Performance Estimation of a Codesign System

LUIZA DE MACEDO MOURELLE and NADIA NEDJAH
Department of Systems Engineering and Computation
State University of Rio de Janeiro
Rua São Francisco Xavier, 524, sala 5022-D, Maracanã, Rio de Janeiro, RJ
BRAZIL

Abstract:- The purpose of this work is to predict the performance of a codesign system, in terms of execution time, considering the hardware function characteristics and those of the target architecture. This allows us to decide whether the critical function should be implemented in hardware before its physical implementation. We can save design time and costs, specifically those related to the synthesis process. The analysis considers two common interface mechanisms. We show that the bus interface, handshake protocol and bus arbitration of the target architecture together with the memory accesses of the hardware function are the characteristics that affect the performance of the codesign system most. We provide an estimate of the execution time that includes these parameters. Finally, we validate our estimation based on a case study.

Key-Words:- Performance estimation, hardware/software codesign, interface mechanism, memory configurations.

1 Introduction

Our work is concerned with software acceleration, using the codesign methodology [4]. Therefore, we start off from an application, specified as a C program, and we identify performance critical regions based on profiling results [2]. The critical region selected for hardware implementation is, then, synthesised into an FPGA, that will serve as a coprocessor in the target architecture. Our codesign system consists of a hardware subsystem, which is the coprocessor responsible for the execution of the critical region, and a software subsystem, which is the main processor, e.g. a microcontroller, responsible for the execution the adapted C code [1-5].

Two different communication mechanisms were investigated: busy-wait and interrupt. In the first one, the main processor reads a memory-mapped control register continuously, leading to the problem of bus contention. In the second one, the main processor waits for an interrupt request of the coprocessor, indicating completion of the operation, thus reducing bus contention, but introducing an additional time overhead corresponding to the interrupt acknowledge cycle.

A specific example is proposed and a simulation procedure is followed, based on our VHDL model of the codesign system [10-12], that will enable us to study details of the architecture, such as bus arbitration, handshake completion, busy-wait cycle and interrupt acknowledge cycle.

Section 2 presents a general formalisation for the execution time of the hardware function, in order to estimate the performance of the codesign system. Section 3 presents a case study, when the characteristics of the target architecture under consideration and those of a critical function, selected for hardware implementation, are introduced. Section 4 validates the formal result. This is done by comparing it to the result obtained from simulation. Section 5 presents some conclusions and future work.

2 General formalisation

A general formalisation for the execution time could be produced, in order to allow the performance of different system architectures to be evaluated, before any implementation is performed. We can express the execution time of the hardware subsystem as:

\[
time_{hw} = time_{pi} + time_{ci} + time_{ex} + time_{co} + time_{po}
\]

where \( time_{pi} \) and \( time_{ci} \) are the times needed to transfer the required parameters from the software subsystem to the hardware subsystem and control from the software subsystem to the hardware subsystem respectively; \( time_{ex} \) represents the execution time of the function into the hardware subsystem; \( time_{co} \) and
The times corresponding to control transfer, \( time_{ci} \) and \( time_{co} \), are related to the interface protocol between the two subsystems. Basically, \( time_{ci} \) is required to start the hardware subsystem operation. This can be done using a control register, located in the hardware subsystem. This type of implementation is discussed in the next section. On the other hand, \( time_{co} \) is required by the software subsystem to identify the end of the hardware subsystem operation, which corresponds to the handshake completion time. This depends on the interface mechanism employed: busy-wait or interrupt. With the former, \( time_{co} \) represents the handshake completion time with busy-wait, say \( time_{cobw} \) while with the latter, \( time_{co} \) represents the handshake completion time with interrupt, say \( time_{coat} \). We discuss both implementations in the Section 3.

The analysis of \( time_{ex} \) is carried out using the function’s VHDL code implemented by the hardware subsystem. It is the sum of two other components:

- \( time_{iops} \): time required by the hardware subsystem to execute the internal operations, without memory accesses;
- \( time_{mem} \): time needed by the hardware subsystem to execute memory accesses only.

The time corresponding to the internal operations, \( time_{iops} \), is estimated from the function’s VHDL code, without taking into account the memory accesses that might exist. The time corresponding to the memory accesses from the hardware subsystem, \( time_{mem} \), depends on the interface between the hardware subsystem and the main memory. Basically, we can write \( time_{mem} \) in terms of the memory read cycle time (\( time_{rdmem} \)), the size of the data-in and their numbers (\( nnemin \)), the memory write cycle time (\( time_{wrmem} \)), the size of the data-out and its number (\( nnemout \)), and the bus bandwidth as:

\[
\begin{align*}
time_{rdmem} &= \sum_{i=1}^{nnemin} \frac{sizemin_{i}}{sizebus} \\
time_{wrmem} &= \sum_{i=1}^{nnemout} \frac{sizememout_{i}}{sizebus} \\
\end{align*}
\]

wherein, \( sizemin_{i} \) and \( sizememout_{i} \) stand for the size of the \( i \)th data-in and data-out respectively.

3 Case study

In this section, we present the target architecture and its characteristics. Some of these are the time taken by the microcontroller and the coprocessor in a memory access, the time taken for transferring control between the two subsystems, the arbitration time, the handshake completion time, the busy-waiting cycle time and the interrupt cycle time. Afterwards, we introduce the estimate expression based on these parameters.

3.1 The target architecture

The target codesign system, shown in Figure 1, can be divided into the software subsystem, which is responsible for the execution of the adapted C code [1-5], and the hardware subsystem, which is responsible for the execution of the critical region (hardware function). Therefore, the former consists of the
microcontroller, while the latter consists of the coprocessor implemented by the FPGA XC4010.

Both subsystems communicate through the main system bus. Integer and pointer parameters are passed to and from the coprocessor via memory-mapped registers, while data arrays are stored in the shared memory, with their related pointers passed to the coprocessor. Two commonly used interface mechanisms were applied: busy-wait and interrupt.

The MC68332 [6,7] is a 32-bit integrated microcontroller, with a 24-bit address bus and 16-bit data bus. The microcontroller runs at 16.78MHz, which gives a main system clock cycle of 60ns, and the coprocessor at 8.39MHz, which gives a coprocessor system clock cycle of 120ns, due to the Xilinx/FPGA internal delays [8,9]. The handshake protocol uses an 8-bit control register, located in the bus interface controller. The microcontroller starts the coprocessor operation by writing into the control register, located in the bus interface controller. The microcontroller starts the coprocessor operation by writing into the control register and setting signal `ncopro_st` to ‘0’. From the simulation results, we observe that it takes 240ns for the microcontroller to write into the control register (`time_c`), 240ns for the microcontroller to read/write into the shared memory 16-bit data, 600ns for the microcontroller to read (`timerdhw`) or write (`timewrhw`) into the coprocessor data registers and 300ns for the coprocessor to read (`timerdacc`) or write (`timewracc`) into the shared memory.

When the coprocessor finishes operation, there is the time corresponding to the handshake completion. The coprocessor sets signal `ncopro_dn` to ‘0’, which corresponds to one of the bits in the control register. As soon as the microcontroller identifies the end of coprocessor operation, it writes into the control register, setting signal `ncopro_st` to ‘1’ and completing the handshake. When using the busy-wait mechanism, the handshake completion time (`time_cobw`) will depend on the busy-wait cycle time and the time taken for signal `ncopro_st` to go to ‘1’.

From the simulation results, we obtained a handshake completion time between 300ns and 540ns. When using the interrupt mechanism, the handshake completion time (`time_coint`) will depend on the interrupt acknowledge cycle time and the time taken by the microcontroller to set signal `ncopro_st` to ‘1’. From the simulation results, seen in Figure 2, we obtained a handshake completion time of 11160ns.

In a shared bus architecture, there will be always the need for bus arbitration, whenever the coprocessor wants to make a memory access. When using the busy-wait mechanism, there might be bus contention, since the microcontroller might be in a busy-wait cycle, which takes 300ns. When the coprocessor finishes operation, it sets signal `copro_dn` to ‘0’, which corresponds to one bit in the control register. Therefore, during the busy-wait cycle, the microcontroller reads the control register and checks, internally, the end of coprocessor operation. In this case, the bus arbitration time (`time_arbbw`) takes between 180ns and 360ns. Figure 3 shows the simulation results during a bus arbitration, when using busy-wait.

When using the interrupt mechanism, there is no bus contention, since the microcontroller is halted after starting the coprocessor, and the bus arbitration time (`time_arbin`) takes 180ns. Figure 4 shows the simulation results during a bus arbitration, when using interrupt.

**3.2 The Hardware Function**

The C program of Figure 5 contains the function `example` to be synthesised in hardware, whose body is based on one main loop, containing two others. During synthesis, the partitioning tool takes the C source program and generates two outputs: the VHDL specifcation of the function `example` and the modified C program. The declaration part of the modified C
program contains the necessary specifications for passing parameters to the coprocessor. Since parameter passing is done via memory-mapped registers, it includes assignments that define the addresses of each parameter, according to their sizes. It also implements the “hardware” call/return mechanism.

Fig. 3: Bus arbitration when using busy-wait

typedef short int array1 [10000];
void example (array1 table, short int i, short int o, short int a) {
    short int c, j, temp;
    for(c=0; c<i; c++) {
        for (j=0; j<o; j++)
            temp += 1;
        for (j = 0; j < a; j++)
            table[c]=temp;
    }
    array1 table; short int i = 10;
    short int o, a = 1;
    main() {
        example (table,i,o,a); exit(0);
    }

Fig. 4: Bus arbitration without contention

The execution time of the hardware function itself \( time_{ex} \) is carried out based on the function’s VHDL code. Applying the expressions defined in the Section 2 for \( time_{ex} \), we have for the internal operations of our example the following:

\[
    time_{pi} = 600 \times \left(\frac{8}{16} + \frac{32}{16} + 3 \times \frac{16}{16}\right) = 360\text{ns}
\]

For the memory cycle time \( time_{mem} \), we consider the memory write cycle time only \( time_{wracc} \) plus the arbitration time \( time_{arb} \), since we are using a shared bus configuration. In the last one, when using busy-waiting, the arbitration time is \( time_{arb bw} \) and when using interrupt, the arbitration time is \( time_{arb int} \), as discussed in Section 2. Besides this, we must consider an extra time (600ns) corresponding to the internal operations for the preparation of the memory access itself, such as address calculation and data buffer initialisation. This is a characteristic of our example and is evaluated from the function’s VHDL code. Hence, the final expression for \( time_{mem} \) when using busy-waiting \( (time_{mem bw}) \), becomes:

\[
    time_{mem bw} = (600 + 300 + 360) \times \left(\frac{16}{16} \times a \times i\right) = 1260 \times a \times i
\]

and the expression for \( time_{mem} \), when using interrupt \( (time_{mem int}) \), becomes:

\[
    time_{mem int} = (600 + 300 + 180) \times \left(\frac{16}{16} \times a \times i\right) = 1080 \times a \times i
\]

In our example, there are no returned parameters from the function. Therefore, we will analyse \( time_{pi} \) only. There are four parameters to be passed to the function: 32-bits \( table \), 16-bit \( i \), 16-bits \( o \) and 16-bits \( a \). Note that \( ixo \) indicates the number of internal operations and \( ixo \) represents that of memory accesses. Besides these, there is an 8-bit coded parameter \( inst \) passed to the coprocessor to select the part of the VHDL state machine related to the function, since there is the possibility of synthesising more than one function. Parameter passing is done through memory-mapped registers. The first parameter is a 32-bit pointer to the array \( table \), which requires two bus accesses. The next parameters are 16-bit integers, requiring only one bus access each. Parameter \( inst \) is not one of the function’s formal parameters, but is part of the parameter passing process, consisting of 8 bits and taking one bus access too. Therefore, the time to send parameters to the hardware function is:

\[
    \text{nstime}_{pi} = \frac{8}{16} \times 360 + \frac{32}{16} \times 360 + 3 \times \frac{16}{16} \times 360 = 360\text{ns}
\]
Finally, we consider the time to transfer control from the coprocessor to the microcontroller (\(\text{time}_\text{co}\)). This corresponds to the handshake completion time, discussed in Section 3.1. Hence, it depends on the interface mechanism employed. When using busy-waiting, \(\text{time}_\text{co}\) corresponds to \(\text{time}_\text{cobw}\) and, when using interrupt, \(\text{time}_\text{co}\) corresponds to \(\text{time}_\text{coint}\).

The estimate for the execution time of the hardware function can, now, be defined in terms of the parameters introduced in Section 2 and developed in Section 3. We provide below two expressions, according to the interface mechanism employed:

\[
\text{time}_\text{exbw} = 600 + (640 + 360 \times o + 1260 \times a) \times i \\
\text{time}_\text{exint} = 600 + (640 + 360 \times o + 1080 \times a) \times i
\]

### 3.3 The estimate

Now that we have the parameters introduced in the previous sections, we must put all together to produce the estimate expression for the execution time of the hardware subsystem (\(\text{time}_\text{hw}\)). We cannot forget to consider the interface mechanism employed. Therefore, the estimate expression, when using busy-wait and when using interrupt is, respectively:

\[
\text{time}_\text{hwbw} = 4980 + (640 + 360 \times o + 1260 \times a) \times i \\
\text{time}_\text{hwint} = 15500 + (640 + 360 \times o + 1080 \times a) \times i
\]

### 4 Validation

We have a model of the codesign system and the function’s VHDL code. Simulation results were, then, obtained for different values of parameter \(i\), using a value of 10 for parameter \(o\) and a value of 1 for parameter \(a\).

\[
\text{time}^e_{\text{hwbw}} = 5220 + 5880 \times i \quad \text{and} \quad \text{time}^e_{\text{hwint}} = 16080 + 5760 \times i
\]

We make the distinction between the estimate value and that obtained by simulation by superscripting the parameter using \(e\) or \(s\), accordingly. Applying the same parameter values to our estimate expressions, we obtain the following:

\[
\text{time}^e_{\text{hwbw}} = 4980 + 5700 \times i \quad \text{and} \quad \text{time}^e_{\text{hwint}} = 15500 + 5520 \times i
\]

Table 1 shows the error introduced by the estimation for different numbers of memory accesses, with \(i\) and \(o\) both equal to 10. Note that the error decreases as the number of memory accesses increases. The maximum error introduced is about 3%.

When the application includes more than ten memory accesses, the error is less than 1%.

<table>
<thead>
<tr>
<th>(a)</th>
<th>(\text{time}^e_{\text{hwbw}}) (ns)</th>
<th>(\text{time}^s_{\text{hwbw}}) (ns)</th>
<th>error(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>49380</td>
<td>49620</td>
<td>0.483676</td>
</tr>
<tr>
<td>1</td>
<td>61980</td>
<td>64020</td>
<td>3.186504</td>
</tr>
<tr>
<td>2</td>
<td>74580</td>
<td>76620</td>
<td>2.66249</td>
</tr>
<tr>
<td>3</td>
<td>87180</td>
<td>89220</td>
<td>2.286483</td>
</tr>
<tr>
<td>10</td>
<td>175380</td>
<td>177420</td>
<td>1.149814</td>
</tr>
</tbody>
</table>

Table 1: Error introduced by the estimation with busy-wait (\(i = 10, o = 10\))

Table 2 shows the error introduced by the estimation for different numbers of internal operations, with ten iterations and a single memory access per iteration. Observe that the error decreases as the number of internal operations increases. The maximum error introduced is about 8%. When the application includes more than 100 internal operations, the error is less than 0.5%.

<table>
<thead>
<tr>
<th>(o)</th>
<th>(\text{time}^e_{\text{hwbw}}) (ns)</th>
<th>(\text{time}^s_{\text{hwbw}}) (ns)</th>
<th>error(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>25980</td>
<td>28020</td>
<td>7.85194</td>
</tr>
<tr>
<td>1</td>
<td>29580</td>
<td>31020</td>
<td>4.868154</td>
</tr>
<tr>
<td>2</td>
<td>33180</td>
<td>34020</td>
<td>2.531646</td>
</tr>
<tr>
<td>3</td>
<td>36780</td>
<td>37860</td>
<td>2.936378</td>
</tr>
<tr>
<td>10</td>
<td>61980</td>
<td>64020</td>
<td>3.291384</td>
</tr>
<tr>
<td>100</td>
<td>385980</td>
<td>388020</td>
<td>0.528525</td>
</tr>
<tr>
<td>200</td>
<td>745980</td>
<td>748020</td>
<td>0.273466</td>
</tr>
<tr>
<td>300</td>
<td>1105980</td>
<td>1108020</td>
<td>0.184452</td>
</tr>
</tbody>
</table>

Table 2: Error introduced by the estimation with busy-wait (\(i = 10, a = 1\))

Table 3 presents the error introduced by the estimation for different numbers of memory accesses, with ten iterations and ten internal operations per iteration. Notice that the error increases with the number memory accesses. However, it does not grow as fast as the latter and seems to stabilise at about 10%.

<table>
<thead>
<tr>
<th>(a)</th>
<th>(\text{time}^e_{\text{hwint}}) (ns)</th>
<th>(\text{time}^s_{\text{hwint}}) (ns)</th>
<th>error(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>59900</td>
<td>60480</td>
<td>0.958995</td>
</tr>
<tr>
<td>1</td>
<td>70700</td>
<td>73680</td>
<td>4.044517</td>
</tr>
<tr>
<td>2</td>
<td>81500</td>
<td>85680</td>
<td>4.878618</td>
</tr>
<tr>
<td>3</td>
<td>92300</td>
<td>97680</td>
<td>5.507781</td>
</tr>
<tr>
<td>10</td>
<td>167900</td>
<td>181680</td>
<td>7.584764</td>
</tr>
<tr>
<td>100</td>
<td>1139900</td>
<td>1261680</td>
<td>9.65221</td>
</tr>
<tr>
<td>200</td>
<td>2219900</td>
<td>2460480</td>
<td>9.777767</td>
</tr>
<tr>
<td>300</td>
<td>3299900</td>
<td>3660480</td>
<td>9.850621</td>
</tr>
</tbody>
</table>

Table 3: Error introduced by the estimation with interrupt (\(i = 10, o = 10\))
Table 4 presents the error introduced by the estimation for different numbers of internal operations, with ten iterations and a single memory access per iteration. Observe that the error decreases as the number of internal operations increases. The maximum error introduced is about 8%. When the application includes more than 100 internal operations, the error is less than 1%.

<table>
<thead>
<tr>
<th>$a$</th>
<th>$time_{hwint}^e$ (ns)</th>
<th>$time_{hwint}$ (ns)</th>
<th>error(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>34700</td>
<td>37680</td>
<td>7.908705</td>
</tr>
<tr>
<td>1</td>
<td>38300</td>
<td>41280</td>
<td>7.218992</td>
</tr>
<tr>
<td>2</td>
<td>41900</td>
<td>44880</td>
<td>6.639929</td>
</tr>
<tr>
<td>3</td>
<td>45500</td>
<td>48480</td>
<td>6.146865</td>
</tr>
<tr>
<td>10</td>
<td>70700</td>
<td>73680</td>
<td>4.044517</td>
</tr>
<tr>
<td>100</td>
<td>394700</td>
<td>397680</td>
<td>0.749346</td>
</tr>
<tr>
<td>200</td>
<td>754700</td>
<td>757680</td>
<td>0.393063</td>
</tr>
<tr>
<td>300</td>
<td>1114700</td>
<td>1117680</td>
<td>0.266624</td>
</tr>
</tbody>
</table>

Table 4: Error introduced by the estimation with interrupt ($i = 10, a = 1$)

The results presented in Table 1 and 2 were based on the busy-wait interface mechanism, while those shown in Table 3 and 4 were based on the interrupt interface mechanism.

5 Conclusions

Our aim is to provide the designer with a mean to estimate the performance of the codesign system for software acceleration, before the implementation of the candidate function in hardware is decided. This would avoid the time and costs associated to the synthesis process, in the case the resulted implementation does not satisfy the expected performance, in terms of execution time. In this case, another candidate function could be chosen instead.

First of all, we obtained some timing characteristics of the codesign system based on simulation results. These were mainly related to the microcontroller accesses to the coprocessor control register and data buffers, the coprocessor memory accesses, the bus arbitration time, the busy-wait cycle time, the interrupt acknowledge time and the handshake completion time. From previous works, we concluded that the coprocessor memory accesses played a key role in the overall performance of the system.

The results obtained for the single-port shared memory validate the proposed estimate expression for the execution time of the hardware subsystem, based on the characteristics of the codesign system and those of the hardware function. The error associated with the estimation was evaluated. We should, however, recall that an error was expected, since some features could not be considered in our approach, such as the synchronisation between the two subsystems, as they operate with different frequencies, and the arbitration time during busy-wait, for which we consider the worst case, in spite of varying between 180ns and 360ns. Despite this, the estimate expression gives the necessary information to predict the performance of the codesign system, in order to decide on the best critical function for hardware implementation and the appropriate interface mechanism.

References:


