# Combining Software Synthesis and Hardware/Software Interface Generation to Meet Hard Real-Time Constraints

Steven Vercauteren
IMEC
Kapeldreef 75,
B-3001 Leuven, Belgium
vercaut@imec.be

Jan Van Der Steen
Philips ITCL
Interleuvenlaan 74-76,
B-3001 Leuven, Belgium
jan.vandersteen@leu.ce.philips.com

Diederik Verkest
IMEC
Kapeldreef 75,
B-3001 Leuven, Belgium
verkest@imec.be

## Abstract

This paper presents an orchestrated combination of software synthesis and automatic hardware/software interface generation to meet hard real-time constraints. The target applications are digital communication systems, which are specified as concurrent communicating processes. The overall approach is theoretically founded and is demonstrated on an industrial strength design, with promising results.

## 1 Introduction

To enable flexible low-cost designs in a short design cycle, most modern embedded systems integrate one or more programmable processor cores onto a single VLSI chip. Thanks to technology, more functionality can be moved to software because microprocessors can deliver the desired performance, obviating the need for much custom logic.

Since embedded systems are concurrent by nature, they are naturally expressed as concurrent programs, specified in terms of communicating processes. Thus, the design burden is shifting to *scheduling* concurrent processes assigned to a single-thread programmable processor, and realizing the communication with the other processes. The latter problem is known as the *hardware/software interfacing* problem. Moreover, embedded systems are subject to stringent timing constraints.

Traditionally, hand-crafted solutions or real-time kernels have been used to solve the scheduling problem. Real-time kernels are specialized operating-systems [7], designed to react *fast* to external events. Kernels trade optimality for generality, which may cause a significant run-time and memory overhead – prohibitive in many cases for embedded systems where performance is paramount and memory is scarce.

*Software synthesis* [4, 5, 8, 13] is an alternative approach to real-time kernels. The aim is to fully exploit the information captured in the system specification to avoid the need for a run-time executive, while still meeting timing constraints.

*Hardware/software interfacing* is considered an error prone and time consuming task leaving little room for optimization or exploration [3]. Recent efforts have led to promising CAD support in this area [1, 2, 5, 6].

In this paper, we present a *complete* solution for embedding real-time software in a customized system architecture, that avoids the use of a run-time executive, where possible. The target applications are digital communication systems. We start from a concurrent process system specification. The processes assigned to a programmable processor core are scheduled using a software synthesis approach, if possible. For this, we build on the key observation that any processor (core) has an inherent fixed-priority preemptive scheduler (cf. interrupts). At the same time, the hardware/software interfaces are automatically generated, partly based on a parameterized library solution. Further, the final implementation is guaranteed to meet real-time constraints.

The remainder of this paper is organized as follows. Section 2 reviews our system specification model. Section 3 describes the goal of this work, as well as the underlying assumptions. Section 4 presents our software synthesis approach. Section 5 details the hardware/software interfacing procedure. Section 6 extensively discusses a case study. Finally, conclusions are drawn in Section 7.

## 2 System Specification Model

In this work, we adhere to the CoWare data model [1]. In this model, a system is hierarchically composed of *processes* whose *ports* are connected by point-to-point communication channels. In turn, a leaf process is composed of concurrent *threads*, all described in the same specification language. Threads of the same process communicate via shared variables; threads of different processes communicate by *Remote Procedure Call (RPC)* over an interconnecting channel. Ports through which an RPC is activated (serviced) are called *master* (*slave*) ports and depicted as filled (open) boxes. Threads that are called upon by RPC are called *slave threads*. The other threads are called *autonomous threads*; their code is executed in an infinite loop. The reader is referred to [1] for details.



0-7695-0078-1/99 $10.00 © 1999 IEEE

## 3 Preliminaries

Our target applications are digital communication systems. These systems are periodic by nature and typically consist of multiple hardware accelerators, control loops (for tracking, acquisition, etc.) and a reactive system, implemented in software. Typically, the reactive system is specified as a process with a single low-rate autonomous thread (for monitoring display information and commands from the user interface) and multiple higher-rate slave threads (for steering the control loops). Therefore, we believe that the following assumptions are realistic for our application domain.

We assume that threads assigned to a processor core are specified in C and merged into a single process. This process consists of one or more slave threads $\tau_1 \ldots \tau_{n-1}$ and one autonomous thread $\tau_n$. Every thread is periodic and leads to an infinite sequence of thread instances, called *jobs*. A thread $\tau_i$ is characterized by a tuple $\sigma_i \equiv (C_i, T_i, D_i, R_i)$ where $C_i$ is the (worst case) execution time of thread $\tau_i$, $T_i$ is the invocation period, $D_i$ is the deadline, and $R_i$ is the first invocation time. The $k$-th job of thread $\tau_i$ is ready for execution at time $R_i + (k-1)T_i$ and, in order to meet its deadline, must be completed no later than $(R_i + (k-1)T_i + D_i)$. The deadlines are less than or equal to the periods, and the deadlines of the slave threads are less than the deadline of the autonomous thread (cf. above). Communication between the different threads is 'unblocked'. The $C_i$ figures accommodate time slots for external communication, which take the form of *delay constraints* that must be met in the implementation.

**Notation 3.1** *delay_constraint*$(p)$ denotes the maximal allowed delay that may be incurred on port $p$ due to the presence of the hardware/software interface.

**Our goal** is to find a $\sigma_i$, for each $\tau_i$, that is 'meaningful' from a system-level perspective [8, 13], and to map the threads – in an automated way – on to a programmable processor core using a software synthesis approach (thus without a kernel), such that the deadlines of all jobs and the delay constraints of all ports are met. If run-time support is unavoidable, our method must notify this. In this case, we provide a (customized) real-time kernel [13], but this is beyond the scope of this paper.

## 4 Software Synthesis Approach

Our software synthesis approach is based on prior results from scheduling theory and on the observation that each processor core has an inherent 'hardware scheduler'.

The algorithms used in practice for scheduling time critical systems are *priority-driven preemptive* algorithms: at each instant of time, the processor is assigned to the highest priority job which is ready to run, preempting – if necessary – a lower priority job. A priority assignment is *feasible* if the deadlines of all jobs are met using such an assignment. For the feasibility check, a worst-case approach is followed. The longest response time for any job occurs when it is invoked simultaneously with all higher priority jobs; this situation is called the *critical instant*. Liu and Layland [11] have proved the following important result, assuming periodic threads (as in our case) and fixed priorities (i.e. the priority of a thread cannot change in time):

**Theorem 4.1** A fixed priority assignment is feasible provided the deadline of the first job of each thread starting from the critical instant is met.

When a scheduling algorithm produces a feasible priority assignment for a set of threads, this set is said to be *schedulable* by that algorithm. A scheduling algorithm is *optimal* when each possible set of threads for which a feasible priority assignment exists, is schedulable by that algorithm. When the deadlines are less than or equal to the periods (as in our case), Leung and Whitehead [10] have proved the following:

**Theorem 4.2** The deadline monotonic algorithm is optimal among all fixed priority algorithms.

With the deadline monotonic algorithm, the thread having the smallest deadline is assigned the highest priority. In other words, $\tau_i$ has a higher priority than $\tau_j$, whenever $D_i < D_j$. This priority assignment is referred to as the *Inverse-Deadline Priority Assignment (IDPA)*.

If we then look at today's processor cores, the above result suggests that the cores themselves can act as optimal schedulers! For example, assume a process composed of 16 slave threads $\tau_1 \ldots \tau_{16}$ and one autonomous thread $\tau_{17}$ (with $D_1 < \ldots < D_{16} < D_{17}$) that are constrained such that IDPA is a feasible priority assignment. Assume SPARC to be the selected processor. This processor has 16 interrupts $I_1 \ldots I_{16}$, where $I_i$ can preempt $I_j$, whenever $i < j$. If thread $\tau_{17}$ is placed into the `main` function, thread $\tau_i$ is assigned to interrupt (routine) $I_i$, for $1 \leq i \leq 16$, and all port delay constraints are met, the resulting system will meet all deadlines.

The general design flow is as follows. Starting point is timing analysis [13], which results in a 'viable' $\sigma_i$, for each thread $\tau_i$. Assume the selected processor has interrupts $I_1 \ldots I_m$, where $I_i$ can preempt $I_j$ whenever $i < j$. In order to check that the threads are schedulable on the selected processor, two conditions are examined.

The **first condition** verifies whether the slave threads have not more deadline levels than there are interrupt levels:

$$m \geq |\bigcup_{1 \leq i \leq n-1} \{D_i\}| \quad (1)$$

If Equation 1 is satisfied, we can pursue an interrupt assignment that satisfies the following ($I(\tau_i)$ denotes the interrupt assigned to thread $\tau_i$):

$$\forall i,j \in \{1,\ldots n-1\} : (I(\tau_i) = I_m \wedge I(\tau_j) = I_k) \Rightarrow (D_i = D_j \Rightarrow m = k) \wedge (D_i < D_j \Rightarrow m < k) \quad (2)$$

Equation 2 can be understood as follows. The deadline monotonic algorithm assigns priorities to threads inversely proportional to their deadline times (cf. IDPA). The absolute value of the assigned priorities is irrelevant; what is important is the *partial order* put on the different priorities. Equation 2 then defines an assignment of threads to the interrupt (routines) of a processor, that respects this partial order. In other words, let $I(\tau_i) = I_m$ and $I(\tau_j) = I_k$. Then $m = k$ if $D_i = D_j$, and $m < k$ ($I_m$ can thus preempt $I_k$) if $D_i < D_j$. An interrupt assignment that satisfies Equation 2 will be referred to as an *IDPA consistent interrupt assignment*. In this case, the selected processor provides a hardware implementation of the deadline monotonic scheduling algorithm! The question remains, however, whether IPDA is indeed a feasible priority assignment for the set of threads $\{\tau_1, \ldots, \tau_n\}$. This is checked by the second condition.

The **second condition** verifies whether the deadlines of the threads can be met using a deadline monotonic scheduling scheme. For this, the worst-case response time of each thread needs to be known. Based on the concept of critical instant (Theorem 4.1), Joseph and Pandya [9] have derived an exact analysis to find the worst-case response time $W_i$ for a thread $\tau_i$, assuming fixed priority, independent threads and deadlines less than periods (i.e. $D_i \leq T_i$, for all $i$):

$$W_i = C_i + \sum_{\tau_j \in hp(i)} C_j \lceil \frac{W_i}{T_j} \rceil \qquad (3)$$

$$\text{with } hp(i) = \{\tau_j \neq \tau_i \mid D_j \leq D_i\}$$

Equation 3 can be understood as follows. If the response time of $\tau_i$ is $W_i$, meanwhile there are at most $\lceil \frac{W_i}{T_j} \rceil$ requests from $\tau_j$ (with $\tau_j$ having a priority greater than or equal to $\tau_i$), which altogether require $C_j \lceil \frac{W_i}{T_j} \rceil$ time to execute.

A necessary and sufficient schedulability test can be readily derived from Equation 3:

**Theorem 4.3** *A fixed priority assignment for a set of threads $\{\tau_1, \ldots, \tau_n\}$, with $D_i \leq T_i \; \forall i$, is feasible iff*

$$W_i \leq D_i \qquad \forall i \qquad (4)$$

If Equation 1 and Equation 4 are satisfied, the selected processor provides a hardware implementation of the deadline monotonic scheduling algorithm, which, moreover, meets all deadline times. However, the response time analysis of Equation 3 is as good as the thread execution times used. These (worst-case) execution times assume delay constraints for every port, that need to be satisfied when generating the hardware/software interface. In a next step, we then attempt to generate a hardware/software interface that implements an IDPA consistent interrupt assignment (cf. Equation 2) and that meets the delay constraints of the different ports. This is explained in Section 5.

If Equation 1 or Equation 4 is not true, we reconstrain and/or select another processor (if possible), until both conditions are satisfied. If this does not succeed, by Theorem 4.2, the set of threads cannot be scheduled using a fixed priority preemptive scheme, or at least cannot be scheduled without the use of a run-time executive [1].

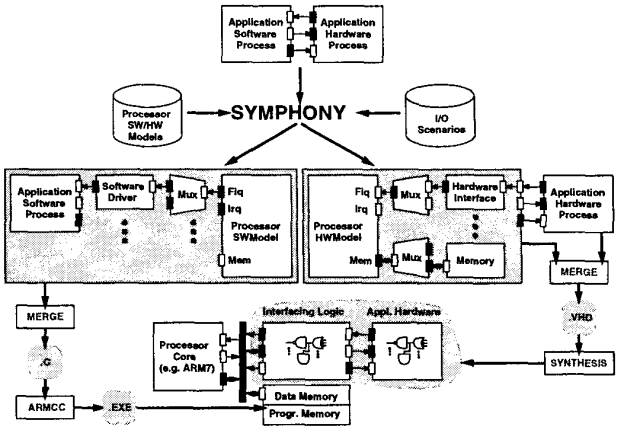


Figure 1: SYMPHONY as a solution to co-implementation

## 5 Hardware/Software Interface Generation

This section presents our approach to hardware/software interface generation. The approach is embodied in the SYMPHONY tool, that is part of the CoWare environment [1]. We first give an intuitive introduction and explain some key concepts. Afterwards, the actual procedure is detailed.

### 5.1 Once Over Lightly

Figure 1 depicts the basic build-up of SYMPHONY. Assume we have an application software process, specified in C, communicating via a number of ports with another application hardware process, specified in VHDL. To realize this communication at the hardware level, we replace the application software process by a new process, that is behaviorally equivalent, but that is specified in VHDL. This new process is obtained as follows. SYMPHONY reads in the application software process, a *software model* and *a hardware model* of the processor core, and a library of *I/O scenarios*. An I/O scenario (e.g. memory mapped I/O, interrupt driven I/O) consists of a software driver and a hardware counterpart for implementing a specific channel type on a particular processor. The SYMPHONY toolbox assigns I/O scenarios on a per-channel basis. Some of the selected I/O scenarios, however, make use of the same processor port. In this case, (de)multiplexing software and logic need to be inserted in the form of additional processes. Eventually, SYMPHONY produces two processes. The first process represents the software view, has no ports and consists of the processor software model, software drivers from the selected I/O scenarios and multiplexer processes. This hierar-

[1] The underlying assumption is that a dynamic priority preemptive scheduling algorithm cannot be implemented without the use of run-time executive.

chy can be flattened away using the *merge transformation*[2]. The result is a C description that is compiled by the vendor-supplied cross-compiler to an executable that will run on the involved processor core. The second process is the desired VHDL process and consists of the processor hardware model (eventually replaced by the 'real' processor core), memory, interfacing hardware from the selected I/O scenarios and multiplexer processes. For the path to implementation, the second process is merged with the application hardware process into a single VHDL description, that is compiled by existing hardware synthesis tools to a gate-level hardware implementation.

## 5.2 Key Concepts

A *software model* of a programmable processor core is a process, specified in C, that represents the processor's interface from a programmer's point of view. The software model identifies the software-controlled ports that can be used to get data in and/or out of the processor core (memory mapped, co-processor port, ...), the (priority of the ) interrupt ports, etc. An interrupt port, for example, is typically modeled as a master port; after merging, the code of the callee slave thread ends up in the corresponding interrupt routine.

A *hardware model* of a programmable processor core is a process, specified in VHDL[3], that represents the processor's interface from an external point of view.

An *I/O scenario* model is a process that has two ports, referred to as its *software port* and its *hardware port*. Basically, an I/O scenario describes how an RPC to its software port eventually activates an RPC from its hardware port (or vice versa), thereby crossing a processor's boundary. It consists of two processes: the first process is composed of a *software driver (process)* and a processor software model; the second process is composed of a *hardware interface (process)* and a processor hardware model. The reader can refer to [1] for more details.

An I/O scenario $I$ is said to be *compatible* with a port $p$ and a processor $P$, when (1) the hardware port of $I$ is of the same type as $p$, and (2) the software model and hardware model apply to $P$.

**Notation 5.1** Let $p$ be a port, and $P$ be a processor. Then, $io\_scenarios(p, P)$ denotes the set of I/O scenarios that are compatible with $p$ and $P$.

The *delay* of an I/O scenario is the *extra time* it takes for an RPC to cross the hardware/software boundary using this I/O scenario. For an interrupt based I/O scenario, this figure accommodates the reaction latency and the context switch delay. In the sequel, we are only interested in the *effective delay* of an I/O scenario, that is the delay of the I/O scenario incremented with the delay due to multiplexing.

[2] In the process of merging, each callee thread is in-lined in the code of the caller thread at the place of the RPC call.

[3] In practice, we also allow instruction-accurate C models to speed up the simulations, but this is beyond the scope of this paper.

**Notation 5.2** Let $I$ be an I/O scenario. Then, $eff\_delay(I)$ denotes the effective delay of $I$.

The *priority level* of an I/O scenario is the priority level of the processor's interrupt that is used by the I/O scenario. For I/O scenarios, that are not interrupt based, the priority level is assumed to be zero.

**Notation 5.3** Let $I$ be an I/O scenario. Then, $priority(I)$ denotes the priority level of $I$.

## 5.3 Procedure

In our earlier work [1], timing constraints were not considered, and I/O scenarios were selected manually. In this work, we aim at automating the I/O scenario selection process, such that Equation 2 is satisfied (i.e. an IDPA consistent interrupt assignment) and all port delay constraints are met. Then, our approach has an important advantage: to calculate the delay incurred on a port $p$, one *only* has to consider the effective delay of the I/O scenario assigned to $p$. The delay of being preempted or halted by another thread does not have to be reckoned with, as this is already done through Equation 3.

Before proceeding, we need to introduce several definitions.

**Definition 5.1** Let $\tau_p$ denote the slave thread associated with slave port $p$, and $I^{max}$ ($I^{min}$) denote the IDPA consistent interrupt assignment that uses as high (low) as possible interrupt priority levels. The *priority slack of* $p$ is defined by $priority\_slack(p) \equiv [I^{min}(\tau_p), I^{max}(\tau_p)]$

The *priority slack* of a slave port defines the range of interrupt priorities that are possible for implementing the corresponding slave thread using an IDPA consistent interrupt assignment.

**Definition 5.2** Let $p$ be a port, and $P$ be a processor. The delay slack of $p$ is defined by

$$delay\_slack(p, P) \equiv delay\_constraint(p) - MIN\{eff\_delay(s) | s \in io\_scenarios(p, P)\}$$

The *delay slack* of a port is defined to be the difference between its delay constraint and its fastest compatible I/O scenario that can be found in the library.

The procedure for selecting an optimal combination of I/O scenarios is detailed below.

**Algorithm 5.1 (Selection of I/O Scenarios)**
global $SUCCESS$ // $true$ initially
global $P$ // Selected processor

```
AssignIOScenarios(portListslackList, allocList) {
 while (portList ≠ ∅ and SUCCESS) {
  SUCCESS = false
  Update(slackList) // (Re)Compute delay slacks and priority slacks
  p ∈ portList with smallest delay slack
  portList = portList \ {p}
  list = io_scenarios(p, P) // List of Compatible I/O Scenarios
  while (list ≠ ∅ and not(SUCCESS)) {
   s = FindCandidate(list, slackList, allocList)
   if (SUCCESS) {
    list = list \ {s}
    Assign(allocList, (p, s))
```

```
    if (s accesses a 'free' processor port ) {
      res = AssignIOScenarios(portList, slackList, allocList)
      // This is a backtracking point
      if (SUCCESS) return
}}}}
return allocList
}
```

The procedure $AssignIOScenarios$ is called with three input arguments: $portList$ is the list of ports that (still) need an I/O scenario assignment; $slackList$ contains the priority slack and the delay slack of each port; $allocList$ keeps track of all I/O scenario assignments.

$AssignIOScenarios$ is a recursive procedure, which attempts to assign an I/O scenario to every port of $portlist$ in order of increasing delay slack. A port with a small delay slack can afford less delay due to multiplexing than a port with a large delay slack. Therefore, as a heuristic, the former port must be able to decide earlier on its I/O scenario, and thus on the processor ports it needs to access.

To find an I/O scenario for port $p$, the procedure $FindCandidate$ is called. This procedure selects the (compatible) I/O scenario with the smallest effective delay, whose priority level is within the priority slack of $p$, and which does not violate the delay constraints of $p$ and any of the ports that have been treated before (this may be possible due to increased multiplexing). Both the delay slack and the priority slack of a port depend on previous I/O scenario assignments, and need to be (re)evaluated after each iteration. If $FindCandidate$ does not find a suitable I/O scenario, the $AssignIOScenarios$ procedure returns ($SUCCESS = false$) to an earlier stored backtracking point, if it exists.

As a heuristic, this algorithm marks a backtracking point (by making a recursive call) when it assigns an I/O scenario that uses an as yet unconnected processor port. The intuitive motivation is that it is considered a privilege for a port to be the first to impose upon a particular processor port. If the recursive call is successful, this means that a feasible solution has been found, and all recursive calls return.

Despite its heuristic nature, it can be proven [14] that – under realistic assumptions – the algorithm finds a solution, if it exists.

# 6 Experimental Results

To assess the viability of our approach, we have experimented with a direct-sequence spread-spectrum satellite receiver [12], used in the design of a *Satellite ISDN-rate Mobile Base-station ASIC (SIMBA)*. We first give a short overview of SIMBA. We then describe the experiments and the obtained results.

## 6.1 System Description

The SIMBA design mainly consists of two parts, as shown in Figure 2.

The first part is the forward path, which has to operate at a very high speed. Therefore, it is implemented in hardware. It consists of an Automatic Gain Control (AGC) block, a down-convertor (to convert the bandpass spread-spectrum signal to the baseband), a chip matched filter, an interpolator and a correlator.

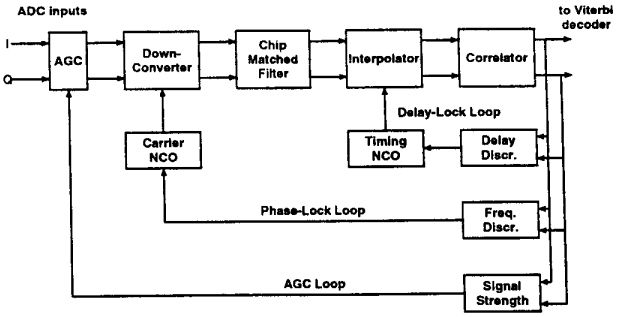


Figure 2: The SIMBA satellite receiver

The second part consists of a number of feedback loops: an AGC-loop to achieve a constant signal strength at the output; a Phase Locked Loop (PLL) to insure a correct frequency shift in the demodulator; a Delay Locked Loop (DLL) to keep the sender and the receiver synchronized. These loops are not computationally intensive. In addition, their design is particularly error-prone. Therefore, these loops are implemented in software.

The SIMBA also contains an important (but not time-critical) control block: it includes a user interface (to change parameters such as the local code sequence and the frequency shift of the down-convertor) and performs a number of checks on the correct operation of the receiver. This control block is also implemented in software.

For the implementation, the ARM processor has been chosen, which is a small ($4.8mm^2$ for a $0.6\mu m$ process) processor core with a very low power consumption (about 2 mW/MHz or 80 mW for the maximum clock speed of 40 MHz). This processor has two interrupts: the IRQ (Interrupt ReQuest) and the FIQ (Fast Interrupt reQuest), where the latter can preempt the former.

## 6.2 Implementation with Symphony

The SIMBA has been described in CoWare. The complete software functionality is specified as a single process consisting of three slave threads $\tau_1$ (PLL), $\tau_2$ (DLL) and $\tau_3$ (AGC-loop), and one autonomous thread $\tau_4$ (control block). All these threads are periodic.

The control block is most naturally specified as an autonomous thread that periodically checks whether there are inputs from the user. A rate of 10 Hz (i.e. a period of 100 ms) has been found to be sufficient.

The PLL and AGC-loop run at the *chip* [4] rate; the DLL runs at the (lower) bit rate. The ratio between the two rates is equal to the length of the code sequence of the spread spectrum system (typically 32).

[4] In spread spectrum systems, each bit is multiplied with a code sequence. To distinguish with the original *bits*, the zeroes and ones of the product sequence are called *chips*.

For the SIMBA, we have considered a bit rate of 2 kbit/s, and a code sequence of length 32. Thus, the chip rate is 64 kchips/s (i.e. the speed of the telephone network). This results in the following thread periods: $500\mu s$ ($T_2$) and $15.6\mu s$ ($T_1 = T_3$). The deadlines are taken equal to the periods. Further, the threads run independently; the communication only occurs through shared variables. Table 1 gives an overview.

| thread $\tau_i$ | function | period $T_i$ ($\mu$s) | execution time $C_i^m$($\mu$s[5])(cycles) | $C_i$ ($\mu$s) |
|---|---|---|---|---|
| $\tau_1$ | PLL | 15.6 | 5.85 (234) | 7 |
| $\tau_2$ | DLL | 500 | 6.20 (248) | 7 |
| $\tau_3$ | AGC | 15.6 | 2.67 (107) | 3 |
| $\tau_4$ | control | $10^5$ | – | $10^4$ |

Table 1: Time figures for the different threads

The assumptions of Section 3 are thus valid. Hence, we can apply the techniques presented in this paper.

Equation 1 is readily verified; the slave threads do not have more deadline levels (viz. 2) than there are interrupt priority levels (viz. 2).

Before proceeding with Equation 4, let us return to Table 1. Here, good estimates are given for the thread execution times: the figures $C_i^m$ are measured on an instruction set simulator of the ARM; the (worst-case) figures $C_i$ include some extra delay for the hardware/software interface. For the control thread, a rough estimation is used, as its execution time depends on the interaction with the user.

The worst-case response times of the different threads are computed by means of Equation 3. The worst-case response time of thread $\tau_i$ is the smallest $W_i$ that satisfies this equation, and is computed by iteration. The results are given below:

$$W_1 = 7\mu s + \lceil \tfrac{10}{15.6} \rceil 3\mu s = 10\mu s$$
$$W_2 = 7\mu s + \lceil \tfrac{27}{15.6} \rceil 7\mu s + \lceil \tfrac{27}{15.6} \rceil 3\mu s = 27\mu s$$
$$W_3 = 3\mu s + \lceil \tfrac{10}{15.6} \rceil 7\mu s = 10\mu s$$
$$W_4 = 10ms + \lceil \tfrac{29ms}{15.6\mu s} \rceil 7\mu s + \lceil \tfrac{29ms}{500\mu s} \rceil 7\mu s + \lceil \tfrac{29ms}{15.6\mu s} \rceil 3\mu s = 29ms$$

As can be seen, the worst-case response times $W_i$ do not exceed the respective deadlines. By Theorem 4.3, the deadlines of all thread instances can then be met, provided that the hardware/software interface does not require more than the allocated delay (cf. above).

To generate the hardware/software interface, the SYMPHONY tool has been called upon. The thread $\tau_2$ is assigned to the IRQ interrupt. The threads $\tau_1$ and $\tau_3$ are both assigned to the FIQ interrupt; this requires (de)multiplexing hardware (software) in the form of additional processes. These multiplexer processes add about 10 cycles to the the execution times of both threads. As a result, the 'effective' execution times of thread $\tau_1$ and thread $\tau_4$ amount to $6.1\mu s$ and $2.93\mu s$, respectively. These 'effective' execution times are still less than their worst-case estimates, used in above equations. As a result, the final implementation is guaranteed to meet all timing constraints.

Finally, the complete hardware/software system has been simulated extensively, to gain confidence in the obtained result.

[5]The maximal clock speed of 40 MHz is assumed.

## 7 Conclusions

In this paper, we have presented an orchestrated combination of software synthesis and automatic hardware/software interface generation. We have derived necessary and sufficient conditions for mapping an application software process – according to our assumptions – on to a programmable processor core, without using a run-time executive. We build on the key observation that any processor (core) has an inherent fixed-priority preemptive scheduler (cf. interrupts), and partly rely on a parameterized library solution to solve the well-known interfacing problem. The overall approach been tested on a spread-spectrum ASIC design, where it has provided significant gain.

## Acknowledgments

The authors would like to thank Filip Thoen for the numerous insightful discussions on the software synthesis problem, and Paul Coene for providing the VHDL descriptions of the SIMBA. This research is funded by the MEDEA AT-403 SMT project.

## References

[1] I. Bolsens et al. Hardware/software Co-design of Digital Telecommunication Systems. *Proceedings of the IEEE*, 85(3):391–418, March 1997.

[2] CoWare Inc. `http://www.coware.com/`

[3] F.George. Cadence Design Systems Inc. Block-based Design: Creating a System on a Chip. *Electronic Design*, pages 86–92, July 8 1996.

[4] M. Chiodo et al. A Formal Methodology for Hardware/Software Co-Design of Embedded Systems.0000 *IEEE Micro*, 14(4):26–36, August 1994.

[5] P. Chou et al. The Chinook Hardware/Software Co-Design System. In *Proceedings of ISSS*, pages 22–27, Cannes, September 1995.

[6] J.M. Daveau. *Spécifications Systèmes et Synthèse de la Communication pour le Co-design Logiciel/Matériel*. PhD thesis, TIMA-INPG, Grenoble, December 1997.

[7] K. Ghosh et al. A Survey of Real-Time Operating Systems. Technical report, Georgia Institute of Technology, February 1994. nr. GIT-CC-93/18.

[8] R. Gupta et al. A Co-synthesis Approach to Embedded System Design Automation. *Design Automation for Embedded Systems*, 1(1-2):69–120, January 1996.

[9] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29:390–395, October 1986.

[10] J.Y.T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks. *Performance Evaluation*, 2:237–250, December 1982.

[11] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20:46–61, January 1973.

[12] B. Sklar. *Digital Communications: Fundamentals and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[13] F. Thoen, G. Goossens, J. Van Der Steen, and H. De Man. Multi-thread Graph A System Model for Real-Time Embedded Software Synthesis. In *SASIMI*, 1996.

[14] S. Vercauteren. *Hardware/Software Co-Design of Application-Specific Heterogeneous Architectures*. PhD thesis, Katholieke Universiteit Leuven, December 1998.