Exploiting Partial Run-Time Reconfiguration for High-Performance Reconfigurable Computing

ESAM EL-ARABY, IVAN GONZALEZ, AND TAREK EL-GHAZAWI
NSF Center for High-Performance Reconfigurable Computing (CHREC),
ECE Department, The George Washington University
801 22nd Street NW, Washington, DC 20052, USA
{esam, ivangm, tarek}@gwu.edu

Run-Time Reconfiguration (RTR) has been traditionally utilized as a means for exploiting the flexibility of High-Performance Reconfigurable Computers (HPRCs). However, the RTR feature comes with the cost of high configuration overhead which might negatively impact the overall performance. Currently, modern FPGAs have more advanced mechanisms for reducing the configuration overheads, particularly Partial Run-Time Reconfiguration (PRTR). It has been perceived that PRTR on HPRC systems can be the trend for improving the performance. In this work, we will investigate the potential of PRTR on HPRC by formally analyzing the execution model and experimentally verifying our analytical findings by enabling PRTR for the first time, to the best of our knowledge, on one of the current HPRC systems, Cray XD1. Our approach is general and can be applied to any of the available HPRC systems. The paper will conclude with recommendations and conditions, based on our conceptual and experimental work, for the optimal utilization of PRTR as well as possible future usage in HPRC.

Categories and Subject Descriptors: C.1.3 [**Processor Architecture**]: Other Architecture Styles - *Adaptable architectures, Heterogeneous (hybrid) systems*.
General Terms: Design, Experimentation, Measurement, Performance
Additional Key Words and Phrases: High Performance Computing, Field Programmable Gate Arrays (FPGA), Reconfigurable Computing, Dynamic Partial Reconfiguration

---

1. INTRODUCTION

Reconfigurable Computers (RCs) have recently evolved from accelerator boards to stand-alone general purpose RCs and parallel reconfigurable supercomputers called High Performance Reconfigurable Computers (HPRCs). Examples of such supercomputers are SRC-7 and SRC-6 [SRC 2006], SGI Altix/RASC [Silicon Graphics 2007], and Cray $XT5_h$ and Cray XD1 [Cray 2006]. In these systems, FPGAs are used to implement coprocessors to accelerate in hardware the critical functions causing the poor performance of the general purpose processors, following HW/SW codesign approaches. Several efforts have proved the significant speedup obtained by these systems for many

---

This research was supported by the NSF Center for High-Performance Reconfigurable Computing (CHREC).
Esam El-Araby, Ivan Gonzalez, and Tarek El-Ghazawi; ECE Department, The George Washington University, 801 22nd Street NW, Washington, DC 20052, USA.

http://doi.acm.org/10.1145/1462586.1462590

different applications [Aggarwal et al. 2006; Buell et al. 2004; Buell and Sandhu 2003; Court and Herbordt 2007; El-Araby et al. 2004; El-Araby et al. 2005; Harkins et al. 2005; Kindratenko and Pointer 2006; Michalski et al. 2003; Storaasli 2002] .

However, one limitation of reconfigurable computing is that some large applications require more hardware resources than are available, and the complete design cannot fit in a single FPGA chip. One solution to this problem is (Full) Run-Time Reconfiguration (RTR). RTR, or FRTR as we will call it in our discussions, is an approach that divides applications into a number of modules with each module implemented as a separate circuit. These modules are dynamically uploaded onto the reconfigurable hardware as they become needed. Recent generations of FPGAs support Partial Run-Time Reconfiguration (PRTR) where application modules can be dynamically uploaded and deleted from the FPGA chip without affecting other running modules. In other words, in the FRTR approach the FPGA is fully configured while in the PRTR only parts of the FPGA are configured / reconfigured. The reconfiguration latency (time) introduces a significant overhead for FRTR. This is because most existing FPGAs use relatively slow interfaces for device configuration. Reconfiguration latency is a challenge in reconfigurable computing as it can offset the performance improvement achieved by hardware acceleration when dynamic FRTR is considered [El-Ghazawi et al. 2008]. For example, applications on some systems spend a considerable amount of their execution time performing reconfiguration [Bondalapati and Prasanna 1999; Buell et al. 2007; El-Ghazawi et al. 2008; Gokhale et al. 2006; Tripp et al. 2005].

As configuration time could be significant, eliminating or reducing this overhead becomes a very critical issue for reconfigurable systems. There have been significant efforts directed to address this problem within the domain of embedded systems by proposing / utilizing either FRTR or PRTR [Hasan et al. 2007; Hymel et al. 2007; Hübner and Becker 2006; Jeong et al. 1999; Ullmann et al. 2004]. On the other hand, many solutions based on hardware caching techniques, virtual memory models, and configuration pre-fetching algorithms have been proposed to utilize PRTR [Li et al. 2000; Li and Hauck 2002; Taher 2005; Taher et al. 2005] for HPRCs. Nevertheless, those proposals were based on simulation experiments with assumptions about PRTR that are far in the future beyond the current status of the technology.

In this work, we investigate the performance potential of PRTR on HPRCs from a practical perspective. We provide a formal analysis of the execution model supported by experimental work. Our work enables PRTR on HPRCs for the first time, to the best of our knowledge, by utilizing one of the current HPRC systems, Cray XD1. Our approach

is general and can be applied to any of the available HPRC systems. We also discuss our theoretical and experimental results highlighting the performance bounds of PRTR on HPRCs augmented with suggestions for possible future directions.

This paper is organized such that section 2 provides a brief discussion of run-time reconfiguration and the concept of hardware virtualization as well as the current status of partial reconfiguration. Section 3 describes our analytical model and explains the formulation steps of this model. Section 4 shows the experimental work and presents the implementation of a partially reconfigurable architecture in Cray XD1. The experimental results for a set of hardware functions are shown in section 4. Section 5 provides a discussion of results and future directions. Finally, section 6 summarizes the conclusions.

2. RUN-TIME RECONFIGURATION

In most HPRC systems, FPGA devices are used as malleable coprocessors where components of the application can be implemented as hardware functions and be configured as needed. However, although the capacity of current FPGAs has grown significantly, a second look at hardware acceleration shows that this technique, at least in its conventional way, is not suitable to improve the performance of applications when the number of functions to be executed in hardware exceeds the chip area. The same problem happens when the number of simultaneously running applications in a given workload requiring hardware acceleration is increased.

2.1 Hardware Virtualization

Most of the proposed solutions in many previous research work [Li et al. 2000; Li and Hauck 2002; Taher 2005; Taher et al. 2005] is to reproduce the same strategies adopted in Operating Systems to support virtual memory such as dynamic loading, partitioning, overlaying, segmentation, and paging, etc. The basic idea behind these techniques is to virtually enlarge the size of the FPGA from the point of view of the applications. Therefore, the concept of “virtual hardware” is an effective and efficient technique to increase the availability of hardware resources, implement larger circuits or reduce the costs by adopting smaller FPGA when the performance can still be satisfied. The possibility to apply this concept requires using special capabilities of the FPGAs namely Full Run-Time Reconfiguration (FRTR) and/or Partial Run-Time Reconfiguration (PRTR). For example, PRTR has been proposed [Taher 2005] for multitasking and for cases of single applications that can change the course of processing in a non-deterministic fashion based on data. In this model, hardware functions are grouped into

hardware reconfiguration blocks (pages) of fixed size, where multiple pages can be configured simultaneously. By grouping only related functions that are typically requested together, processing spatial locality can be exploited. However, all these proposed techniques assume that the applications and related hardware functions are known previously and FRTR and/or PRTR are well supported on the system. Currently, this is true for FRTR while it is not the case for PRTR. Also, they do not take into consideration the architectural limitation of using partial reconfiguration on current HPRCs. To the end user, HPRC systems when compared to embedded systems are "closed black box" systems. Users do not have the possibility to modify the system nor have access to the FPGA configuration ports. They can only use the API functions provided by the vendor. With this regard, most of previous work is based on simulations rather than investigating such practical issues.

2.2 Partial Reconfiguration

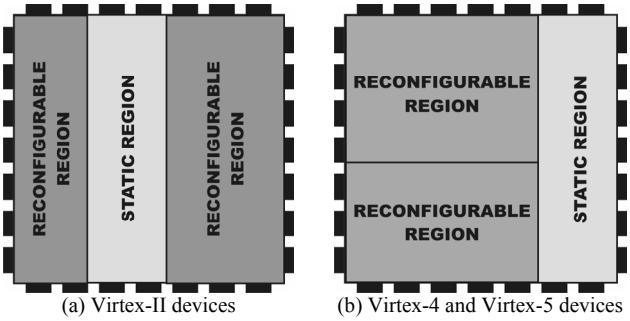
Hardware, like software, can be designed modularly, by creating subcomponents which can then be instantiated by higher-level components. In many cases it is useful to be able to swap out one or several of these subcomponents while the FPGA is still operating. Normally, reconfiguring an FPGA requires it to be held in a reset state while an external controller reloads a design onto it. Partial reconfiguration allows for critical parts of the design to continue operating while a controller, which can be inside or outside the FPGA, loads a partial design into a reconfigurable module.

Partial reconfiguration is supported by different FPGA vendors like Atmel and Xilinx. Xilinx FPGAs are the most popular partial reconfigurable devices among the PRTR community. Starting from the Virtex family, all Xilinx FPGAs can be partially reconfigured at run-time, that is, part of the chip configuration can be changed while the remaining parts continue their normal operation. The minimal unit that can be reconfigured is a frame, which is the smallest addressable segment of the configuration memory space. However, it is possible to change just one bit of the FPGA configuration, as long as the remaining bits of the frame enclosing it are unchanged. If some bits of the new frame do not change with respect to the existing configuration, it is guaranteed that there will be no glitches on these bits during the reconfiguration.

From the functionality of the design, partial reconfiguration can be divided into two groups, i.e. dynamic partial reconfiguration and static partial reconfiguration. Dynamic partial reconfiguration, also known as an active partial reconfiguration, permits changing a part of the device while the rest of an FPGA is still running. In static partial

reconfiguration the device is not active during the reconfiguration process. In other words, while the partial data is sent into the FPGA, the rest of the device is stopped (in the shutdown mode) and brought up after the configuration is completed. Additionally, there are two styles of partial reconfiguration of FPGA devices from Xilinx, i.e. module-based and difference-based. In our experiments we followed the module-based style. Module-based partial reconfiguration allowed us to reconfigure distinct modular parts of the design [Xilinx 2004].

Partial reconfiguration has to be supported by the design automation tools. They should allow the modification of some blocks of the design while maintaining the rest unchanged, and they should also ensure that the placement and routing of the block being modified does not overlap with other modules. Xilinx's solution to this problem is Early Access Partial Reconfiguration flow [Xilinx 2006] which is based on the Modular Design flow [Xilinx 2004]. In current versions of this software, Xilinx supports partial reconfiguration on Virtex II, Virtex II Pro, Virtex 4, and Virtex 5 FPGA lines. Modular Design flow permits building the final FPGA layout from separated modules, each located in a rectangular section of the device. First each module is implemented (mapped, placed and routed) separately, and then in a final assembly phase they are merged to construct the definitive layout. For example, in the layout shown in Figure 1 there are three regions used as configuration space for different application modules. One is a static region and the other two regions are dynamically reconfigurable regions typically called Partially Reconfigured Regions (PRRs). To change the hardware function of one of the regions using partial reconfiguration, the selected module for a given region is re-implemented as a new design and then merged with other modules, previously created, for the static region and all remaining PRRs. As a result, only the PRR dedicated to the new module changes in the new layout, because the static region and other PRRs remain unmodified.



(a) Virtex-II devices (b) Virtex-4 and Virtex-5 devices

Fig. 1. Examples of partial reconfiguration arrangements

Early Access Partial Reconfiguration ensures that both the placement and routing for a module will be confined to a rectangular area of the FPGA [Xilinx 2006]. However, a problem arises when trying to interconnect two regions, since the tool does not allow making external connections to other regions. The solution is to use a component just for interconnection purposes, which does not belong to any of the regions being connected. This component which is called bus-macro ensures the communication across the reconfigurable region boundaries and serves as a fixed routing bridge that connects the reconfigurable region with the remaining parts of the design. Xilinx implements the bus macro [Xilinx 2006] using pairs of look-up tables (LUTs): One LUT will be located in the area reserved for the first region, and the other in the space for the second region. Depending on the type of the selected bus macro, i.e. either "right2left" or "left2right", the communication goes from one region to the second or vice versa. This component is implemented as a hard macro to avoid the routes going through region boundaries changing when re-implementing the partially reconfigurable region. Bus macros were useful for our experimental work. They enabled us to establish communication links between neighbor PRRs.

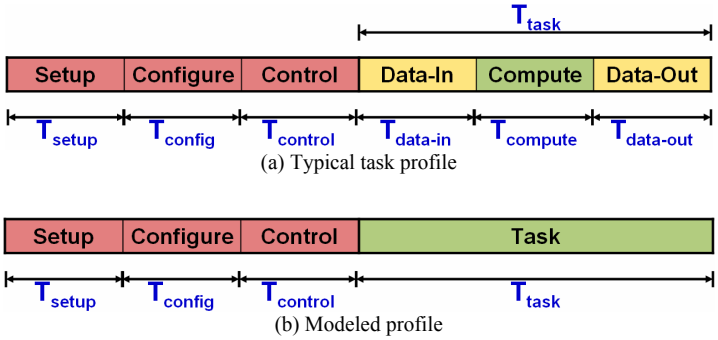
3. EXECUTION MODEL FORMULATION

In order to investigate the performance potential of PRTR on HPRCs and before conducting our experimental work, we derive a formal analysis of the execution model. This analysis would provide us with theoretical expectations which would serve as a frame of reference against which we can project our experimental results. In addition, it helps us gain in-depth insight about the boundaries and/or conditions for performance gain using PRTR. In achieving this objective, our approach is based on leveraging previous work and concepts that were introduced for solving similar and related problems. For example, we include in our analytical model the concept of configuration caching as proposed in [Li et al. 2000; Li and Hauck 2002; Taher 2005; Taher et al. 2005]. In addition, we follow an approach in the derivation of the model similar to what has been proposed in [El-Araby 2005; El-Araby et al. 2006; Hadley and Hutchings 1995; Smith 2002; Smith and Peterson 2002; Taher et al. 2005].

3.1 Analysis

In our analysis we assume that the system receives some applications as input, these applications are all designed around a common hardware library. Each application requires on the average a few hardware functions (tasks) that need to be executed on the

reconfigurable system. The execution cycle for any task, i.e. function call, on an HPRC consists of the computation time, the total I/O time and an overhead time [El-Araby 2005; El-Araby et al. 2006; Taher et al. 2005]. The I/O time is the time necessary to transfer data between the microprocessor and the FPGA. The overhead time consists of setup time, configuration time, and transfer of control time [El-Araby 2005; El-Araby et al. 2006; Taher et al. 2005] as shown in Figure 2(a). The transfer of control time is the time necessary to start a configured task. The setup time is the time spent for pre-fetching related tasks for configuration. In other words, the setup time is the time taken by the configuration caching algorithm to decide whether to configure or not to configure certain tasks which can equivalently be considered as the decision latency. Tasks need to be configured only if they do not exist on the FPGA when needed. This, of course, is based on the assumption that a pre-fetching algorithm as proposed in [Li et al. 2000; Li and Hauck 2002; Taher 2005; Taher et al. 2005] is being utilized. It is also assumed that pre-fetching and/or caching hardware tasks can be performed when the FPGA is divided into at least two PRRs. The baseline for our analysis is FRTR. In other words, we will consider PRTR with respect to FRTR to investigate the relative performance gain to that baseline. This will focus our discussions on applications that are broken down into hardware tasks only. Software tasks are excluded from our analysis because, we think, that would add unnecessary complications to model the partitioning schemes as well as the profiles of scheduling among software and hardware tasks. In addition, we assume that each task is fully characterized by its time requirement, T_{task}, as shown in Figure 2(b). The I/O and computations of each task can be overlapped to further enhance the overall execution time as proposed in [El-Araby 2005; El-Araby et al. 2006]. However, the distribution of the time requirement for each task among data transfer and computations is not included in our model because it can be equivalently represented and masked out, for simplification, by the overall time requirement, T_{task}.



(a) Typical task profile

(b) Modeled profile

Fig. 2. Task profile on an HPRC

The configuration pre-fetching (caching) algorithms as proposed in [Taher 2005; Taher et al. 2005] can be characterized by two parameters:

- The decision latency (time) which is the setup time needed by the algorithm to make the configuration decision (i.e. to configure or not to configure)
- The hit ratio of the caching algorithm which represents the percentage of the tasks that have been successfully pre-fetched to the FPGA and need not be reconfigured when needed

The following notation will be used in our mathematical model:

- n_{calls} is the total number of function (task) calls
- n_{config} is the number of (re-)configurations performed
- $T_{setup} = T_{decision}$ is the average setup time which equals the pre-fetching latency
- $T_{control}$ is the average transfer of control time
- T_{task} is the average task execution time requirement
- $T_{config} = T_{FRTR}$ is the full configuration time for FRTR
- T_{PRTR} is the average partial configuration time for PRTR
- H is the hit ratio of the caching algorithm
- M is the miss ratio of the caching algorithm ($M = 1-H$)
- T^{FRTR}_{total} is the total execution time of FRTR
- T^{PRTR}_{total} is the total execution time of PRTR
- S is the speedup or performance gain of using PRTR relative to FRTR

The total execution time for the case of FRTR, as shown in Figure 3, can be derived as follows:

$$T^{FRTR}_{total} = \sum_{i=1}^{n_{calls}} \left(T_{config_i} + T_{control_i} + T_{task_i} \right), \quad where \quad T_{task_i} = T_{data-in_i} + T_{compute_i} + T_{data-out_i}$$

$$\Rightarrow T^{FRTR}_{total} = n_{calls} \left(T_{FRTR} + T_{control} + T_{task} \right) \tag{1}$$

It is worth to mention that $T_{decision}$ is not included in the derivation of the total execution time for FRTR. This is because configuration pre-fetching is only needed in the case of PRTR. When we normalize the variables with respect to the full configuration time, T_{FRTR}, equation (1) can be rewritten as:

$$X^{FRTR}_{total} = n_{calls}(1 + X_{control} + X_{task})$$

$$where \; X^{FRTR}_{total} = \frac{T^{FRTR}_{total}}{T_{FRTR}}, \; X_{control} = \frac{T_{control}}{T_{FRTR}}, \; and \; X_{task} = \frac{T_{task}}{T_{FRTR}} \tag{2}$$

Figure 4 shows the execution profiles of tasks using PRTR. In this scenario, tasks can be categorized as either missed tasks, see Figure 4(a), or pre-fetched (hit) tasks, see Figure 4(b). As shown in Figure 4(a), the FPGA is assumed to be divided into at least

two PRRs in order to simultaneously pre-fetch/cache missed tasks while other tasks are executing.

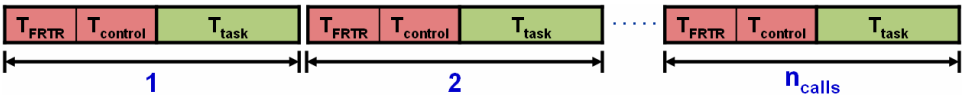
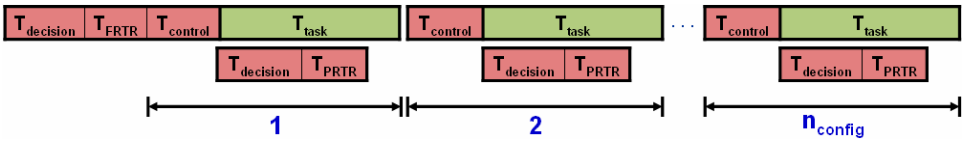
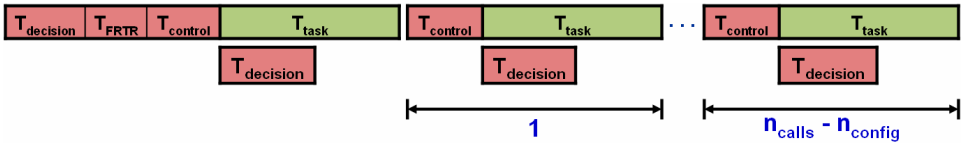


Fig. 3. Typical execution profile using FRTR



(a) Execution profile for missed tasks (i.e. tasks that need to be configured) using PRTR



(b) Execution profile for pre-fetched tasks (i.e. tasks that do not need to be configured) using PRTR

Fig. 4. Execution profile using PRTR

Missed tasks are the tasks that do not exist on the FPGA when needed for execution while hit tasks are the tasks that have been previously pre-fetched to the FPGA and are available for execution when needed. In this scenario, the total execution time would be reduced by the amount of configuration overhead for the hit tasks by overlapping their configuration with the execution of previous tasks. Therefore, the total execution time for the case of PRTR, as shown in Figure 4, can be derived as follows:

$$T_{total}^{PRTR} = \left(T_{decision_1} + T_{FRTR}\right) + \sum_{i=1}^{n_{calls}} T_{control_i} + \sum_{i=1}^{n_{config}} T_{missed_i} + \sum_{i=1}^{n_{calls}-n_{config}} T_{hit_i}$$

$$\Rightarrow T_{total}^{PRTR} = \left(T_{decision} + T_{FRTR}\right) + T_{total}^{control} + T_{total}^{missed} + T_{total}^{hit}, \quad where \tag{3}$$

$$T_{total}^{control} = \sum_{i=1}^{n_{calls}} T_{control_i} = n_{calls} T_{control},$$

$$T_{total}^{missed} = \sum_{i=1}^{n_{config}} T_{missed_i} = \sum_{i=1}^{n_{config}} \max\left(T_{task_i}, T_{decision_{i+1}} + T_{PRTR_{i+1}}\right) = n_{config} \max\left(T_{task}, T_{decision} + T_{PRTR}\right),$$

and

$$T_{total}^{hit} = \sum_{i=1}^{n_{calls}-n_{config}} T_{hit_i} = \sum_{i=1}^{n_{calls}-n_{config}} \max\left(T_{task_i}, T_{decision_{i+1}}\right) = \left(n_{calls} - n_{config}\right) \max\left(T_{task}, T_{decision}\right)$$

Normalizing with respect to T_{FRTR}, equation (3) can be rewritten as follows:

$$X_{total}^{PRTR} = \left(X_{decision} + 1\right) + X_{total}^{control} + X_{total}^{missed} + X_{total}^{hit}$$

$$\Rightarrow X_{total}^{PRTR} = n_{calls} \times \left[\left(\frac{1 + X_{decision}}{n_{calls}} \right) + X_{control} + X_{missed} + X_{hit} \right], \ where \tag{4}$$

$$X_{decision} = \frac{T_{decision}}{T_{FRTR}}, \ X_{control} = \frac{T_{control}}{T_{FRTR}}, \ X_{task} = \frac{T_{task}}{T_{FRTR}}, \ X_{PRTR} = \frac{T_{PRTR}}{T_{FRTR}}, \ X_{total}^{PRTR} = \frac{T_{total}^{FRTR}}{T_{FRTR}},$$

$$X_{total}^{control} = \frac{T_{total}^{control}}{T_{FRTR}} = n_{calls} X_{control}, \ X_{total}^{missed} = \frac{T_{total}^{missed}}{T_{FRTR}} = n_{calls} X_{missed}, \ X_{total}^{hit} = \frac{T_{total}^{hit}}{T_{FRTR}} = n_{calls} X_{hit},$$

$$X_{missed} = \frac{n_{config}}{n_{calls}} \max\left(X_{task}, X_{decision} + X_{PRTR}\right), \ and$$

$$X_{hit} = \left(1 - \frac{n_{config}}{n_{calls}}\right) \max\left(X_{task}, X_{decision}\right)$$

As defined earlier, n_{config}, is the number of (re-)configurations corresponding to the missed tasks. It is obvious that the number of configurations, n_{config}, is less than or equal to the total number of function calls, n_{calls}. Therefore, if we define the ratio of the number of configurations to the total number of calls as the pre-fetching miss-ratio, $M = n_{config}/n_{calls}$, equation (4) can be rewritten as:

$$X_{total}^{PRTR} = n_{calls} \times \left[\left(\frac{1 + X_{decision}}{n_{calls}} \right) + X_{control} + X_{missed} + X_{hit} \right], \ where \tag{5}$$

$$X_{missed} = M \cdot \max\left(X_{task}, X_{decision} + X_{PRTR}\right), \ X_{hit} = H \cdot \max\left(X_{task}, X_{decision}\right),$$

$$M = \frac{n_{config}}{n_{calls}} \equiv Miss\ ratio, \ and \ \ H = 1 - \frac{n_{config}}{n_{calls}} = 1 - M \equiv Hit\ ratio$$

The performance gain (speedup) of PRTR in reference to FRTR can be expressed as follows by combining equations (2) and (5):

$$S = \frac{T_{total}^{FRTR}}{T_{total}^{PRTR}} = \frac{X_{total}^{FRTR}}{X_{total}^{PRTR}} = \frac{1 + X_{control} + X_{task}}{\left(\frac{1 + X_{decision}}{n_{calls}} \right) + X_{control} + X_{missed} + X_{hit}}$$

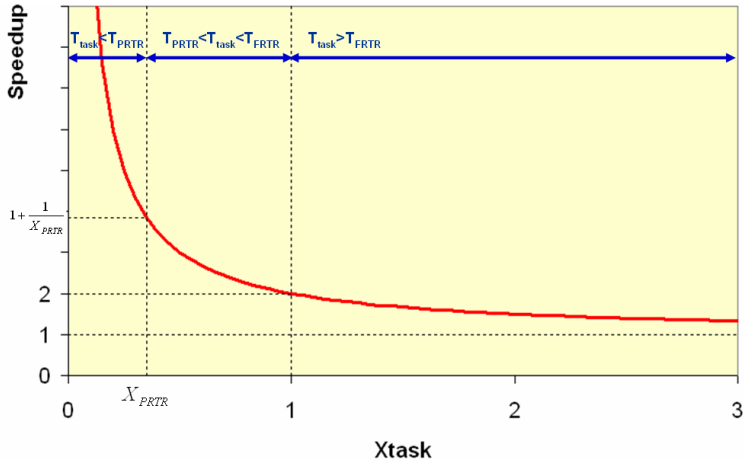
$$\Rightarrow S = \frac{1 + X_{control} + X_{task}}{\frac{\left(1 + X_{decision}\right)}{n_{calls}} + X_{control} + M \cdot \max\left(X_{task}, X_{decision} + X_{PRTR}\right) + H \cdot \max\left(X_{task}, X_{decision}\right)} \tag{6}$$

In order to estimate the upper bound of the performance of PRTR, we take the limit of equation (6) as the number of function calls increases indefinitely. This will help us estimate the asymptotic behavior of PRTR with respect to FRTR as follows:

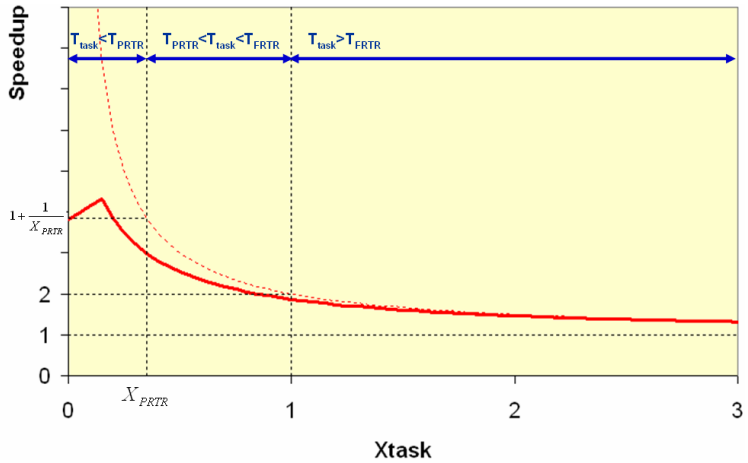
$$S_\infty \equiv \lim_{n_{calls}\to\infty} S$$

$$\Rightarrow S_\infty = \frac{1 + X_{control} + X_{task}}{X_{control} + M \cdot \max\left(X_{task}, X_{decision} + X_{PRTR}\right) + H \cdot \max\left(X_{task}, X_{decision}\right)} \qquad (7)$$

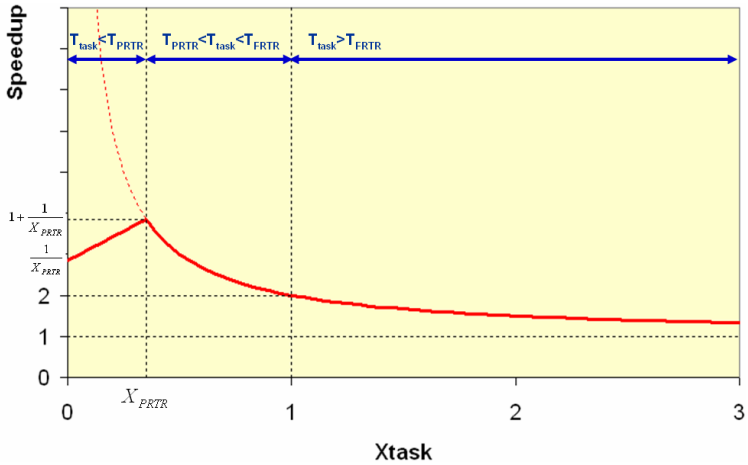
Figure 5 shows the asymptotic speedup of PRTR as given by equation (7) when minimal pre-fetching latency, i.e. $X_{decision}=0$, is assumed as well as zero overhead of transfer of control, i.e. $X_{control}=0$. These overheads will reduce the final speedup if non-zero values are considered. Figure 5 shows the bounds and conditions under which PRTR shows an asymptotic behavior. It can be seen in Figure 5 that PRTR speedup for tasks characterized by higher execution requirements than the full configuration time, i.e. $X_{task}>1$, cannot exceed twice that of FRTR no matter how efficient the pre-fetching algorithm used is. The efficiency of the pre-fetching algorithm affects the speedup only when the task time requirement is less than the full configuration time and is comparable to the partial configuration time, i.e. $X_{PRTR} < X_{task} < 1$ or $0 < X_{task} < X_{PRTR}$, see Figure 5. For highly efficient pre-fetching characterized by high hit rate, i.e. $H \cong 1$, and $M \cong 0$, the speedup decreases monotonically with the task time requirement no matter how large or small the partial configuration overhead is. In this case, the speedup depends on the ratio between the task time requirement and the full configuration time. On the other hand, for much less efficient pre-fetching algorithms, characterized by low hit rate, i.e. $H \cong 0$, and



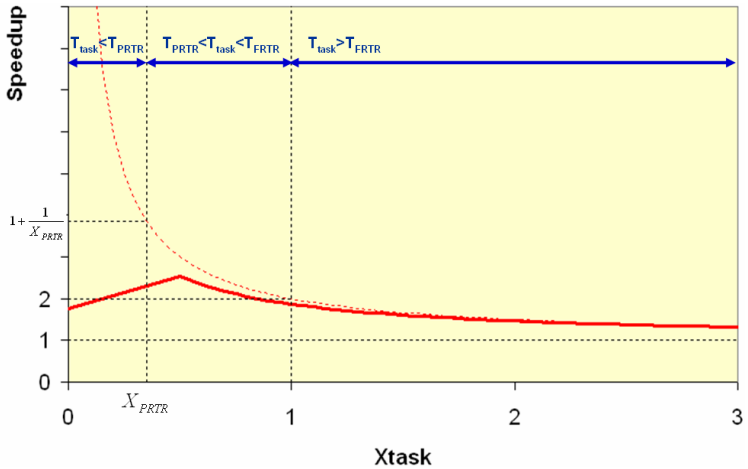
(a) $X_{decision}=0$, $X_{control}=0$, $H=1$, $M=0$, $X_{PRTR}<1$



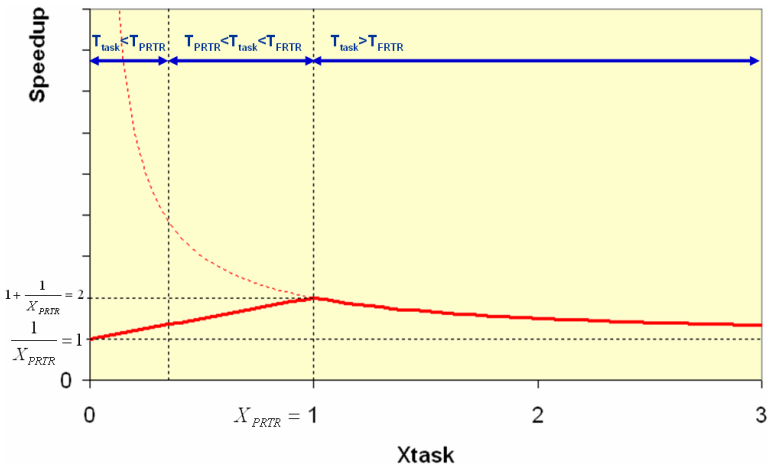
(b) $X_{decision}=0.15$, $X_{control}=0.15$, $H=1$, $M=0$, $X_{PRTR}<1$



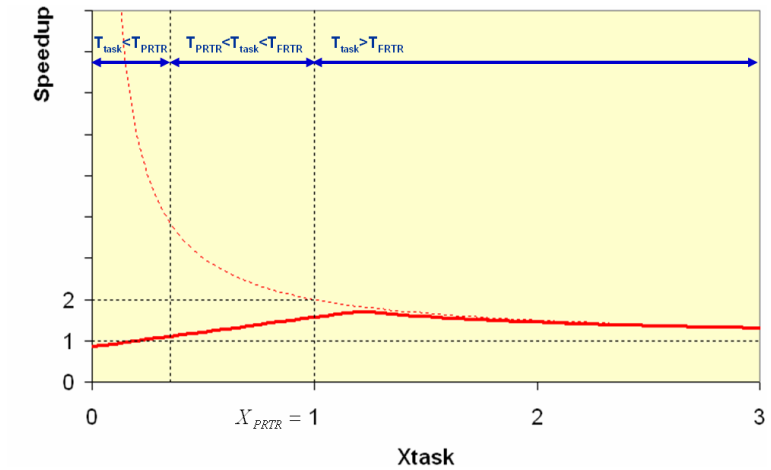
(c) $X_{decision}=0$, $X_{control}=0$, $H=0$, $M=1$, $X_{PRTR}<1$



(d) $X_{decision}=0.15$, $X_{control}=0.15$, $H=0$, $M=1$, $X_{PRTR}<1$



(e) $X_{decision}$=0, $X_{control}$=0, H=0, M=1, X_{PRTR}=1



(f) $X_{decision}$=0.2, $X_{control}$=0.2, H=0, M=1, X_{PRTR}=1

Fig. 5. Asymptotic speedup of PRTR

$M \cong 1$, the speedup reaches its maximum only for those tasks whose time requirement is equal to the partial configuration time, i.e. $X_{task}=X_{PRTR}$, see Figure 5. In this case, the speedup depends on the ratio between the average partial configuration time and the full configuration time.

4. EXPERIMENTAL WORK

Our experiments have been performed on Cray XD1, one of the current HPRCs [Cray 2006]. The Cray XD1 is a multi-chassis system. Each chassis contains up to six nodes (blades). Each blade consists of two 64-bit AMD 2.4 GHz Opteron processors, one Rapid Array Processor (RAP) that handles the communication, an optional second RAP, and an optional Application Accelerator Processor (AAP). The AAP is a Xilinx Virtex-II Pro

XC2VP50-7 FPGA with 16 MB of QDR-II SRAM local memory. The application acceleration subsystem acts as a coprocessor to the AMD Opteron processors, handling the computationally intensive and highly repetitive algorithms that can be significantly accelerated through parallel execution. Figure 6 shows Cray XD1 system architecture.

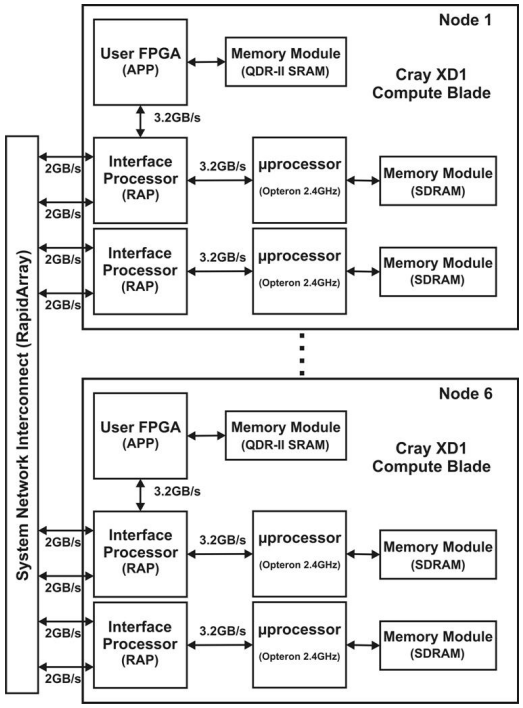


Fig. 6. Cray XD1 architecture

4.1 Partial reconfiguration in Cray XD1: setup and requirements

On Xilinx FPGAs, only the JTAG and the parallel (also known as SelectMap) configuration interfaces support partial reconfiguration. High-end families like Virtex-II, Virtex-4 and Virtex-5 feature an internal access to the parallel interface, i.e. the Internal Configuration Access Port (ICAP), specifically designed for self-reconfiguration. These ports operate at a maximum of 66 MHz (8-bit configuration data) for the Virtex-II Pro devices available in Cray XD1.

Support for RTR (FRTR) in Cray XD1 is performed by one of the vendor's software API functions. This configuration function, when called, downloads a full bitstream using one of the external configuration interfaces previously mentioned, most probably SelectMap in this case. The configuration function, however, returns an error for partial bitstreams because of a simple check on the size of the bitstream. In other words, partial reconfiguration is not natively supported on Cray XD1. Therefore, in order to enable

partial reconfiguration we see it necessary to modify the vendor's configuration function by doing the following:

- Do not check the bitstream size
    - Partial bitstreams have an undefined size from a few bytes to the maximum of full bitstream
- Do not check the DONE signal of the configuration interface
    - This is typically overlooked
    - Partial bitstreams are downloaded when the FPGA is already configured, which means this signal will be always enabled during the reconfiguration process which will fail the check test

However, modifications to the vendor API libraries are not usually possible. These libraries are not open to the user to modify. Therefore, our work-around approach was to use the only available configuration interface, i.e. ICAP. The use of this interface requires the implementation of an additional control circuit to receive the partial bitstream from the host memory, through the conventional data transfer channel between the host and the FPGA, and send it to the internal configuration port. This solution presents two disadvantages. First, the ICAP port is slower than the dedicated external configuration ports which results in higher reconfiguration time. Second, it is necessary to share the communication link between the host and the FPGA for transferring both the configuration bitstreams and needed data. However, this would not heavily impact the overall performance because the communication link in Cray XD1 is a dual channel link, i.e. it has two independent channels one for data input and another for data output. Therefore, it is possible to overlap the execution of tasks with configurations of other tasks as assumed by our analytical model and explained in section 3.1. In this case, partial reconfiguration can only be performed after the data has been transferred from the host to the FPGA (data input), thus overlapping the configuration with either computation time or the data transfer from the FPGA to the host (data output). Although, these two problems impact the final performance, the proposed approach enables PRTR on Cray XD1, and can be applied to any of the available HPRC systems.

Figure 7 shows the implemented control unit in order to support partial reconfiguration. This control unit includes a small buffer using internal BRAM memories to store the partial bitstream. This buffer is necessary because the ICAP has a transfer rate of 66 MB/s while the Hypertransport channel bandwidth reaches 1.6 GB/s. In addition, buffering the configuration bitstreams in internal BRAM memories allows overlapping the transfer of input and /or output data with the configuration of partial bitstream. While

the ICAP is reading the configurations from the BRAM memory, it would be possible to transfer data. Moreover, an additional state machine is implemented to control the communication between the host and BRAM memory as well as the communication between the BRAM memory and the ICAP.

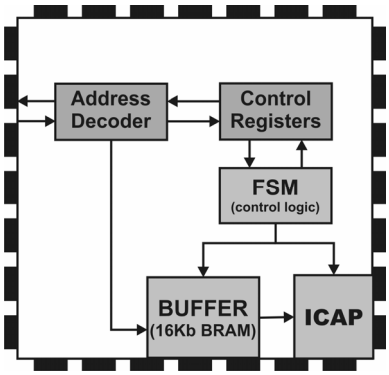
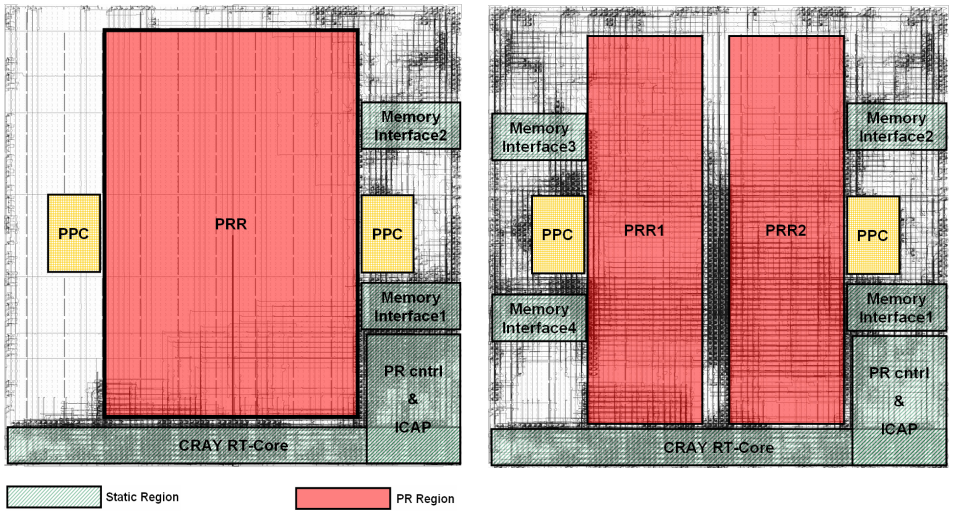


Fig. 7. Internal circuit to support partial reconfiguration using the ICAP configuration port

4.2 Partial reconfiguration in Cray XD1: dynamic scenarios

In Cray XD1 each FPGA is connected to four memory banks. Also, Cray provides a service (interface) block, called Rapid Transport (RT) core, that manages the access to these memories and the communication with the host. The RT core supports several mechanisms of data transfer between the user logic on the FPGA and the host processor. In a typical scenario the host sends the data to the local memory of the FPGA and the user logic reads the data from memory, processes the data, and returns the result back to memory which is then read back by the host. Additionally, there is a DMA mechanism that allows the user logic to initiate the transfer of data in both directions, i.e. write to and read from the host memory directly.

In order to simplify the process of enabling partial reconfiguration, we will assume that the hardware functions use local memory banks to read and write the data, while the DMA capabilities are not used. In this configuration, a maximum of four hardware functions can be implemented if one memory bank is used as input and output. However, the final configuration (FPGA layout) that we used in our experiments supported both single and dual Partially Reconfigurable Regions (PRRs) in addition to the static region, see Figure 8. In the single PRR layout the four banks are available for use by the implemented functions in that PRR region. In the dual PRRs layout, two memory banks are assigned for each region. This is due to the limitations of partial reconfiguration in Virtex-II Pro devices, e.g. a frame includes a whole column of logic resources. Furthermore, the available resources for user logic are limited because XC2VP50 FPGA



(a) Single region layout (static region + 1 PRR) (b) Dual region layout (static region + 2 PRRs)

Fig. 8. FPGA layouts in Cray XD1

in Cray XD1 is not relatively a large device and the two PowerPC (PPC) hard cores occupy a fair amount of the FPGA fabric resources.

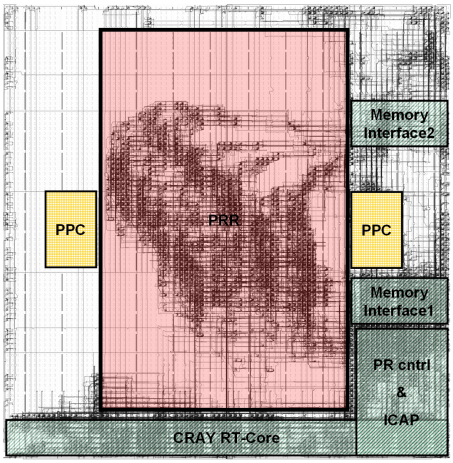
Another important design consideration which is imposed by partial reconfiguration requirements is the implementation of FIFOs between each memory bank and its associated PRR. FIFOs reduced the impact of the fixed allocation of bus macros required to interconnect the PRRs with each others or with the static region. Also, FIFOs simplified the interface with the hardware functions and relaxed the constraint of minimum delay (maximum clock frequency) for the hardware functions. Furthermore, the implementation of FIFOs guaranteed data availability for the hardware functions when the memory was being read.

Finally, it is worth to mention that the interface services block, i.e. RT core provided by Cray, the reconfiguration control unit, and FIFOs are included in the static region. The remaining area of the device is available for the dynamic PRRs, see Figure 8.

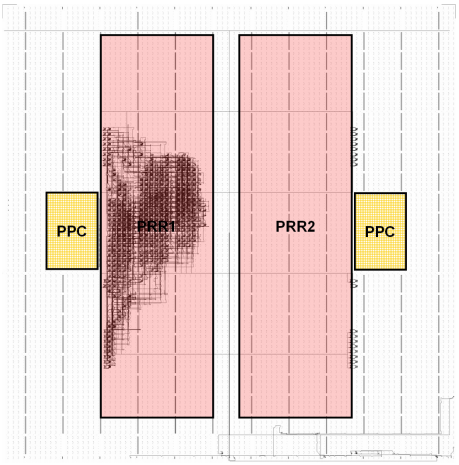
4.3 Experimental Results

For our experiments we selected the application of image feature extraction. In this particular application object edges were of interest and were extracted after first reducing high-frequency noise components. Two different algorithms were used for noise reduction. The final images were transferred back to the microprocessor for quality

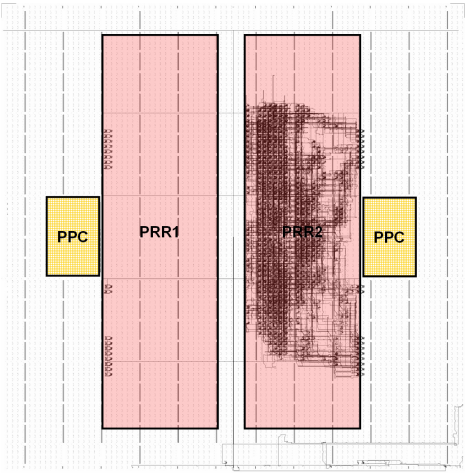
(a) Median filter core in PRR



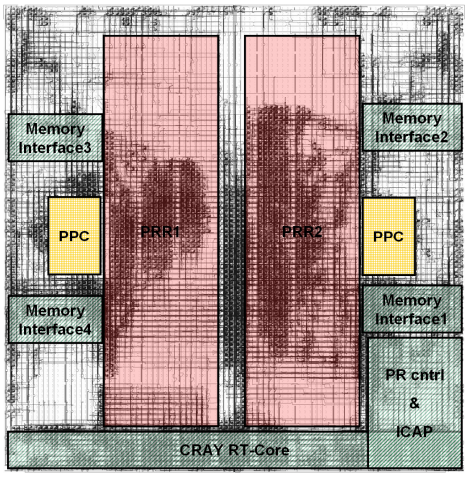
(b) Complete layout for Median filter (Single PRR)



(c) Sobel filter core in PRR1



(d) Smoothing filter core in PRR2



(e) Complete layout for Sobel and Smoothing filters (Dual PRRs)

Fig. 9. FPGA layouts for some image processing cores in Cray XD1

checks. More specifically, this application required the execution of a sequence of image processing functions, namely median filtering followed by sobel edge detection as well as smoothing filtering also followed by sobel edge detection, see Table I. These functions were implemented as hardware functions (cores or tasks) and were executed using both the single and dual PRR layouts. Figure 9 shows the two FPGA layouts for some of the implemented cores. Table II shows data transfer times, configuration times as well as the bitstream size associated with each layout configuration that we considered. The estimated configuration times for each region are calculated based on the size of the region, i.e. bitstream size, and the maximum throughput of the configuration port, i.e. 66 MB/s for SelectMap in this case. These estimates represent a lower bound, i.e. best case scenario, for the configuration times. In addition, the estimated values for data transfers were based on the theoretical maximum bandwidth between the microprocessor and the FPGA as published in the datasheet [Cray 2006] of the testbed. This bandwidth is approximately 1422 MB/s in each [Cray 2006]. The measured values were different from the estimated due to overhead introduced by Cray API configuration function for the case of full configuration, and by the ICAP configuration scheme we used for the partial reconfiguration cases. Furthermore, on Cray XD1 there is a performance gap between microprocessor-initiated input transfers and output transfers. Output transfers from the

Table I. Hardware functions and their resource requirements

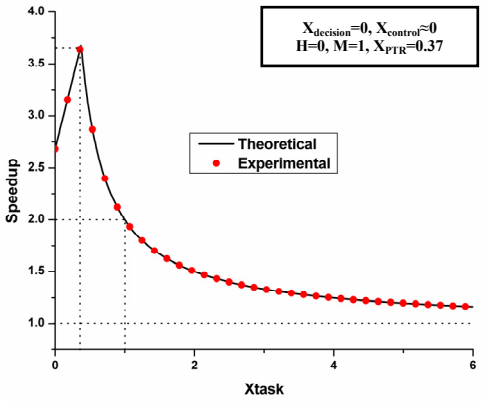
Hardware Function	LUTs	FFs	BRAM	Frequency (MHz)
Static Region	3,372 (7%)	5,503 (11%)	25 (10%)	200
PR Controller	418 (0%)	432 (0%)	8 (3%)	66
Median Filter	3,141 (6%)	3,270 (6%)	NA	200
Sobel Filter	1,159 (2%)	1,060 (2%)	NA	200
Smoothing Filter	2,053 (4%)	1,601 (3%)	NA	200

Table II. Experimental values for model parameters

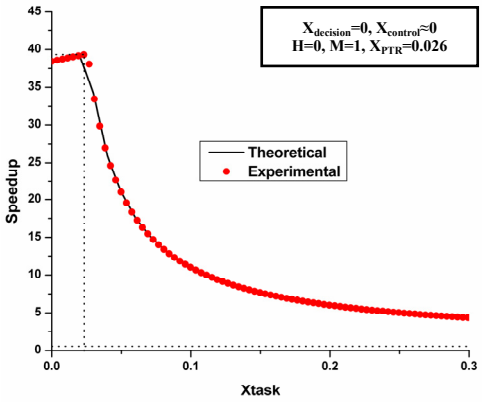
	Data Size (Bytes)	Time (msec)		Normalized Configuration time, X_{PRTR}	
		Estimated	Measured	Estimated	Measured
Full Configuration	2381764	36.09	1678.04	1	1
Single PRR	887784	13.45	43.48	0.37	0.026
Dual PRR	404168	6.12	19.77	0.17	0.012
Input Transfer	4194304	2.95	3.00	NA	NA
Output Transfer	4194304	2.95	641.10	NA	NA

FPGA to the microprocessor require the processor to wait for a response from the FPGA (in other words, the requested data). There is no mechanism for the processor to issue burst read requests to the FPGA or to have multiple outstanding read requests. As a result, the microprocessor can write to the FPGA much more efficiently than it can read. This fact is explicitly stated by Cray in [Cray 2006] and verified by our experiments as shown in Table II.

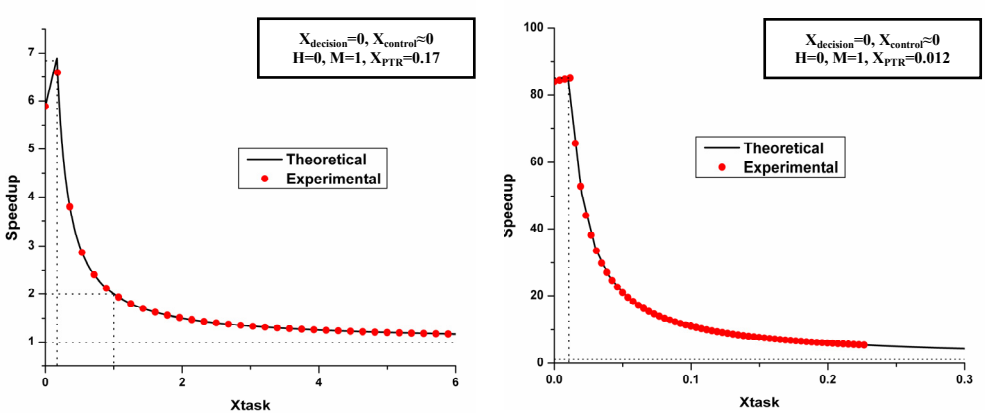
It is worth to mention that the pre-fetching mechanism adopted for our experiments is a worst-case implementation. The goal of our experiments was to show the independent performance behavior of PRTR compared to that of FRTR with minimal contribution from the pre-fetching techniques. Based on our previous discussion in section 3.1, this case can be considered as the one in which the least efficient pre-fetching algorithm was implemented. In other words, our hypothetical configuration pre-fetching always misses tasks when needed and always reconfigures the called tasks. This can be modeled by



(a) Single PRR results normalized w.r.t. estimated configuration time, T_{FRTR}=36.09 ms



(b) Single PRR results normalized w.r.t. measured configuration time, T_{FRTR}= 1678.04 ms



(c) Dual PRR results normalized w.r.t. estimated configuration time, T_{FRTR}=36.09 ms

(d) Dual PRR results normalized w.r.t. measured configuration time, T_{FRTR}= 1678.04 ms

Fig. 10. Experimental results of PRTR in Cray XD1

$X_{decision}=0$, $M=1$, $H=0$. In addition, the transfer of control time was measured to be minimal compared to other parameters. The task time requirement was varied by changing the amount of data transferred to/from and processed by the task. In other words, this was performed by changing the image size. The parameters that we measured in our experiments for both cases of single and dual PRR layouts were as follows:

- $n_{calls} \cong \infty$, $T_{decision} = 0$, $T_{control} \cong 10$ µsec
- $H = 0$, $M = 1$

5. DISCUSSION AND FUTURE DIRECTIONS

Figure 10 shows the results collected in our experiments for both scenarios of a single PRR and dual PRRs. It can be seen that the results are in good agreement with what is predicted by the model. However, the experimental results are slightly deviated from the theoretical expectations because of the transfer of control overhead. It can also be noticed, by comparing Figure 10(a) with 10(c) and 10(b) with 10(d), that the speedup for the case of dual PRR is almost double that of the single PRR layout. This is due to the fact that the size of the single PRR is as twice as that of the dual PRR, see Figure 8 and Table II. It is worth to mention that the entries in Table II under the data size column for the "Input Transfer" and "Output Transfer" refer to the size of the image being filtered while the entries for "Full Configuration", "Single PRR", and "Dual PRR" refer to the size of the corresponding configuration bitstream in bytes.

As shown in the experimental results, the relative positioning of the task time requirements with respect to the full configuration time affects significantly the achieved speedup. For example, in the best configuration scenario the full configuration time is estimated to take only 36 ms, see Table II, while most of the data-intensive tasks require

larger execution time given the I/O bandwidth, i.e. 1422 MB/s, on Cray XD1. In this case, PRTR speedup is bounded to twice the speedup of FRTR, see Figure 10(a) and 10(c). For less data-intensive tasks, the PRTR cannot exceed 7 times the speedup of FRTR for dual PRR layout and 3.86 times the speedup of FRTR for a single PRR layout, see Figure 10(a) and 10(c). This speedup is dependent on the ratio between the partial configuration time and the full configuration time, i.e. X_{PRTR}. However, in a realistic situation on Cray XD1 the full configuration time, as shown in Table II, is much larger, i.e. 1.7 seconds, than the requirements for the majority of tasks including those tasks that are data-intensive. Only in this case, where FRTR overhead is high, PRTR is more beneficial. The peak speedup, again depending on X_{PRTR}, can reach up to 87x higher than the speedup of FRTR for dual PRR layout and up to 40x for single PRR layout, see Figure 10(b) and 10(d). In other words, in order to achieve the optimal speedup of fully dynamic partial reconfigurable systems through PRTR, the partitions (PRRs) must be so fine grained to match the task time requirements, i.e. $X_{PRTR}=X_{task}$. This would reduce the configuration overhead and increase the system density in terms of the number of Partially Reconfigured Regions (PRRs) per chip.

Given the analytical findings as well as the experimental results, we conclude that PRTR support on HPRCs can be beneficial from the performance perspective. However, these benefits are insignificant performance offsets for a broad range of applications as compared to those of FRTR. Moreover, given the current status of the technology these benefits are associated with many conditions and practical requirements. These practical considerations might overweight the gains especially when productivity is added to the picture. For example, the current design cycle for PRTR increases exponentially with the number of implemented tasks and PRRs. All permutations among the tasks across all PRRs must be implemented before PRTR is utilized. This increases dramatically the development time. With future support of Operating Systems for PRTR, we see PRTR as compared to FRTR is far more beneficial for versatility purposes, multi-tasking applications, and hardware virtualization than it is for plain performance. Nevertheless, improving versatility and providing more efficient support for multi-tasking and hardware virtualization will positively impact the overall performance.

6. CONCLUSIONS

In this paper we presented an effort of High-Performance Reconfigurable Computing (HPRC) support for Partial Run-Time Reconfiguration (PRTR). We investigated the performance potential of PRTR on HPRCs from both theoretical and practical

perspectives. In doing so, we derived a formal and an analytical model of PRTR on HPRC systems relative to the baseline of Full Run-Time Reconfiguration (FRTR). The model provided us with theoretical expectations which served as a frame of reference against which we projected our experimental results. In addition, it helped us gain in-depth insight about the boundaries and/or conditions for possibilities of performance gain using PRTR. In achieving this objective, our approach was based on leveraging previous work and concepts that were introduced for solving similar and related problems. For example, we included in our analytical model the concept of configuration caching (pre-fetching) which is usually associated with PRTR.

In conducting the experimental work, we utilized one of the current HPRC systems, Cray XD1. We discussed the issues of PRTR support on HPRCs and provided recommendations for vendor support. We also discussed the requirements and the setups for PRTR on Cray XD1. Our setup included the design of a special configuration control unit managing the configuration of different layouts of Partially Reconfigured Regions (PRRs). The approach we followed for Cray XD1 is general and can be applied to any of the available HPRC systems.

Based on our analytical and experimental findings, we see hardware virtualization and multi-tasking using PRTR from a versatility perspective as good directions for further investigations.

REFERENCES

AGGARWAL, V., GEORGE, A.D., AND SLATTON, K.C. 2006. Reconfigurable Computing with Multiscale Data Fusion for Remote Sensing. In *Proceedings of the 2006 ACM/SIGDA 14th International Symposium on Field Programmable Gate Arrays (FPGA 2006)*, Monterey, California, USA.

BONDALAPATI, K., AND PRASANNA, V.K. 1999. Dynamic precision management for loop computations on reconfigurable architectures. In *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '99)*, 249-258, 1999.

BUELL, D.A., EL-GHAZAWI, T.A., GAJ, K., AND KINDRATENKO, V. 2007. Guest Editors' Introduction: High-Performance Reconfigurable Computing. *IEEE Computer* 40(3), 23-27, 2007.

BUELL, D.A., DAVIS, J.P., QUAN, G., AKELLA, S., DEVARKAL, S., KANCHARLA, P., MICHALSKI, E.A., AND WAKE, H.A. 2004. Experiences with a reconfigurable computer. In *Proceedings of Engineering of Reconfigurable Systems and Algorithms*, Las Vegas, Nevada, 21-24 June 2004.

BUELL, D.A., AND SANDHU, R. 2003. Identity management. *IEEE Internet Computing* 7(6), 26-28, November/December 2003 (guest editors' introduction).

COURT, T.V., AND HERBORDT, M.C. 2007. Families of FPGA-Based Accelerators for Approximate String Matching. *ACM Microprocessors & Microsystems* 31(2), 135-145, March 2007.

CRAY INC. 2006. Cray XD1™ FPGA Development (S-6400-14), 2006.

EL-ARABY, E., TAHER, M., GAJ, K., EL-GHAZAWI, T., CALIGA, D., AND ALEXANDRIDIS, N. 2006. System-Level Parallelism and Concurrency Maximisation in Reconfigurable Computing Applications. *International Journal of Embedded Systems (IJES)* 2(1/2), 62-72, 2006.

EL-ARABY, E., TAHER, M., EL-GHAZAWI, T., AND LE MOIGNE, J. 2005. Prototyping Automatic Cloud Cover Assessment (ACCA) Algorithm for Remote Sensing On-Board Processing on a Reconfigurable

Computer. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT 2005)*, Singapore, December, 2005.

EL-ARABY, E. 2005. A System-Level Design Methodology for Reconfigurable Computing Applications. *A Thesis for the Master of Science Degree in Computer Engineering*, Department of Electrical and Computer Engineering, The George Washington University, January 2005.

EL-ARABY, E., EL-GHAZAWI, T., LE MOIGNE, J., AND GAJ, K. 2004. Wavelet Spectral Dimension Reduction of Hyperspectral Imagery on a Reconfigurable Computer. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT 2004)*, Brisbane, Australia, December, 2004.

EL-GHAZAWI, T., EL-ARABY, E., HUANG, M., GAJ, K., KINDRATENKO, V., AND BUELL, D. 2008. The Promise of High-Performance Reconfigurable Computing. *IEEE Computer* 41(2), 69-76, February 2008.

FIDANCI, D., POZNANOVIC, D., GAJ, K., EL-GHAZAWI, T., AND ALEXANDRIDIS, N. 2003. Performance and Overhead in a Hybrid Reconfigurable Computer. *Reconfigurable Architectures Workshop (RAW 2003), Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS) Workshops*, Nice, France, 22-26 April, 2003.

GOKHALE, M., GRAHAM, P., WIRTHLIN, M.J., JOHNSON, D.E., AND ROLLINS, N. 2006. Dynamic reconfiguration for management of radiation-induced faults in FPGAs. *IJES* 2(1/2), 28-38, 2006.

HADLEY, J.D., AND HUTCHINGS, B.L. 1995. Design methodologies for partially reconfigured systems. In *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, ATHANAS, P., AND POCEK, K.L., Eds., Napa, CA, , 78-84, April 1995.

HARKINS, J., EL-GHAZAWI, T., EL-ARABY, E., AND HUANG, M. 2005. Performance of Sorting Algorithms on the SRC 6 Reconfigurable Computer. In *Proceedings of the IEEE International Conference on Field-Programmable Technology (FPT 2005)*, Singapore, December, 2005.

HASAN, M.Z., AND ZIAVRAS, S.G. 2007. Runtime Partial Reconfiguration for Embedded Vector Processors. In *Proceedings of the 4th International Conference on Information Technology (ITNG'07)*, 983-988, Las Vegas, NV, USA, April 2007.

HÜBNER, M., AND BECKER, J. 2006. Exploiting Dynamic and Partial Reconfiguration for FPGAs – Toolflow, Architecture, and System Integration. In *Proceedings of the 19th SBCCI Symposium on Integrated Circuits and Systems Design*, Ouro Preot, Brazil, 2006.

HYMEL, R., GEORGE, A.D., AND LAM, H. 2007. Evaluating Partial Reconfiguration for Embedded FPGA Applications. In *Proceedings of High-Performance Embedded Computing Workshop (HPEC 2007)*, MIT Lincoln Lab, Lexington, MA, 18-20 September, 2007.

JEONG, B., YOO, S., AND CHOI, K. 1999. Exploiting Early Partial Reconfiguration of Run-Time Reconfigurable FPGAs in Embedded Systems Design. In *Proceedings of the ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays (FPGA 1999)*, 247, Monterey, California, USA, 1999.

KINDRATENKO, V., AND POINTER, D. 2006. A case study in porting a production scientific supercomputing application to a reconfigurable computer. In *Proceedings IEEE Symposium on Field-Programmable Custom Computing Machines - FCCM'06*, 13-22, 2006.

LI, Z., AND HAUCK, S. 2002. Configuration Prefetching Techniques for Partial Reconfigurable Coprocessor with Relocation and Defragmentation. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA 2002)*, 187-195, 2002.

LI, Z., COMPTON, K., AND HAUCK, S. 2000. Configuration Caching Management Techniques for Reconfigurable Computing. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines (FCCM '00)*, 87-96, 2000.

MICHALSKI, A., GAJ, K., AND EL-GHAZAWI, T. 2003. An Implementation Comparison of an IDEA Encryption Cryptosystem on Two General-Purpose Reconfigurable Computers. In *Proceedings of FPL 2003*, 204-219, Lisbon, Sept. 2003.

SILICON GRAPHICS INC. 2007. Reconfigurable Application-Specific Computing User's Guide (007-4718-005), January 2007.

SMITH, M.C., AND PETERSON, G.D. 2002. Analytical Modeling for High Performance Reconfigurable Computers. In *Proceedings of the SCS International Symposium on Performance Evaluation of Computer and Telecommunications Systems*, July 2002.

SMITH, M.C. 2002. Analytical Modeling of High Performance Reconfigurable Computers: Prediction and Analysis of System Performance. *A Dissertation Proposal for the Doctor of Philosophy Degree in Electrical Engineering*, The University of Tennessee, Knoxville, March 2002.

SRC COMPUTERS INC. 2006. SRC Carte™ C Programming Environment v2.2 Guide (SRC-007-18), August 2006.

STORAASLI, O. 2002. Scientific Applications on a NASA Reconfigurable Hypercomputer. *5th MAPLD International Conference*, Washington, DC, USA, September, 2002.

TAHER, M. 2005. Exploiting Processing Locality for Adaptive Computing Systems. *A Dissertation Proposal for the Doctor of Philosophy Degree*, Department of Electrical and Computer Engineering, The George Washington University, May 2005.

TAHER, M., EL-ARABY, E., AND EL-GHAZAWI, T. 2005. Configuration Caching in Adaptive Computing Systems Using Association Rule Mining (ARM). *Dynamic Reconfigurable Systems Workshop (DRS 2005)*, Innsbruck, Austria, March 2005.

TRIPP, J.L., MORTVEIT, H.S., HANSSON, A.A., AND GOKHALE, M. 2005. Metropolitan road traffic simulation on FPGAs. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2005 (FCCM 2005)*, 117- 126, 18-20 April 2005.

ULLMANN, M., GRIMM, B., HÜBNER, M., AND BECKER, J. 2004. An FPGA Run-Time System for Dynamical On-Demand Reconfiguration. In *Proceedings of IEEE Parallel and Distributed Processing Symposium*, Santa Fe, NM, 26-30 April, 2004.

XILINX INC. 2006. Early Access Partial Reconfiguration User Guide. *User Guide 208 (v1.1)*, March 2006.

XILINX INC. 2004. Two Flows for Partial Reconfiguration: Module Based or Difference Based. *Xilinx Application Note XAPP290 (v1.2)*, September 2004.