Run Time Mapping for Dynamic Reconfiguration Management in Embedded Systems

Pascal Benoit, Lionel Torres, Gilles Sassatelli, Michel Robert, Nicolas Saint-Jean

Run-time mapping for dynamic reconfiguration management in embedded systems

Pascal Benoit*, Lionel Torres, Gilles Sassatelli, Michel Robert and Nicolas Saint-Jean

LIRMM, Montpellier Institute of Computer Science, Robotics and Microelectronics, University of Montpellier 2,
CNRS, UMR 5506, 161, rue Ada, 34392 Montpellier Cedex 5, France
Fax: (+33) (0)-467-85-00
E-mail: Pascal.Benoit@lirmm.fr
E-mail: Lionel.Torres@lirmm.fr
E-mail: Gilles.Sassatelli@lirmm.fr
E-mail: Michel.Robert@lirmm.fr
E-mail: Nicolas.Saint-Jean@lirmm.fr
*Corresponding author

Abstract: Dynamic reconfiguration provides attractive features such as hardware flexibility and adaptability. Unfortunately, the lack of programming tools to manage it has limited its use in current SoC. This paper presents a method to abstract dynamic reconfiguration management at design time. Dynamic hardware multiplexing is a generic principle based on a scheduler dedicated to the management of reconfigurable resources at run-time. Formal background, implementation, simulation results and validations are exposed to illustrate the contribution of this study.

Keywords: reconfigurable computing; run-time resource management; self-adaptability; task relocation; task duplication; TD; task allocation and scheduling; hardware and software control; embedded systems.

Reference to this paper should be made as follows: Benoit, P., Torres, L., Sassatelli, G., Robert, M. and Saint-Jean, N. (2010) 'Run-time mapping for dynamic reconfiguration management in embedded systems', *Int. J. Embedded Systems*, Vol. 4, Nos. 3/4, pp.276–291.

Biographical notes: Pascal Benoit obtained his PhD from the University of Montpellier in 2004. He then joined the Department of Electrical Engineering at the University of Karlsruhe in Germany where he worked as a Scientific Assistant. Since 2005, he is an Associate Professor at the University of Montpellier in the Flexible Architectures Group at the LIRMM Microelectronic Research Department.

Lionel Torres obtained his PhD in 1996 from the University of Montpellier in France. From 1996 to 1997, he was involved in ATMEL Company as IP Core Methodology Engineer. He is currently a Professor at the University of Montpellier and the Head of the Microelectronic Department of the LIRMM laboratory.

Gilles Sassatelli holds a Full-Time Researcher position at CNRS. He obtained his PhD in 2002 in Microelectronics. He is responsible for the flexible parallel and reconfigurable architectures group at LIRMM.

Michel Robert is a Professor at the University of Montpellier 2. He is nominated as a member from the French University Institute in 1997, Advisor at the Research Ministry, and Scientific Co-Director from the STIC (Information and Communication Technology and Science) Department of CNRS (2002–2004). He is currently the Vice President of CNFM (National Committee of Microelectronic Education) since 1999, and the Director of LIRMM since 2005.

Nicolas Saint-Jean obtained his Master in 2005. He is currently a PhD student at LIRMM.

1 Introduction

The explosion of standards in 3G and more generally in wireless systems goes along with a widening spectrum of applications that portable devices have to support. Designing a chip for every single device tends to become less feasible (SoC complexity, lifecycle of multimedia product, increasing non-recurring engineering costs for deep-submicron technologies). A considered way to overcome these problems relies on flexibility, i.e., reusing

the same chip for a range of products, applications, or even several generations of the same product. This approach allows sharing the non-recurring engineering costs, and the design stage comes down to a software customisation phase. Reconfigurable architectures provide this flexibility at the price of a silicon area overhead highly dependent on the level of flexibility.

1.1 Related works

Recently, FPGA cores integration has become a reality in embedded systems. Several companies as eASIC (http://www.easic.com/) and M2000 (http://www.m2000.fr/) offer dense customisable FPGA macros that can be used as configurable logic associated to a general purpose processor. Using reconfigurable logic provides an increased flexibility since it then becomes feasible to modify the configuration after the chip fabrication, to upgrade the device during its life cycle, at the price of a reduced silicon density. Dynamic reconfiguration is a third level of flexibility and consists in modifying the configuration of the device at run-time. For typical reconfigurable architectures such as FPGA, run-time reconfiguration is generally costly as it requires a large amount of data. To overcome this drawback, coarse grain reconfigurable architectures have been studied and developed (Table 1) for the past 15 years.

More recently, we have observed a growing interest in dynamic reconfiguration management techniques in the literature. Table 2 exposes a synthetic overview of different dynamic configuration management techniques. In the literature, most of the approaches target fine grain partially reconfigurable devices [XC6200 (Shirazi et al., 1998; Burns et al., 1997; Robinson and Lysaght, 1999), Virtex-II Pro (Curd, 2003), Virtex II (Blodget et al., 2003; Carvahlo et al., 2004; Huebner et al., 2004; Ullman et al., 2004) or Altera-like architectures (Danne and Platzner, 2005). Some of the proposed schemes provide a support for the relocation of tasks (Burns et al., 1997; Robinson and Lysaght, 1999; Carvahlo et al., 2004; Danne and Platzner, 2005; Huebner et al., 2004) where they are handled as a fixed-size or variable-size rectangle that can be placed on the FPGA resource. Each method may differ in the placement and scheduling, aiming at optimising cost functions. In the first methods (Shirazi et al., 1998; Burns et al., 1997), the scheduling was performed statically by heuristics and in Robinson and Lysaght (1999), Carvahlo et al. (2004), Huebner et al. (2004) and Ullman et al. (2004), this is processed dynamically. A configuration controller is then generally supposed to execute tasks as a loader in an operating system, according to a calculated scheduling. This one can be directly implemented in the reconfigurable logic on the FPGA (Curd, 2003; Blodget et al., 2003; Carvahlo et al., 2004; Huebner et al., 2004; Ullman et al., 2004). Real-time scheduling of hardware tasks on FPGA has been studied in Danne and Platzner (2005) and in Huebner et al. (2004), a dynamic placement of tasks is performed thanks to a NoC architecture. In Ullman et al. (2004), an adaptive priority scheme allows a partial reconfiguration of relocatable tasks. An interesting solution for multi-tasking was also proposed in the SCORE project (Caspi et al., 2000), a multi-threaded computational model and architecture that rely on a scalable dynamic scheduling. The RAW architecture (Taylor et al., 2002) allows dynamic switching of tasks but with a compiled programme.

Table 1 Examples of dynamically reconfigurable architectures

Name	*Grain*	*Type*	*Comment*	*Ref.*
AT40K	Fine	Stand-alone	Cell reconfig.	Atmel Corporation
Xilinx	Hybrid	Stand-alone RSoC	Partial reconfig.	Xilinx Corporation
SCORE	Fine	Stand-alone	Mesh-based	Caspi et al. (2000)
RAW	Coarse	Stand-alone	Switch-box	Taylor et al. (2002)
Piperench	Coarse	Stand-alone	Pipeline	Goldstein et al. (200)
MorphoSys	Coarse	Co-processor	Mesh-based	Singh et al. (2000)
Systolic ring	Coarse	Co-processor	Circular pipeline	Sassatelli et al. (2002)

Table 2 Examples of dynamic reconfiguration management techniques

Name	*Sched.*	*Reloc.*	*Replic.*	*Config. manager*	*Location*	*Ref*
XC6200	Static	No	No	No	-	Shirazi et al. (1998)
XC6200	Static	Yes	No	No	-	Burns et al. (1997)
XC6200	Dyn.	Yes	No	No	-	Robinson and Lysaght (1999)
V2-Pro	Static	No	No	SW PowerPC	FPGA	Curd (2003)
V2	Static	No	No	SW µBlaze	FPGA	Blodget et al. (2003)
SCORE	Static	No	No	HW	Dedicated module	Caspi et al. (2000)
RAW	Static	No	No	Compiler	-	Taylor et al. (2002)

1.2 Our approach

Dynamic reconfiguration management techniques become mandatory in a SoC context where next generation circuits will have to be able to handle several applications simultaneously in an adaptive manner. In the literature, most of the approaches target fine grain partially reconfigurable circuits. But a major drawback of fine grain architectures is the latency introduced by their reconfiguration time. Coarse grain reconfigurable architectures allow to significantly reduce the configuration time, then allowing an increased reactivity to dynamic environments. For these kinds of architectures, only compiler support has been proposed to manage the dynamic reconfiguration. Our purpose is to explore a dynamic reconfiguration technique based on a run-time controller support allowing an adaptive placement and scheduling of unpredictable task scenarios.

The objective of this paper is then to present a technique to handle dynamic configuration with the hardware, allowing the application designer to abstract dynamic reconfiguration management constraints. This technique is based on a run-time scheduler implementing an algorithm called dynamic hardware multiplexing (DHM). Thanks to this approach, the configuration of each application is generated at design time. At run-time, the scheduler takes care of the placement and modifies the original configuration in order to adapt the computational resources to handle several applications or to implement different qualities of service. The efficiency of our approach is characterised by two-metrics showing the increase in the resource usage and multi-application management. The suggested method has been designed and validated on a coarse grain reconfigurable architecture.

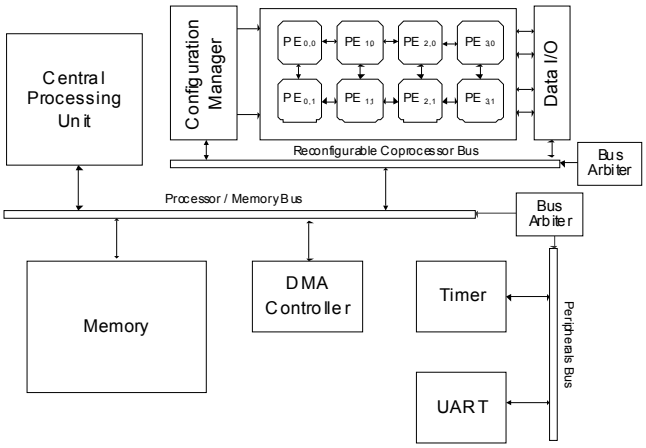
This paper is organised as follows: Section 2 presents a context analysis to derive the formal framework and characterisation metrics; Section 3 describes the dynamic HW multiplexing algorithms; Section 4 illustrates the implementation of DHM carried out on a coarse grain reconfigurable architecture; simulation results are exposed in Section 5 and HW/SW DHM schedulers validations are discussed in Section 6; finally, conclusions and perspectives are drawn.

2 Context analysis and formalisation

Self-adaptability represents a real challenge in the SoC context. One promising approach to achieve it consists in using a reconfigurable co-processing unit as illustrated in Figure 1.

To achieve this objective of adaptability, it is necessary to provide a middleware layer to allow a smart configuration management (the 'configuration manager' in the figure). This unit must be able to communicate relevant information to the operating system so that it allows a local and a global supervision of the co-processor configurations. In this section, we develop a formal framework for this issue, in order to define a solution to multi-application handling through dynamic reconfiguration.

Figure 1 Simplified overview of the targeted SoC



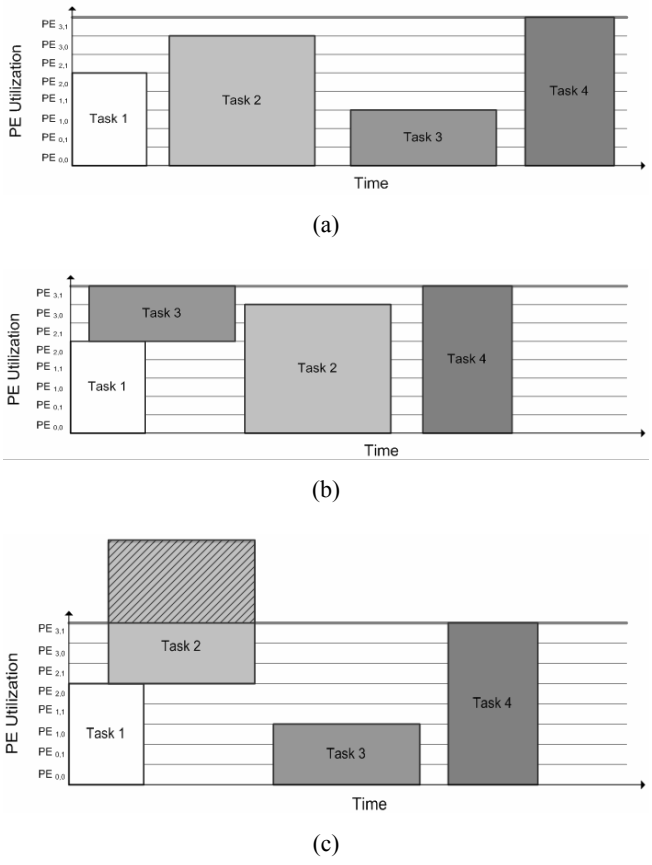
2.1 Execution scenario

We assume a set θ of random tasks potentially assignable to the reconfigurable co-processor as a hardware task (which assumes that this task is time-consuming in a software fashion implementation). The cardinal of this set represents the number of requests from the OS. We assume the system is running during n cycles (from 0 to n). Each task $T_k \in \theta$ is executed during n_k cycles (n_k is a non-deterministic variable, e.g., depends on how long the user wants to run the application). At each cycle of [0, n], the co-processor is running a set of tasks $\Gamma \subseteq \theta$. At cycle n, the co-processor has handled *card* ε tasks where ε is the set of executed tasks ($\varepsilon \subseteq \theta$).

The reconfigurable co-processor is modelled as a set of homogenous processing elements P (further along, our model will consider the memory bandwidth between the co-processor and the memory as a set of memory channels C). Each task T_k is represented by a set of processing elements P_k (and further along by a set of memory channels C_k). Figure 2 illustrates three possible scenarios (a, b, c) with four tasks, θ = {Task 1, Task 2, Task 3, Task 4}, and with different starting points. The co-processor has eight processing elements as depicted in Figure 1. The time is represented on the X-axis in Figure 2, and the resource utilisation (PE) on the Y-axis. In scenario (a), all requests are handled by the reconfigurable co-processor; there is no particular observation in this example: each new task starts after the completion of the previous task. In scenario (b), Task 3 is started before the completion of Task 1. There are three PE required and three are available. The task is potentially assignable but this requires a relocation of the initial configuration. In scenario (c), Task 2 is started before the completion of Task 1. However, Task 2 and Task 1 cannot be executed simultaneously as they require more than the total of available PE (the area above $P_{3,1}$). This entails that the system must be able to stall either the execution of Task 1 on the reconfigurable co-processor if Task 2 has a higher priority (pre-emption) and make it

possible to execute Task 2 or Task 1 alternatively on another resource of the system (the CPU, i.e.).

Figure 2 Execution scenarios of task processing and co-processor utilisation



2.2 Task configuration management

The management of the configurations depends on a set of conditions. From the previous observations, we define the three following lemmas:

Lemma 1: A task $T_k \in \theta$ can be handled if the number of required resources is currently available i.e., $card(P_k) \leq card(P) - \Sigma_{Tr\in\Gamma}\ card(P_r)$ and $card(C_k) \leq card(C) - \Sigma_{Tr\in\Gamma}\ card(C_r)$.

Lemma 2: A task $T_k \in \theta$ is allocated to the reconfigurable accelerator if *Lemma 1* is verified and if the configurations of the running tasks are compatible with the configuration of T_k i.e., $(\Sigma_{Tr\in\Gamma}\ P_r) \cap P_k = \varnothing$ and $(\Sigma_{Tr\in\Gamma}\ C_r) \cap C_k = \varnothing$.

Lemma 3: If Lemma 1 or Lemma 2 are not respected and if $T_k \in \theta$ has a higher priority than at least one of the running tasks, then this task is temporarily stalled for lemma 1 and lemma 2 analyses. If both are verified, the stalled task is permanently removed from the co-processor and migrated to another resource of the system (the CPU, i.e.). Or else, another task with lower priority is also temporary stalled and so on. This step is done until the compatibility is found; otherwise the OS chooses an alternative resource to execute T_k.

The critical part of previous propositions is to determine a new configuration from the initial one, to fit the current state of the reconfigurable co-processor resources. This process is commonly named 'relocation' of a hardware task. This has been studied in the literature for FPGA but from our knowledge, never deeply explored for coarse grain reconfigurable architectures. In this case, the relocation process can be formalised as follows: if $f(T_i)$ is the function implemented by T_i the function *transform*() must verify $f(T_k) = f(transform(T_k))$, In other words, *transform*() is a relocation process able to generate a new configuration $\{P_k, C_k\}$, functionally equivalent to the configuration of T_k. This function depends on the targeted architecture, i.e., the interconnection topology, the homogeneity of the PE, type of I/Os, memories, etc.

The example exposed in Figure 3 illustrates this proposition. The coarse grain reconfigurable architecture (a) is a four PE structure. Each PE has the same reconfigurable functionality, i.e., each one can implement the same predefined set of arithmetic functions. Assuming that this reconfigurable architecture has to compute the following function: $f(a, b) = (a + b) * (a - b)$, one possible configuration can be the one exposed in (b). But the same functionality can be also achieved with the configuration in (c). This functionally equivalent configuration is actually obtained with a modulo relocation process, i.e.:

$$f\left(P_{i,j}\right) = f\left(P_{i,(j+t)\%2}\right),$$

where t is a translation of the functionality to the adjacent PE, and two because of the two parallel PE.

The modulo-relocation process can be adapted to any architecture based on a torus or a ring topology. Anyway, this methodology can also be applied to any architecture with respect to its properties in terms of PE functionality, interconnection topology, I/Os… Moreover, as we will see in the next section, the 'modulo relocation' process can be used for other purposes, such as task duplication (TD), i.e., to increase the intrinsic performances.

2.3 Characterisation metrics

In order to evaluate the quality of the configuration management, it is necessary to define metrics. The multi-tasking efficiency (MT_{eff}) computes the percentage of tasks handled by the reconfigurable co-processor compared to the number of OS requests:

$$MT_{eff} = 100 \cdot \frac{card(\varepsilon)}{card(\theta)}$$

This metric represents the task acceptance ratio and it is basically dependant on the system workload.

Figure 3 Coarse grain reconfigurable architecture example (a) and (b) (c) implementations of (a + b)*(a – b)

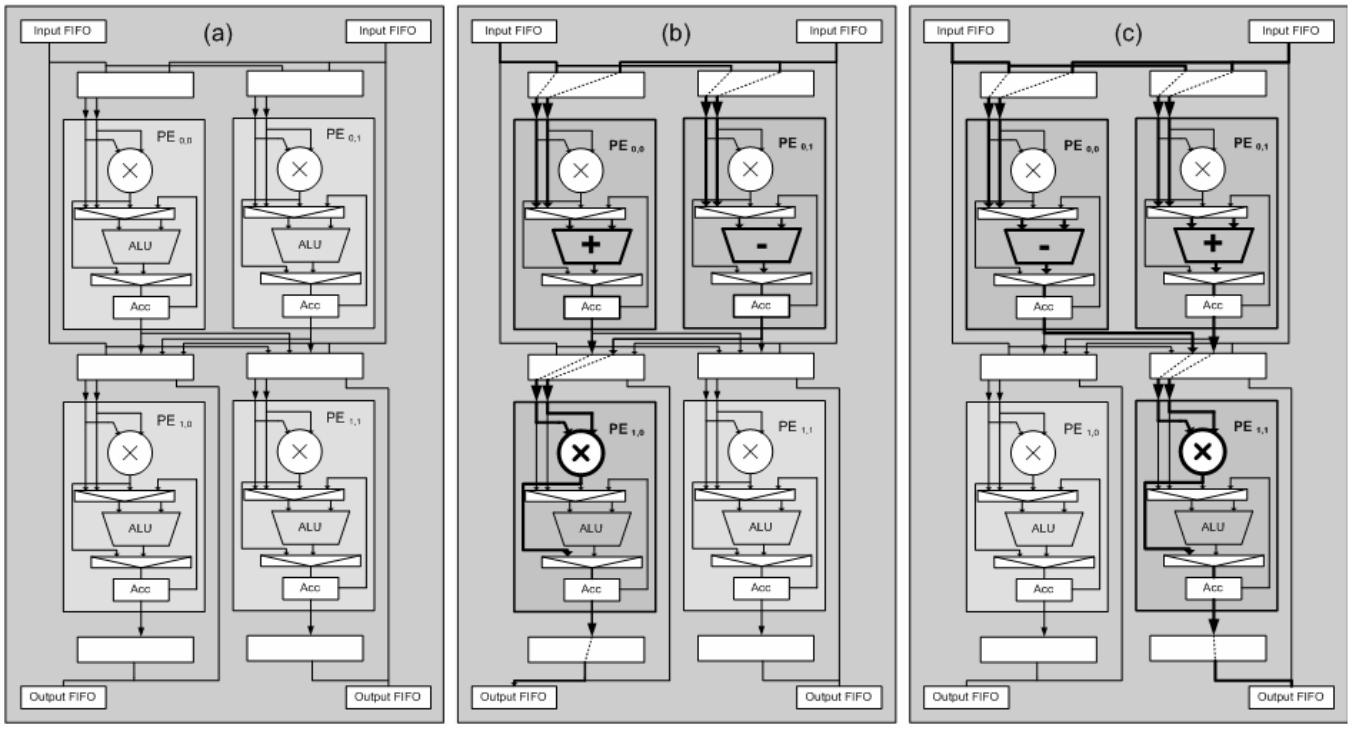
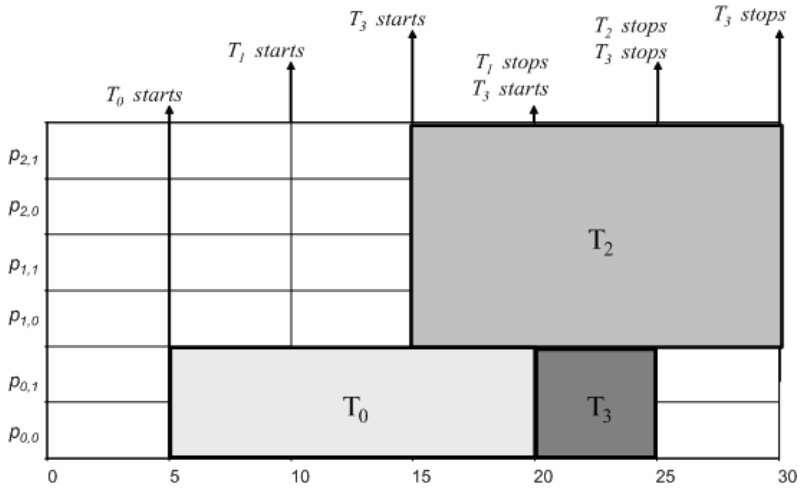


Figure 4 Task scenario modelled with the proposed formalism

Tk	P_k	Start cycle	Stop cycle	n_k
T_0	{1,1,0,0,0,0}	5	20	15
T_1	{1,1,1,1,1,0}	10	25	15
T_2	{1,1,1,1,0,0}	15	30	15
T_3	{1,1,0,0,0,0}	20	25	5

Card P= 6, n=30
θ ={T0, T1, T2,, T3 }, card θ = 4
ε={T0, T2, T3 }, card ε=3
*Mteff (%) = 100*2/3= 75%*
*Peff (30)= (15*2+4*15+5*2)*100/(6*30) = 55%*
*WL= (15*2+5*15+4*15+5*2)*100/(6*30) = 97%*
*R=(25/30)*100= 83%*



The system workload is dependent on the load that each task of θ implies on a given architecture, during a given time. Thus, we define the metric WL as follows:

$$WL = 100 \cdot \frac{\sum_{T_k \in \theta} card\left(P_k\right) \cdot n_k}{n\,card(P)}$$

We define another metric, *R*, to represent the time usage ratio:

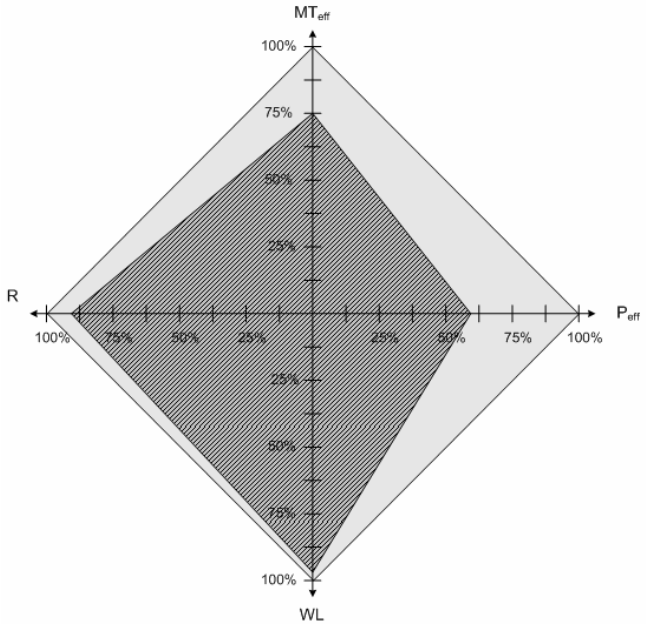
$$R = 100 \cdot \frac{\sum_{T_k \in \varepsilon} n_k}{n}$$

The processing efficiency (P_{eff}) is defined to calculate the space and time utilisation percentage ratio of the co-processing resources:

$$P_{eff} = \frac{100}{\left(n \cdot card(P)\right)} \cdot \sum_{t_k \in \varepsilon} card\left(P_k\right) \cdot n_k$$

Figure 4 depicts an example of a set of tasks $\theta = \{T_0, T_1, T_2, T_3\}$ handled by a reconfigurable architecture with $P = \{p_{0,0}, p_{0,1}, p_{1,0}, p_{1,1}, p_{2,0}, p_{2,1}\}$. This example shows that for this set of tasks, a multi-tasking efficiency of 75% is achieved with a 55% overall processing efficiency and relative P_{eff} is 66% with a time usage ratio R = 83% (Figure 5).

Figure 5 Architecture characterisation with the proposed metrics



3 Dynamic hardware multiplexing (DHM)

DHM is the proposed approach to address the objective of system-on-chip adaptability with reconfigurable hardware. It is based on a dedicated application design flow and a reconfigurable system platform with a so-called dynamic hardware multiplexer.

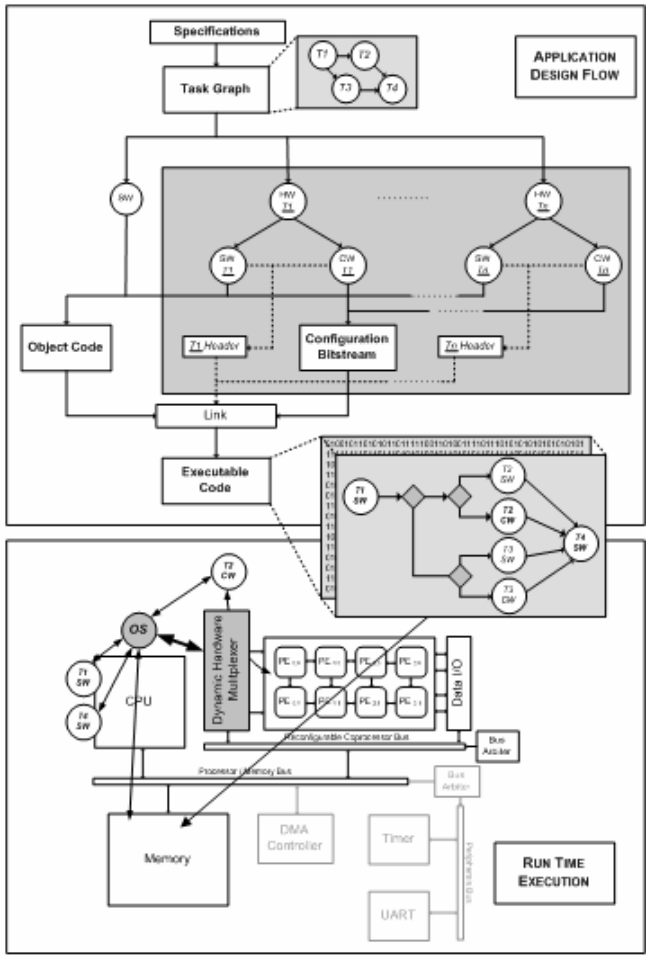
3.1 DHM design flow

The proposed design flow is depicted in Figure 6. It is based on a two-phase process: the first step is performed at design-time and the second one at run-time.

The application is represented as a task graph. A first analysis allows to partition the graph and to classify each task. The non-critical tasks are compiled as software tasks (SW). Each critical task (time/power consuming) is given a priority level. Then, they are both compiled as SW tasks and as HW configurations (configware or CW). A header is attached to each HW task: it contains information about the task priority level, the number of required operators, etc. The compiled SW tasks produce an object code, and the HW tasks a configuration bitstream. They are finally linked together in a unique executable which includes both object codes and configuration bitstreams.

The task allocation is performed at run-time under OS control. A dedicated system call is used for this purpose: each time a task has been assigned to HW, the header is sent to the OS which communicates a co-processing request to the DHM unit: this one analyses the task header and compares the task requirements and the state of reconfigurable hardware (number of available operators, priorities of running tasks, etc.). The DHM unit takes the decision either to accept or reject the co-processing request. In the first case, the configuration bitstream is loaded into the reconfigurable co-processor, in the second one; the task object code is scheduled by the OS on the CPU.

Figure 6 DHM design flow



The DHM unit acts as a dedicated service of the OS for HW management. This service is detailed in the following section.

3.2 DHM services: mapping analysis, relocation and replication

The DHM unit is implemented as an OS service allowing an adaptive task mapping on a given reconfigurable architecture. It is composed of two procedures: the first one is dedicated to the mapping analysis and relocation (simultaneous multi-tasking mapping algorithm), and the second one to the replication (TD mapping algorithm).

3.2.1 Mapping analysis and task relocation

The first function of the DHM unit is to perform a mapping analysis regarding the co-processing (*OsTaskRequest*) request through the header. The OsTaskRequest is structured as follows:

```
typedef struct OsTaskRequest {
   On;
   TaskId;
```

```
  Header;
  * ConfigTaskAddress;
}OsTaskRequest;
```

The *On* field is a Boolean that indicates if the task starts or ends up. The *TaskId* is a number attributed by the OS to identify the task. The *ConfigTaskAddress* is a pointer to the configuration bitstream memory location. The Header field is defined as follows:

```
typedef struct ConfigurationHeader {
  nOp;
  nChannels;
  Topology[N];
  Priority;
}ConfigurationHeader;
```

This header allows to determine whether the task can be mapped. The number of required resources in terms of operators and memory channels (*nOp* for the number of processing units, *card* (*Pi*) as previously mentioned, and *nChannels*, the number of DMA channels required, *card*(*Ci*)) is first analysed in Proposition 1 (i.e., Lemma 1). Then, a topological compatibility test (*Topology*[*N*], or as previously mentioned {*Pi*, *Ci*}) is realised in Proposition 2 (i.e., Lemma 2). If both propositions are true, the task T_k is directly allocated: the corresponding *configuration bitstream* is then loaded in the configuration memory without changes. If Proposition2 is not verified, the *transform*() function is executed to compute a new configuration with an equivalent functionality (as exposed in the previous section) until a compatible solution is found or all the equivalent configurations have been tested (*TransformationIsPossible* becomes false). Consequently, a map function is used to load the configuration and apply the required transformations on the topology. When a task needs more than the available resources or cannot be adapted to the current state of running tasks, tasks are sorted by priority levels and mapped accordingly. When a task has to be removed or when it cannot be handled by the co-processor, the DHM unit generates a CPU equivalent context (initial or current) and transmits it to the OS: the SW task is then executed by the CPU.

3.2.2 Task replication

Task replication has several advantages like increasing the processing efficiency (P_{eff}) or minimising power consumption (if a computational task is duplicated, the operating frequency can be divided by 2 with the same performance): this is the role of the TD mapping function (Figure 8). The principle is quite simple and similar to the task relocation: after sorting the running tasks by priority levels, it tries to map a second instance of each task on the free resources. This process is executed each time the SMT mapping is executed and when a new task request is pending, a background routine suspends all duplicated tasks.

Figure 7 Task mapping analysis and relocation process

```
1: process SmtMapping()
2:   Γ←∅
3:   wait for OsTaskRequest
4:     if OsTaskRequest.On then
5:       k ← OsTaskRequest.TaskId
6:       Tk.Header ← OsTaskRequest.Header
7:       if Proposition1 then
8:         if Proposition2 then
9:           ConfigBitStream ← OsTaskRequest→ConfigAddress
10:          Map(Tk, ConfigBitStream)
11:          Γ ← Γ ∪ Tk
12:        else
13:          do
14:            Tk.Header.Config ←Transform(Tk.Header.Config)
15:            if Proposition2 then
16:              ConfigBitStream ← OsTaskRequest→ConfigAddress
17:              Map(Tk, Tk.Header.Config, ConfigBitStream)
18:              Γ ← Γ ∪ Tk
19:              UnMapped ← false
20:            end if
21:          while UnMapped && TransformationIsPossible
22:          if UnMapped
23:            for each Ti ∈ Γ
24:              i ← QuickSortLowPriority(Γ)
25:              if Tk.Header.Priority > Ti.Header.Priority
26:                UnMap(Ti, ConfigBitStream)
27:                Γ← Γ -{ Ti }
28:                go to 8
29:              end if
30:            end for
31:          end if
32:        end if
33:      else
34:        for each Ti ∈ Γ
35:          i ← QuickSortLowPriority(Γ)
36:          if Tk.Header.Priority > Ti.Header.Priority
37:            UnMap(Ti, ConfigBitStream)
38:            Γ← Γ -{ Ti }
39:            go to 7
40:          end if
41:        end for
42:      end if
43:    else
44:      k ← OsTaskRequest.TaskId
45:      Γ← Γ -{ Tk }
46:    else
47:      TrMapping(Γ)
48:    end if
49:  end while
50: end process
```

Figure 8 Task relocation process

```
51: process TdMapping(Γ)
52:  for each Ti ∈ Γ do
53:     k ← QuickSortHighPriority(Γ)
54:     dTk.Header ← Tk.Header
55:     if Proposition1 then
56:       if Proposition2 then
57:         ConfigBitStream ← (Tk.(Header→ConfigAddress))
58:         MapAgain(Tk,dTk.Header.Config,ConfigBitStream)
59:       else
60:         do
61:           dTk.Header.Config ← Transform(dTk.Header.Config)
62:           if Proposition2 then
63:             ConfigBitStream ← (Tk.(Header→ConfigAddress)
64:             MapAgain(Tk, dTk.Header.Config, ConfigBitStream)
65:             Duplicated ← true
66:           end if
67:       while !Duplicated && TransformationIsPossible
68:     end if
69:   end if
70:  end for
71: end process (go to 3, wait for OsTaskRequest)
```

3.3 DHM interaction with the OS micro kernel

The micro kernel is globally considered to operate as a classical pre-emptive and interrupt driven operating system. It manages the task-control by keeping track of the status or state of each task. In our approach, a task can typically be in any one of the following states:

1 executing

2 ready

3 suspended or blocked

4 co-executing.

A task is said to be *executing* when it is actually running as SW code in the CPU. Tasks in the ready state are those that are ready to run but are not running. Tasks that are waiting on a particular resource, and thus are not ready, are said to be in the *suspended* or *blocked* state. A task identified as critical at design time is set as a HW task and can then be executed either as SW or CW code depending on the co-processing resource availability. When the micro kernel loads such a task, it generates a request to the DHM unit which performs a mapping analysis. During this time, the task is set to *suspended* mode in the micro kernel task list. When the analysis is completed, it generates an interrupt to the micro kernel which indicates whether the task can be implemented in SW or in CW fashion. In the first case, the task is set to *ready* and in the second one to *co-executing*.

When a process is killed, if the task is *executing* or *ready*, it is simply removed from the task list of scheduling and dispatching units of the OS. If it is *suspended* or *co-executing*, an OS request is sent to the reconfigurable co-processor to indicate that the task is being resumed. In this way, the resources involved are released and the task is removed from the OS task list.

4 Case study

4.1 Targeted architecture

The systolic ring is a customisable coarse grain reconfigurable model. It features a compact DSP-like coarse grain reconfigurable datapath, the *Dnode*, which is the building block of the architecture. The architecture is configured by a microinstruction code loaded from the memory to a local sequencer using dedicated registers as a configuration memory. Figure 9 describes this architecture.

Figure 9 Processing element architecture

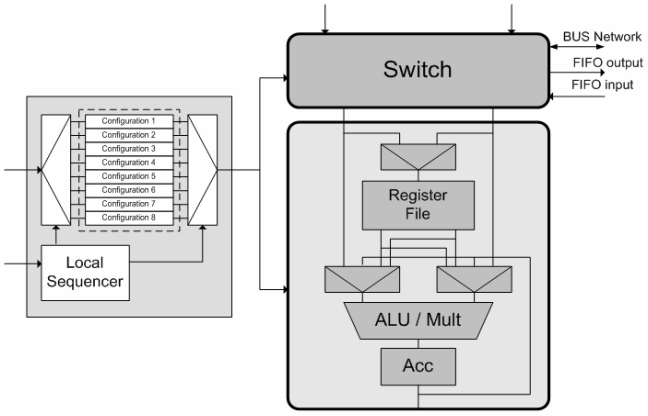
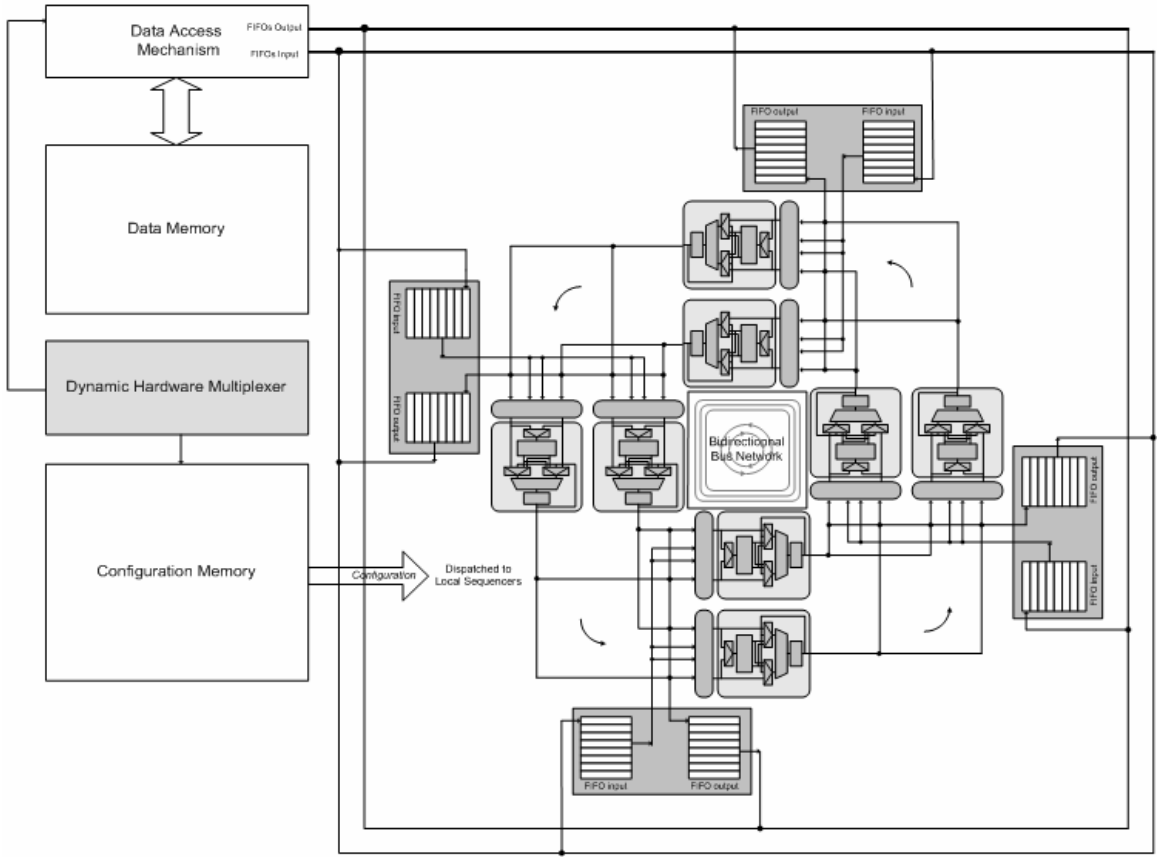


Figure 10 System overview

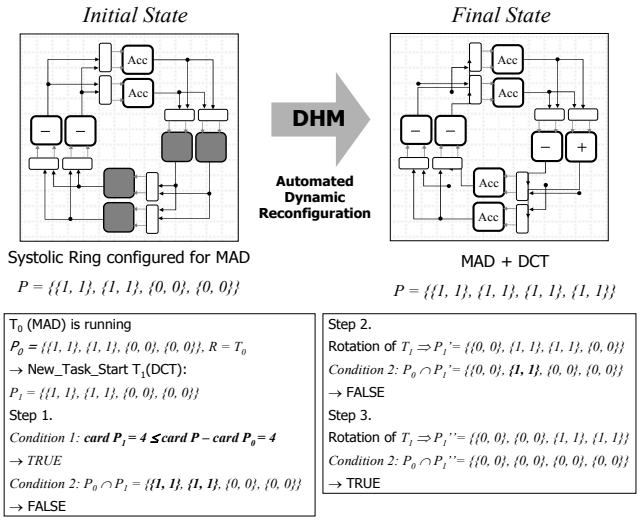


The specific structure of the operating layer is depicted in Figure 10. The ring topology allows an efficient implementation of pipelined datapath. The switch components establish a full connectivity between two layers. The systolic ring also provides an alternative connection bus network which proves useful for recursive operations, i.e. It allows to feedback data to previous layers implemented from each switch in the structure. Each switch in the architecture has a read access on each switch's bus.

Each configuration programme line is able to set the configuration of an entire Dnode layer (2 Dnodes on the 8 Dnodes systolic ring depicted in Figure 10) each cycle. Up to 12.5% of the Dnodes can be reconfigured at each cycle in the actual version, but this can be tailored, especially when d/l varies, d being the number of Dnodes per layer and l the number of layers. An assembler/simulator environment has been developed. This environment also generates the object code running on the global sequencer and dynamically managing the configuration.

With the formalism developed in Section 3, we can describe the architecture as $P = \{\{p_{0,0}, p_{0,1}, \ldots, p_{0,d-1}\}, \{p_{1,0}, p_{1,1}, \ldots, p_{1,d-1}\}, \ldots, \{p_{l-1,0}, p_{l-1,1}, \ldots, p_{l-1,d-1}\}\}$ and $C = \{c_0, c_1, \ldots, c_1\}$ with dim $P = 1 \times d$ and $dim\ C = 1$. The total number of processing elements of the architecture is then given by card $P = l.d$.

Figure 11 Task mapping with rotation-based transformation scheme



4.2 Transform process implementation

The proposed DHM algorithms can be applied to any coarse grain reconfigurable architecture supporting run-time partial reconfiguration. The only architecture-dependent function is the transformation process applied to modify the configuration. This can be easily derived from the topology of the targeted architecture. For instance, the systolic ring architecture is based on a homogenous ring topology where each processing element is able to implement the same set F of arithmetic and logic functions. Thus, a simple rotation (an example is depicted in Figure 11) of the configuration following the dataflow direction (i) is functionally-equivalent. The following formula formalises this lemma and is used in the DHM algorithm in order to find equivalent implementations:

$$\forall r, i \in [0..l-1] \text{ and } j \in [0..d-1], f\left(p_{i,j}\right) = f\left(p_{(i+r)\%l,j}\right)$$
$$\text{and } f\left(c_j\right) = f\left(c_{(i+r)\%l}\right) \text{ where } f \in F$$

4.3 Multi-tasking scenario with image processing kernels

Image processing is well known as a time consuming application field. Therefore, its implementation is often inefficient on general purpose processors making alternative approaches attractive. Several test benches have been implemented on the systolic ring in order to validate our purpose on real world applications. We focus here on the kernel acceleration which represents generally more than 80% of the CPU time required on a general purpose processor.

Figure 12 details the even-odd frequencies decomposition of a discrete cosine transform (DCT) (Chen et al., 1977) implementation. This decomposition splits the calculations into two matrix products. Once the first layer calculations have been carried out, the sums and differences of the samples are sent to the following layer which performs the multiplication-accumulation.

Several studies have proven that the wavelet transform is an efficient alternative to the classical DCT, and thus, it has been chosen for the JPEG2000 standard. Our implementation uses the lifting scheme (Sweldens, 1998) algorithm and operates a 2D direct transform on a 1,024 × 768 pixels 16 bits coded image. One pixel sample is computed each clock cycle, thanks to the use of 4 Dnodes, two based on local-mode implementation (Figure 13) and two global buses.

Block matching, and especially full search block matching (FSBM) algorithm is the most popular implementation, also recommended by several standard committees for motion estimation (Park and Burleson, 1997) (MPEG-video-, and H.261 – videoconferencing – standards). Figure 14 illustrates its implementation on the systolic ring.

Figure 15 depicts the scheduling and the mapping of previous tasks obtained thanks to the use of the DHM unit with the systolic ring on several scenarios. Thanks to the relocation scheme, the same task can be mapped on different processing elements with the same behaviour and functionality as expected at design time.

Figure 12 DCT implementation onto the systolic ring (see online version for colours)

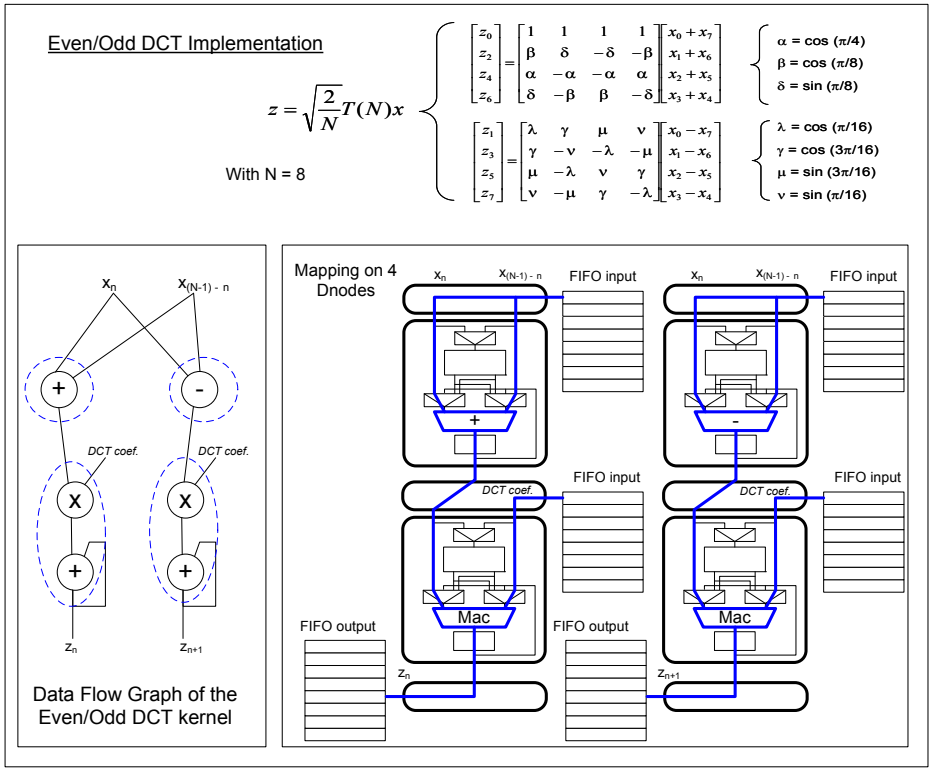


Figure 13 Wavelet transform implementation onto the systolic ring (see online version for colours)

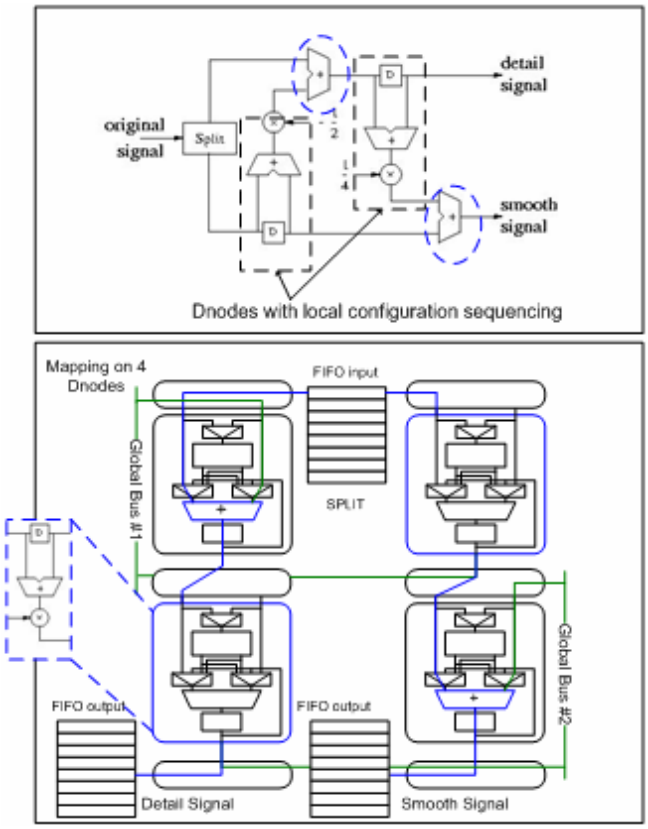


Figure 14 Block matching implementation onto the systolic ring (see online version for colours)

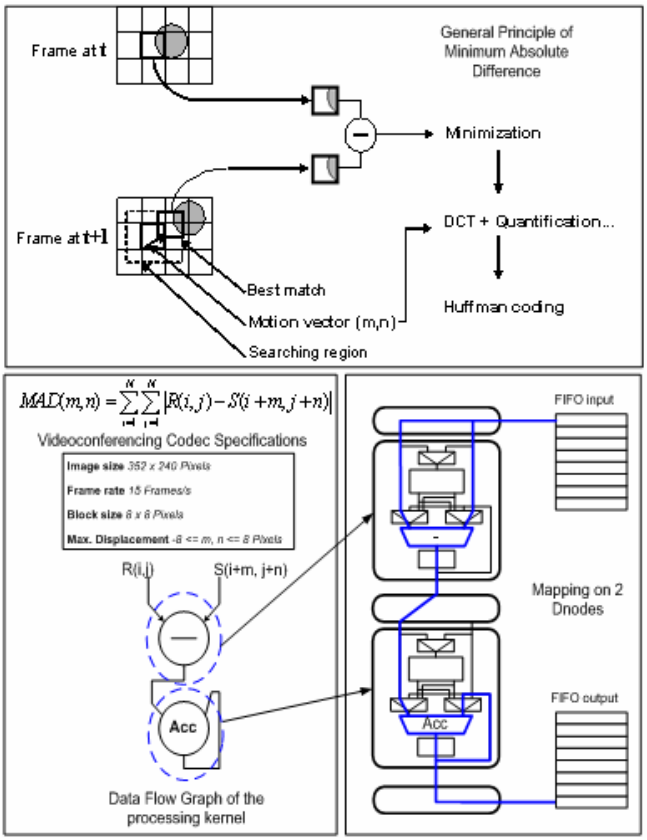
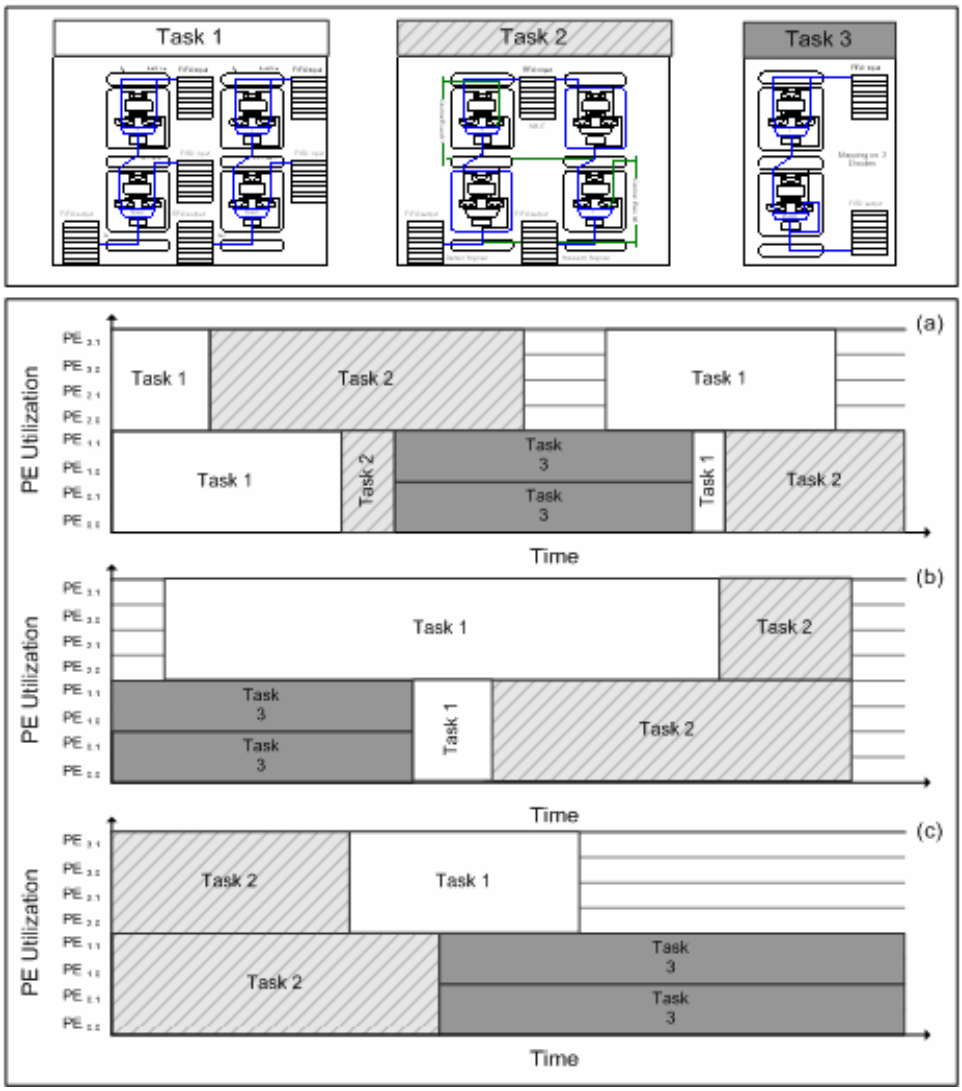


Figure 15 Task mapping/scheduling in three different scenarios (see online version for colours)

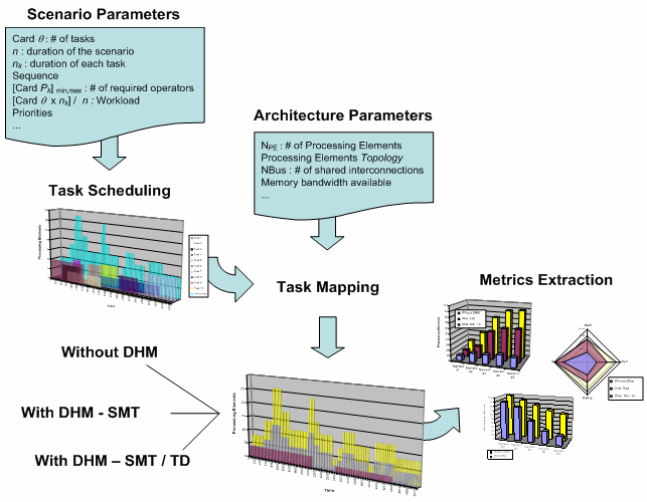


5 Evaluation and experimental results

The DHM unit has been implemented and attached to the systolic ring, and then tested on three DSP kernels. Simulations have demonstrated the run-time adaptability on different application scenarios. In order to generalise our approach on larger sets of scenarios, a simulation framework has been developed and is presented in this section.

Figure 16 depicts the simulation environment developed to explore and characterise the proposed approach. The generation of task scheduling and the DHM mapping is completely automated allowing to produce results on a large number of scenarios. The simulation framework is composed of a C++ programme with constants defined to tune the scenario characteristics or architecture generic parameters. This programme is interfaced with a spreadsheet programme calculating and probing the resulting task mapping. Also, the previous metrics are automatically computed and plotted for characterisation.

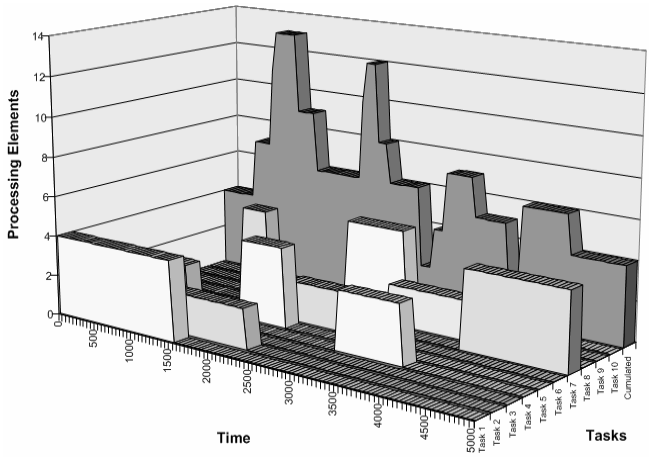
Figure 16 The simulation environment (see online version for colours)



Two instances of the systolic ring have then been experimented: an eight processing element version (l = 4, d = 2) and a 32 processing element instance (l = 8, d = 4). For a given architecture, ten scenarios are generated with a random task sequence and a given range of scenario parameters. For each scenario, the parameters have been drawn lots: the task requirements in terms of processing elements and memory bandwidth, the workload (number of tasks from 2 to 80 ($card\ \theta \in [2, 80]$, on a given time n with variable task duration n_k), and the task sequence. Each generated scenario corresponds to a task scheduling. Three mapping procedures are then performed: one without the DHM capabilities, other ones with simultaneous multi-tasking procedure (DHM-SMT), and the last one with SMT and task duplication (DHM-SMT-TD). In order to extract statistical results, a total of 300 scenarios have been simulated for a given workload. Thus, the presented values in the following figures correspond to a mean value on 300 samples.

Figure 17 depicts the resulting task scheduling (ten tasks) with a relative workload for both tested architectures (80% for 8-PE systolic ring, and 20% for 32-PE systolic ring). The Blue histogram represents the cumulated resource requirements at a given time. The size and shape of the tasks were inspired from the image processing kernels presented in the previous section. During a run-time duration equal to 5,000 seconds, ten tasks are run. We observe a maximum of 13 resources required and a mean value around six processing elements.

Figure 17 Example of a generated task-scheduling



We have depicted in Figure 18 and Figure 19 the resulting placement and scheduling on both 8-PE systolic ring and 32-PE systolic ring. In Figure 18, we notice that without any relocation possibility, the co-processor is underexploited and it rejects a lot of task requests. With DHM SMT, the resource utilisation is clearly higher. Because of its relocation feature, the task acceptance is improved. Thanks to the TD of DHM, we observe on the yellow histogram that the maximum of available resource can be reached while on the other ones it is either seldom (with SMT) or never. However, this technique does not increase the task acceptance.

Figure 18 Task scheduling on 8-PE

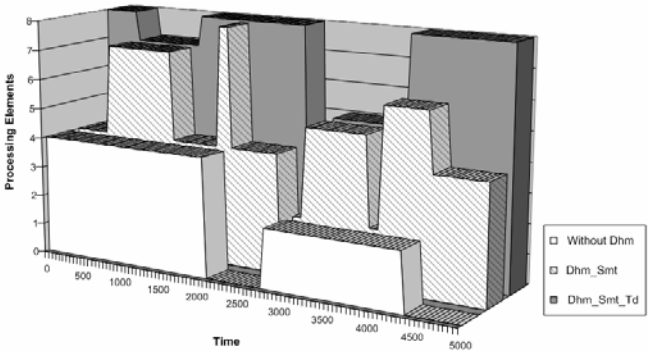


Figure 19 Task scheduling on 32-PE

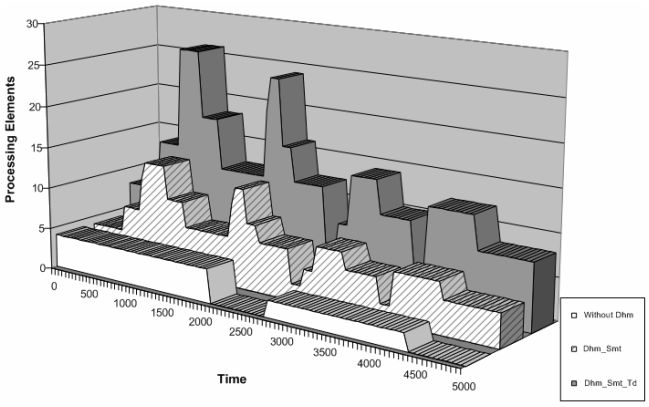
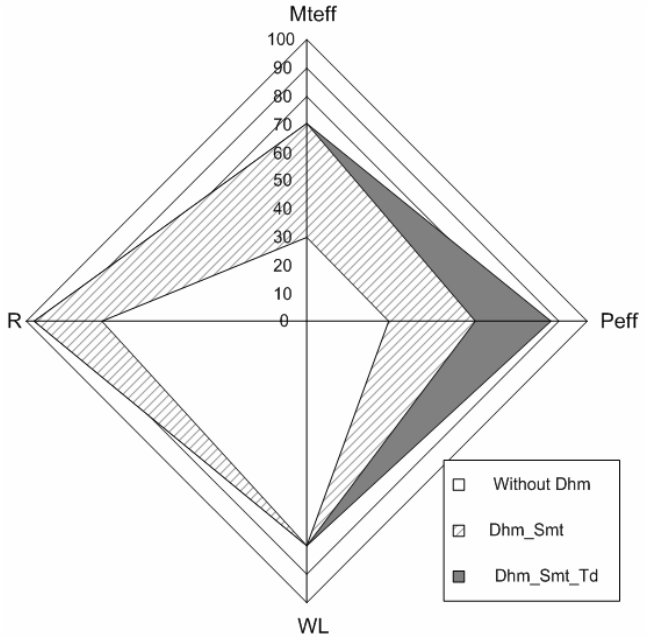


Figure 20 Characterisation with 8-PE



In Figure 19, with the same scenario, we show that the resulting placement/scheduling are exactly the same without any relocation feature. This results in a high under-usage of the available resources. With DHM SMT, all the task requests are accepted and thanks to the TD feature, a maximum of 26 parallel operations is reached. However, the systolic ring with 32-PE remains underused compared to the

large number of available resources. This can be explained by the fact that resulting workload is four times lower.

Figure 21 Characterisation with 32 PE

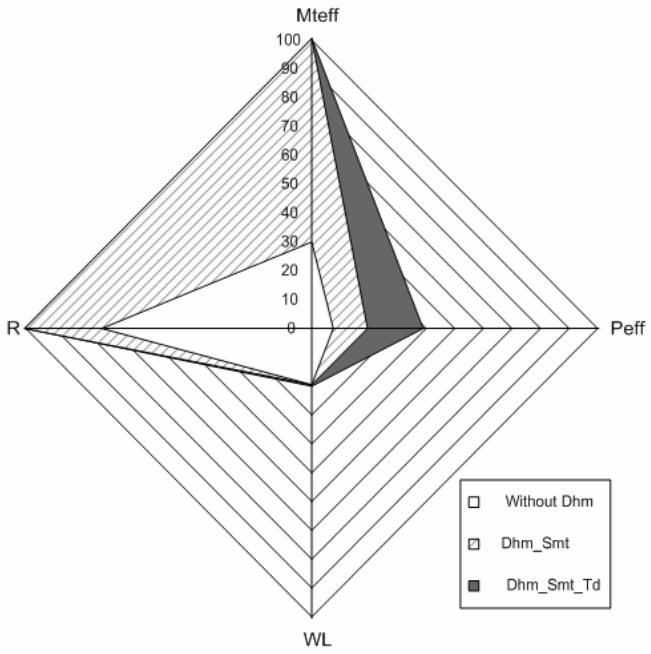
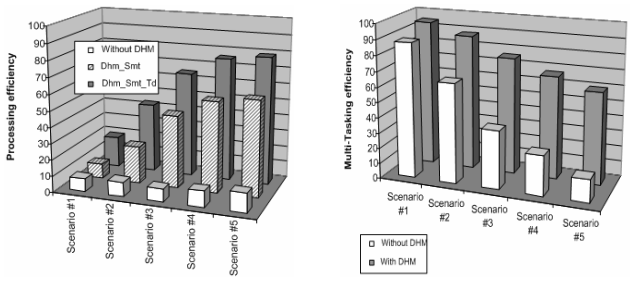


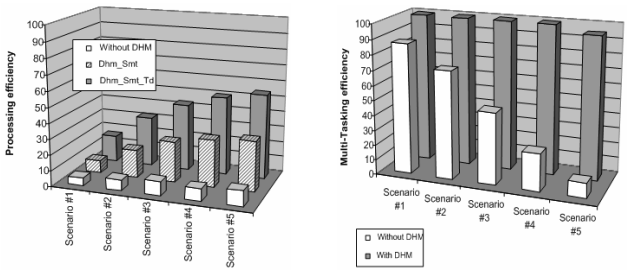
Figure 22 and Figure 23 show the evolution of the MTeff and Peff performances on different system workloads. There are five different workload levels compared in this survey: from very low workloads (scenario #1) to very high workloads (scenario #5). Please, notice that in each case, the WL produced by these scenarios is always four times lower for 32-PE systolic ring. The histograms plotted here highlight the difference between two different instances of the systolic ring (8-PE in Figure 22, and 32-PE in Figure 23). The results presented above were done for a high workload scenario (scenario #4). We observe here the evolution in Peff and MTeff. Basically, Peff increases in both cases but grows quickly in the smallest systolic ring. It reaches a value between 80% to 90% for high and very high workloads, with a typical value around 70%. However, the MTeff decreases to a value around 60% for very high workloads. For the biggest systolic ring, the increase in Peff is slower but it reaches a value of 60% for the very high workloads, and still the task acceptance is higher than 90%.

Figure 22 Peff and MTEff on 8-PE with different task schedules



Note: With a workload from 20 to 100.

Figure 23 Peff and MTEff on 32-PE with different task schedules

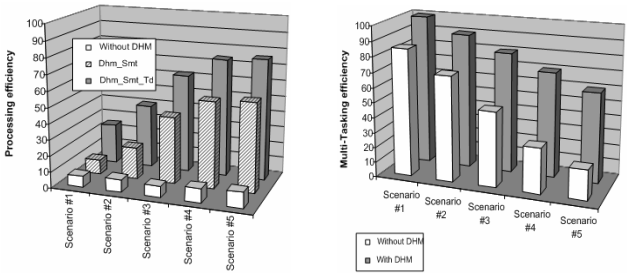


Note: Same scenarios as Figure 22 resulting in a 0.05 to 25 workload.

Figure 20 and Figure 21 characterise both scenarios and architectures thanks to the metrics suggested in Section 3. These results here emphasise the improvements obtained thanks to DHM through SMT and TD features. On the smallest systolic ring, the processing efficiency is clearly increased from 30% to almost 90%. The multi-tasking efficiency is also basically improved as it is more than doubled (from 30% to 70%) and the co-processor is used almost all the time (more than 95% of run-time). In Figure 21, we observe that all the tasks have been accepted (multi tasking efficiency equals 100%) and the systolic ring is used 100% of the run-time. But the processing efficiency is lower than the previous one as the set of tasks used for the simulation has a lower workload (20%).

When the set of tasks simulated is compounded with bigger sized tasks (Figure 24), the obtained results show an improved processing efficiency but the multi-tasking efficiency is lowered down.

Figure 24 Task scheduling on 32-PE



6 HW and SW implementations

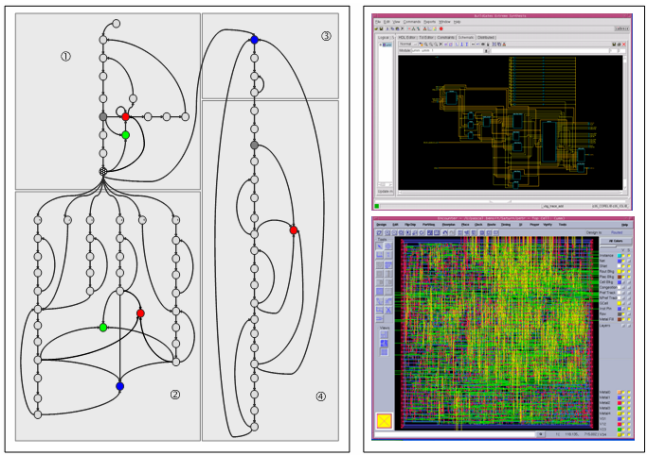
To evaluate the cost and performances of hardware and software implementations, we have designed both of them. In this section, we show the obtained results and we analyse the perspectives of each one.

6.1 HW DHM: Saturn ring system

The *Saturn* controller is a DHM unit implementation for the systolic ring architecture. This unit is a dedicated *configuration processor*. It is mainly composed of a programme counter (PC) that allows the generation of the programme memory addresses. Data coming from this memory are either headers or configuration words. Different logic blocks are used for topology matching analysis

between the current state of the reconfigurable accelerator and the header fields. When a compatibility is found (after a configuration rotation or not), a configuration word is read from the memory and then modified accordingly. Dedicated registers are used to store the state of the systolic ring resources (e.g., Dnodes) at each cycle in order to allocate them if necessary.

Figure 25 Task scheduling on 32 PE (see online version for colours)



The Saturn controller is a finite state machine (FSM) designed to make decisions as quickly as possible for fast dynamic reconfiguration. It is used to schedule the different steps of the DHM management, by generating chip enable (ce) signals to the datapath. The complexity of this FSM is about 52 states in the first version, where TD is not possible. Enabling this implies 20 additional states in the FSM.

Table 3 Performance and overhead of HW DHM

Saturn	*Area (mm²)*	*Min. lat (cycles)*	*Max. lat (cycles)*	*Power cons. (mW/MHz)*
DHM SMT	0.38	6	25	0.10
DHM SMT-TD	0.40	6	25	0.12

Saturn has been designed in behavioural VHDL and then synthesised with Cadence design flow as illustrated in Table 3. The design kit used is AMS 0.35 µ with four metal layers (3.3 V). The maximum accessible frequency in this technology for the synthesised hardware is 30 MHz.

6.2 SW DHM: the plasma processor

We have used a soft processor to implement the DHM algorithm with a software programme. This one is a MIPS-like processor. It is based on a 32 bits CPU, a local memory and an UART. The CPU is based on a three stage pipelined unit, an ALU, a shifter and a multiplier.

The DHM programme is first compiled with a *gcc* cross compiler with the third level of optimisation allowing a gain on the performances around 20%. This code is then loaded into the local memory and executed indefinitely. Two different implementations have been tested: one optimised for memory, the other optimised for speed. The results are summarised in the table below. Several memories were synthesised from the manufacturer website following the size required for the code. The maximum accessible frequency for the synthesised processor is 30 MHz for an area equal to 1.44 mm^2.

Table 4 Performance and overhead of SW DHM

Plasma	*Mem. (KB)*	*Min. lat. (cycles)*	*Max. lat. (cycles)*	*Power cons (mW/MHz)*
DHM1 SMT	23	3,806	65,902	1.25
DHM1 SMT-TD	25	3,806	65,902	1.36
DHM2 SMT	16	1,200	188,359	1.12
DHM2 SMT-TD	18	1,200	188,359	1.16

6.3 Performance comparisons

HW and SW-DHM controllers have been designed both for a systolic ring instance composed with 8 Dnodes. After synthesis, the designs have been validated by simulations. For HW-DHM, the whole algorithm takes less than 25 cycles (0.8µs@30MHz) to dynamically modify and (re-)allocate a pre-defined configuration. The area overhead is about 0.4 mm^2, which represents less than 10% of the systolic ring. For SW-DHM, the 32 bits processor has a silicon area of 1.44 mm^2 (about 36% of the systolic ring area) and implies in the worst case up to 65,902 cycles (2.1mms@100MHz) to perform the same allocation task. These results are listed in Table 5.

Table 5 Comparison of performance and overhead of HW and SW DHM

	HW-DHM scheduler	*SW-DHM scheduler*
Area (mm^2)	0.40	1.44
Time (µs)	0.83	2,186.73
1/(A.T)	3	3.10^{-4}

Not surprisingly, these results clearly show the superiority of the dedicated approach with a global performance [1/AT (DeHon, 1998)] 10,000 times better than the SW-DHM. However, the HW-DHM requires much more design effort, while the SW-DHM can be easily changed by simply reprogramming the processor. Moreover, the software approach is far more scalable as it requires no area overhead for different instances. In terms of latency introduced by the mapping process, the quality of service obtained with hardware DHM is better than the software DHM as it is less than a microsecond, which certifies a very fast reconfiguration. However, for some applications, a maximum latency of 2 ms can be acceptable (i.e., audio applications require less than 8 ms).

Table 6 Our contribution compared to the existing solutions

Name	*Sched.*	*Reloc.*	*Replic.*	*Config. manager*	*Location*	*Ref*
XC6200	Static	No	No	No	-	Shirazi et al. (1998)
XC6200	Static	Yes	No	No	-	Burns et al. (1997)
XC6200	Dyn.	Yes	No	No	-	Robinson and Lysaght (1999)
V2-Pro	Static	No	No	SW PowerPC	FPGA	Curd (2003)
V2	Static	No	No	SW µBlaze	FPGA	Blodget et al. (2003)
SCORE	Static	No	No	HW	Dedicated module	Caspi et al. (2000)
RAW	Static	No	No	Compiler	-	Taylor et al. (2002)
Our approach	Dyn.	Yes	Yes	HW or SW	Dedicated module	

7 Conclusions and future works

In this article, we have proposed and implemented a method for dynamic reconfiguration management. This method is based on a DHM algorithm allowing an abstraction, at design-time, of the dynamic reconfiguration management. The DHM algorithm aims at exploiting more efficiently the processing resources thanks to an adaptive online scheduling technique. A case study on the systolic ring architecture has proven the feasibility of the proposed approach. The simulation results show an improved processing efficiency and dynamic reconfiguration is directly and automatically managed by the hardware. The HW-scheduler offers a clearly better performance/area trade-off, while the SW-DHM approach is attractive from the point of view of its flexibility. The features of our approach compared to related works are depicted in Table 6.

A 'fine-grain' implementation could also be considered but we strongly believe that our method takes its whole sense on coarse grain architectures and proposed DHM is very easy to adapt to any processing element interconnection topology. The main limitation stands actually on the processing element programming. We are therefore exploring a new architectural support for DHM based on a mesh network of RISC processors that we can directly address with a C code.

References

Blodget, B., McMillan, S. and Lysaght, P. (2003) 'A lightweight approach for embedded reconfiguration of FPGAs', in *DATE'03*, pp.399–400, Germany.

Burns, J., Donlin, A., Hogg, J., Singh, S. and Wit, M. (1997) 'A dynamic reconfiguration run-time system', in *FCCM'97*, pp.66–75, USA.

Carvahlo, E., Calazans, N., Briao, E.W. and Moraes, F.G. (2004) PADReH – a framework for the design and implementation of dynamically and partially reconfigurable systems', *17th Symposium on Integrated Circuits and Systems Design – SBCCI 2004*, pp.10–15, ACM Press, New York.

Caspi, E., Chu, M., Huang, R., Yeh, J., Wawrzynek, J. and DeHon, A. (2000) 'Stream computations organized for reconfigurable execution (SCORE)', *FPL 2000*, pp.605–614.

Chen, W.H., Smith, C.H. and Fralick, S. (1977) 'A fast computational algorithm for the discrete cosine transform', *IEEE Trans. Commun.*, September, Vol. COM-25, pp.1004–1009.

Curd, D. (2003) 'Dynamic reconfiguration of RocketIO MGT attributes', *Xilinx Application Note XAPP660, V2.1*, November.

Danne, K. and Platzner, M. (2005) 'A heuristic approach to schedule periodic real-time tasks on reconfigurable hardware', *FPL 2005*, pp.568–573.

DeHon, A. (1998) 'Comparing computing machines', in *Configurable Computing: Technology and Applications*, Vol. 3526, pp.124–133.

Goldstein, S.C., Schmit, H., Budiu, M., Cadambi, S., Moe, M., Reed Taylor, R. (2000) 'PipeRench: a reconfigurable architecture and compiler', *IEEE Computer*, Vol. 33, No. 4, pp.70–77.

Huebner, M., Ullmann, M., Braunn, L., Klausmann, A. and Becker, J. (2004) 'Scalable application-dependent network on chip adaptability for dynamical reconfigurable real-time systems', *FPL'2004, LNCS 3203*, pp.1037–1041.

Park, S.R. and Burleson, W. (1997) 'Reconfiguration for power saving in real-time motion estimation', *ICASSP*.

Robinson, D. and Lysaght, P. (1999) 'Modeling and synthesis of configuration controllers for dynamically reconfigurable logic systems using the DCS CAD framework', in *FPL'99, Lecture Notes in Computer Science*, Vol. 1673, UK.

Sassatelli, G., Torres, L., Benoit, P., Gil, T., Diou, C., Cambon, G. and Galy, J. (2002) 'Highly scalable dynamically reconfigurable systolic ring-architecture for DSP applications', *DATE 2002*, pp.553–558.

Shirazi, N., Luk, W. and Cheung, P. (1998) 'Run-time management of dynamically reconfigurable designs', in *FPL'98, Lecture Notes in Computer Science*, Vol. 1482, Springer-Verlag, Heidelberg, Estonia.

Singh, H., Lee, M-H., Lu, G., Kurdahi, F.J., Bagherzadeh, N., Chaves Filho, E.M. (2000) 'MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications', *IEEE Trans. Computers*, Vol. 49, No. 5, pp.465–481.

Sweldens, W. (1998) 'The lifting scheme: a construction of second generation wavelets', *SIAM Journal on Mathematical Analysis*, Vol. 29, No. 2, pp.511–546.

Taylor, M.B., Kim, J.S., Miller, J., Wentzlaff, D., Ghodrat, F., Greenwald, B., Hoffmann, H., Johnson, P., Lee, J-W., Lee, W., Ma, A., Saraf, A., Seneski, M., Shnidman, N., Strumpen, V., Frank, M., Amarasinghe, S.P. and Agarwal, A.

(2002) 'The raw microprocessor: a computational fabric for software circuits and general-purpose programs', *IEEE Micro*, Vol. 22, No. 2, pp.25–35.

Ullman, M., Hübner, M., Grimm, B. and Becker, J. (2004) 'On-demand FPGA run-time system for dynamical reconfiguration with adaptive priorities', Becker, J., Platzner, M. and Vernald, S. (Eds.): *FPL 2004, LNCS 3203*, pp.454–463, Springer-Verlag.

Websites

Atmel Corporation, available at http://www.atmel.com.

eASIC, available at http://www.easic.com/.

M2000 – Embedded FPGA, available at http://www.m2000.fr/.

Xilinx Corporation, available at http://www.xilinx.com.