Fixed Priority Timing Analysis of Real-Time Systems with Limited Parallelism

N.C. Audsley and K. Bletsas
Dept. of Computer Science, University of York, UK. [neil,bletsas]@cs.york.ac.uk

Abstract

Conventional real-time system implementations assume a von Neumann processor-memory architecture with at most one software process executing at any time. Scheduling approaches for such systems multiplex the processor among runnable processes. Fixed priority scheduling, notably, utilises timing analysis to determine offline the run-time timing behaviour of the system. Recent alternative real-time system implementations combine processor-memory architectures with reconfigurable hardware for speedup. Such reconfigurable hardware can execute many functions in parallel. This leads to a limited parallel system where a single software process can execute on the processor at any time, in parallel with a number of functions implemented in hardware. Conventional fixed priority timing analysis is unsuitable for this model, as fundamental assumptions are compromised, eg. the notion of a critical instant. This paper proposes new timing analysis aimed at such limited parallel systems, illustrated by an example system utilising Field Programmable Gate Arrays as the reconfigurable hardware resource.

1 Introduction

Architectures with processor and memory arranged on a single bus are typically assumed for real-time systems; at most one software process can then execute at any time; timing analysis models the multiplexing of the single processor among competing software processes. The analysis built up for such architectures is comprehensive, typified by the body of work based on fixed priority scheduling [11, 9, 10, 3, 4].

Recently developed real-time system implementations employ both reconfigurable hardware and a conventional processor, as target for the application functionality [7, 16, 17, 18]. Thus, part of a software process is implemented in hardware so as to speed-up overall execution. The reconfigurable hardware can be seen as a number of application specific co-processors, where each one can speed-up the execution of the application (relative to a software implementation). Also, the co-processors can execute in parallel. In the resulting limited parallel architecture, at most one software process can execute in parallel with a number of co-processors.

Traditional co-processor use is exemplified by how floating point operations are achieved on a typical processor [12]: a running software process wishing to execute such an operation issues the latter to the floating point unit and the processor pauses until the floating point operation is complete. This typifies a master/slave approach to computation in co-processors, with the processor pausing whilst the co-processor executes, even if another runnable process could take advantage of the processor. An approach only viable if co-processor operations are short, wasting few cycles. The limited parallel model introduced here differs in that when a process is waiting for a hardware operation to complete, another runnable process can be scheduled.

Currently available reconfigurable devices support this model. Field Programmable Gate Arrays (FPGAs) [15, 19, 1] can contain several million programmable gates: the Xilinx Virtex-II [19] FPGA can combine subsystems such as audio/image/signal processing, communications, networking etc. Such complex functions require many clock cycles to execute, thus the "master /slave" model becomes inefficient.

Reconfigurable logic can speed up overall system performance notably [6, 7] and improve timing predictability [16, 17, 18] as no software scheduling is involved and the timing behaviour of logic is well-understood [19, 15].

Timing analysis for conventional architectures is inappropriate for limited parallel systems. For

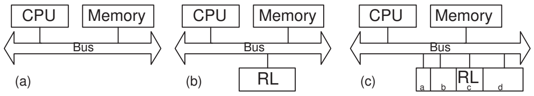

Figure 1: Architecture for Real-Time Systems

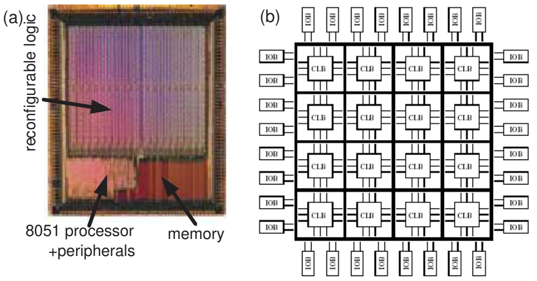

Figure 2: (a) Triscend E5 (b) FPGA structure

example, the critical instant (worst-case scenario) forms the basis for all conventional fixed priority timing analysis - this worst-case no more applies under limited parallelism. Note that timing analyses developed for multiprocessor or distributed [13, 8] systems are not applicable - both allow multiple software elements to execute in parallel on different processors and do not allow multiple application functions to execute in parallel on a single processor.

This paper focuses on such limited parallel systems. It identifies a new worst-case scenario and provides suitable response-time analysis. It is structured as follows. In the next section, further background is given on the computational capabilities of systems involving reconfigurable hardware, providing motivation for the rest of the paper. In sections 3 and 4, timing analysis for limited parallel systems based on reconfigurable logic is given. In section 5, that analysis is assessed. Section 6 provides a case study: an implementation of a limited parallel system using FPGAs as the reconfigurable logic element. Section 7 provides concluding remarks.

2 Background/Motivation

We provide motivation for the limited parallel model, by reviewing architectures and implementations of actual embedded real-time systems. These are typically von Neumann processor-memory hierarchies (Figure 1(a)). Devices are memory mapped, on the same processor memory bus (or perhaps separated from it by a bridge to a slower I/O bus [12]). Reconfigurable logic may be introduced as another device on the bus (Figure1(b)), or perhaps as a range of separate devices on the bus. This is shown in Figure 1(c): The reconfigurable logic is configured into m separate partitions, each of which can function in parallel with the rest. Figure 1(c) illustrates the concept of limited parallelism where a single software process can execute on the processor in parallel with upto m separate hardware functions each implemented on a partition in the reconfigurable logic device.

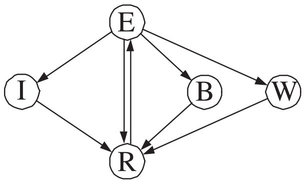

Figure 3: Extended Process Model

This model is supported by a wide range of technologies. Firstly, hybrid processors which contain processor and reconfigurable logic on the same device. Figure 2(a) shows a typical device, the Triscend E5 [14]. Xilinx offers similar products [19]. Devices consisting solely of reconfigurable logic are common in the embedded systems world. The most widely used form of large-scale single device reconfigurable logic is the Field-Programmable Gate Array (FPGA). Even on most hybrid devices, the reconfigurable part is a (limited) FPGA. FPGAs are discussed further, as part of our case study in section 6.

3 Timing Analysis

3.1 Process Model

A process set $\Delta = \{\tau_1,\ \tau_2,\ ...,\ \tau_n\}$ is scheduled by a fixed priority scheme: the highest-priority runnable process executes on the processor at any time. Priorities are unique, ranging from 1(lowest)..n(highest). Each process τ_i has a period T_i and deadline D_i, where $D_i \leq T_i$. For a sporadic process, T_i denotes the minimum inter-arrival time between successive executions of it. The worst-case response time of τ_i is R_i. Whilst executing, processes cannot voluntarily suspend; also (a restriction later removed) they share no resources, hence do not block.

The limited parallel model allows part of a process to be executed in a coprocessor implemented in reconfigurable hardware. As this is not reflected in the traditional state model of a process, we introduce a new state termed *waiting*. The extended model is shown in Figure 3 with states (labeled by initial: Idle, Ready, Executing, Blocked, Waiting) and allowed transitions. The processor is always assigned to the highest-priority process among those ready, with that process switching to executing state. When control is passed to a hardware co-processor, the respective process transitions from executing to waiting. It switches to ready state when the co-processor function terminates. Note that the scheduler may need to reconsider the process to execute whenever a co-processor function terminates, as the process switching to ready state may be of higher priority than the one executing.

The part of a process implemented by a coprocessor is termed a *gap*, as it forms an interruption in the conventional processor based execution of a process. A process can invoke a number of hardware co-processor functions during an invocation, hence can contain a number of gaps. The total time of computation for a specific invocation of a process τ_i equals the sum of the execution times in software and on the co-processor. Assuming no correlation between the two, $C_i = X_i + G_i$, where C_i, X_i, G_i are the upper bounds for overall, software and coprocessor execution times. If a process τ_i has no gaps, it requires all its execution time on the processor and the amount of outstanding computation is reduced only when it is scheduled. But if $G_i > 0$, the process may require upto X_i scheduled on the processor, but may at some point initiate a hardware co-processor operation. Note that a hardware operation initiated by τ_i at time t will complete at $t+g$ (where g is the length of the gap) and is not dependent on any other process. At any time upto n processes could be in the waiting state having invoked hardware co-processor operations.

3.2 Observations

The process model introduced above for limited parallel systems clearly varies from the master/slave model on which traditional timing analysis is based. However, the latter is in fact a special (limited) case of the limited parallel model where no process initiates hardware co-processor operations. Formally, for master/slave systems: $\forall \tau_i \in \Delta \bullet G_i = 0$; bounds on worst-case response times can then be derived by [3]:

$$R_i = C_i + \sum_{j \in hp(i)} \lceil R_i / T_j \rceil * C_j \qquad (1)$$

hp(i) being the set of processes having higher priority than τ_i. This holds under the assumptions of the process model above (ie. no blocking; no voluntary process suspension).

Now assume one or more processes with $G>0$. One approach is to pause execution of the process initiating the hardware operation until that operation is complete - this reflects the approach taken with floating point operations (ie. short hardware operations). The above analysis is still applicable (even if some processes have $G > 0$), noting that the processsor will be idle during hardware operation execution. As G increases as a proportion of C, this becomes more inefficient and unsatisfactory. Consider the process set of Table 1. A simulation of this system under the traditional approach is shown in Figure 4(a). Shaded areas denote execution on a co-processor. During these intervals, there is no execution on the processor. The result is that τ_D misses a deadline - even under optimal priority assignment (ie. deadline monotonic [9]). Under the limited parallel approach, when a process invokes a hardware operation, another process can be executed by the processor (ie. the first process is now in the waiting state). Clearly, response times are reduced: Figure 4(b) demonstrates how limited parallelism makes the previously unschedulable process set feasible.

process	T	D	X	G	C	priority
τ_A	12	4	2	1	3	4
τ_B	12	8	2	2	4	3
τ_C	12	9	2	0	2	2
τ_D	18	10	3	0	3	1

Table 1: An Example Process Set

3.3 Identifing The Worst Case

A process may interfere with lower priority proceses only while executing on the processor, ie. in the executing state. When it is in waiting state, computation is performed on the coprocessor, with the processor available to other

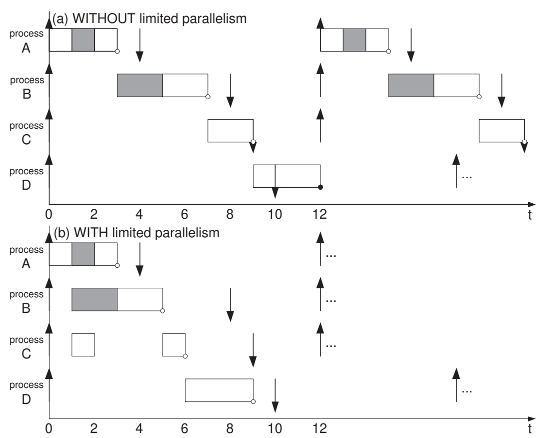

Figure 4: Parallelism Reduces Response Times

processes that are in ready state. Thus, each instance of τ_j can interfere with lower-priority processes for at most X_j time units. For an instance of τ_i, cumulative interference equals the individual interferences exerted by each $\tau_j \in hp(i)$ summed:

$$I_i = \sum_{j \in hp(i)} I_{j \to i}$$

Under the conventional master/slave model, worst-case interference on τ_i occurs for simultaneous process releases. This is termed the critical instant [11]. For the limited parallel model, this is no more the worst case. Consider the trivial example in Figure 5. If higher-priority processes A, B start with gaps and all releases coincide, the lowest-priority process can execute on the processor during that time interval and terminate, with no interference (Figure 5(a)). But if it is released one time unit later, its response time time increases (Figure 5(b)).

We now identify the worst-case interference a process can suffer due to higher-priority processes executing on the processor. Intuitively this worst case is based upon Figure 5(b). Initially, the execution pattern of processes is considered. Given that a process can contain zero or more hardware co-processor invocations, we must consider where these gaps occur within the process execution. Figure 6 shows invocations of the same process, with the gaps in different places due to control-flow variations.

The *execution distribution* of an activation of a process τ_i consists of C_i execution steps. It is

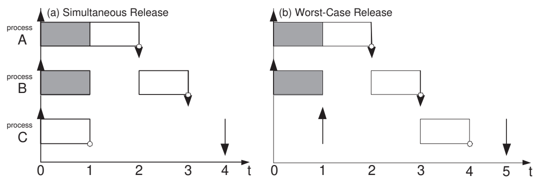

Figure 5: There Is No More A Critical Instant

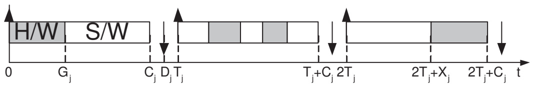

Figure 6: Per Invocation Variability For Gaps

formally defined as:

$$f_i(t) = \begin{cases} 1 & \text{if } \tau_i \text{ is in executing state after} \\ & \text{having computed for } t \text{ time units} \\ 0 & \text{otherwise} \end{cases}$$

Hence, $f_i(t)$ defines which units of C_i are software execution and which belong to gaps. This enables the description of execution patterns such as those given in Figure 6. Clearly, for an invocation of τ_i there is exactly X_i units of execution on the processor so $\sum_{t=0}^{C_i-1} f_i(t) = X_i$. When all the software is executed at the start or end of an invocation of τ_i at t_0 (and hence co-processor invocations occur at the end/start), then respectively:
$f_{i_{L(t_0)}} = true \iff \forall t \in [0 \leq t \leq X-1] \bullet f_i(t+t_0)=1$,
$f_{i_{R(t_0)}} = true \iff \forall t \in [G \leq t \leq C-1] \bullet f_i(t+t_0)=1$

To describe software execution patterns within an invocation of τ_i at t_0, the start times of software execution blocks need to be defined. A software block starts at t if and only if:
$start_i(t) = true \iff$
$(f_i(t) = 1) \wedge (t = t_0 \vee (t > t_0 \wedge f_i(t-1) = 0))$
Software block lengths can be defined as:

$$g_i(t) = \begin{cases} 0 & if\ \not{s}tart_i(t) \\ \max n \in \mathbb{N}^+ \bullet f_i(t+m)=1, & \\ \forall m \in \{0, 1, ..., n-1\} & else \end{cases}$$

Theorem 1: *The time spent in executing state (thus interfering with any lower-priority processes) by a process τ_j over a time interval starting at t_0 is maximized when:*

- *τ_j is first released at $t_0 - G_j$ with the first transition to executing state (for X_j time units) occuring as late as possible at t_0, and*

- *in subsequent instances, software execution for X_j precedes hardware execution for G_j.*

Proof: Let τ_1,τ_2,...,τ_m be a priority-ordered (lowest to highest) process set. A process $\tau_j \in$hp(i) exerts maximum interference to an instance of τ_i when the time its activations spend in executing state from release to completion of τ_i is maximized. The actual interference τ_i suffers due to τ_j depends on:

- The relative offset of the releases of τ_i, τ_j.
- The execution distributions for τ_j.

Without loss of generality, gaps can float and merge (like beads on an abacus) anywhere within the activation of τ_j - for the worst-case, we select the placement maximizing $I_{j \to i}$.

We define h(t) as the sum of all software execution blocks of τ_j started at or by time $t>0$:

$$h_j(t) = \sum_{u=-\infty}^{t} g_j(u)$$

Then maximum interference by an instance of τ_j on τ_i over $[t_\alpha, t_\beta]$ (with $0 \leq t_\alpha \leq t_\alpha$) is:

$$h_j(\tau_\beta) - \sum_{-\infty}^{\tau_\alpha - 1} f_j(u)$$

Hence interference by τ_j upon τ_i (released at $t \geq 0$) is given by:

$$I_{j \to i} = h_j(t + R_i) - \sum_{-\infty}^{t-1} f_j(u)$$

It remains to determine the times of releases of τ_j that maximize its interference on τ_i. We define t_j^0 as the latest release of τ_j at or before t_0, ie. $t_j^0 \leq t_j$. Then, the sequence of releases of τ_j starting at t_j^0 is t_j^0,t_j^1,t_j^2,... where: $t_j^k = t_j^0 + k * T_j$.

The interference from the 0-th release of τ_j is maximized if $f_{j_R}(t_j^0) = true$ because then software execution in the time-window $[t_0..t_j^0 + C_j - 1]$ is maximized. Each activation of τ_j can exert (using any offset for the sequence) interference of at most X_j. We do not know *a priori* how many invocations of τ_j actually interfere with τ_i in the general case but by choosing $f_{j_L}(t_j^a) \forall a \geq 1$ we ensure that if one of those is actually interfering, contributed interference is maximized and equal to its software execution (X_j in the worst case).

The invocations of τ_j actually interfering with τ_i are at most $\lceil (R_i + t_0 - t_j^0)/T_j \rceil$. Thus, maximizing the numerator (choosing the appropriate offset $t_0 - t_j^0$) maximizes interference. But $t_0 - t_j^0$ and interference from the 0-th release can not increase independently. For $t_0 - t_j^0 = -G_j$ interference from the release at t_j^0 is maximal at X_j (as $f_{j_R}(t_j^0) = true$). Shifting t_j^0 any more to the left by $v \leq X_j - 1$ reduces interference by an equal amount of samples, so the numerator does not change. This proves the theorem for a two-process set. Since interferences from distinct higher priority processes are independent (as no resources are shared and as inspection of the equations shows), it also holds for n processes.
□

We term this worst-case scenario *critical higher-priority execution distribution.* Interestingly, conditions for worst-case interference on distinct processes are mutually exclusive (unless $G_i = 0 \forall i$; a degenerate case).

3.4 Response Time Analysis

For the worst-case given by the previous section, the number of releases of τ_j that can interfere with τ_i is $\lceil (R_i + G_j)/T_j \rceil$. This expression is analogous to the inclusion of jitter in conventional fixed priority timing analysis [3]. Hence:

$$R_i = C_i + \sum_{j \in hp(i)} \lceil (R_i + G_j)/T_j \rceil * Xj \qquad (2)$$

The response time equation is solved by forming a recurrence relationship, similarly as in [3]:

$$w_i^{n+1} = C_i + \sum_{j \in hp(i)} \lceil (w_i^n + G_j)/T_j \rceil * X_i$$

The recurrence is initiated by $w^0 = C_i$ and terminates when $w^{n+1} = w^n$.

4 Priority Assignment

Under the conventional master/slave model the Deadline Monotonic Priority Ordering (DMPO) is optimal [9]; under the limited parallel model not. For example, the process set described in Table 2 is not feasible under DMPO but under reversed priorities, it is (see Figure 7). For the limited parallel model, priorities can be optimally assigned by the approach described in [2]. Starting with the lowest priority, processes are tested for feasibility at that priority level. If none is feasible then neither is the whole set. But if one is found feasible then it is assigned that priority and the procedure continues with the remaining processes and priority levels. Provided the worst-case response time of a process can not increase if priorities are exchanged with a higher-priority process, this algorithm is valid and optimal; this property holds for the response time analysis of section 3.4.

5 Assessing The New Model

A synopsis of the models discussed follows:

τ	T	D	X	G	C
process A	16	9	1	7	8
process B	14	6	4	0	4

Table 2: An Example Process Set

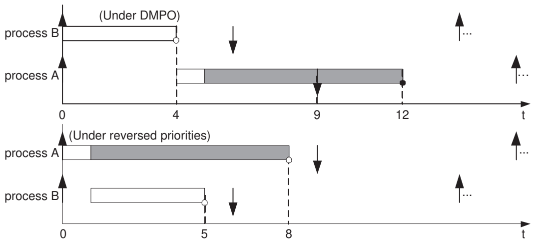

Figure 7: DMPO Is *Not* Optimal

1. Conventional Master/Slave (section 3.2):
   • processor idle while hardware executes
   • worst-case interference on τ_i is for coincident higher priority process releases (ie. critical instant [11])
   • response time derived by equation (1).

2. Limited Parallelism (sections 3.3, 3.4):
   • worst-case interference on τ_i occurs at a critical higher-priority execution distribution (section 3.3)
   • response time derived by equation (2).

If $\forall\tau_i\in\Delta\bullet G_i{=}0$, these models are equivalent; if some $G_i\neq 0$, this section proves that response times under the limited parallel model equal/better than those under the conventional model (under the respective worst cases):

Initially, consider the limited parallel model and the critical higher priority distribution for a two process system. The intervals available for τ_i to execute are shown in Figure 8(a) (NB. the index j is omitted for reasons of clarity). The first such interval has length $T_j{-}G_j{-}X_j$ while any subsequent ones have length $T_j{-}X_j$ (as a result of the critical execution distribution). If C_i lies between 1 and $T_j{-}G_j{-}X_j$, τ_i will suffer interference of X_j. If it equals $T_j{-}G_j{-}X_j{+}1$ it will suffer additional interference from the instance of τ_j released at $t{=}T{-}G$ and so on. This is generalised formally as: $I^{LP}_{j\to i}(C_i){=}(n{+}1){*}X_j \iff n{*}(T_j{-}X_j){-}G_j{<}C_i{\leq}(n{+}1){*}(T_j{-}X_j){-}G_j$ for $n{\in}\{1,2,...\}$. A similar expression is derived for the worst-case under the conventional model: $I^{MS}_{j\to i}(C_i){=}(m{+}1){*}(X_j{+}G_j) \iff$

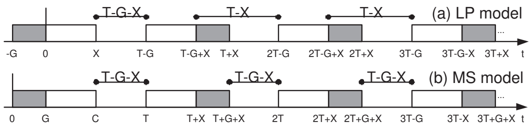

Figure 8: Time Windows For Execution

$m{*}(T_j{-}X_j{-}G_j){<}C_i{\leq}(m{+}1){*}(T_j{-}X_j{-}G_j)$ for $m{\in}\{0,1,...\}$. Figure 8 juxtaposes visualisations of the expressions for the two models.

Theorem 2: *For a n-process set* $I^{LP}_{j\to i}{\leq}I^{MS}_{j\to i}$.

Proof: For a two-process system, where τ_j interferes with τ_j, $\exists n,m{\in}\mathbb{N}$ such that:
$n(T_j - X_j) - G_j < C_i \leq (n+1)(T_j - X_j) - G_j$,
$m(T_j - X_j - G_j) < C_i \leq (m+1)(T_j - X_j - G_j)$.
Then $n < ((m{+}1)(T_j{-}X_j{-}G_j){+}G_j)/(T_j{-}X_j)$.
We have $I^{MS}_{j\to i}{=}(m{+}1)(X_j{+}G_j), I^{LP}_{j\to i}{=}(n{+}1)X_j$.
If $(G_j{=}0){\vee}(X_j{=}0)$ the proof is trivial. Else, it suffices to show that $I^{MS}_{j\to i}{-}I^{LP}_{j\to i}{>}0$:
• If $m{=}0$ then n must also be 0. Then $I^{MS}_{j\to i}{=}X_j + G_j{<}X_j{=}I^{LP}_{j\to i}$
• If $m{\geq}1$: $I^{LP}_{j\to i}(C_i){<}(\frac{(m+1)(T_j-X_j+G_j)+G_j}{T_j-X_j}{+}1)X_j$
$\Rightarrow I^{LP}_{j\to i}(C_i){<}\frac{(m+1)(T_j-X_j-G_j)X_j+G_jX_j+(T_j-Xj)X_j}{T_j-X_j}$
Then $I^{MS}_{j\to i}{-}I^{LP}_{j\to i}{>}\frac{A}{T_j-X_j}$, where
$A = [mX_j + mG_j - (m+1)](T_j - X_j) + X_jT_j + G_j(T_j - X_j) + mG_j$.
It suffices to show that $A{>}0$ and for that it suffices to show that $B = mX_j{+}mG_j{-}(m{+}1) \geq 0$.
And $B = (X_j + G_j - 1)m - 1 \geq m - 1 \geq 0$.

In a n-process set, respective interferences from higher-priority processes are independent. So the property just proven for a two-process system holds for them, thus also for their sum. This proves the theorem.
□

6 Realisation On FPGAs

Unlike the program once, logic-term PAL / GAL devices, Field Programmable Gate Arrays (FPGAs) provide reprogrammability and logic functions to the system designer [19, 1]. This allows them to provide more complex functionality than an equivalently sized programmable logic device. A typical sea-of-gates FPGA consists of an array of functional blocks, connected by programmable interconnects (see Figure 2(b)). Functional blocks contain the logic elements of the design. Views on granularity of block func-

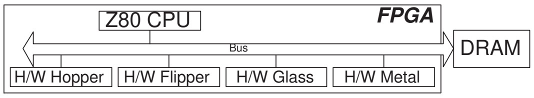

Figure 9: FPGA-Based Realisation

tionality differ among FPGA families but overall device functionality is the same. Programmable interconnects route signals between logic blocks, and come in varying lengths to suit the range of connection distances required. By connecting logic blocks, larger logic expressions are realised. Typical actual devices may involve over 100,000 functional blocks and also provide considerable memory blocks every few columns of functional blocks (upto 4 MBytes on large devices).

I/O blocks, around the edge of the FPGA, provide the means to interface to/from external signals. Typically FPGAs are I/O pin rich (upto 500 I/O pins per device, depending upon the exact packaging). Most FPGAs support a wide range of bus and interface standards so can be placed directly on a bus without any glue logic. Data storage forms a large part of most systems, and so to keep data access speed high, FPGAs generally contain some form of memory. Most large devices include fairly large (2-8 kbit) RAM blocks around the array, configurable in the word width of data. In some devices smaller memory blocks can be constructed out of logic blocks, for local storage of small data items.

FPGAs can be used for the physical realisation of the limited parallel architectures depicted in Figures 1(b), 1(c). An FPGA connected to the processor-memory bus can provide one or more functions. These typically have high cost when implemented in software and can be sped up if implemented in hardware, eg. digital signal/image processing or parallel algorithms. Multiple parallel functions in an FPGA (ie. as Figure 1(c)) require either internally multiplexing a single bus connection or using the many I/O pins for many separate connections.

FPGAs are reprogrammable; configuration takes some milliseconds. The program for the device can be specified in languages such as Ada [16, 17, 18], Handel-C [5], VHDL. In any case, the program is synthesised to a *net list*: an abstract circuit description. This is fitted to the device it is to be implemented on. Finally the fitted design is converted to the configuration bit stream used for programming the FPGA.

6.1 An Example System

In the example system (Figure 9), a Xilinx SpartanIIE connected to RAM is programmed with a softcore Z80 processor plus hardware functions for speed-up. The application is a marble chute sorter (see Figure 10). Marbles (from a random collection) released down the chute from a hopper are detected as either metal/glass half way down the chute, a flipper at the bottom controlling which container they end up in. The functionality consists of the following processes:

- Hopper: releases marbles down the chute;
- Metal: driven on interrupt from metal sensor;
- Glass: likewise, from light sensor;
- Flipper: controls where marbles end up.

The flipper and hopper releases are offset by the time (0.2s) it takes for marbles to fall from the hopper to past the sensors. The hardware functions implemented are effectively device drivers for the hopper, sensors and flipper. For the sensors, the driver de-bounces sensor inputs and timestamps readings. Hence, the sensor processes begin with the hardware function and end with software execution (ie. $f_R = true$). The hopper and flipper processes have the hardware function somewhere in the middle of their lifecycle. The timing requirements are given in Table 3. Process offsets are omitted, as they are ignored in response time calculation for the limited parallel model. Placing functions in hardware enables tighter response times; note that if execution was all in software, deadlines would be missed due to additional interference.

τ	T	D	X	G	C	P	R
metal	400	40	10	10	20	4	20
glass	400	40	10	10	20	3	30
hopper	400	50	5	5	10	2	30
flipper	400	50	15	5	20	1	45

Table 3: Marble Chute Process Set

7 Conclusions

Under the introduced *limited parallel* model for real-time systems, software processes can invoke complex hardware functions at any time, the processor being assigned to another runnable process immediately. Again, at most one process can execute on the processor at any time but they can all execute hardware functions in

Figure 10: Marble Chute

parallel. The increased parallelism required new appropriate timing analysis; we identify a worst-case scenario for interference and provide response time analysis for it. In this worst-case scenario (the *critical higher-priority execution distribution*) process releases do not coincide, unlike the classical critical instant. Conventional analysis, if used, has to treat hardware execution as normal software execution (ie. by idling the processor during hardware execution). An approach proven inferior to the new model.

The limited parallel model reflects engineering reality: Hardware co-processors are often utilised to speed-up overall system performance, typically in devices such as FPGAs and hybrid processors. The case study given (implementation on an FPGA) illustrates the efficacy of the timing analysis developed.

As for open research issues: Firstly, the analysis must incorporate software blocking. While this is straightforward (by adding a blocking term in the response time equation, as in conventional analysis), the priority assignment method then becomes non-optimal. Blocking must also be extended to hardware shared resources. Secondly, moving functions to hardware exchanges processor time with FPGA space. An open issue is the consideration of FPGA space constraints together with timing analysis at the design stage, for implementation decisions.

References

[1] Altera Corporation. *Altera Product Information : http://www.altera.com/products*, 2003.

[2] N. C. Audsley. On Priority Assignment in Fixed Priority Scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[3] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell, and A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling. *Software Eng. J.*, 8(5):284–292, 1993.

[4] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 3rd edition, 2001.

[5] Celoxica Ltd. *Celoxica Product Information : http://www.celoxica.com*, 2003.

[6] S. N. D. Gajski, F. Vahid and J. Chong. System-Level Exploration with SpecSyn. In *Proc. DAC*, pages 812–817, 1998.

[7] G. DeMicheli. *Readings in Hardware / Software Codesign*. Morgan-Kaufman, 2001.

[8] J. Garcia and M. Harbour. Optimized Priority Assignment for Tasks and Messages in Distributed Real-Time Systems. *IEEE Parallel and Distrib. Sys.*, pages 124–131, 1995.

[9] J. Y. T. Leung and M. L. Merrill. A Note on Preemptive Scheduling of Periodic, Real-Time Tasks. *Inf. Pro. Letters*, 11(3), 1980.

[10] J. Y. T. Leung and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Perf. Eval. (Netherlands)*, 2(4):237–250, 1982.

[11] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *J. of the ACM*, 20(1):40–61, 1973.

[12] D. A. Patterson and J. L. Hennessey. *Computer Organisation and Design: The Hardware/Software Interface*. Morgan Kauf., 1998.

[13] K. Tindell. Holistic schedulability analysis for distributed hard real-time systems. Ycs 197, April 1993.

[14] Triscend Corporation. *Triscend Products : http://www.triscend.com/products*, 2003.

[15] J. Wakerly. *Digital Design*. Prentice-Hall, 2000.

[16] M. Ward and N. C. Audsley. Hardware Compilation of Sequential Ada. In *Proc. CASES*, pages 99–107, 2001.

[17] M. Ward and N. C. Audsley. Hardware Implementation of the Ravenscar Ada Tasking Profile. In *Proc. CASES*, pages 59–68, 2002.

[18] M. Ward and N. C. Audsley. Language Issues of Compiling Ada to Hardware. In *Proc. of Ada Europe*, 2002.

[19] Xilinx Corporation. *Xilinx Product Information : http://www.xilinx.com/products*, 2003.