Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 127302, 10 pages
doi:10.1155/2012/127302

Research Article

Using Partial Reconfiguration and Message Passing to Enable FPGA-Based Generic Computing Platforms

Manuel Saldaña,[1] Arun Patel,[1] Hao Jun Liu,[2] and Paul Chow[2]

[1] *ArchES Computing Systems, 708-222 Spadina Avenue, Toronto, ON, Canada M5T 3A2*
[2] *The Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, 10 King's College Road, Toronto, ON, Canada M5S 3G4*

Correspondence should be addressed to Manuel Saldaña, ms@archescomputing.com

Received 12 May 2011; Accepted 22 August 2011

Academic Editor: Marco D. Santambrogio

Partial reconfiguration (PR) is an FPGA feature that allows the modification of certain parts of an FPGA while the rest of the system continues to operate without disruption. This distinctive characteristic of FPGAs has many potential benefits but also challenges. The lack of good CAD tools and the deep hardware knowledge requirement result in a hard-to-use feature. In this paper, the new partition-based Xilinx PR flow is used to incorporate PR within our MPI-based message-passing framework to allow hardware designers to create *template bitstreams*, which are predesigned, prerouted, generic bitstreams that can be reused for multiple applications. As an example of the generality of this approach, four different applications that use the same template bitstream are run consecutively, with a PR operation performed at the beginning of each application to instantiate the desired application engine. We demonstrate a simplified, reusable, high-level, and portable PR interface for X86-FPGA hybrid machines. PR issues such as local resets of reconfigurable modules and context saving and restoring are addressed in this paper followed by some examples and preliminary PR overhead measurements.

1. Introduction

Partial reconfiguration (PR) is a feature of an FPGA that allows part of it to be reconfigured while the rest of it continues to operate normally. PR has been the focus of considerable research because of the many potential benefits of such a feature. For example, it allows the implementation of more power-efficient designs by using hardware on-demand, that is, only instantiate the logic that is necessary at a given time and remove unused logic. PR also allows the virtualization of FPGA resources by time sharing them among many concurrent applications. Alternatively, a single large application may be implemented even if it requires more logic resources than what a single FPGA can provide as long as the logic resources are not required simultaneously. Fault-tolerant systems and dynamic load-balancing are also potential benefits of PR. All these features make PR attractive for applications in the fields of automotive and aerospace design, software radio, video, and image processing, among other markets.

However, there are many challenges for PR to be more widely accepted, for example, dynamic changing of logic adds an extra level of difficulty to verification performance overheads in terms of designed target frequency and higher resource utilization complex design entry tools and PR CAD flows. A PR design requires the partitioning and floorplanning of the entire FPGA into static regions (SRs) and reconfigurable regions (RRs). The SR does not change during the execution of an application, and RRs may be dynamically reconfigured during the execution of the application. This partitioning has added another layer of complexity to FPGA design. To cope with this increased complexity, Xilinx has released a new partition-based ISE 12 solution that simplifies PR design [1].

By looking at Xilinx documentation and comparing the resources in Virtex-E and Virtex 7 devices, we can see that

in about ten years, FPGAs have increased their resources roughly over 18-fold in LUTs, 14-fold in Flip-Flops, and 31-fold in BRAMs. Furthermore, the number of configuration bits for the XC7V2000T FPGA is just under 54 MB. At this rate, handling partial bitstreams may become more practical than handling entire bitstreams. As FPGA sizes continue to increase, new use case models that were not possible before will now start to emerge, and PR can be an enabling technology of such models.

In this paper, we extend previous work [2] on partial reconfiguration to explore new use cases for FPGAs by using PR within a message-passing system to provide generic, predesigned and prerouted computing platforms based on template designs, where the user can dynamically populate the RRs with application cores. These generic templates can then be modified as needed for a particular application and still be released as application-specific templates. The goal is to relieve the user from the burden of having to design the communications infrastructure for an application and focus solely on application cores. Equally important is to do this in a portable manner across platforms. To this end, we add PR capabilities to the ArchES-MPI framework [3, 4], which provides a communication abstraction layer that enables point-to-point communications between high-end X86 and embedded processors, and hardware accelerators.

The rest of the paper is organized as follows: Section 2 mentions some related work in PR and Section 3 introduces the concept of PR within our message-passing framework. Section 4 describes the synchronization process suggested to perform PR, how to store and restore the current status, and how to generate necessary resets. Section 5 shows the hardware platform used to run the tests. Section 6 explains the software flow to create and handle partial bitstreams. Experimental results are shown in Section 7. Section 8 describes an example where four applications reuse the same template bitstream. Finally, Section 9 presents some concluding remarks.

2. Related Work

There has been abundant research on PR in the last decade with much of it focusing on specific aspects of PR such as CAD flows [5], scheduling [6], communications [7], configuration time evaluation frameworks [8], and core relocation [9]. Our long-term goal is to use PR to implement an entire, practical and generic framework that allows hardware designers to create generic and application-specific template platforms that follows a higher-level parallel programming model that is easier to understand by software developers. This paper presents a step in that direction, providing a working framework that includes the software interface, CAD flow, network-on-chip (NoC), and message-passing layer. Future work will focus on adding and optimizing different aspects of the PR framework.

One of the PR aspects is the state store and restore capability before and after PR takes place. An approach for doing this is by reading back parts of the FPGA configuration memory as described in [10]. However, this approach assumes deep knowledge of the bitstream format and it is strongly dependent upon the FPGA architecture, which is too low level for most developers. Additionally, unnecessary data is read back, which increases the storage overhead and the time to perform the context switch. Our approach is to let the user decide what to store and restore using the message-passing infrastructure making the solution more portable and higher-level at the expense of some additional design effort. Research has been done to provide platform-independent PR flows [11]. Similarly, our goal is to achieve more portable PR systems by using our existing MPI infrastructure.

The BORPH research project [12] follows similar goals with our work in the sense that it aims at simplifying the use of FPGAs by inserting high-level and well-known abstractions. PR has been added to BORPH as a way to dynamically configure a hardware engine at runtime and treat it as an operating system process. Communication with the engine is achieved via file system calls, such as open, read, write, and close. In our work, we treat the hardware engines as MPI processes (not operating system processes) and communication is achieved via MPI function calls, such as MPI_Send and MPI_Recv. Our approach is more suitable for parallel programming as it is based on a standard designed specifically for such a purpose.

In most prior art in this field, the configuration controller is an embedded processor (MicroBlaze or PowerPC) using a fixed configuration interface (ICAP or SelectMAP) [13] and the controller also generates the local reset pulse after PR. In our framework, the X86 processor controls the configuration and the actual configuration interface is abstracted away from the user, who does not need to know which interface is used. Also, in our approach, an embedded processor is not necessary to issue the local reset, rather it is generated by using the underlying messaging system already available.

3. Message Passing and PR in FPGAs

The MPI standard [14] is a widely used parallel programming API within the high-performance computing world to program supercomputers, clusters, and even grid applications. Previous work presented TMD-MPI [4] and proved that a subset implementation of the MPI standard can be developed targeting embedded systems implemented in FPGAs. TMD-MPI was initially developed at the University of Toronto and now it is supported as a commercial tool known as ArchES-MPI. It allows X86, MicroBlaze, and PowerPC processors as well as hardware accelerators to all exchange point-to-point messages and work concurrently to solve an application. By using MPI as a communications middleware layer, portability across multiple platforms can be achieved easily. In particular, platforms that include FPGAs have extra benefit from a portability layer as hardware can change all the time, either because of the nature of the FPGA itself or due to changes in the boards that FPGAs are placed on. But most importantly, ArchES-MPI provides a unified programming model and proposes a programming

paradigm to the developer that facilitates the implementation of heterogeneous, multicore, multiaccelerator systems.

The use of PR within our message-passing framework is to allow the creation of predesigned and prerouted platforms that can be distributed as templates to the end users with the purpose of simplifying the system-level design process. By using PR it is possible to create an array of RRs that can be populated with user application cores, known as reconfigurable modules (RMs) at run time. Figure 1 shows some examples of these generic designs for one, four, and eight RRs. Application cores can also be placed in the SR and the ArchES-MPI infrastructure transparently handles the communication between the cores regardless of the region they are placed in.

A typical MPI program has multiple software processes, each with a unique ID number known as the rank. In ArchES-MPI there are software ranks running on processors and hardware ranks running as hardware engines. One MPI hardware rank can have multiple RMs. This means that the rank number assigned to it does not change during the execution of the entire application, just its functionality depending on the RM currently configured. There is another situation (not covered in this paper) where a new RM requires a change in the assigned rank number, which requires the NoC routing tables to be updated dynamically.

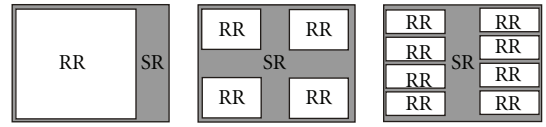

Figure 1: Layout of template-based, generic bitstreams for one, four, and eight reconfigurable regions (RRs) and the static region (SR).

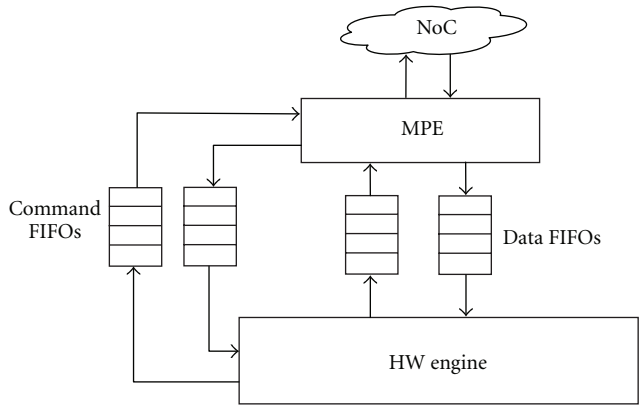

Figure 2: Connection of the application hardware engine and the MPE.

3.1. Message-Passing Engine and Reconfigurable Regions. To processors, ArchES-MPI appears as a software library that provides message-passing capabilities. Hardware engines must use a hardware block called the MPE (message-passing engine), which encapsulates in hardware some of the MPI functionality. The MPE provides the hardware equivalent to MPI_Send and MPI_Recv to a hardware engine in the FPGA. It handles unexpected messages, processes the communication protocol, and divides large messages into smaller size packets to be sent through the NoC. As shown in Figure 2, the MPE is connected between the hardware engine and the NoC. The MPE receives the message parameters and data from the hardware engine, such as the operation (whether it is sending or receiving a message), the destination node id (rank of the process in the MPI environment), the length of the message, and an identification number for the message (the Tag parameter in a normal MPI send/receive operation). After this, the MPE will handle the communications through the NoC with the destination rank. The hardware engine and the MPE form a single endpoint in the message-passing network. The interfaces between the MPE and the hardware engine are four FIFOs (two for commands and two for data), making integration easy. These FIFOs can be asynchronous allowing the hardware engine to operate at different frequencies than its associated MPE.

Based on where the MPE is placed relative to the RR, we can have three possible scenarios. Figure 3(a) shows the MPE placed outside the RR, and it is connected to a user wrapper that controls the MPE and the actual RM through control signals. The RM is a block instantiated within the wrapper. The RM block can be thought of as the main computational pipeline and the wrapper as a higher-level, application-specific controller. One of the wrapper's duties is to control the PR synchronization process (described in the next section). In Figure 3(b), the MPE is also placed outside the RR but with no wrapper (or a very small and generic wrapper) and it is connected directly to the RR. This means that the entire RR is reserved for a single and self-contained RM that must directly control the MPE and handle its own PR synchronization process. This self-contained scenario is more generic than the wrapper-based scenario because it does not contain an application-specific wrapper that may not work with a different application. From the implementation point of view, the user can still design a specific wrapper for a particular application but it would have to be placed in the RR and not in the SR. Finally, Figures 3(c) and 3(d) show the scenario where there is no MPE in the SR. This scenario gives the user the opportunity to implement a sub-NoC that may contain many more ranks or user-defined bridges for off-chip communications such as an Ethernet-based bridge. For brevity, in this paper we only focus on the first two scenarios.

4. Partial Reconfiguration Synchronization

Certain applications may need a way to store the current status of registers, state machines, and memory contents before PR takes place and restore them once the RM is configured back again. This is analogous to pushing and popping variables to a stack before and after a function call. A local reset pulse may also be required to initialize the newly partially reconfigured module to drive registers or state machines to their quiescent state; the global reset cannot be used for this.

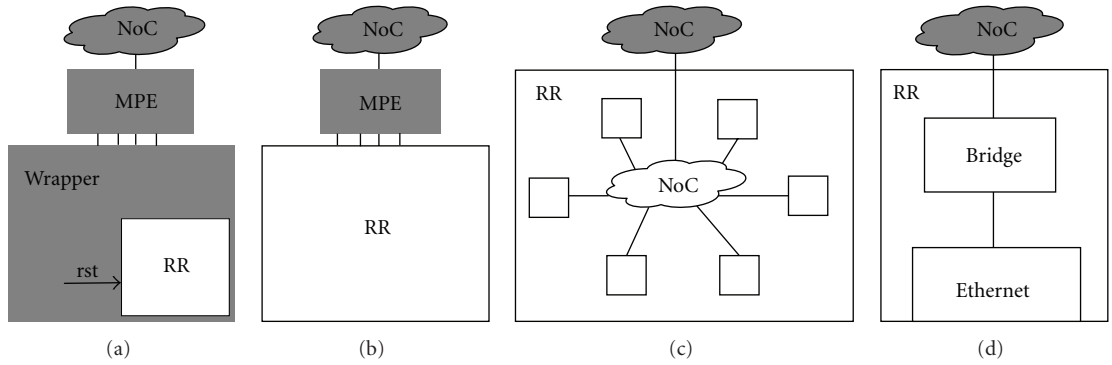

Figure 3: (a) Wrapper-contained RR, (b) Self-contained RR, (c) and (d) no-MPE RRs. (Gray means static and white means dynamic).

4.1. Status Store and Restore. The requirement to store and restore the current status of an RM is solved by sending explicit messages between the configuration controller (a rank within the MPI world) and the rank that is going to be partially reconfigured. Since any rank in MPI can initiate a message, there can be two options to start the PR process based on the initiating entity: the *processor-initiated PR* and the *RM-initiated PR*. This is shown in Figure 4. In the processor-initiated PR, the X86 processor (Rank 0) acts as a master and initiates the reconfiguration synchronization by sending an explicit message ("cfg") to RM_A (Rank 1), which acts as a slave waiting for this message and reacting to it. RM_A then sends an "OK-to-configure" message back to Rank 0 when it is safe to perform the configuration process. The payload of this "OK-to-configure" message can be used to save the current status of RM_A. When Rank 0 receives the "OK-to-configure" message it stores the status (if any) in memory and proceeds with the partial configuration process of RM_B (also Rank 1). Once this is done, a "configuration-done" message is sent to the newly configured module RM_B with the previously stored status data (if any), which is received and used by RM_B to restore its state prior to PR.

The second synchronization option is when the RM is the master and the X86 is the slave. In this case, when a certain condition is met, RM_A sends a "request-to-configure" message along with the status data to Rank 0, which stores the status data and proceeds with the reconfiguration of RM_B. After this is done, Rank 0 sends the "configuration-done" message to the newly configured RM_B along with the status data to restore its state. The user-defined condition that triggers the PR entirely depends on the application.

4.2. Local Reset. The reset issue can be handled easily in the wrapper-contained scenario mentioned in Section 3.1. Since the wrapper controls the MPE it knows when the PR synchronization messages are sent and received and it can assert the reset signal of the RR while PR takes place and release it after the "configuration-done" message is received. In the wrapper-contained scenario the wrapper is placed in the SR and it is aware of when PR has occurred. In contrast, in the self-contained scenario all the application logic is in the RM and it is all being configured, therefore, a different reset mechanism must be implemented. In multi-FPGA systems this is an inherently distributed situation where the RR might be in a different FPGA than where the configuration controller (e.g., embedded processor) is located. Everything has to be done at the endpoints using messagepassing otherwise a reset signal might need an off-chip wire between FPGAs, and as many wires as there are RRs per FPGA, which is not practical.

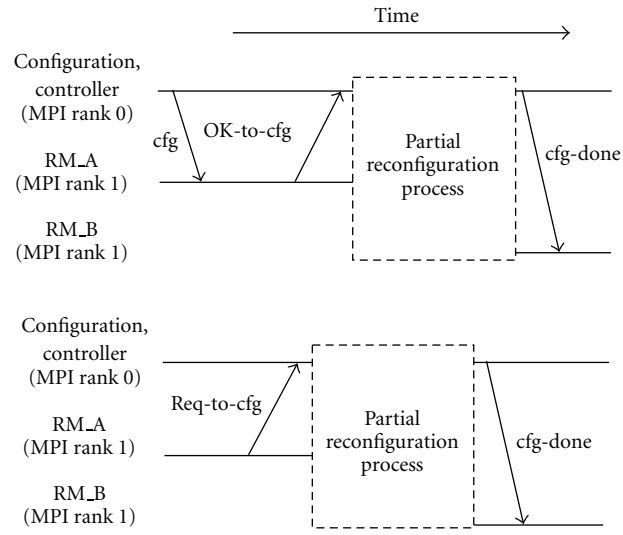

Figure 4: Explicit message synchronization between PR controller and RMs.

Previous work [2] suggested that a possible way to autogenerate a local reset is by using an LUT configured as shift register (SRL) in each RM with all the bits initialized to the reset value (1's if reset is active high) and its input tied to the global system reset signal. The output of the SRL is used to reset the RM. This way, as soon as the RM is configured the RM will be in the reset state and the clock will start shifting the current value of the global reset signal, which should be deasserted assuming the rest of the system is currently running. Eventually, after a given number of cycles (length of the shift register, e.g., 16 cycles) the SRL output will be deasserted and the RM will come out of reset. This process is repeated every time the RM is configured.

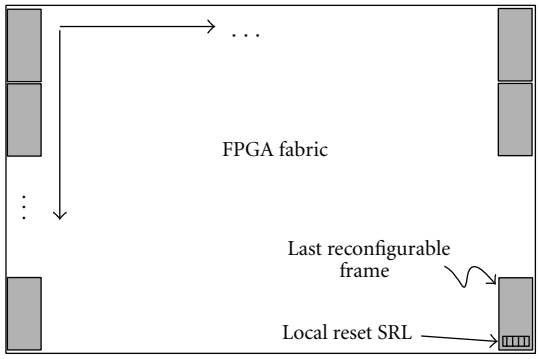

Figure 5: Placement of the reset SRL within the RR.

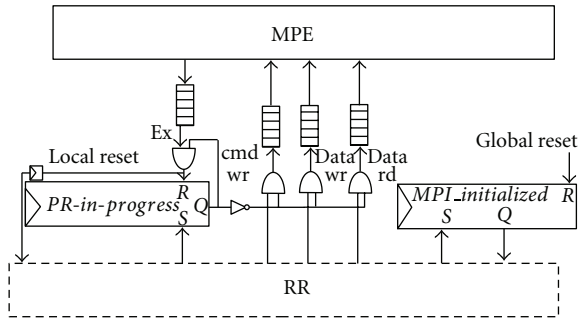

Figure 6: Static registers and decoupling logic between SR and RR.

However, a problem with the approach is that it will not work if the SRL placement is not properly constrained. We believe that partial reconfiguration of an entire RR does not happen atomically, which means that not all the logic of an RM is configured at once, but frame by frame. A PR frame is the smallest reconfigurable building block. A PR frame is 1 CLB wide by 16, 20, or 40 CLBs high for Virtex 4, 5, and 6, respectively. If the reset SRL is configured before the rest of the RM logic is configured then the local reset will be applied to old logic from the previous RM. The actual time it takes to complete PR depends on how fast the partial bitstream is transferred from host memory to the ICAP or SelectMAP interfaces, which in turn depends on what communication mechanism is being used (PCIe, FSB, Ethernet, etc.). In any case, a good solution should not rely on configuration time to be correct.

A possible solution to this problem is to force the CAD tools to place the SRL in the last LUT of the last frame within the RR to be configured as shown in Figure 5. This way we ensure that all the logic from the RM has been configured before the reset SRL and that the local reset will apply to the new RM logic. However, the caveat of this approach is, that to the best of our knowledge, there is no official documentation from Xilinx regarding how PR is done over the RR. For example, Figure 5 assumes that PR follows a top-down and left-right approach, but it could be otherwise.

To avoid the previous problems, the approach used to generate a local reset is to take advantage of the existing message-passing capability and the MPE to produce an external signal to the RM. This approach has the added benefit being platform independent. After an RM receives the configuration message ("cfg") and before PR takes place, the current RM can instruct the MPE to be ready to receive the "configuration-done" message, which will come eventually. The RM's FSM then goes into an idle state waiting for PR to occur. After PR is done and the new RM has been completely configured, the X86 processor sends the "configuration-done" message, which the MPE is already expecting. This causes the MPE to inform the RM of the newly arrived message by writing a control word to the command FIFO between the MPE and the RM. The exists signal (Ex in Figure 6) of the command FIFO is then used to generate the reset pulse for the new RM. After coming out of reset the RM can dequeue the control word and start running, or resume its operation by restoring its previous status (if any) as mentioned in Section 4.1.

There is an additional requirement for RMs that use ArchES-MPI when coming out of reset. The MPI_Init function must be called only once by all ranks at the beginning of the application. This function initializes the MPI environment and in the case of a hardware rank the function initializes the MPEs. This means that after coming out of reset an RM must know whether it is the first time the application is running, and therefore the MPE needs to be initialized, or if the MPE has been initialized already by a previous RM. In the later case the RM must not go through the initialization process again. Instead, it must continue its execution and restore its status (if any) using the "configuration-done" message.

A simple solution for this is to add a set-reset flip-flop to the SR that indicates that the MPE has been initialized. This flag, shown as *MPI_Initialized* in Figure 6, must be set by the first RM to run in the FPGA during the initialization process after global reset. Successive RMs should check this flag to know whether or not to do the initialization process. This flag is analogous to a static variable within a software function that is initialized only once during the first time the function is called.

4.3. Decoupling Logic. Another consideration when using PR is the decoupling between SR and RR logic because the outputs of the RR may be undefined during PR. This may cause undesired reads or writes to FIFOs with invalid data. To this end, the synchronization messages can be used to set and clear a flag to gate the outputs of the RM as shown in Figure 6. When the RM receives the PR message ("cfg" in Figure 4) it sets the *PR-in-progress* flag, which is a set-reset flip-flop placed in SR that will disable the RR outputs during PR. After PR is done, the flag is cleared when the "configuration-done" message is received. This is the same message used to generate the local reset as previously discussed. The decoupling logic occupies very little resources: 158 flip-flops and 186 LUTs per RR. With one RR this means less than 1% of the resources available in the XC5VLX330 FPGA.

5. Development Platform

Portability is one of the benefits of using a communications abstraction layer such as ArchES-MPI, hence adding partial reconfiguration must be done in a portable way as well. ArchES-MPI has been used in a variety of platforms, but the focus here will be on the one shown in Figure 7 as it provides a heterogeneous platform. It is based on a quad-socket motherboard with one Intel Quad-core Xeon processor and the other sockets have Nallatech FSB-socket accelerator modules [15]. The accelerator module consists of a stack of up to three PCB layers. The first layer (bottom-up) contains the base FPGA (XC5VLX110) and it is reserved as a connection interface between the front side bus (FSB) and the upper layers. The second and third layers contain two XC5VLX330 FPGAs each. The FPGAs are directly connected by LVDS buses. Figure 7 also shows a Xilinx evaluation board XUPV5LX110T with a PCIe-x1 link.

The FPGA on the FSB base module drives the SelectMAP interface of the FPGAs in the upper layers to perform PR. In contrast, the PCIe board uses the ICAP port as there is only one FPGA. The FSB_Bridge and PCIe_Bridge are hardware components that are used by the NoC to handle the communications over their respective interfaces and move data packets from the FPGA to shared memory and vice-versa. The bridges are also in charge of receiving the partial bitstreams and writing them to the appropriate configuration interface: ICAP or SelectMAP.

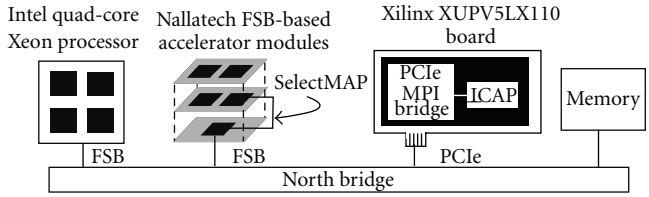

Figure 7: Development platform with FSB and PCIe-based FPGA boards.

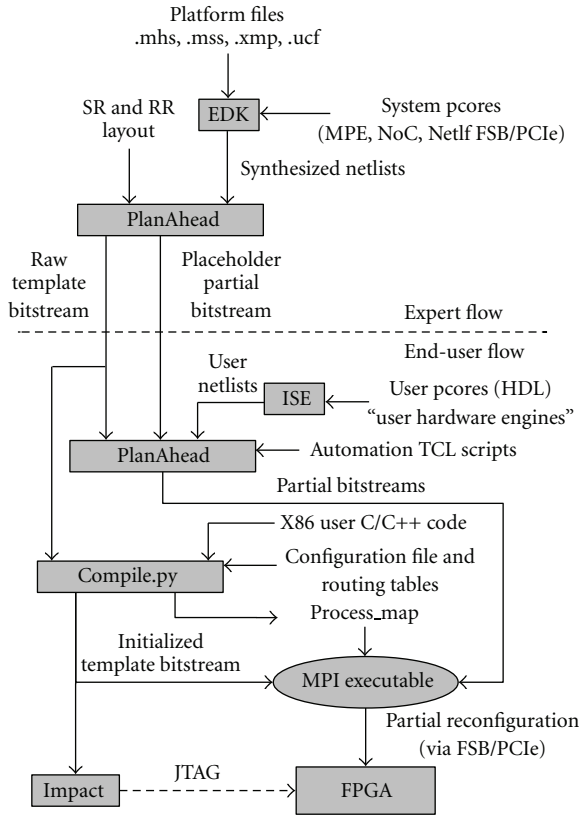

Figure 8: Expert and end-user software flows to generate template bitstreams, partial bitstreams, and the message-passing executables.

6. Software Flow

For the purposes of this paper, which is to incorporate PR into our existing MPI infrastructure, a standard Xilinx flow is followed with some extensions to it. However, any other PR flow that can generate and handle full and partial bitstreams could potentially be used as well.

There are two parts of the CAD flow shown in Figure 8: the first flow is used to create the template bitstreams by partitioning the design into static and reconfigurable regions. The second part of the flow allows the end user to reuse the prebuilt, prerouted template design. The Xilinx EDK/ISE 12.1 suite is used to implement and synthesize the system-level design including the NoC and message-passing infrastructure (system components). The netlists are then used in PlanAhead to layout the RRs, which requires certain expertise. With the new Xilinx Partition-based flow there is no need to explicitly implement bus macros. PlanAhead will automatically insert LUT1 elements where they are required. Finally, the full template bitstream is created and can be distributed to users. Additionally, the partial bitstream of a placeholder RM can also be created. The placeholder (or bootloader RM) is discussed in Section 8.1.

The end user can implement hardware engines as Xilinx pcores in an EDK project. The engines can be coded as HDL or via C-to-Gates tools and synthesize them. The resulting netlist can then be integrated into the PlanAhead project where the predesigned SR is reused and it is only necessary to place and route the RMs and generate user partial bitstreams. The end user does not have to do any layout, but PlanAhead is still necessary to generate the partial bitstreams. However, additional automation TCL scripts can help in simplifying the task of using PlanAhead. The last step is to run an ArchES-MPI Python script (compile.py in Figure 8) to compile the MPI code for all the processors (X86, MicroBlaze, PowerPC) and initialize the BRAMs for those embedded processors in the FPGA. The script also initializes the routing tables for the NoC and generates a hosts file required to run the application.

This host file (process_map in Figure 8) in a typical MPI implementation is known as a machine host file and it is used to specify the host, the type of processor and the executable binary assigned to each MPI rank. The ArchES-MPI library uses a similar concept but it is extended to use bitstreams. This file has two sections, the first section assigns each rank to a host, a processing element, and a binary to execute. If it is a processor, then the binary to execute is an

```
main()
MPI_Init(···);//implicit config. of SRs and initial RMs
...
MPI_Send(···, destRank, CFG_MSG_TAG···);
MPI_Recv(statusData_RM_A,..., destRank,OK_TO_CFG_TAG,···);
ARCHES_MPI_Reconfig(RM_B.bit,boardNum,fpgaNum);
MPI_Send(statusData_RM_B,..., destRank, CFG_DONE_TAG,···);
...
```

Figure 9: Partial reconfiguration code snippet.

ELF file generated by the compiler (e.g., gcc) for X86, or cross-compiler for embedded processors (e.g., mb-gcc). If it is a hardware engine, the binary to execute is the partial bitstream file generated by PlanAhead. The second section assigns the full template bitstreams (as opposed to partial bitstreams) to the FPGAs available in the platform.

The template bitstreams must be downloaded first. For multi-FPGA systems such as the Nallatech FSB-based accelerator modules, there can be many template bitstreams and they all can be downloaded at runtime via FSB. For the PCIe board, JTAG or Flash memory are the only options to configure the FPGA at boot time. Once the template bitstreams are in place, the partial bitstreams can be downloaded as many times as necessary using the FSB or PCIe link without the need to reboot the host machine.

The ArchES-MPI library parses the configuration file at runtime and during the execution of the `MPI_Init()` function the template bitstreams (if there are more than one) are downloaded to the FPGAs. This is completely transparent to the user. After this, the user can explicitly download partial bitstreams by calling the `ARCHES_MPI_Reconfig()` function (as shown in Figure 9) within the source code. This function uses three parameters: the partial bitstream filename, the FPGA board number (if there are multiple FPGA boards), and the FPGA number (in case a given board has many FPGAs). The partial bitstream file includes information that determines which RR to use. In addition, the user can send and receive the synchronization messages described in Section 4. The code uses MPI message tags (user defined) to distinguish the configuration messages from the application messages. Note that the same code can be used regardless of the type of communication interface (PCIe or FSB) or the type of configuration interface (ICAP or SelectMAP); it is all abstracted away from the user.

7. The Vector Engine Example

The overall performance of a parallel application programmed using message passing depends on many different factors, such as communication-to-computation ratio, memory access times, and data dependencies, among others. When using PR, additional factors must be included, such as the partial bitstream size, the number of RRs, and the PR granularity of the application, which is the ratio between the amount of computation per PR event. It is beyond the scope of this paper to provide a complete evaluation of PR overhead by exploring all the possible parameters. Instead,

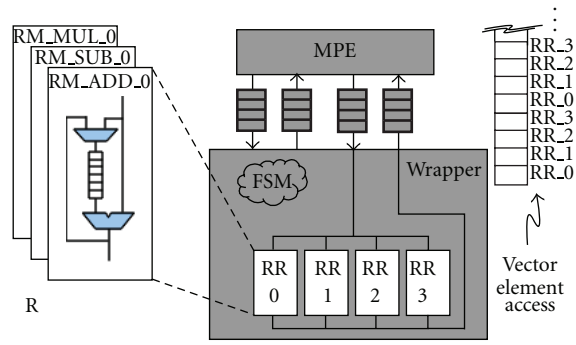

Figure 10: Vector Engine with 4 functional units (RRs) and three RMs.

we focus on measuring the PR overhead relative to the PR granularity. A simple vector engine example is used to verify that PR works in real hardware and to provide initial overhead measurements; it is not intended to be a fast vector operation accelerator.

This is an example of a wrapper-contained scenario (see Figure 3) because application logic is placed in the static region. The vector engine core is shown in Figure 10. The vector elements are distributed across the four functional pipelines of the engine (i.e., stride access of four). Each pipeline is an RM that can dynamically change the operation to perform (add, subtract, or multiply). Also, each pipeline includes an FIFO to store the vector elements.

The wrapper issues Send and Receive commands to the MPE and the data goes directly in and out of the pipelines. The wrapper decodes commands sent as messages from the master rank (Rank 0), which is the X86 processor; the vector engine is Rank 1. The vector commands can be LOAD (load a vector into the accumulator), ACC (perform the operation and store the results back into the accumulator), and STORE (flush the accumulator and send it back to the master rank). The wrapper controls the pipeline via control signals (e.g., enable, done, command, and vector_size).

There are four RRs and only one Vector Engine (i.e., one slave rank). The partial bitstream size is 130872 bytes per RR, and the vector size is set to 4000 floating point numbers. The experiment consists of performing 100 LOAD-ACC-STORE (LAS) cycles and performing PR every certain number of cycles. This setup allows us to control the amount of communication and computation per PR event. Note that the point of the experiment is to measure the overhead for a given setup and not to compare FSB and PCIe performance. The results are shown in Table 1. The worst-case scenario is when PR is performed every LAS cycle (i.e., fine PR granularity) with a degradation of 425-fold for PCIe and 308-fold for FSB compared to the case where no PR is performed at all. The best-case scenario is when PR is performed only once (i.e., coarse PR granularity) for the entire run with an overhead factor of 5 and 4 for PCIe and FSB, respectively. From those numbers, only 0.5% of the time is spent in the synchronization messages, the rest is spent in

Table 1: PR overhead at different PR event granularities.

Num of PR events	PCIe		FSB	
	Exec. time (s)	Times slower	Exec. Time (s)	Times slower
100	18.695	425	6.466	308
10	1.912	43	0.666	32
1	0.231	5	0.085	4
no PR	0.044	—	0.021	—

transferring the partial bitstreams. Note that there are four of them, one for each pipeline (RR).

These numbers are important to measure further improvements to the PR infrastructure, but they are not representative of the performance degradation expected of any application that uses this PR framework. Even for the same application, the overhead is less for a smaller partial bitstream, which is proportional to the size of the RR. The RR was drawn to an arbitrary size and the pipeline only uses 13% of LUTs and FFs, and 25% of BRAMs and DSPs of the RR. There was no attempt to minimize the RR size or to compress the partial bitstream. The MPE, NoC, and other infrastructure use 11044 LUTs (15%), 15950 FFs (23%), 80 BRAMs (54%), and 3 DSPs (4%) of the static region on the XC5VLX110.

The FSB-based board has less runtime overhead than the PCIe-based board because the FSB has less latency and more bandwidth than the PCIe-x1 link. Also, the FPGA in the FSB board runs at 133 MHz while the FPGA on the PCIe board runs at 125 MHz.

8. Generic Computing Platform Example

The vector accumulator is an application-specific template example because the wrapper implemented in the SR is part of the application. This section presents an example of a generic template bitstream that can be reused by four different applications, therefore, no application-related logic can be placed in the SR. In any case, the objective is not to show hardware acceleration but to prove functionality and that the same template bitstream is reused by four different applications. All of this is done without the user having to design the communications infrastructure or dealing with low-level hardware details.

Figure 11 shows the analogy between software and hardware binaries. The software binaries (.elf files) are loaded by the operating system and executed by X86 processor cores, and the partial bitstreams (.bit files) are loaded by the ArchES-MPI library and *executed* by the FPGA fabric within the RRs. In this case, PR is used to download the corresponding hardware engine to the FPGA at the beginning of each application. Figure 11 shows four RRs, this means that we could instantiate up to four hardware engines of the same application. Technically, it is also possible to have hardware engines of different applications in different RRs. However, our goal is to test that the same RR can be reused by different hardware engines.

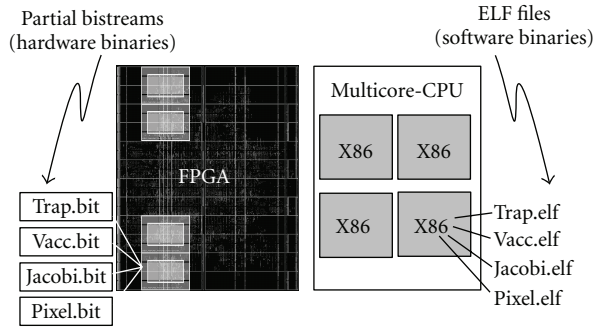

Figure 11: Mapping binaries to a generic template bitstream and X86 multicore CPU.

The first application is the same vector accumulator discussed in Section 7 but now in a self-contained form. This means the wrapper is now also in the RR, not in the SR as before. This implies some minor changes to the wrapper's FSM, basically it was necessary to remove the functionality to generate the local reset to the floating point pipelines. Now the reset is generated as described in Section 4.2, external to the wrapper using the MPE and messages. Now the wrapper itself needs to be reset as well. The same reset mechanism is used for all four different application engines. The vector accumulator is hand coded in VHDL.

The second application computes the area under the curve for a given function using the trapezoidal method. This application is based on the software MPI parallel implementation presented in Pacheco [16]. Impulse-C [17], a C-to-Gates compiler, is used to generate the hardware engine. With the addition of some macros introduced in previous work [18], it is possible to translate MPI calls in the C code to MPE commands in VHDL. This application requires at least two ranks but is designed to use more. In this case, one software rank and one hardware rank; both ranks perform the same computation on different data.

The third application is a simple Jacobi iteration method for approximating the solution to a linear system of equations to solve the Laplace equation with finite differences. The implementation of this application is based on previous work [19], and it was written in C and MPI for embedded processors. This application uses three ranks, one master and two slaves. The master and one slave are software ranks, and the other slave is a hardware rank. Again, Impulse-C is used to generate the hardware engine with commands to the MPE.

Finally, the fourth application is another simple MPI program written in C to invert pixels from image frames coming from a webcam or a file stored in the host's hard drive. This application uses OpenCV [20], an open source library for computer vision that provides an easy way to access the Linux video device and to open, read, and write video and image files. This application uses three ranks, Rank 0 (X86) captures the frames and send them to Rank 1 (hardware engine) where the pixels are inverted and then it sends the processed frame to Rank 2 (X86) where the frame is stored in a file or displayed on the screen. The hardware engine is hand coded in VHDL.

Although there has not been any attempt to optimize the size and placement of the RRs, the FPGA in Figure 11 has been partitioned following an educated guess. This guess comes from the experience of having to design the low-level LVDS communication interface for the FSB-based platform (See Figure 7), which is the one used in this example. The FPGA in this platform has abundant LVDS I/O logic to communicate with other FPGAs above, sideways, and below. This logic is placed and concentrated towards the middle of the chip closer to specific I/O banks. Therefore, placing the RR around the middle of the chip would complicate timing closure because the RR would displace the LVDS logic creating longer paths and delays. Additionally, the XC5VLX330 FPGA only has two columns of DSP blocks on the left side of the chip—it is not symmetrical, therefore, there are no RRs on the right side for this particular template. A different template may include DSP-enabled and non-DSP-enabled RRs. In any case, these are exactly the kind of decisions that the end user, that is, the application developer, should not need to worry about. The place where the I/O banks or DSPs are located is completely irrelevant to the trapezoidal method or the Jacobi algorithm. Instead, the template bitstream has already been created by a hardware engineer familiar with the platform who can layout the RRs for the user to work with.

The four applications were compiled, implemented, and run on the FSB-based platform and all the output results were correct for each application. The RR itself occupies 6.8% of LUTs and FFs, 4.9% of BRAMs, and 16.6% of DSPs available in the XC5VLX330 FPGA. The partial bitstreams for the four application RMs have the same size (477666 bytes), although they do not require the same resources. Table 2 presents the utilization percentage with respect to the RR not the entire FPGA. In this case, all the engines fit within that RR. If this would not have been the case then another template with a larger RR would have been selected. The decision of what template bitstream to use from a collection of them can be done by a higher-level tool that matches the resources required by an application and resources available in a template. This is an optimization problem that can be addressed in future research.

8.1. Boot Loader Reconfigurable Module. When a template bitstream is downloaded to an FPGA each RR must have an initial RM configured. The question is what RM should be placed at the beginning. In our generic platform a simple placeholder RM has been created to be the initial RM. This placeholder is in fact a small bootloader for partial bitstreams that merely follows the synchronization steps described in Section 4. It instructs the MPE to receive a PR message ("cfg" message in Figure 4) indicating that a new RM is going to be configured. The bootloader RM will then issue another receive command to the MPE for the eventual "configuration-done" message, then it disables all the outputs from the RR by setting the *PR-in-progress* flag and finally it will go to an idle state waiting for PR to occur. Once the application RM is configured and out of reset the application begins. Just before the application is about to finish, during the call to the MPI_Finalize function (all ranks must call it) the bootloader RM is restored in all the RRs to set the FPGA ready for the next application. This last step is analogous to a software process that finishes and returns the control of the processor to the operating system. Similarly, when the application RM finishes it returns the control of the MPE to the bootloader RM.

Table 2: Resource utilization of four application hardware engines (Percentages are relative to the RR).

	RR	Pixel	Trapezoidal	Vector acc	Jacobi
LUTs	9920	154 (2%)	6397 (65%)	1476 (20%)	5271 (54%)
FFs	9920	112 (2%)	3993 (41%)	1553 (16%)	4460 (45%)
DSPs	32	0	32 (100%)	8 (25%)	13 (41%)
BRAM	16	0	0	4 (25%)	2 (13%)

9. Conclusions

The main contribution of this paper is the concept of embedding partial reconfiguration into the MPI programming model. We have presented an extension to the ArchES-MPI framework that simplifies the use of PR by abstracting away the hardware details from the user while providing a layer of portability. Such layer allows PR to be performed regardless of the configuration interface (ICAP or SelectMAP) and independent of the communication channel used (PCIe or FSB) to communicate with an X86 processor, which is used as a configuration controller. Also, we demonstrated a method to generate local resets without the need of an embedded processor, and provided an alternative to actively store and restore the status of reconfigurable modules via explicit messages. Very little additional logic and code were required on top of the existing message-passing infrastructure to enable PR. In addition, the concept of template-based bitstreams was introduced as a way to use this PR framework to create reusable and generic computing platforms. Four different applications were actually implemented and executed consecutively reusing the same template bitstream. An approach like this simplifies and speeds up the design process of creating multiaccelerator-based computing systems by allowing third-party hardware experts to provide prebuilt accelerators, I/O interfaces, NoC, and memory controllers, letting the designer focus on the actual value-added application logic. The initial performance overhead numbers obtained set a reference point to measure future improvements to the PR framework.

Acknowledgments

The authors acknowledge the CMC/SOCRN, NSERC, Impulse Accelerated Technologies, and Xilinx for the hardware, tools, and funding provided for this paper.

References

[1] Xilinx, Inc., "Partial Reconfiguration User Guide," http://www.xilinx.com/support/documentation/sw_manuals//xilinx12_2/ug702.pdf.

[2] M. Saldaña, A. Patel, H. J. Liu, and P. Chow, "Using partial reconfiguration in an embedded message-passing system," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig '10)*, pp. 418–423, December 2010.

[3] ArchES Computing, Inc., http://www.archescomputing.com.

[4] M. Saldaña, A. Patel, C. Madill et al., "MPI as an abstraction for software-hardware interaction for HPRCs," in *Proceedings of the 2nd International Workshop on High-Performance Reconfigurable Computing Technology and Applications (HPRCTA '08)*, pp. 1–10, November 2008.

[5] P. Lysaght, B. Blodget, J. Mason, J. Young, and B. Bridgford, "Invited paper: enhanced architectures, design methodologies and CAD tools for dynamic reconfiguration of Xilinx FPGAS," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 1–6, August 2006.

[6] O. Diessel, H. ElGindy, M. Middendorf, H. Schmeck, and B. Schmidt, "Dynamic scheduling of tasks on partially reconfigurable FPGAs," *IEE Proceedings: Computers and Digital Techniques*, vol. 147, no. 3, pp. 181–188, 2000.

[7] C. Bobda, A. Ahmadinia, M. Majer, J. Teich, S. Fekete, and J. van der Veen, "DyNoC: a dynamic infrastructure for communication in dynamically reconfigurable devices," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 153–158, August 2005.

[8] K. Papadimitriou, A. Anyfantis, and A. Dollas, "An effective framework to evaluate dynamic partial reconfiguration in FPGA systems," *IEEE Transactions on Instrumentation and Measurement*, vol. 59, no. 6, pp. 1642–1651, 2010.

[9] C. Rossmeissl, A. Sreeramareddy, and A. Akoglu, "Partial bitstream 2-D core relocation for reconfigurable architectures," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS '09)*, pp. 98–105, August 2009.

[10] H. Kalte and M. Porrmann, "Context saving and restoring for multitasking in reconfigurable systems," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 223–228, August 2005.

[11] D. Koch and J. Teich, "Platform-independent methodology for partial reconfiguration," in *Proceedings of the 1st Conference on Computing Frontiers (CF '04)*, pp. 398–403, ACM, New York, NY, USA, 2004.

[12] H.-H. So, A. Tkachenko, and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," in *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '06)*, pp. 259–264, October 2006.

[13] L. Möller, R. Soares, E. Carvalho, I. Grehs, N. Calazans, and F. Moraes, "Infrastructure for dynamic reconfigurable systems: choices and trade-offs," in *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design (SBCCI '06)*, pp. 44–49, ACM, New York, NY, USA, 2006.

[14] The MPI Forum, "MPI: a message passing interface," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, pp. 878–883, ACM, New York, NY, USA, November 1993.

[15] Nallatech, http://www.nallatech.com/.

[16] P. S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann, 1997.

[17] Impulse Accelerated Technologies, http://www.impulseaccelerated.com/.

[18] A. W. House, M. Saldaña, and P. Chow, "Integrating high-level synthesis into MPI," in *Proceedings of the 18th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM '10)*, pp. 175–178, May 2010.

[19] M. Saldaña, D. Nunes, E. Ramalho, and P. Chow, "Configuration and programming of heterogeneous multiprocessors on a multi-FPGA system using TMD-MPI," in *Proceedings of the IEEE International Conference on Reconfigurable Computing and FPGA's, (ReConFig '06)*, pp. 1–10, September 2006.

[20] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.