Memory accesses are processed in order, and both the instruction cache and the data cache are considered perfect (i.e. the memory latency is constant). Branch prediction is also perfect, i.e. every branch is well predicted. Size information on the simulated processor is given in Table 1.

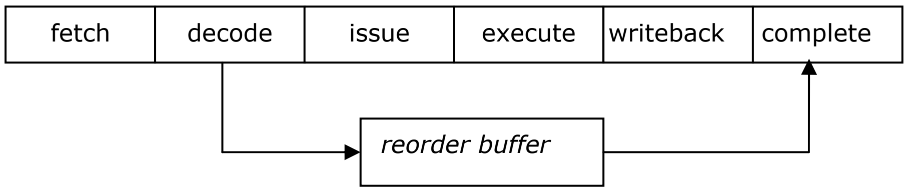


Figure 5. ***Pipeline of the simulated processor.***

pipeline width	2-way
fetch queue size	16
instruction cache	perfect (100% hit rate)
branch predictor	perfect (no mispredictions)
max. number of pending branches	3
re-order buffer size	16
number of functional units (latency)	
integer add (1 cycle)	2
integer mul/div (6 cycles)	1
floating-point add (3 cycles)	1
floating point mul/div (6/15 cycles)	1
load/store (2 cycles)	1
data cache	perfect (100% hit rate)

Table 1. ***Simulated processor***

3.4 Determining the *real* WCET and estimating it by the IPET method

3.4.1 Determining the real WCET

Determining the *real* worst-case execution times requires simulating every possible execution path. While it is generally too costly in time for applications that come from the real world, we can afford it for the very simple code that we intend to dissect.

For our example code, it seems obvious that the longest path is followed whenever the input array is sorted in the reverse order. Then, it would be easy to initialize the array that way and then to simulate the program once with this input data. However, due to possible timing anomalies [LuSt99b], it is difficult to guarantee that the path that seems to be the longest one at first sight is actually the worst-case path. For this reason, we decided to measure all the possible paths.

We got round the problem of determining the corresponding input data sets by using the symbolic execution mode of our simulator. This mode consists in initialising every input data with the « *unknown* » value, in propagating « *unknown* » values through the computations and, each time a branch with an « *unknown* » condition is encountered, in exploring both possible paths. At the end, when all the possible paths have been measured, the WCET is the longest execution time.

3.4.2 Estimating the WCET by the IPET method

We also have evaluated the WCET of the program using the IPET method [LiMa95]. This method consists in expressing the overall execution time of the program by the sum of the execution times of the basic blocks weighted by their respective numbers of instances on a given execution path. Edges that link basic blocks in the Control Flow Graph can also be taken into account, with negative execution times, to reflect the gain due to the overlapping of successive basic blocks in the pipeline. The overall execution time is then maximised under some constraints that express the possible control flow,(we used the *lp_solve* tool that implements integer linear programming algorithms). Structural constraints link the numbers of instances of blocks and edges in accordance with the CFG structure: they were built automatically by a simple Perl script. Other constraints express the results of the flow analysis (loop bounds, etc.) and were added by hand. The execution times of the basic blocks as well as the gains related to sequences of two blocks were obtained using the simulator described in sections 3.2 and 3.3.

3.5 Flow analysis

Note that symbolic execution might generate unfeasible paths. For example, our knowledge of the high-level semantics of the program under study in this report makes us conclude that not every pair of array elements can be unsorted at a given iteration of the external loop. The last pair of elements, for instance, is sorted after the first iteration. Thus, in every subsequent iteration of the external loop, and in the last iteration of the internal loop (the one that processes the last pair of array elements), only one of the two paths should be explored (since the condition is always false). Symbolic execution does not have this kind of considerations and explores the two paths. This will not distort our results since we did not include any constraint expressing the infeasibility of one of them when evaluating the WCET by the IPET method either.

4. General results

4.1 Real and estimated WCETs

4.1.1 Real WCET

Figure 6 shows the results generated by our simulator used in the "symbolic execution" mode. Since we fixed the size of the array to 2, there are only two possible paths depending on whether the array is already sorted at initialization or not. As expected, the longest execution path is observed for an initially unsorted array. Its execution time, which is the **actual WCET**, is **70 cycles**.

```
Path #1:
  blocks: 0 - 1 - 2 - 5 - 6 - 8 - 9 - 10 - 6 - 7 - 11 - 1 - 2 - 3 - 4 - 12
  execution time = 70
Path #2:
  blocks: 0 - 1 - 2 - 5 - 6 - 8 - 10 - 6 - 7 - 11 - 1 - 4 - 12 -
  execution time = 47

Worst-Case Path: 0 - 1 - 2 - 5 - 6 - 8 - 9 - 10 - 6 - 7 - 11 - 1 - 2 - 3 - 4 - 12
(WCET=70)
```

Figure 6. ***Results of simulation with symbolic execution***

4.1.2 Estimated WCET

In order to evaluate the WCET by the IPET method, we first measured (by simulation) the individual execution times of basic blocks. Results are given in Table 1.

block	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
time	10	10	9	6	6	8	9	6	18	27	9	9	10

Table 2. ***Measured individual execution times of basic blocks (in cycles)***

We also measured all the possible sequences of two basic blocks extracted from the Control Flow Graph. From the execution time t_{i-j} of the sequence composed of blocks *i* and *j*, and from the individual execution times t_i and t_j of the two blocks, one can compute the gain due to the overlapping of the two blocks in the pipeline:

$$g_{i-j} = t_{i-j} - t_i - t_j$$

The measured execution times of all the possible two-block sequences as well as the corresponding gains are given in Table 3.

seq.	**0-1**	**1-2**	**1-4**	**2-3**	**2-5**	**3-4**	**5-6**	**6-7**	**6-8**	**7-11**	**8-10**	**9-10**	**10-6**	**11-1**	**8-9**	**4-12**
time	14	11	10	9	11	7	11	9	19	10	20	30	12	13	35	11
gain	-6	-8	-6	-6	-6	-5	-6	-6	-8	-5	-7	-6	-6	-6	-10	-5

Table 3. ***Execution times of two-block sequences and corresponding gains due to pipelined execution***

The *lp_solve* input file corresponding to our example is given in Figure 7: `xi` stands for the number of instances of block `i` in an execution path, and `xitoj` stands for the number of transitions from block `i` to block `j` in the path. The first line specifies the objective which is to maximize the overall execution time. The weights associated to the numbers of instances in the expression of the overall execution times are the individual execution times of basic blocks (e.g. block 0 lasts 10 cycles) and the gains due to pipelined execution (e.g. the sequence of blocks 1 and 2 is executed in 11 cycles, and then the corresponding gain is 11–10–9 = -8 cycles). The first set of constraints describes the CFG structure. The second set adds flow information: the first constraint of this set specifies the maximum number of executions of the condition testing of the external loop; the second bounds the number of iterations of the internal loop, in relation with the number of iterations of the external loop (via `x5to6`).

```
max: 10 x0 + 10 x1 + 9 x2 + 6 x3 + 6 x4 + 8 x5 + 9 x6 + 6 x7 + 18 x8 + 27
x9 + 9 x10 + 9 x11 + 10 x12 + -6 x0to1 + -8 x1to2 + -6 x1to4 + -6 x2to3 +
-6 x2to5 + -5 x3to4 + -6 x5to6 + -6 x6to7 + -8 x6to8 + -5 x7to11 + -7
x8to10 + -6 x9to10 + -6 x10to6 + -6 x11to1 + -10 x8to9 + -5 x4to12;

x0 = 1;
x0 = x0to1;
x1 = x0to1 + x11to1;
x1 = x1to2 + x1to4;
x2 = x1to2;
x2 = x2to3 + x2to5;
x3 = x2to3;
x3 = x3to4;
x4 = x1to4 + x3to4;
x4 = d15;
x5 = x2to5;
x5 = x5to6;
x6 = x5to6 + x10to6;
x6 = x6to7 + x6to8;
x7 = x6to7;
x7 = x7to11;
x8 = x6to8;
x8 = x8to10 + x8to9;
x9 = x8to9;
x9 = x9to10;
x10 = x8to10 + x9to10;
x10 = x10to6;
x11 = x7to11;
x11 = x11to1;
x12 = x4to12;
x12 = 1;

x1 <= 2;
x6 <= 2 x5to6;
int x0;
```

Figure 7. ***Specifications for WCET evaluation by the IPET method***

The results returned by *lp_solve* include the solution found for the variables to maximize the overall execution time while respecting the flow constraints, as well as the value of the WCET. In the case of our example, we obtained the values given in Figure 8.

block	0	1	2	3	4	5	6	7	8	9	10	11	12
xi	1	2	2	1	1	1	2	1	1	1	1	1	1

seq	0-1	1-2	1-4	2-3	2-5	3-4	5-6	6-7	6-8	7-11	8-10	9-10	10-6	11-1	8-9	4-12
xitoj	1	2	0	1	1	1	1	1	1	1	0	1	1	1	1	1

WCET = 68 cycles

Figure 8. ***Results of WCET evaluation by IPET***

4.1.3 Comparing the real and the estimated WCETs

It appears that the WCET estimated using the IPET method is underestimated by 2 cycles against the actual WCET (e.g. by about 2.9%).

4.2 Long timing effects

As many as 155 possible sequences of more than two basic blocks have been extracted from the Control Flow Graph (we filtered out all the sequences that could be built from the CFG but that did not belong to any of the two possible paths). While most of these sequences do not exhibit any long timing effect, other ones generate either positive or negative effects. Table 4 gives the distribution of long timing effects: 9 sequences of three blocks and more exhibit a positive LTE.

long timing effect	-2	-1	0	1	2
# sequences	2	5	139	8	1

Table 4. ***Distribution of long timing effects***

As we said in introduction, the length of the sequences that might generate long timing effects is not bounded. The distribution of the lengths of the sequences associated with a long timing effect (non-null) is given in Table 5, and the distribution of the lengths of the sequences associated with a *positive* long timing effect is given in Table 6. We can observe that one sequence as long as the longest path (16 blocks) generates a LTE (which is negative). Positive LTEs, that should not be ignored to estimate a safe WCET, happen for sequences as long a 6 blocks in this example. In [Engb02], it has been shown how positive LTEs could span over complete execution paths.

sequence length	3	4	5	6	7	8	9	10	11	12	13	14	15	16
# sequences	18	18	18	18	17	15	13	11	9	7	5	3	2	1

Table 5. ***Distribution of the length of sequences with*** **non-nul** ***LTEs***

sequence length	3	4	5	6	7	8	9	10	11	12	13	14	15	16
# sequences	2	5	1	1	0	0	0	0	0	0	0	0	0	0

Table 6. ***Distribution of the length of sequences with*** **positive** ***LTEs***

Table 7 shows the sequences that belong to the longest path and have a positive LTE.

sequence of basic blocks	**LTE**	**# instances**	**impact**
1 – 2 – 3	+1	1	+1
1 – 2 – 3 - 4	+1	1	+1
1 – 2 – 3 – 4 – 12	-2	1	-2
2 – 3 – 4 – 12	+2	1	+2
2 – 5 – 6 – 8 – 9	-1	1	-1
6 - 7 - 11	-1	1	-1
6 – 7 – 11 - 1	+1	1	+1
6 – 8 – 9	+1	1	+1
10 – 6 – 7 – 11	+1	1	+1
10 – 6 – 7 – 11 – 1	-1	1	-1
		TOTAL	**+2**

Table 7. ***Impact of long timing effects on the WCET of the example program***

5. Analysing execution traces

In this section, we will trace and comment the execution of some blocks sequences to highlight some sources of long timing effects.

5.1 First example

Let us see how the sequence of blocks 2-3-4-12 passes through the pipeline. Figure 9 shows the numbered instructions of each of the 4 basic blocks of the sequence. The way these blocks are processed in the pipeline, either alone or in sequences, is described in Figure 10.

block 2	2a	`100001ac:`	`lwz`	`r0,32(r31)`	
	2b	`100001b0:`	`cmpwi`	`r0,0`	→ *depends on 2a (`r0`)*
	2c	`100001b4:`	`ble`	`100001c0`	→ *depends on 2b (implicit register `CR`)*
block 3	3a	`100001b8:`	`b`	`100001bc`	
block 4	4a	`100001bc:`	`b`	`100002b4`	
block 12	12a	`100002b4:`	`lwz`	`r11,0(r1)`	
	12b	`100002b8:`	`lwz`	`r31,-4(r11)`	→ *depends on 12a (`r11`)*
	12c	`100002bc:`	`mr`	`r1,r11`	→ *depends on 12a (`r11`)*
	12d	`100002c0:`	`blr`		

Figure 9. ***Sequence of blocks 2-3-4-12.***

From the traces given in Figure 10, we draw:

$t_2 = 9$ $t_3 = 6$ $t_4 = 6$ $t_{12} = 10$	$t_{2\text{-}3} = 9$ $t_{3\text{-}4} = 7$ $t_{4\text{-}12} = 11$	$t_{2\text{-}3\text{-}4} = 10$ $t_{3\text{-}4\text{-}12} = 12$	$t_{2\text{-}3\text{-}4\text{-}12} = 17$

and then :

$$\delta_{\mathbf{2\text{-}3}} = t_{2\text{-}3} - t_2 - t_3 = \mathbf{-6}$$

$$\delta_{\mathbf{3\text{-}4}} = t_{3\text{-}4} - t_3 - t_4 = \mathbf{-5}$$

$$\delta_{\mathbf{4\text{-}12}} = t_{4\text{-}12} - t_4 - t_{12} = \mathbf{-5}$$

$$\delta_{\mathbf{2\text{-}3\text{-}4}} = t_{2\text{-}3\text{-}4} - t_2 - t_3 - t_4 - \delta_{2\text{-}3} - \delta_{3\text{-}4} = \mathbf{0}$$

$$\delta_{\mathbf{3\text{-}4\text{-}12}} = t_{3\text{-}4\text{-}12} - t_3 - t_4 - t_{12} - \delta_{3\text{-}4} - \delta_{4\text{-}12} = \mathbf{0}$$

$$\delta_{\mathbf{2\text{-}3\text{-}4\text{-}12}} = t_{2\text{-}3\text{-}4\text{-}12} - t_2 - t_3 - t_4 - t_{12} - \delta_{2\text{-}3} - \delta_{3\text{-}4} - \delta_{4\text{-}12} - \delta_{2\text{-}3\text{-}4} - \delta_{3\text{-}4\text{-}12} = \mathbf{+2}$$

No long timing effect is observed for the sequence of blocks 2-3-4. Block 3 is not executed the same way depending on whether it is preceded by block 2 or not: block 2 delays its completion by one cycle. However, thanks to the superscalar pipeline, both blocks complete at the same cycle and the gain $\delta_{2\text{-}3}$ reaches its maximum absolute value (-6 cycles, i.e. all the execution of block 3 overlaps the execution of block 2). If we examine sequence 3-4 when it is executed alone and when it is preceded by block 2, it appears that, in both cases, block 4 completes one cycle after block 3. The distortion of block 3 after block 2 is accounted for by $\delta_{2\text{-}3}$ and then $\delta_{2\text{-}3\text{-}4}$ is nul. In sequence 3-4-12, block 12 passes through the pipeline exactly as when it is executed alone. This is why $\delta_{3\text{-}4\text{-}12}$ also equals zero. On the contrary, the execution of block 12 after the sequence 2-3-4 is distorted: this is due to the limitation of the number of pending branches in the pipeline to three. Then the instructions of block 12 cannot be issued until the branch that ends block 2 has completed. This delay is responsible for the positive timing effect $\delta_{2\text{-}3\text{-}4\text{-}12}$. Note that, while block 12 is delayed by three cycles, $\delta_{2\text{-}3\text{-}4\text{-}12}$ only equals two cycles thanks to the gain related to the simultaneous completion of blocks 2 and 3.

cycle	fetch		decode		issue		execute		writeback		complete	
1	2a	2b										
2	2c		2a	2b								
3			2c		2a	2b						
4					2c		2a					
5												
6							2b		2a			
7							2c		2b		2a	
8									2c		2b	
9											2c	

cycle	fetch		decode		issue		execute		writeback		complete	
1	3a											
2			3a									
3					3a							
4							3a					
5									3a			
6											3a	

cycle	fetch		decode		issue		execute		writeback		complete	
1	4a											
2			4a									
3					4a							
4							4a					
5									4a			
6											4a	

cycle	fetch		decode		issue		execute		writeback		complete	
1	12a	12b										
2	12c	12d	12a	12b								
3			12c	12d	12a	12b						
4					12c	12d	12a					
5							12d					
6							12b	12c	12a	12d		
7									12c		12a	
8									12b			
9											12b	12c
10											12d	

cycle	fetch		decode		issue		execute		writeback		complete	
1	2a	2b										
2	2c		2a	2b								
3	3a		2c		2a	2b						
4			3a		2c		2a					
5					3a							
6							2b	3a	2a			
7							2c		2b	3a	2a	
8									2c		2b	
9											2c	3a

cycle	fetch		decode		issue		execute		writeback		complete	
1	3a											
2	4a		3a									
3			4a		3a							
4					4a		3a					
5							4a		3a			
6									4a		3a	
7											4a	

cycle	fetch		decode		issue		execute		writeback		complete	
1	4a											
2	12a	12b	4a									
3	12c	12d	12a	12b	4a							
4			12c	12d	12a	12b	4a					
5					12c	12d	12a		4a			
6							12d				4a	
7							12b	12c	12a	12d		
8									12c		12a	
9									12b			
10											12b	12c
11											12d	

cycle	fetch		decode		issue		execute		writeback		complete	
1	2a	2b										
2	2c		2a	2b								
3	3a		2c		2a	2b						
4	4a		3a		2c		2a					
5			4a		3a							
6					4a		2b	3a	2a			
7							2c	4a	2b	3a	2a	
8									2c	4a	2b	
9											2c	3a
10											4a	

cycle	fetch		decode		issue		execute		writeback		complete	
1	3a											
2	4a		3a									
3	12a	12b	4a		3a							
4	12c	12d	12a	12b	4a		3a					
5			12c	12d	12a	12b	4a		3a			
6					12c	12d	12a		4a		3a	
7							12d				4a	
8							12b	12c	12a	12d		
9									12c		12a	
10									12b			
11											12b	12c
12											12d	

cycle	fetch		decode		issue		execute		writeback		complete	
1	2a	2b										
2	2c		2a	2b								
3	3a		2c		2a	2b						
4	4a		3a		2c		2a					
5	12a	12b	4a		3a							
6	12c	12d	12a	12b	4a		2b	3a	2a			
7			12c	12d			2c	4a	2b	3a	2a	
8									2c	4a	2b	
9											2c	3a
10					12a	12b					4a	
11					12c	12d	12a					
12							12d					
13							12b	12c	12a	12d		
14									12c		12a	
15									12b			
16											12b	12c
17											12d	

Figure 10. ***Execution traces of blocks 2, 3, 4 and 12.***

5.2 Second example

We will now consider the sequence of blocks 1-2-3-4. These blocks contain the instructions listed in Figure 11 and their execution traces through the pipeline are shown in Figure 12.

block 1	1a	`1000019c:`	`lbz`	`r9,36(r31)`	
	1b	`100001a0:`	`clrlwi`	`r0,r9,24`	*→ depends on 1a (`r9`)*
	1c	`100001a4:`	`cmpwi`	`r0,0`	*→ depends on 1b (r0)*
	1d	`100001a8:`	`bne`	`100001bc`	*→ depends on 1c (implicit register `CR`)*
block 2	2a	`100001ac:`	`lwz`	`r0,32(r31)`	
	2b	`100001b0:`	`cmpwi`	`r0,0`	*→ depends on 2a (`r0`)*
	2c	`100001b4:`	`ble`	`100001c0`	*→ depends on 2b (implicit register `CR`)*
block 3	3a	`100001b8:`	`b`	`100001bc`	
block 4	4a	`100001bc:`	`b`	`100002b4`	

Figure 11. ***Sequence of blocks 1-2-3-4.***

From these traces, we get:

$t_1 = 10$	$t_{1\text{-}2} = 11$	$t_{1\text{-}2\text{-}3} = 12$	$t_{1\text{-}2\text{-}3\text{-}4} = 14$
$t_2 = 9$	$t_{2\text{-}3} = 9$	$t_{2\text{-}3\text{-}4} = 10$	
$t_3 = 6$	$t_{3\text{-}4} = 7$		
$t_4 = 6$			

and then :

$$\delta_{\mathbf{1\text{-}2}} = t_{1\text{-}2} - t_1 - t_2 = \mathbf{-8}$$
$$\delta_{\mathbf{2\text{-}3}} = t_{2\text{-}3} - t_2 - t_3 = \mathbf{-6}$$
$$\delta_{\mathbf{3\text{-}4}} = t_{3\text{-}4} - t_3 - t_4 = \mathbf{-5}$$
$$\delta_{\mathbf{1\text{-}2\text{-}3}} = t_{1\text{-}2\text{-}3} - t_1 - t_2 - t_3 - \delta_{1\text{-}2} - \delta_{2\text{-}3} = \mathbf{+1}$$
$$\delta_{\mathbf{2\text{-}3\text{-}4}} = t_{2\text{-}3\text{-}4} - t_2 - t_3 - t_4 - \delta_{2\text{-}3} - \delta_{3\text{-}4} = \mathbf{0}$$
$$\delta_{\mathbf{1\text{-}2\text{-}3\text{-}4}} = t_{1\text{-}2\text{-}3\text{-}4} - t_1 - t_2 - t_3 - t_4 - \delta_{1\text{-}2} - \delta_{2\text{-}3} - \delta_{3\text{-}4} - \delta_{1\text{-}2\text{-}3} - \delta_{2\text{-}3\text{-}4} = \mathbf{+1}$$

The timing effects of the two-blocks sequences are due to complete (for sequence 2-3) or partial (for sequences 1-2 and 3-4) overlapping of the blocks in the pipeline. Since block 3 fully overlaps with block 2, $\delta_{2\text{-}3}$ equals t_3: block 3 is executed "for free" in the sequence. On the other hand, the only partial overlapping in sequences 1-2 and 3-4 leads to $\delta_{1\text{-}2}$ lower than t_2 and $\delta_{3\text{-}4}$ lower than t_4 (in absolute values). As far as sequence 1-2-3 is concerned, the positive effects $\delta_{1\text{-}2\text{-}3}$ comes from the fact that block 3 can not complete at the same cycle as block 2 when preceded by block 1. The case of sequence 2-3-4 is similar to those commented for the first example (see section 5.1) and the corresponding timing effect is nul. Finally, in sequence 1-2-3-4, the issue of block 4 is delayed, because of the limited number of pending branches, until the completion of block 1. This three-cycle delay partially overlaps the one-cycle delays of sequences 1-2 and 3-4, and this is why $\delta_{1\text{-}2\text{-}3\text{-}4}$ only equals +1.

cycle	fetch		decode		issue		execute		writeback		complete	
1	1a	1b										
2	1c	1d	1a	1b								
3			1c	1d	1a	1b						
4					1c	1d	1a					
5												
6							1b		1a			
7							1c		1b		1a	
8							1d		1c		1b	
9									1d		1c	
10											1d	

cycle	fetch		decode		issue		execute		writeback		complete	
1	2a	2b										
2	2c		2a	2b								
3			2c		2a	2b						
4					2c		2a					
5												
6							2b		2a			
7							2c		2b		2a	
8									2c		2b	
9											2c	

cycle	fetch		decode		issue		execute		writeback		complete	
1	3a											
2			3a									
3					3a							
4							3a					
5									3a			
6											3a	

cycle	fetch		decode		issue		execute		writeback		complete	
1	4a											
2			4a									
3					4a							
4							4a					
5									4a			
6											4a	

cycle	fetch		decode		issue		execute		writeback		complete	
1	1a	1b										
2	1c	1d	1a	1b								
3	2a	2b	1c	1d	1a	1b						
4	2c		2a	2b	1c	1d	1a					
5			2c		2a	2b						
6					2c		1b	2a	1a			
7							1c		1b		1a	
8							1d	2b	1c	2a	1b	
9							2c		1d	2b	1c	
10									2c		1d	2a
11											2b	2c

cycle	fetch		decode		issue		execute		writeback		complete	
1	2a	2b										
2	2c		2a	2b								
3	3a		2c		2a	2b						
4			3a		2c		2a					
5					3a							
6							2b	3a	2a			
7							2c		2b	3a	2a	
8									2c		2b	
9											2c	3a

cycle	fetch		decode		issue		execute		writeback		complete	
1	3a											
2	4a		3a									
3			4a		3a							
4					4a		3a					
5							4a		3a			
6									4a		3a	
7											4a	

cycle	fetch		decode		issue		execute		writeback		complete	
1	1a	1b										
2	1c	1d	1a	1b								
3	2a	2b	1c	1d	1a	1b						
4	2c		2a	2b	1c	1d	1a					
5	3a		2c		2a	2b						
6			3a		2c		1b	2a	1a			
7					3a		1c		1b		1a	
8							1d	2b	1c	2a	1b	
9							2c	3a	1d	2b	1c	
10									2c	3a	1d	2a
11											2b	2c
12											3a	

cycle	fetch		decode		issue		execute		writeback		complete	
1	2a	2b										
2	2c		2a	2b								
3	3a		2c		2a	2b						
4	4a		3a		2c		2a					
5			4a		3a							
6					4a		2b	3a	2a			
7							2c	4a	2b	3a	2a	
8									2c	4a	2b	
9											2c	3a
10											4a	

cycle	fetch		decode		issue		execute		writeback		complete	
1	1a	1b										
2	1c	1d	1a	1b								
3	2a	2b	1c	1d	1a	1b						
4	2c		2a	2b	1c	1d	1a					
5	3a		2c		2a	2b						
6	4a		3a		2c		1b	2a	1a			
7			4a		3a		1c		1b		1a	
8							1d	2b	1c	2a	1b	
9							2c	3a	1d	2b	1c	
10									2c	3a	1d	2a
11					4a						2b	2c
12							4a				3a	
13									4a			
14											4a	

Figure 12. ***Execution traces for blocks 1, 2, 3 and 4***

6. Concluding remarks

The examples examined in the previous section show how a basic block can have a timing impact on a distant subsequent block. We can identify two kinds of positive timing effects:

- the **last block** of the sequence is **delayed** for a reason directly related to the first block. This is mainly due to resource contention, in the broad sense of the word:
    - conflicts for the use of pipeline slots or of functional units (especially long-latency non-pipelined units)

    - full queues (in tests not reported here, we have observed that LTEs were generated by the limited capacity of the reorder buffer)
    - other specific restrictions, like the one that limits the number of pending branches in the pipeline

  The LTEs $\delta_{2\text{-}3\text{-}4\text{-}12}$ and $\delta_{1\text{-}2\text{-}3\text{-}4}$ belong to this category.

- the **gains** related to **shorter sequences** are **reduced** due to resource contention: this can be observed for sequence 1-2-3.

We feel that inter-block data dependences are also likely to produce LTEs. However, we have found that they are very rare in practice.

We are convinced that many other sources of long timing effects can lie in more and more sophisticated processor architectures. This is why we argue for solutions that would eliminate them.

References

[Engb02] J. Engblom. ***Processor Pipelines and Static Worst-Case Execution Time Analysis.*** PhD thesis, University of Uppsala, 2002.

[LiMa95] Y.-T. Li, S. Malik. ***Performance Analysis of Embedded Software using Implicit Path Enumeration.*** ACM SIGPLAN Notices, vol. 30, n°11, 1995.

[LuSt99a] T. Lundqvist, P. Stenström. ***An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution***. Real-Time Systems, vol. 17, n°2. March 1999

[LuSt99b] T. Lundqvist, P. Stenström. ***Timing Anomalies in Dynamically-Scheduled Microprocessors***. IEEE Real-Time Systems Symposium (RTSS), 1999.