Power-Performance Trade-Offs for Reconfigurable Computing

Juanjo Noguera
Research & Development Dept.
InkJet Commercial Division - Hewlett-Packard
jnoguera@bpo.hp.com

Rosa M. Badia
Computer Architecture Dept. (DAC)
Technical University of Catalonia (UPC)
rosab@ac.upc.es

ABSTRACT

In this paper, we explore the system-level power-performance trade-offs available when implementing streaming embedded applications on fine-grained reconfigurable architectures. We show that an efficient hardware-software partitioning algorithm is required when targeting low-power. However, if the application objective is performance, then we propose the use of dynamically reconfigurable architectures. This work presents a configuration-aware data size partitioning approach. We propose a design methodology that adapts the architecture and used algorithms to the application requirements. The methodology has been proven to work on a real research platform based on Xilinx devices. Finally, we have applied our methodology and algorithms to the case study of image sharpening, which is required nowadays in digital cameras and mobile phones.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles – *adaptable architectures, heterogeneous (hybrid) systems.*

General Terms

Algorithms, Performance, Design.

Keywords

HW/SW partitioning, dynamically reconfigurable architectures, task/configuration scheduling, power-performance trade-offs.

1. INTRODUCTION

The continued progress of Reconfigurable Computing (RC) has enabled the Programmable-System-On-Chip to become a reality, combining a wide range of complex functions on a single die. An example is the Virtex-II Pro from Xilinx, which integrates a core processor (PowerPC405), embedded memory and configurable logic [1]. Additionally, the importance of having on-chip programmable logic regions in System-on-Chip platforms is becoming increasingly evident. Partitioning an application among software and programmable logic hardware can substantially improve performance, but such partitioning can also improve power consumption by performing computations more effectively and by allowing for longer microprocessor shutdown periods [11].

Dynamic Reconfiguration has emerged as a particularly attractive technique to increase the effective use of programmable logic blocks. *Dynamic Reconfiguration* allows the change of the device configuration *on the fly* during application execution. However, this attractive idea of time-multiplexing the needed device configuration does not come for free. The *reconfiguration latency* has to be minimized in order to improve application performance. *Temporal partitioning* [8] and *multi-context scheduling* [5] techniques can be used to minimize this penalty.

We could summarize that there are many *system-level approaches* to reconfigurable computing, which could be divided in two broad categories: (1) HW/SW partitioning for statically reconfigurable architectures; and (2) temporal partitioning and task/configuration scheduling for dynamically reconfigurable architectures.

In this paper, we explore the system-level power-performance trade-offs for reconfigurable computing. We show that the use of a particular approach (i.e. HW/SW partitioning for statically reconfigurable or task/configuration scheduling for dynamically reconfigurable architectures) depends on the application requirements (i.e. power or performance).

Moreover, we explain that in the task/configuration scheduling approach, the dynamically reconfigurable architecture should process large blocks of data, which require the use of external off-chip memory resources. The execution of large blocks of data in part of the reconfigurable architecture is required to hide the reconfiguration process, which could be running in a different part of the reconfigurable architecture. Thus, this idea increases the performance because the reconfiguration latency is hidden, but it also increases the power-consumption due to the use of external memory resources. On the other hand, the approach based on HW/SW partitioning for statically reconfigurable architectures, processes small blocks of data that can be stored in on-chip memory resources, which means that we reduce the overall system power-consumption.

The paper is organized as follows: section 2 explains the related work. In section 3, we introduce our target architecture. The proposed design methodology for embedded systems is presented in section 4. Section 5 introduces the concept of configuration-aware data partitioning. In section 6, we explain the benchmarks, the experimental set-up and the obtained results. Finally, the conclusions of this paper are presented in section 7.

CODES+ISSS'04, September 8–10, 2004, Stockholm, Sweden.

2. RELATED WORK

In [4] an integrated algorithm for HW/SW partitioning and scheduling, temporal partitioning and context scheduling is presented. However, the paper does not address any power-performance trade-offs. A technique for application partitioning between configurable logic and an embedded processor is given in [11]. This paper shows that such partitioning helps to improve both performance and energy. However, the paper only considers statically configurable logic, and does not consider run-time reconfigurable architectures. A different approach for coarse-grained reconfigurable computing is presented in [10]. In the paper, a data scheduler algorithm is proposed to reduce the overall system energy. However, the paper does not consider the benefits of HW/SW partitioning.

2.1 Contributions of this paper

This paper explores the system-level power-performance trade-offs for fine-grained reconfigurable computing. More in detail, the paper compares, in terms of energy savings and performance improvements, the two key approaches existing in reconfigurable computing: (1) partitioning an application between software and configurable hardware; and (2) task/configuration scheduling for dynamically reconfigurable architectures. To the best of our knowledge, this open issue has not been addressed in previous research efforts.

3. TARGET ARCHITECTURE

The target architecture is a heterogeneous architecture, which includes an embedded processor, a given number of dynamically reconfigurable processors (DRPs), an on-chip L2 multi-bank memory sub-system and external DRAM memory resources. An example of this architecture is shown in figure 1, where we can see a 4-DRP based architecture. This architecture follows the Chip Multi-Processor (CMP) paradigm. The data that must be transferred between tasks executed in the DRP processors is stored in the on-chip L2 memory sub-system. Each dynamically reconfigurable processor can be independently reconfigured. The proposed target architecture supports *multiple reconfigurations running concurrently*, which is not the case on most of the run-time reconfigurable architectures proposed in the literature.

Each DRP processor has a local L1 memory buffer. A hardware-based data pre-fetching mechanism is proposed to hide the memory latency. Each DRP has a point-to-point link to the L2 buffers (in order to simplify the figure 1, this is not shown in the picture). However, this is shown in figure 2.a., which shows the internal architecture of a DRP processor. There are three main components in this architecture: (1) the load unit; (2) the store

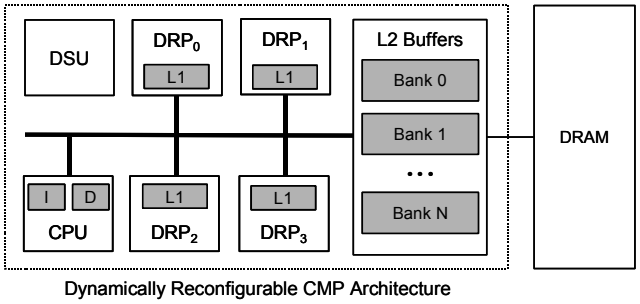


Figure 1: Dynamically Reconfigurable CMP Architecture

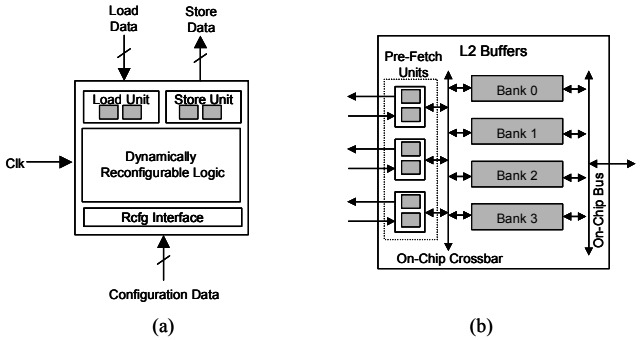


Figure 2: (a) Dynamically Reconfigurable Processor; (b) Architecture of the L2 on-chip memory sub-system

unit; and (3) the dynamically reconfigurable logic. The DRPs are single-context devices. It can be observed in figure 2.a. that the load and store units have internal L1 data buffers. More in detail, each unit (i.e., load and store) has two internal buffers. This approach enables the possibility of having three processes running concurrently: (1) the load unit receiving data for the next computation; (2) the reconfigurable logic is processing data from a buffer in the load unit and storing this processed data in a buffer of the store unit; finally (3) the store unit is sending the previous processed data to the L2 memory sub-system.

The on-chip L2 memory sub-system is based on a multi-bank approach (see figure 2.b). Each one of these banks is logically divided in two independent sub-banks (i.e. this enables reading from one sub-bank while concurrently writing to the other sub-bank of the same physical bank). These buffers interact from one side with the data pre-fetch units (left-hand side of figure 2.b) using a crossbar, and from the other side with an on-chip bus that interacts with the external DRAM memory controller. In this L2 memory sub-system there must be as many data pre-fetch units as number of DRP processors in the CMP architecture.

Finally, the proposed architecture includes for each DRP a dedicated hardware-based configuration pre-fetch unit. This is not shown in the pictures in order to simplify the figures. Thus, the architecture supports the transfer of data in one DRP overlapped with the reconfiguration of a different DRP.

4. DESIGN METHODOLOGY FOR EMBEDDED SYSTEMS

The proposed design methodology for embedded systems is depicted in figure 3. We can observe that it is divided in three steps: (1) application phase; (2) static phase; (3) dynamic phase.

4.1 Application Phase

The proposed methodology assumes that the input application is specified as a task-graph, where nodes represent tasks (i.e. coarse-grained computations) and edges represent data dependences. Each edge has a weight to represent the amount of data that must be transferred between tasks. There are no control dependencies. Finally, each task has associated a task type (i.e. in the task-graph specification we could have several tasks implementing the same type of computation).

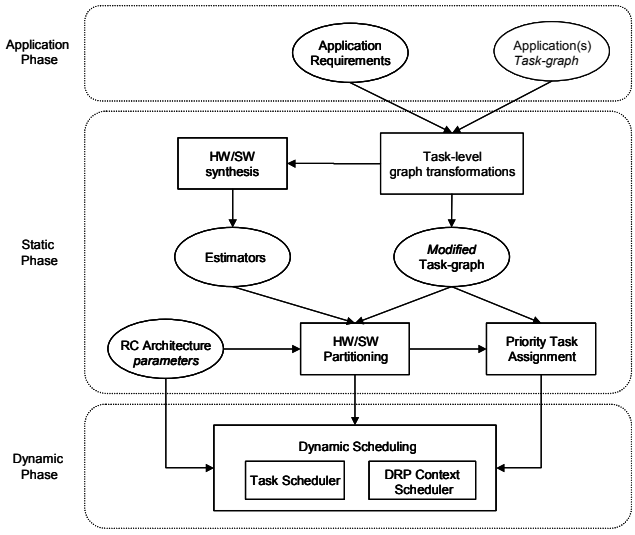


Figure 3: Design methodology for embedded systems

4.2 Static Phase

In this phase there are four main processes: (1) *Task-level graph transformations;* (2) *HW/SW synthesis*; (3) *HW/SW partitioning*; and (4) *Priority task assignment*.

We can apply some *task-level graph transformation techniques* in order to increase the architecture performance. These transformations include: loop pipelining, loop unrolling and/or task (configuration) replication. The output of this step is the modified task graph.

The HW/SW synthesis is the process of implementing the tasks found in the application. The output of this process is a set of *estimators*. Typical estimators are HW execution time, SW execution time, HW area and reconfiguration time. These estimators could be obtained using accurate implementation tools (i.e. compiler, logic synthesis and place&route tools) or using high-level estimation tools.

The HW/SW partitioning process decides which tasks are mapped to hardware or software depending on: (1) the architecture parameters (i.e. number of DRP processors, external DRAM size, etc.); (2) the modified task-graph; and (3) the task's estimators. Note that the application requirements do not affect *directly* the HW/SW partitioning process, but they do affect this process *indirectly* using the modified task-graph. The partitioning algorithm must take into account the configuration pre-fetch technique in its implementation.

Finally, in the static phase we also find the *Priority Task Assignment* process. In this process, we statically assign to each task a priority of execution. This information will be used during run-time to decide the execution order of the tasks. An example of priority function is the critical-path analysis.

4.3 Dynamic Phase

This phase is responsible for the scheduling of the tasks but also for the scheduling of the DRP's reconfigurations. The *Task Scheduler* and *DRP Context Scheduler* co-operate and run in parallel during application run-time execution. Their functionality is based on the use of a look-ahead strategy into the list of tasks ready for execution (i.e tasks which predecessors have *finished* its execution). At run-time, the task scheduler assigns tasks to DRP's and decides the execution order of the tasks found in the list of ready for execution. The DRP context (configuration) scheduler is used to minimize reconfiguration overhead. The objective of the DRP context scheduler is to decide: (1) which DRP processor must be reconfigured, and (2) which reconfiguration context, or hardware task from the list of tasks ready for reconfiguration (i.e tasks which predecessors have *initiated* its execution), must be loaded in the DRP processor. This scheduler tries to minimize this reconfiguration overhead by overlapping the execution of tasks with DRP reconfigurations. These algorithms are implemented in *hardware* using the Dynamic Scheduling Unit (DSU) found in our architecture (see figure 1) [7]. Several research efforts in the field of *SoC* design propose moving into hardware functionality that traditionally has been assigned to operating systems [9].

5. RECONFIGURATION-AWARE DATA PARTITIONING

5.1 Motivation

It has been demonstrated that the parameters of the reconfigurable architecture (i.e. number of DRP processors; reconfiguration time) have a direct impact into the performance given by the HW/SW partitioning process [6]. The partitioning process must take into account the reconfiguration overhead, and also the configuration pre-fetching technique for reconfiguration latency minimization. This is summarized in the next expression, which shows how the execution time of a task mapped to hardware can be modified.

$$\overline{ET_i^{HW}} = ET_i^{HW} + \alpha_R \cdot \left(T_R - \overline{T_{EXE}}\right) \qquad \textbf{(1)}$$

where:

- α_R is the probability of reconfiguration, which is a function of the number of tasks mapped to hardware and the number of DRP processors.
- T_R is the reconfiguration time needed for a DRP processor to change its context (configuration).
- $\overline{T_{EXE}}$ is the average executing time for all tasks.

On the other hand, in the design of embedded systems, we would like to minimize the amount of accesses to external memory. Reducing the number of data transfer to external memory resources helps to reduce the overall system-level power consumption. Thus, data transfers between tasks should be kept to a size that fits into the on-chip L2 memory sub-system.

In streaming embedded applications, we could assume that, in general, the execution time of a task implemented in hardware or software is proportional to the size of the data that must be processed. Thus, if the data is stored in on-chip memory resources with a smaller capacity, then we could conclude that the average execution time (T_{EXE}) of the tasks will be smaller when compared to the reconfiguration time (we are assuming reconfiguration times in the order of 800us-1.4ms). If this is the case, and applying expression (1) we will have a significant reconfiguration overhead (because $T_R >> T_{EXE}$), which may prevent moving the task from software to hardware. In order to overcome this limitation and reduce the reconfiguration overhead, we could increase the amount of data to be processed by the task. Increasing

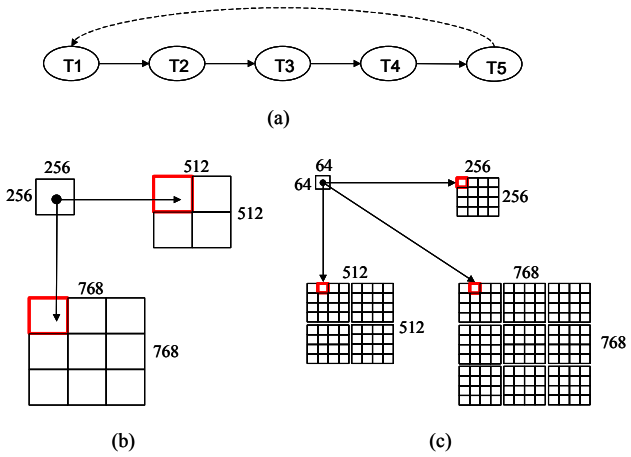


Figure 4: (a) Initial task graph; (b) data partitioning for dynamically reconfigurable architectures; (c) data partitioning for HW/SW partitioning

the amount of data means that we will be forced to use external memory resources. Using this approach, we increase the performance (because more tasks could be mapped to hardware) but we also increase the overall system-level power consumption.

5.2 Data partitioning for reconfigurable architectures

As previously stated, the data partitioning will mainly drive the use of a given approach: (1) hardware/software partitioning for statically reconfigurable architectures using on-chip memory resources; or (2) task/configuration scheduling for dynamically reconfigurable architectures using off-chip memory resources.

Thus, the type of used data partitioning approach (off-chip or on-chip memory) will be decided based on the application requirements. Also, the type of data partitioning will give us the number of iterations of the task graph. This is shown in figure 4, where we can observe an example of a linear task graph. In figure 4.b. we can see a possible data partitioning. We can observe that the size of the blocks of data is large. The volume of the processed data must be such that, at least, the task execution time equals the reconfiguration time. However, the number of iterations of the task graph will be small. In the opposite case, we have the situation where we process small blocks of data, but we have a large number of iterations of the task graph (see figure 4.c).

6. EXPERIMENTS AND RESULTS

6.1 Image Sharpening Benchmarks

The proposed dynamically reconfigurable architecture is addressing streaming data (computation intensive) embedded applications. That is, applications with a large amount of data-level parallelism. It is not the goal of the proposed architecture to address control-dominated applications.

Image processing applications are a good example of the type of applications that we are addressing. These kind of applications are becoming more and more sensible to power consumption, specially if we consider the increasing market-share of digital cameras or mobile phones with embedded cameras, which require of this type of image processing techniques. In this sense, we have selected three applications that implement an image sharpening application (see figure 5).

The three benchmarks follow the same basic process: (1) transform the input image from RGB to YCrCb color space; (2) image quality improvements processing the luminance (mainly using sliding window operations like 3x3 linear convolutions); and finally (3) transform from YCrCb back to RGB color space.

Three different input data-sets (image size) have been used in the three applications: (1) 256x256; (2) 512x512; and (3) 768x768.

6.2 Prototype Implementation

A prototype of the proposed architecture has been designed and implemented. The Galapagos system is a PCI-based system (64bit/66MHz). It is based on leading-edge FPGA's from Xilinx and high-bandwidth DDR SDRAM memory (left-hand side of the figure 6). This reconfigurable system is based on a Virtex-II Pro device. The device used is a XC2VP20, which includes two PowerPC processors. The dynamically scheduling unit (DSU, in figure 1) and the data pre-fetch units of the L2 memory subsystem (see figure 2) have been mapped to the Virtex-II pro device. This device also includes the SDRAM memory controller. The design of these blocks has been done in verilog HDL, and the implementation has been done using Synplicity (synthesis) and Xilinx (place&route) tools.

The DRP processors of our CMP architecture are implemented in the Galapagos system using three Virtex-II devices (i.e. XC2V1000). The load and store units have been implemented using Virtex-II on-chip memory resources. The size of the buffers in the load/store units is 2KB each buffer (i.e. 4KB for each unit). The width of the memory words is 64bits. Figure 6 shows a picture of the Galapagos system in a PC environment.

6.3 Tasks Performance Results

Figure 7 shows the execution time of the unsharp masking application running on: (1) an embedded processor, PowerPC405 (300MHz), which processes blocks of data of 64x64pixels; (2) a DRP processor from the Galapagos System (60MHz) processing blocks of 64x64 pixels; and (3) a DRP processor processing blocks of data of 256x256 pixels. See figure 4.

It is interesting to note the order of magnitude that has been obtained in the implementation of the *blur* task (3x3 linear convolution). It is not the objective of this paper to explain the

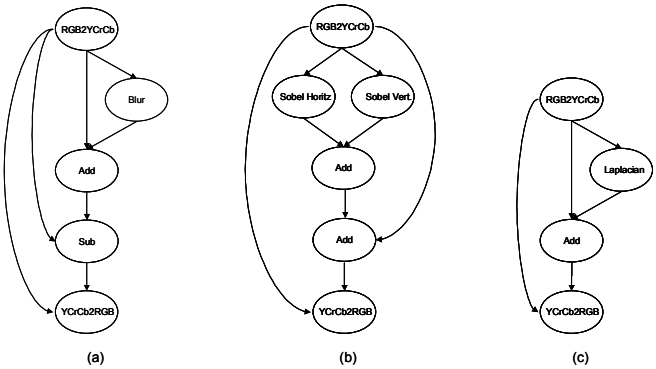


Figure 5: Image sharpening benchmarks; (a) unsharp masking; (b) sobel filter; (c) laplacian filter

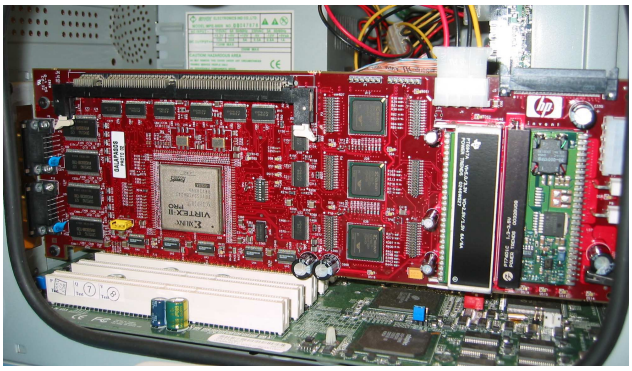


Figure 6: The prototyping platform – the Galapagos system

details of the implementation of the several tasks in hardware. These tasks have been designed in verilog HDL, simulated using Modelsim, and implemented using Synplicity (synthesis) and Xilinx (place&route) tools.

In order to reduce the reconfiguration overhead, we have used the *partial reconfiguration* capability of the Virtex-II devices. In this sense, the Virtex-II resources used by the hardware tasks, have been fixed to be in the center of the device, where we time-multiplex the required task (see figure 8). The left side and right side of the device are used by the load and store units of the DRP (see figure 2.a). The load and store units are not reconfigured.

Using this capability of the Virtex-II devices, we have reduced the reconfiguration time of a Virtex-II device XC2V1000 from *8ms (full/complete device reconfiguration) to 1.4ms (partial device reconfiguration)*, using a reconfiguration clock of 66MHz.

6.4 Tasks Power Results

The power consumption for the implementation of the tasks is shown in figure 9. Figure 9.a. shows the power consumption for a hardware implementation (with on-chip and off-chip memory), while figure 9.b. shows the power consumption for a software implementation (embedded PowerPC405 processor core).

The power consumption for a XC2V1000 device during configuration is 1300mW. There are three components to this power consumption: (1) 200mW used by the device itself [2]; (2) the power consumption of the configuration pre-fetch unit in the Virtex-II Pro (100mW); and (3) the power consumption of the external memory which stores the device configurations (1000mW)[3]. The power consumption of the device in the execution of a task has also two components: (1) the power consumption of the data pre-fetch units implemented in the Virtex-II Pro (150mW); and (2) the hardware task power consumption, which is in average 450mW (i.e. power of a hardware task running at 60MHz). This average result has been obtained implementing a gate-level accurate simulation after the

	RGB2YCrCb	Blur	Add/Sub	YCrCb2RGB
PowerPC 300MHz - 64x64	852us	1860us	564us	1266us
Galapagos DRP 60MHz - 64x64	205us	69us	137us	207us
Galapagos DRP 60MHz - 256x256	3276us	1092us	2184us	3276us

Figure 7: Hardware/Software task execution time

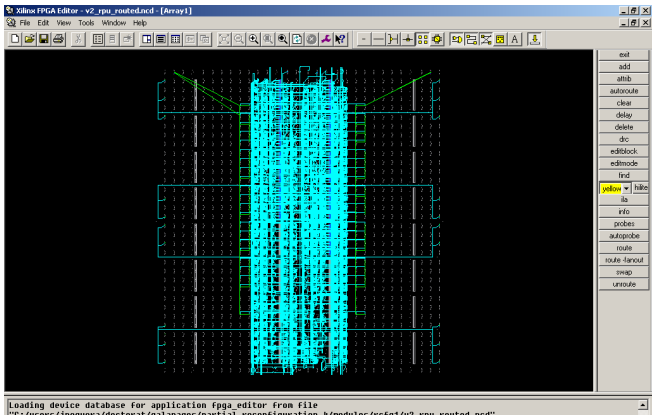


Figure 8: Final place&route on the virtex-II device

place&route process for all the tasks. We should add 1000mW when using external memory resources. Finally, the power consumption when the device is in the idle state is 150mW [2]. This idle state represents the case when the device is powered but a clock-gating mechanism is applied to its input clock signal (remember that the clock signal of each DRP can be controlled independently; see figure 2.a). This power consumption in the idle state is required to store the configuration context in the device (i.e. SRAM cells that store the device configuration bits). The CPU power consumption is shown in figure 9.b., which in execution includes the power of the processor core [1] and the used on-chip memory resources (150mW).

6.5 Energy-Performance Trade-Offs Results

In this section, we explain the energy-performance trade-offs results obtained in the image sharpening benchmarks.

More in detail, figure 10.a. shows the performance results, and figure 10.b shows the energy consumption results. In each picture we can observe the obtained results for the different used data-sets. Also, for each data-set we present three results corresponding to the following implementations:

(1) Dynamic reconfiguration (*seq_rtr*); In this case, we have to use external DRAM, in order to process an amount of data that, at least equals the reconfiguration time. That is, we should process "relatively" large blocks of data. Because the blocks of data are large, then we have to iterate the task-graph a small number of times. Using this approach, all the tasks in the applications are mapped to hardware.

(2) HW/SW partitioning (*seq_hw_sw*); In this case, we use on-chip memory because we process small blocks of data. As the blocks of data are small, we have to iterate the task-graph a large number of times. Using this approach, we map to hardware the most three critical tasks.

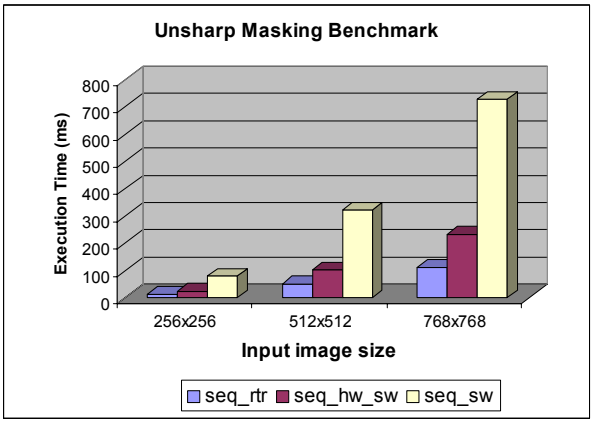
	DRP+On-Chip memory	DRP+Off-Chip memory
Power Reconfiguration	Not apply	1300mW
Power Execution	600mW	1600mW
Power Idle/Wait	150mW	150mW

(a)

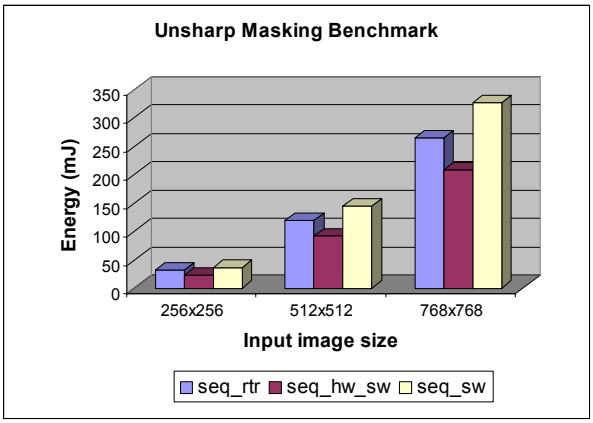
	Embedded CPU (PowerPC405)
Power Execution	450mW
Power Idle/Wait	50mW

(b)

Figure 9: Hardware/Software task power-consumption



(a)



(b)

Figure 10: (a) performance (execution time) results; (b) energy consumption results

(3) Software implementation (*seq_sw*); same as in (2).

The performance results have been obtained from real executions on the Galapagos system. The execution generates a *log file* with the state changes of the three Virtex-II devices (i.e. DRP's) and embedded CPU (i.e. PowerPC). We have obtained the energy from: (1) the power consumption of the components as described in figure 9; and (2) the execution log file (which gives information about the time that a device changes its state).

In figure 10.a., we can observe that the software implementation obtains the worst performance results. The use of the HW/SW partitioning algorithm contributes to a major improvement in performance, since critical tasks are mapped to the configurable hardware. Moreover, we can see that the dynamic reconfiguration technique helps to improve performance even more.

Figure 10.b. shows that HW/SW partitioning obtains better energy savings when compared to the software solution. We can also see that dynamic reconfiguration increases the energy consumption when compared to the HW/SW partitioning despite of its improvements in performance. Finally, in both figures we can observe that performance and energy are proportional to the size of processed data (input data-set).

The results obtained show that the dynamic reconfiguration technique obtains an average performance improvement of 107% (execution time reduction) when compared to the HW/SW partitioning approach. However this benefits in terms of performance also come with an average increase of 27.16% in energy consumption for the image sharpening benchmarks.

7. CONCLUSIONS

In this paper we have explored the system-level power-performance trade-offs for fine-grained reconfigurable computing. We have shown that the use of a given approach (i.e. HW/SW partitioning for statically reconfigurable or task/configuration scheduling for dynamically reconfigurable architectures) depends on the application requirements (i.e. power or performance). Thus, in streaming applications when the objective is energy-efficiency then HW/SW partitioning for statically reconfigurable logic is the most favorable solution. On the other hand, if the application objective is performance then task/configuration scheduling for dynamically reconfigurable architectures is the optimum solution.

ACKNOWLEDGMENTS

Juanjo Noguera acknowledges the support of the HP-IPG Resident Fellowship program.

This work is funded by CICYT-TIC project TIC2001-2476-CO3-02 and DURSI project 2001SGR00226.

REFERENCES

[1] http://www.xilinx.com/virtex2pro

[2] http://www.xilinx.com/virtex

[3] http://www.micron.com

[4] K. Chatta, R. Vemuri, "Hardware-Software Codesign for Dynamically Reconfigurable Architectures". Proc. FPL'99.

[5] R. Maestre *et al.*, "A Framework for Reconfigurable Computing: Task Scheduling and Context Management", IEEE Trans. on VLSI Systems.Vol. 9, No. 6, Dec. 2001.

[6] J. Noguera, R. M. Badia, "HW/SW Codesign Techniques for Dynamically Reconfigurable Architectures", IEEE Trans. on VLSI Systems. Vol. 10. Issue 4. August 2002.

[7] J. Noguera, R. M. Badia, "Multitasking on Reconfigurable Architectures: Micro-architecture Support and Dynamic Scheduling". ACM TECS. May 2004.

[8] K. Purna, D. Bhatia, "Temporal Partitioning and Scheduling Data Flow Graphs for Re-configurable Computers", IEEE Trans. on Computers, vol. 48, No. 6. June 1999.

[9] B. E. Saglam (Akgul) and V. Mooney, "System-on-a-Chip Processor Synchronization Support in Hardware," Proc. of DATE'01, pp. 633-639, March 2001.

[10] M. Sánchez-Élez *et al.*, "A Complete Data Scheduler for Multi-Context Reconfigurable Architectures", Proc. DATE'02, Paris, France, 2002.

[11] G. Stitt, F. Vahid, S. Nemetebaksh; "Energy Savings and Speedups from Partitioning Critical Software Loops to Hardware in Embedded Systems". ACM TECS. Jan.2004