ON-CHIP SOFTWARE TOOLS FOR HARDWARE MULTITASKING ON PARTIALLY RECONFIGURABLE FPGAS

By

AURELIO FEDERICO MORALES VILLANUEVA

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2015

© 2015 Aurelio Federico Morales Villanueva

To my mother Lida, I wish you were here with us, and to my father Elfren

ACKNOWLEDGMENTS

This work was supported by the Unidad Coordinadora del Programa de Ciencia y Tecnología (FINCyT), Perú, under contract N° 121-2009-FINCyT-BDE and in part by the I/UCRC Program of the National Science Foundation, under grants EEC-0642422 and IIP-1161022. The author gratefully acknowledges the support of Universidad Nacional de Ingeniería – Lima, Perú, the Presidencia del Consejo de Ministros in Perú, through FINCyT, and the tools provided by Xilinx.

Special thanks to Dr. Alan George, director of the NSF Center for High-Performance Reconfigurable Computing (CHREC), who gave me the chance to be part of this prestigious center as a research volunteer. And finally, I would like to thank my Ph.D. advisor, Dr. Ann Gordon-Ross, for all of her guidance and support throughout the last several years.

TABLE OF CONTENTS

page

ACKNOWLEDGMENTS 4

LIST OF TABLES 7

LIST OF FIGURES 9

ABSTRACT 11

CHAPTER

1 INTRODUCTION 13

2 BACKGROUND AND RELATED RESEARCH 19

2.1 Previous Work 19
2.2 Virtex-5 FPGA architecture 23
2.2.1 Device Layout and Resources 24
2.2.2 Device Configuration 25
2.2.3 Cost Model for Partial Bitstream Size 29

3 ON-CHIP CONTEXT SAVE AND RESTORE, AND HARDWARE TASK RELOCATION SOFTWARE 33

3.1 CSR and HTR Overview 34
3.2 Context Save (CS) of Hardware Tasks 35
3.3 Merge of Initial Bitstream and Saved Context 37
3.4 Saved Context Relocation of Hardware Tasks (HTR) 38
3.5 Context Restore (CR) of Hardware Tasks 40
3.6 CSR and HTR Portability across FPGA Device Families 41
3.7 Summary 45

4 ON-CHIP DISTRIBUTED DYNAMIC RESOURCE MANAGEMENT SOFTWARE . 49

4.1 DDRM Concepts and Definitions 50
4.2 DDRM Overview 56
4.3 DDRM Operations 58
4.4 Summary 68

5 EXPERIMENTAL RESULTS 75

5.1 Experimental setup 75
5.2 CSR Experimental Results 81
5.3 HTR Experimental Results 84
5.4 DDRM Experimental Results 88

6 CONCLUSIONS 106

REFERENCES.. 109

BIOGRAPHICAL SKETCH.. 114

LIST OF TABLES

Table page

2-1 Parameters used in the partial bitstream size cost model 31

2-2 Specific values from Table 2-1 for Virtex-4/5/6 FPGA device families 32

3-1 Truth table for the CSR merge process 46

4-1 Global table for local and remote PRM relocation for node consistency in DDRM with two nodes 70

4-2 Local table for node consistency in DDRM showing the currently assigned PRMs in node "1" from Table 4-1 70

5-1 CS bitstream and partial bitstream sizes (in KB) used in the CSR, HTR, and DDRM experiments 93

5-2 Execution times (ms) for $T_{reconfig_prr}$ 93

5-3 Execution times (ms) for CS (T_{cs}) in CSR 94

5-4 Execution times (ms) for the merge process (T_{merge}) in CSR 94

5-5 Execution times (ms) for CR (T_{cr}) in CSR 94

5-6 Execution times (ms) for CS (T_{cs}), context relocation ($T_{relocate}$), and CR (T_{cr}) for small-to-large PRR HTR 95

5-7 Execution times (ms) for CS (T_{cs}), context relocation ($T_{relocate}$), and CR (T_{cr}) for large-to-small PRR HTR 95

5-8 DDRM execution times (ms) for T_{exe1} with respect to the number of PRM flip-flops. 95

5-9 DDRM execution times (ms) for T_{res1} with respect to the number of PRM flip-flops. 96

5-10 DDRM execution times (ms) for T_{exe2} with respect to the number of PRM flip-flops (part 1 of 2) 96

5-11 DDRM execution times (ms) for T_{exe2} with respect to the number of PRM flip-flops (part 2 of 2) 96

5-12 DDRM execution times (ms) for T_{res3} with respect to the number of PRM flip-flops (part 1 of 2) 97

5-13 DDRM execution times (ms) for T_{res3} with respect to the number of PRM flip-flops (part 2 of 2) 97

5-14 DDRM execution times (ms) for T_{exe4} with respect to the number of PRM flip-flops (part 1 of 2) 97

5-15 DDRM execution times (ms) for T_{exe4} with respect to the number of PRM flip-flops (part 2 of 2) 98

5-16 DDRM execution times (ms) for $T_{exeres2}$ with respect to the number of PRM flip-flops (part 1 of 2) 98

5-17 DDRM execution times (ms) for $T_{exeres2}$ with respect to the number of PRM flip-flops (part 2 of 2) 98

LIST OF FIGURES

Figure page

2-1 Virtex-5 LX110T FPGA fabric layout with four sample PRRs 32

2-2 Partial bitstream structure for Virtex-5 FPGAs 32

3-1 On-chip context save and restore (CSR) and hardware task relocation (HTR) flows 47

3-2 Multiple flip-flop updates for CSR merge process 47

3-3 Single flip-flop update for context relocation (HTR) process 47

3-4 Multiple flip-flop updates in a word boundary for context relocation (HTR) process 48

4-1 Portion of the distributed dynamic resource management (DDRM) flow showing the first time execution (or resumption) of a PRM prm_{ijq} in a predefined PRR prr_{ij} 71

4-2 Details of the DDRM flow for the steps performed in the boxes with vertical lines in Figure 4-1 for the first time execution (or resumption) of prm_{ijq} in predefined prr_{ij}. 72

4-3 DDRM flow continued, showing the first time execution (or resumption) of a locally relocated prm_{ijq} in a local candidate prr_{it} as prm_{itq} 72

4-4 Detailed steps of the DDRM flow showing the steps performed in the boxes with horizontal and vertical lines from Figure 4-3 for the first time execution (or resumption) of a locally relocated prm_{ijq} in a local candidate prr_{it} as prm_{itq} 73

4-5 DDRM flow continued, showing the execution of the remote relocation of prm_{ijq} to a remote candidate PRR n in a node m (prr_{mn}) as prm_{mnq} 73

4-6 DRRM flow continued, showing the steps performed in the dark gray box in Figure 4-5 for execution of the remote relocation of prm_{ijq} to a remote candidate PRR n in node m (prr_{mn}) as prm_{mnq}. 74

5-1 Execution times (ms) for $T_{reconfig_prr}$ with respect to the number of PRM flip-flops 99

5-2 Execution times (ms) for CS (T_{cs}) in CSR with respect to the number of PRM flip-flops 99

5-3 Execution times (ms) for the merge process (T_{merge}) in CSR with respect to the number of PRM flip-flops 100

5-4 Execution times (ms) for CR (T_{cr}) in CSR with respect to the number of PRM flip-flops 100

5-5 Execution times (ms) for CS (T_{cs}) in HTR with respect to the number of PRM flip-flops 101

5-6 Execution times (ms) for context relocation ($T_{relocate}$) in HTR with respect to the number of PRM flip-flops 101

5-7 Execution times (ms) for CR (T_{cr}) in HTR with respect to the number of PRM flip-flops 102

5-8 DDRM execution times (ms) for T_{exe1} with respect to the number of PRM flip-flops 102

5-9 DDRM execution times (ms) for T_{res1} with respect to the number of PRM flip-flops 103

5-10 DDRM execution times (ms) for T_{exe2} with respect to the number of PRM flip-flops 103

5-11 DDRM execution times (ms) for T_{res3} with respect to the number of PRM flip-flops 104

5-12 DDRM execution times (ms) for T_{exe4} with respect to the number of PRM flip-flops 104

5-13 DDRM execution times (ms) for $T_{exeres2}$ with respect to the number of PRM flip-flops 105

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

ON-CHIP SOFTWARE TOOLS FOR HARDWARE MULTITASKING
ON PARTIALLY RECONFIGURABLE FPGAS

By

Aurelio Federico Morales Villanueva

August 2015

Chair: Ann Gordon-Ross
Major: Electrical and Computer Engineering

Partially reconfigurable (PR) field-programmable gate arrays (FPGAs) partition the FPGA into one static region and multiple PR regions (PRRs). This partitioning affords faster reconfiguration time, as compared to configuring the entire FPGA since the PRRs are reconfigured using smaller partial bitstreams. PR FPGAs also afford isolated reconfiguration since only the reconfigured PRR halts execution while the remainder of the FPGA continues to operate. Faster and isolated reconfiguration enable flexible hardware task multiplexing in the PRRs, and, to fully exploit this PRR time multiplexing, higher priority tasks should be able to preempt lower priority tasks, and the preempted tasks should be able to resume execution in any PRR with sufficient resources. This preemption/resumption requires saving/restoring the preempted task execution context and relocating the task to another PRR.

Some prior works address the involved challenges, but these works only provide partial solutions and impose limitations and/or overheads that prevent portability of these works across different FPGA device families. The research presented here presents a set of methods and software tools for hardware multitasking on PR FPGAs to address prior works' limitations.

First, we present on-chip context save and restore (CSR) software to enable task preemption/resumption in the same PRR, preserving the execution state of preempted hardware tasks, without disrupting operations in the static region and other PRRs in the FPGA. Second, we introduce on-chip hardware task relocation (HTR) software to enable a task execution state to be saved, and relocated to and restored in any PRR in the FPGA with sufficient resources. Finally, using our on-chip HTR software, we present on-chip distributed dynamic resource management (DDRM) for PR FPGAs to enable context relocation of hardware tasks between different physical FPGAs on an interconnected network. Experimental results evaluate CSR, HTR, and DDRM execution times, enabling designers to tradeoff task/PRR granularity and CSR/HTR/DDRM execution times based on application requirements.

CHAPTER 1
INTRODUCTION

Partially reconfigurable (PR) field programmable gate arrays (FPGAs) partition the FPGA fabric into one static region and one or more PR regions (PRRs). This partitioning enables PRRs to time multiplex hardware tasks, where incoming tasks can be scheduled to any *candidate* PRR, which is any free/idle PRR (not currently executing another task) with sufficient resources. This resource time multiplexing reduces the total FPGA area requirements and power consumption, since smaller FPGA devices can be used. Also, this partitioning can improve overall PR system performance compared to not using PRRs or using non-PR FPGAs.

PRRs are reconfigured using small partial bitstreams, which affords faster reconfiguration times as compared to using full bitstreams that are used to configure the entire FPGA. Reconfiguring a PRR is isolated, where only the scheduled PRR halts operation during the reconfiguration, while the static region and the other PRRs continue executing. This isolated PRR reconfiguration enables maximum flexibility and adherence to task priority requirements using task preemption and resumption, where higher priority tasks can preempt lower priority tasks on scheduled PRRs.

Task preemption/resumption pauses/resumes task execution by saving/restoring the task's execution state (i.e., context). Resumed tasks can be scheduled to any candidate PRR, wherein a candidate PRR now includes any PRR with sufficient resources that is currently executing a lower priority task, which will be preempted. Task preemption requires a context save (CS) operation, which reads the task's execution state from the FPGA and saves this context off-chip in a CS bitstream.

Task resumption requires a context restore (CR) operation using the previously saved CS bitstream to restore the task's execution state on the scheduled PRR. To effectively restore the task's execution state, the CS bitstream is merged with the task's initial partial bitstream (created at synthesis), where the merged bitstream is created using bitstream manipulations using bit-level and 32-bit word-level bit masking.

There exist some prior works on CS and CR—collectively referred to as context save and restore (CSR)—to the same PRR [14][22][23][24][26][35]. However, requiring that CR only be to the same PRR is too restrictive, and hardware tasks should be able to resume execution in any candidate PRR. Hardware task relocation (HTR) alleviates this restriction by relocating and resuming task execution in any candidate PRR, which affords improved performance/task throughput, and maximizes resource usage, especially for application domains such as dynamic load balancing of hardware tasks and task migration across different network-connected FPGAs (e.g., local wired network, wireless network, etc.).

HTR is more challenging than CSR because the relocation and resumption of a preempted task is a complex process. HTR is relatively easy when relocating tasks between homogenous PRRs (same size, shape, and resource distribution, but different fabric location), since the bitstream manipulation is relatively simple [26][29]. However, relocating tasks between only homogeneous PRRs impose many restrictions [2][3][11] that prevent portability across different FPGA device families. HTR for task relocation between different-sized heterogeneous PRRs (different shape, resource distribution, and fabric location) is significantly more challenging.

Prior works addressed some of the HTR challenges, such as bitstream relocation (BR) between homogeneous PRRs [4][7][8][9][10][16][30][36][39][41][42] or between same-sized heterogeneous PRRs [1][2]. However, BR does not save/restore the task's current execution state, and as a consequence, the task must be restarted when the task is relocated and resumed in another PRR, which introduces performance-degrading re-execution overhead that may involve seconds/minutes to re-generate data. We note that these prior works on HTR/BR do not address task relocation between different-sized heterogeneous PRRs.

Prior works on HTR evaluated off-chip and on-chip implementations. In off-chip HTR between homogeneous PRRs [26][29], the FPGA is attached to a host CPU that executes software-based HTR, imposing a dedicated communication bus between the host CPU and the FPGA, which introduces reconfiguration time overhead due to this communication. Additionally, this host CPU-FPGA dedicated communication bus prevents autonomous HTR implementation (i.e., the FPGA *must* have an attached CPU, and cannot execute autonomously in remote environments).

To enable autonomous HTR and eliminate this CPU-FPGA communication overhead, on-chip HTR can be implemented entirely on the FPGA either using custom hardware [3][17][18] or software running on a soft-core processor in the static region, or a combination of these two solutions [19][20]. Custom hardware on the FPGA generates area overhead, imposes device-specific constraints that limits portability across different FPGA device families, and reduces the task’s maximum operating frequency. Alternatively, on-chip software-based HTR enables system portability across different FPGA device families, does not generate area overhead, does not impose device-

specific constraints, and does not reduce the task's maximum operating frequency. We note that, to the best of our knowledge, there is no prior work on autonomous HTR that is portable across different FPGA device families that supports CR of hardware tasks between different-sized heterogeneous PRRs.

This research addresses prior CSR/BR/HTR works' limitations (e.g., area overhead due to using custom and non-portable on-chip hardware, host CPU-FPGA communication overhead, relocated task re-execution, etc.).

In the first phase of this research, we introduce on-chip CSR software for any heterogeneous PRR, where we leverage PR FPGAs' features for hardware multitasking on the same PRR, preserving the execution state of preempted hardware tasks, without disrupting operations in the static region and other PRRs, and where the on-chip CSR software executes on a soft-core processor in the FPGA's static region.

In the second phase of this research, we extend our on-chip CSR software to enable hardware multitasking between different PRRs in the same FPGA, and present on-chip HTR software for relocating hardware tasks' contexts between different-sized heterogeneous PRRs in order to maximize shared resources (PRRs) utilization, and maximize tasks throughputs.

In the third phase of this research, we further extend our HTR to function across multiple networked-interconnected FPGAs to enable dynamic CR of hardware tasks between physically-distributed devices. This extension provides additional improvements in task throughput due to more candidate PRRs, improvements in resource usage per FPGA, and vastly increases the application-domain applicability. To the best of our knowledge, no prior work proposes such a flexible HTR implementation.

We leverage our HTR for multiple interconnected FPGAs and present distributed dynamic resource management (DDRM), where each FPGA in the network is an autonomous system. DDRM can be used for application domains, such as dynamic load balancing of hardware tasks between FPGAs, distributed fault tolerant systems, and distributed, concurrent, and parallel processing of hardware tasks in a network of FPGAs.

We present a detailed description of the on-chip CSR/HTR/DDRM operations and present implementation results for a Virtex-5 LX110T with a MicroBlaze soft-core processor (used as the reconfiguration controller) running embedded Linux. Even though we show results for a specific FPGA device, since our CSR/HTR/DDRM uses the FPGA's internal configuration access port (ICAP), our CSR/HTR/DDRM software is portable (with minimum changes) to newer Xilinx device families, such as the Virtex-6, 7 series, and Zynq-7000.

Our on-chip CSR/HTR/DDRM executes on autonomous FPGA systems, which does not incur host CPU-FPGA communication overhead, does not introduce device overhead, does not impact the tasks' maximum operating frequencies, and is portable across different FPGA device families with minimum changes. Also, our CSR/HTR/DDRM maximizes PRR usage per FPGA, enabling the execution, preemption, and resumption of hardware tasks between different-sized heterogeneous PRRs without losing the preempted task's execution state, which eliminates seconds/minutes of re-execution time and reduces the task's waiting time by relocating the task to any candidate PRR on the same or a different FPGA. Additionally, our

analyses enable system designers to trade off CSR/HTR/DDRM execution times and task granularity (i.e., the task's PRR size) based on application requirements.

The organization of the remainder of this document is as follows. Background of the Virtex-5 FPGA architecture and device configuration, and related research pertaining to CSR, BR, and HTR is summarized in Chapter 2. Chapter 3 describes our on-chip CSR and HTR software, covering the major steps involved in CSR and HTR, and the portability of CSR/HTR across FPGA device families. Next, Chapter 4 discusses the details of how to extend our on-chip HTR software for multiple networked-interconnected FPGAs and presents the on-chip DDRM software. Chapter 5 then presents the experimental results of our on-chip CSR, HTR and DDRM software. Finally, Chapter 6 presents the conclusions of this research and outlines directions for possible future research.

CHAPTER 2
BACKGROUND AND RELATED RESEARCH

The background and related research in this chapter is divided into two sections. Section 2.1 discusses prior works in the areas of CSR, BR and HTR, and Section 2.2 provides a background of Virtex-5 FPGA architecture, covering the device layout and resources, device configuration, and a cost model for partial bitstream size, which enable system designers to understand the fundamentals of our proposed on-chip software-based tools for hardware multitasking on PR FPGAs.

2.1 Previous Work

There exists some prior works in CSR and of these works, few leverage PR FPGAs. Landaker et al. [31] and Simmler et al. [40] presented off-chip CSR methods where all CSR operations executed in software running on an attached host CPU. Since these works did not leverage PR FPGAs, CSR reconfigured the entire FPGA. Landaker's work reported 407 ms, 465 ms, and 365 ms for CS, bitstream manipulations, and CR execution times, respectively, for a Xilinx Virtex XCV1000 device, while Simmler's work reported 14.4 ms, 83.7 ms, and 12.4 ms for the same execution times on a Xilinx Virtex XCV400 device.

Joswik et al. [24] presented off-chip CSR methods for PRRs, as opposed to the entire FPGA, and reduced CSR times using direct memory access (DMA) for the ICAP. The results showed CS and CR execution times on the order of hundreds of microseconds, and bitstream manipulation execution times on the order of a few milliseconds for a Xilinx Virtex-4 XC4VFX60 device. Additionally, Kalte and Porrmann [26] and Koester et al. [29] extended off-chip CSR between homogeneous PRRs by incorporating an on-chip custom hardware relocator (REPLICA and REPLICA2Pro, for

Kalte's and Koester's works, respectively). Results showed CS execution times ranging from 33 μs to 1.2 ms, CR execution times ranging from 190 μs to 7.0 ms, and relocation times ranging from 0.4 ms to 15.0 ms. However, both works were implemented for older Xilinx PR-capable FPGAs (Virtex-E XCV2000E and Virtex-II XC2V4000 devices, for REPLICA and REPLICA2Pro, respectively) that only supported one-dimensional (1-D) PRRs, and thus are not applicable to newer Xilinx devices that support two-dimensional (2-D) PRRs.

To eliminate the communication overhead between the FPGA and the host CPU, Koch et al. [28], Garcia et al. [14], and Jovanovic et al. [22] presented on-chip CSR hardware solutions (for non-PR FPGAs in Koch's work, and for PR FPGAs in Garcia's and Jovanovic's works). All these works reduced CS and CR times to the order of microseconds using different versions of scan-path chains of flip-flops, a technique used in design for testability (DFT) for very large scale integrated (VLSI) circuits. Additionally, Jozwik et al. [23] presented a task-specific access structure (TSAS) method to perform CSR, which inserted custom logic for each flip-flop and the FPGA's internal memory elements for each task to perform CS and CR, achieving CS and CR execution times from hundreds of microseconds to a few milliseconds. However, even though these methods eliminated communication with a host CPU, these methods incurred significant hardware overhead, lacked portability, reduced the system's maximum operating frequency, and required changes in the design tool flow. In prior work [35], we presented on-chip software CSR, which alleviated these drawbacks, but did not relocate the task to a different PRR.

BR enables task relocation, but prior works did not relocate the task's context to a different PRR. Horta and Lockwood [16], Blodget et al. [4], and Krasteva et al. [30] presented off-chip BR software, and Kalte et al. [25][27] presented on-chip BR hardware support for off-chip BR software. All of these prior works were intended for BR between homogeneous PRRs, however, these methods still incurred the same drawbacks as off- and on-chip CSR. Horta's work presented PARBIT to enable BR on Xilinx Virtex-E devices, Blodget's work introduced XPART to enable BR on Xilinx Virtex-II and Virtex-II Pro devices, and Krasteva's work presented BITPOS to enable BR on Xilinx Virtex-II devices. Krasteva's and Horta's works reported BR execution times of approximately 500 ms, while Blodget's works did not report BR execution times. Kalte's works used REPLICA and REPLICA2Pro, establishing a custom on-chip communication bus to enable all hardware tasks to communicate to each other, and obtained BR execution times on the order of milliseconds.

Becker et al. [1][2] and Carver et al. [7] presented on-chip BR software for same-sized heterogeneous PRRs and homogeneous PRRs, respectively, however, these methods constrained the static region's logic routing from passing through the PRRs, in addition to other constraints. Whereas these constraints reduced the number of partial bitstreams to one per task, the constraints introduced area and performance overheads [7][13]. Becker's and Carver's works reported BR throughputs of up to 4.7 MB/s and 9.5 MB/s, respectively, for Xilinx Virtex-4 devices. The partial bitstreams where stored in external flash memory and in the FPGA's internal random access memory blocks (BRAMs) for Becker's and Carver's work, respectively. Both authors used the on-chip MicroBlaze soft-core processor to execute the BR software.

Corbetta et al. [8][9], Morandi et al. [36], Sudarsanam et al. [42], Dasu and Kallam [10], and Sreeramareddy et al. [41] presented custom on-chip BR hardware for homogeneous PRRs, which was orchestrated using an on-chip soft-core processor. Corbetta's and Morandi's works used 1-D BiRF (for Virtex-II Pro devices) and 2-D BiRF (for Virtex-4/5 devices), and achieved BR throughputs of 5.9 MB/s and 7.3 MB/s for 1-D and 2-D BiRF, respectively. Sudarsanam's, Dasu's, and Sreeramareddy's works used ARC for Virtex-4 devices, and used BRAMs to reduce the BR execution times. Sudarsanam's and Sreeramareddy's works achieved BR throughputs of up to 65.6 MB/s and 86.3 MB/s, respectively. All these prior works used the on-chip MicroBlaze soft-core processor to orchestrate the on-chip BR hardware implementations. Even though Santambrogio et al. [39] used the 2-D BiRF for Xilinx Virtex-4 devices, that work did not use an on-chip processor to perform BR, and resulted in an average BR throughput of 740 MB/s.

There are a few prior works that focus on on-chip HTR. Iturbe et al. [17] proposed a custom hardware communication interface (CIF) that was attached to each hardware task to enable on-chip HTR between arbitrary heterogeneous PRRs for Virtex-4 devices. Also, Iturbe et al. [18] proposed Snake, a novel technique for hardware task allocation to enable inter-task communication between hardware tasks. Iturbe et al. [19][20] leveraged their prior work with CIF and Snake to propose a reliable reconfigurable real-time operating system (R3TOS) using one MicroBlaze and three PicoBlaze on-chip soft-core processors, where inter-task communication between hardware tasks was implemented with custom hardware data relocation task (DRT) blocks. However, Iturbe's CIFs used BRAMs (which are limited in number, and location-

specific FPGA storage resources) as buffers to hold input and output values in hardware tasks, and PRRs needed to be placed close to each other and to the DRTs on the FPGA fabric, imposing area overhead and specific constraints, which prevented portability of Iturbe's on-chip HTR solution across FPGA device families.

In order to address the drawbacks of prior off- and on-chip CSR, BR, and HTR methods (e.g., high communication overhead and device resource overhead, lack of system portability, and reduced maximum operating frequency, respectively) and to improve system performance and flexibility via task preemption/resumption/relocation, we propose in this document on-chip CSR software for CS and CR on any arbitrary 2-D PRR, and on-chip HTR software for 2-D different-sized heterogeneous PRRs. The CSR and HTR software executes on a soft-core processor in the FPGA's static region and uses the ICAP for reconfiguration. As compared to similar prior works, our CSR and HTR do not incur off-chip communication overhead, do not introduce hardware/device overhead, do not impact the tasks' maximum operating frequency, has no special constraints on the PRRs and static region (e.g., as in [1][2][7][11][18][19]), and do not require tool flow changes.

2.2 Virtex-5 FPGA architecture

Since both CSR and HTR are complex processes that require detailed device knowledge, in this section we review the Xilinx Virtex-5 FPGA architecture, which will assist designers in incorporating HTR into their systems. We refer the reader to [50] and [51] for complete information on Virtex-5 device configuration and architecture, respectively.

2.2.1 Device Layout and Resources

The Virtex-5 devices support 2-D PRRs, which allows PRRs to occupy a rectangular fabric area. Prior Virtex families (Virtex, Virtex-E, Virtex-II, and Virtex-II Pro) only supported 1-D PRRs, which required PRRs to span the fabric's entire height. 2-D PRRs offers finer reconfiguration granularity and therefore, increases design flexibility and resource utilization. Virtex-5's can be configured using external interfaces, such as JTAG (serial), SelectMAP (parallel), or the internal ICAP interface (parallel).

Figure 2-1 depicts the Virtex-5 LX110T device's fabric layout, the device used in our experiments, with four sample PRRs and the following resources [51]: configurable logic blocks (CLBs), which implement combinational and sequential logic and each CLB contains eight flip-flops; random access memory blocks (BRAMs) for internal storage; input/output blocks (IOBs), which are mainly used to provide external connections between the FPGA and other devices; digital signal processing blocks (DSPs) implement complex arithmetic functions that cannot be efficiently implemented using CLBs [52]; and clock resources (CLK) provide clock signals to all resources. Each CLB contains one slice pair. Each slice contains four logic-function generators (look-up tables or LUTs), four flip-flops, multiplexers, and carry logic. Slices denoted as SLICEMs support storing data using distributed RAM (or LUTRAM) and shifting data with 32-bit registers, and SLICELs do not support these functions. Across the entire device fabric, 50% of the CLB columns contain one SLICEM and one SLICEL, and the other 50% of the CLB columns contain two SLICELs.

As exemplified in Figure 2-1, the resources are distributed in the device fabric in a row/column organization. The device is logically divided into two halves—the top and bottom—and each half contains four rows and each row contains the same number of

columns. Each column contains a group of frames and the number of frames per column depends on the resource type (CLB, BRAM, etc.). A frame is the minimum unit of information used to write/read to/from the device. For Virtex-5 devices, a frame contains 41 32-bit words.

With respect to HTR, partial regions PRR1 and PRR2 are homogeneous since these PRRs have the same size (same number of columns and rows) and the same resources (CLBs), even though the PRRs' locations in the device fabric are different. Alternatively, partial regions PRR3 and PRR4 are heterogeneous since these PRRs have different sizes, resources, and device fabric locations. A special case of homogeneous PRRs would be if two PRRs have the same resources (CLBs, BRAMs, DSP, etc.), both PRRs begin at the same column and finish at the same column, have the same number of rows, but have different vertical positions. The minimum PRR size on the Virtex-5 is one row and one column.

2.2.2 Device Configuration

A Xilinx PR FPGA can be configured using full or partial bitstreams, which are used to configure the entire device or only a single PRR, respectively. The bitstream's configuration information is organized in configuration frames and is stored in the FPGA's internal configuration memory. A configuration frame establishes the configuration of the resources of a specific column and the routing information to access the resources. For the Virtex-5 device family, CLB, BRAM, DSP, IOB, and CLK columns have 36, 30, 28, 54, and 4 configuration frames, respectively [50]. Additionally, each BRAM column requires 128 data frames for initialization.

Full configuration of the device requires sequential execution of three phases: the setup phase, the bitstream loading phase, and the startup sequence phase [50]. While

the configuration frames are downloading, the device continuously calculates the cyclic redundancy check (CRC) value. After downloading all of the configuration frames, the device verifies the CRC by comparing the calculated CRC with the bitstream's expected CRC, which is included as part of the bitstream. If the CRCs match, the startup sequence phase begins, which initializes the device's flip-flops and BRAMs, and the device enters the *user* mode by asserting the internal end of startup (EOS) signal and the external DONE pin.

Once the device is in user mode, partial reconfiguration of a PRR can be performed using the ICAP. PRR reconfiguration starts from the bitstream loading phase, and the startup sequence phase is not executed since the device is already in user mode, and the reconfigured PRR's flip-flops and BRAMs are reconfigured provided that CRC verification was successful. Since PRRs may contain flip-flops, BRAMs, and DSPs, dedicated clock gating (using buffer BUFGCE) is used for each PRR to enable changes in these resources for each clock transition. Since the FPGA has only one ICAP, it is not possible to perform more than one ICAP operation (read or write) at a time.

Future re-initialization of flip-flops and BRAMs can be forced by toggling the internal global set reset (GSR) signal using the Xilinx user primitive STARTUP_VIRTEX5 in order to execute the startup sequence phase if the CRC verification is skipped [34]. However, since toggling the GSR would re-initialize the entire device with the flip-flops' and BRAMs' initial values as defined in the full bitstream, a protection/unprotection mechanism for the static region and PRRs must be

provided [34]. This mechanism avoids/allows future re-initialization of flip-flops and BRAMs when the GSR is toggled.

Since CSR/HTR leverages the ICAP, all partial bitstreams need to be 32-bit word aligned. Figure 2-2 depicts the initial partial bitstream structure used in CSR/HTR for the Virtex-5. This partial bitstream is the same as the bitstream that is generated by the Xilinx tools, with the exception that the initial comments, which include the name of the native circuit description file (*.ncd) that was used to generate the bitstream and the bitstream's creation date, are extracted, resulting in a 32-bit word aligned file to be used with the ICAP. The initial partial bitstream consists of a sequence of initial words, including the bus width words (0x000000BB and 0x11220044), the synchronization word (0xAA995566), a sequence of initial register writes that includes the RCRC (reset CRC), IDCODE (0x02AD6093), WCFG, FAR (frame address register), and FDRI (frame data register input, which specifies the number of 32-bit configuration words to write into the device) [50], followed by the configuration words (number of which is specified by the FDRI), and ending with the final words, that includes the final register writes (MASK, CTL1, LFRM), the CRC, and DESYNCH (which releases the ICAP and allows other PRRs to be reconfigured) [50].

If the PRR consists of only one row with no BRAM columns, there is only one pair of FAR/FDRI values, which is followed by the configuration words. For a PRR with multiple rows with no BRAM columns, the partial bitstream uses as many pairs of FAR/FDRI values as number of rows in the PRR, which are followed by the configuration words for each row. If the PRR includes BRAM columns, additional pairs of FAR/FDRI are used for the BRAM initialization words, which follow the last row's

configuration words for the rows that contain BRAM columns. Figure 2-2 depicts a sample partial bitstream structure for a PRR with two rows that contain CLBs, DSPs, and BRAMs.

In order to read a task's context (i.e., CS), the command GCAPTURE [50] is sent to the device via the ICAP to capture the flip-flops' values on a single edge transition of the main clock. Reading the flip-flops' values from the device is defined by Xilinx as the *readback capture process*, denoted as *capture* for CSR/HTR purposes, which is different from the *readback verify process* (defined by Xilinx), where the flip-flops' initial values are read. After capturing the PRR's flip-flops' values, the command RCAP (reset capture) is sent via the ICAP to enable future CSs on the same or different PRR. We note that a protected PRR does not allow the task to capture the task's flip-flops and BRAMs values.

Restoring a task's context (i.e., CR) to a PRR requires initializing the PRR's flip-flops' values with the saved flip-flops' values (i.e., the task's context) without interrupting the static region or the other PRRs' executions. In order to initialize a PRR with new flip-flop values, the GSR signal must be toggled, however, since toggling this signal would re-initialize the entire device with the initial values defined in the full bitstream, a protection/unprotection mechanism must be provided.

A PRR/FPGA can be protected using the block type '010' and a special frame, sent to all PRR/FPGA columns, that has all 41 32-bit words set to 0x00000000 except word 21, which has bit 12 and 13 set to '1', which correspond to the internal GWE (global write enable) and GRESTORE signals, respectively. Unprotecting a PRR/FPGA is similar, except that all 41 32-bit words are set to 0x00000000. Protecting the entire

FPGA only needs to be done once, while unprotection/protection of the PRRs is required for each CS and CR.

For newer devices (e.g., the Virtex-6 and -7 series, and the Zynq-7000) and tools (e.g., starting from the Xilinx PlanAhead 14.3 tool [53]) the RESET_AFTER_RECONFIG = TRUE (RaR) constraint may be applied to PRRs in order to avoid the manual unprotection/protection of PRRs and manual protection of the static region after full configuration. The partial bitstream generated with this constraint contains the ICAP command sequence to protect the entire FPGA, unprotect/protect the PRR, and the GRESTORE and START commands [50] to force the startup sequence. However, for the purposes of CSR/HTR, the user requires generation of the CRC with custom hardware, which incurs hardware overhead (1,218 FFs and 5 BRAMs for the Virtex-4) [24], unless the CRC verification is skipped. Since partial bitstreams using the RaR constraint contain the ICAP commands to protect the entire FPGA, these partial bitstreams are extremely large in size as compared to partial bitstreams without using the RaR constraint, which increases the PRR reconfiguration time.

Additionally, for the purposes of physical implementation of CSR/HTR, the user would not have manual control of re-initialization of the PRR's flip-flops and BRAMs after downloading the partial bitstream if the RaR constraint is used, causing the task that is executing in that PRR to terminate. Thus, all of the fundamentals explained in this work for the Virtex-5 are still valid for the newer devices.

2.2.3 Cost Model for Partial Bitstream Size

From the initial partial bitstream structure depicted in Figure 2-2, the partial bitstream size can be calculated using a simple cost model, which is used for the CSR, HTR, and DDRM experiments. Table 2-1 depicts the parameters for partial bitstream

size derivation where IW, FW, FAR_FDRI, CF_{CLB}, CF_{DSP}, CF_{BRAM}, DF_{BRAM}, and FR_{size} are device family dependent. We note that for Virtex-4/5/6 and Series 7 devices, words are 32-bit, however, in other devices, such as Spartan-3/6 devices, words are 16-bit, therefore, $Bytes_{word}$ must be adjusted according to the device family. In Table 2-1, the FAR_FDRI specifies the number of words for setting the FAR and the FDRI registers [50]. The FAR specifies the first frame address in terms of a row and column on the device fabric for configuration words (or initialization words for BRAM columns, if BRAMs are used) in a given PRR row, and the FDRI specifies the number of configuration words (or initialization words for BRAM columns) for the given PRR row. Table 2-2 summarizes the specific values from Table 2-1 for Virtex-4/5/6 device families.

The size of the partial bitstream ($S_{bitstream}$) for a PRR with H rows that contains CLBs, DSPs, and BRAMs is:

$$S_{bitstream} = \{IW + H \times (NCW_{row} + NDW_{BRAM}) + FW\} \times Bytes_{word} \tag{2-1}$$

where IW and FW in (2-1) denote the number of initial and final words in the partial bitstream, respectively. The number of configuration words in a PRR row (denoted as NCW_{row}) in (2-1) is expressed as:

$$NCW_{row} = FAR_FDRI + (NCF_{CLB} + NCF_{DSP} + NCF_{BRAM} + 1) \times FR_{size} \tag{2-2}$$

where FR_{size} in (2-2) denotes the frame size in words. The total number of configuration frames per CLB, DSP, and BRAM columns in a single row of a PRR (denoted as NCF_{CLB}, NCF_{DSP}, and NCF_{BRAM}, respectively) are:

$$NCF_{CLB} = W_{CLB} \times CF_{CLB} \tag{2-3}$$

$$NCF_{DSP} = W_{DSP} \times CF_{DSP} \tag{2-4}$$

$$NCF_{BRAM} = W_{BRAM} \times CF_{BRAM} \quad (2\text{-}5)$$

where W_{CLB}, W_{DSP}, and W_{BRAM} included in (2-3), (2-4), and (2-5) are the number of CLB, DSP, and BRAM columns in the PRR, respectively. Also, CF_{CLB}, CF_{DSP}, and CF_{BRAM} in (2-3), (2-4), and (2-5) are the number of configuration frames per single CLB, DSP, and BRAM column, respectively.

Finally, the number of BRAM initialization words in a PRR row (denoted as NDW_{BRAM}) in (2-1) is:

$$NDW_{BRAM} = FAR_FDRI + (W_{BRAM} \times DF_{BRAM} + 1) \times FR_{size} \quad (2\text{-}6)$$

where DF_{BRAM} included in (2-6) are the number of data initialization frames per single BRAM column.

Table 2-1. Parameters used in the partial bitstream size cost model

Parameter	Description
IW	Number of initial words in the partial bitstream
FW	Number of final words in the partial bitstream
FAR_FDRI	FAR/FDRI initialization words per row
NCW_{row}	Number of configuration words in a PRR row
NDW_{BRAM}	Number of BRAM initialization words in a PRR row
NCF_{CLB}	Number of CLB configuration frames in a PRR row
NCF_{DSP}	Number of DSP configuration frames in a PRR row
NCF_{BRAM}	Number of BRAM configuration frames in a PRR row
CF_{CLB}	Number of configuration frames per CLB column
CF_{DSP}	Number of configuration frames per DSP column
CF_{BRAM}	Number of configuration frames per BRAM column
DF_{BRAM}	Number of data initialization frames per BRAM column
FR_{size}	Frame size in words
$Bytes_{word}$	Number of bytes per word
H	Number of rows in the PRR
$S_{bitstream}$	Size of partial bitstream in bytes

Table 2-2. Specific values from Table 2-1 for Virtex-4/5/6 FPGA device families

Parameter	Virtex-4	Virtex-5	Virtex-6
CF_{CLB}	22	36	36
CF_{DSP}	21	28	28
CF_{BRAM}	20	30	28
DF_{BRAM}	64	128	128
FR_{size}	41	41	81
IW	12	16	20
FW	108	114	113
FAR_FDRI	5	5	5
$Bytes_{word}$	4	4	4

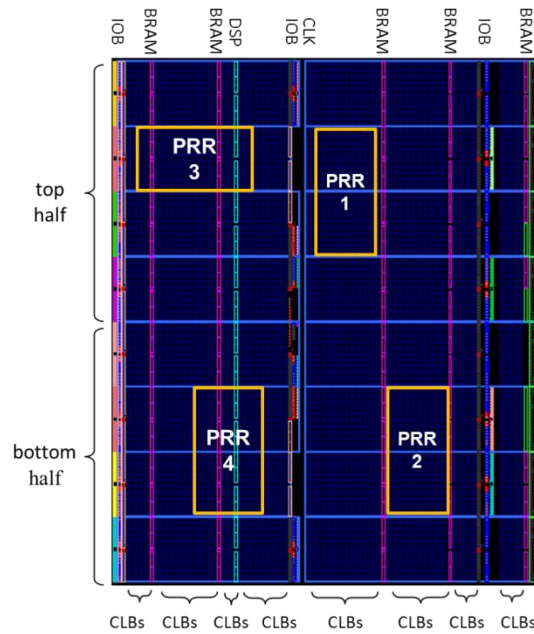

Figure 2-1. Virtex-5 LX110T FPGA fabric layout with four sample PRRs

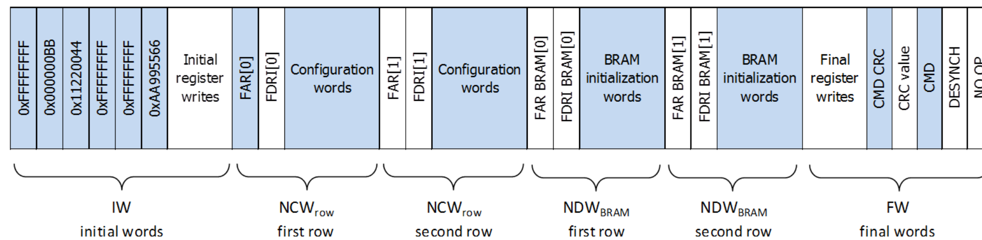

Figure 2-2. Partial bitstream structure for Virtex-5 FPGAs

CHAPTER 3
ON-CHIP CONTEXT SAVE AND RESTORE, AND HARDWARE TASK RELOCATION SOFTWARE

In this chapter, we give a detailed description of the on-chip CSR and HTR software, which correspond to the first and second phases of this research, in order to enable hardware multitasking on the same and different PRRs, respectively, for PR FPGAs.

Our on-chip CSR/HTR for heterogeneous PRRs executes in software on a soft-core processor in the FPGA's static region. We assume that prior to CSR/HTR execution, the applications have already been synthesized and partitioned into multiple hardware tasks, the PRRs [47][48] and soft-core processor have been created, the system contains a scheduler that maps and schedules incoming tasks to PRRs, and all full and initial partial bitstreams and necessary files have been generated. We refer to a hardware task that is executing in a PRR as a partially reconfigurable module (PRM), and we note that since a PRM may be executed in more than one heterogeneous PRR at different times, all PRM's initial partial bitstreams for candidate PRRs must be generated prior to system execution. Even though a PRR may contain a mixture of resources (CLBs, BRAMs, DSPs, IOBs), in this chapter, we detail CSR/HTR for PRMs that use CLB, distributed RAM (LUTRAM) for CLBs, and BRAM resources only, however, our technique is equally applicable to heterogeneous PRRs that contain DSPs and/or IOBs that are not used by the PRM.

The remainder of this chapter is organized as follows. Section 3.1 presents an overview of CSR and HTR. Section 3.2 describes how to perform CS of hardware tasks to produce a CS bitstream, which is the first major step in CSR and HTR. Section 3.3 then describes the merge process, in order to generate a new partial bitstream based

on the CS bitstream, which is the second major step in CSR. Next, Section 3.4 describes how to perform the saved context relocation between different PRRs based on the CS bitstream, which is the second major step in HTR. Section 3.5 then describes how to perform CR of hardware tasks, which is the third major step in CSR and HTR. Next, Section 3.6 discusses the portability of our on-chip CSR and HTR across FPGA device families. Finally, conclusions from the first two phases of this research are summarized in Section 3.7.

3.1 CSR and HTR Overview

We explain CSR and HTR using a system with two heterogeneous PRRs. PRR1 is a candidate PRR for PRM1 and PRM2, and PRR2 is a candidate PRR for PRM2 and PRM3. Even though this is a small example, this system is sufficient for explaining CSR and HTR. Figure 3-1 depicts the CSR and HTR flows for this example. Even though not shown in Figure 3-1, we assume that PRM2 has already executed in PRR1, PRM2 was preempted and PRM2's context was saved, PRM3 is currently executing in PRR2, and PRR1 is ready to execute in PRM1. T_x denotes each step's execution time.

Initialization reconfigures PRR1 with PRM1 by transferring PRM1's initial partial bitstream for PRR1 from external storage to the device via the ICAP and enables FPGA protection to prevent re-initialization of flip-flops and BRAMs in the static region and PRRs on successive executions of the startup phase (Section 2.2.2). $T_{reconfig_prr}$ and $T_{protect_fpga}$ denote the execution times for these steps, respectively, and these times depend on the number of rows and columns in the PRR and FPGA, respectively.

PRM2 can either be resumed in PRR1 or relocated to PRR2. Since CSR requires less execution time than HTR, PRM2 will first attempt to resume execution in PRR1. For example, if PRR1 is free, or is executing a lower priority task and can thus be

preempted by PRM2 (i.e., PRM1 is lower priority than PRM2 in this case), CSR will resume PRM2 in PRR1 by: 1) CS of PRM1; 2) PRM2's saved context (CS bitstream) is merged with PRM2's initial partial bitstream to create the merged partial bitstream for PRR1; and 3) CR of PRM2 on PRR1. If PRR1 is not free or is executing a higher priority task (i.e., PRM1 is higher priority than PRM2 in this case), and PRR2 is available or executing a lower priority task (i.e., PRM3 is of lower priority in this case), HTR will relocate PRM2 to PRR2 by: 1) CS of PRM3; 2) relocate PRM2's saved context to PRR2; and 3) CR of PRM2 on PRR2.

In the following sections we elaborate the sequential major steps in CSR and HTR. The major steps in CSR are CS, merge of initial bitstream with CS bitstream, and CR, while the main steps in HTR are CS, saved context relocation, and CR.

3.2 Context Save (CS) of Hardware Tasks

CS is the first major step in both CSR and HTR. Before reading a PRM's flip-flops' values, the PRR's clock must be stopped to avoid potential setup/hold violations. Next, a sequence of commands (including GCAPTURE) is sent via the ICAP to initiate the capture process and subsequently unprotect the PRR. T_{pre_cs} and $T_{unprotect_prr_cs}$ denote the execution time for these steps, respectively, where $T_{unprotect_prr_cs}$ depends on the number of rows and columns of the PRR. The PRR must be unprotected prior to capturing the flip-flops' and BRAMs' values since performing a capture over a protected PRR captures the flip-flops' and BRAMs' initial values and not the current values at the moment of stopping the PRR's clock. However, performing a capture over DSP columns will always give the DSPs' flip-flops' initial values because the DSP columns do not have specific configuration frames (of all 28) to read the current values of the DSPs' flip-

flops, which is also confirmed in Jozwik's work [23]. We note that IOBs are not captured since the current version of the Xilinx tools no longer support IOBs in PRRs.

After the capture process is initiated and the PRR is unprotected, a sequence of commands executed in a program loop on the soft-core processor read and save the PRM's CLB LUTRAMs and CLB flip-flops' values into a temporary file on a frame-by-frame basis. This software capture process's execution time depends on the number of frames containing PRM flip-flops, LUTRAMs, and BRAMs. Reading from Xilinx devices using the ICAP generates an extra dummy frame and one dummy word that must be read [50], but are later discarded. We read and save the PRM's BRAM values in a different temporary file, but not on a frame-by-frame basis since reading each BRAM column's contents (128 data frames) would greatly increase the CS execution time by reading an extra dummy frame and word for each valid frame. Instead, we read (23 + 1) data frames and one word for five iterations, and the sixth iteration we read (13 + 1) data frames and one word. The BRAMs' values must be captured in this way due to limitations imposed by the Linux driver for the HWICAP, which limits the data to a maximum of 4K bytes per readback [55]. T_{cs_prr} denotes the execution time to read the PRM's BRAMs, CLB LUTRAMs, and CLBs flip-flops' values, including the dummy frames and words and the time to save these values in the temporary files.

After completing the capture process, another sequence of commands (CMD RCAP) is sent to the device prior to re-protecting the PRR, which resets the internal CAPTURE signal to enable future CSs on the same or different PRR. T_{post_cs} denotes the execution time for these commands. Then, the PRR is re-protected to prevent future execution of the startup sequence phase in another PRR from re-initializing the current

PRR's flip-flops' and BRAMs' values. $T_{protect_prr_cs}$ denotes the execution time of this step. Next, a sequence of commands frees the ICAP for subsequent use by the current PRR or another PRR. $T_{desynch}$ denotes the execution time for these commands.

Finally, another program loop executed on the soft-core processor discards the dummy frames and words from the temporary files and saves the PRM's context in a file (CS bitstream). The size of the CS bitstream in 32-bit words is denoted as $1+N+N*41$, where N is the number of frames read and that contain the PRM's flip-flops', LUTRAMs', and BRAMs' values. The first word in the CS bitstream specifies N's value, the following N words specify the N different frame address values that contain the flip-flops', LUTRAMs', and BRAMs' values, and the final $N*41$ words are the contents of the N frames. $T_{cs_bitstream}$ denotes the execution time to generate the CS bitstream.

Thus, the total execution time for CS (denoted as T_{cs}) is:

$$T_{cs} = T_{pre_cs} + T_{unprotect_prr_cs} + T_{cs_prr} + T_{post_cs} + T_{protect_prr_cs} + T_{desynch} + T_{cs_bitstream} \quad (3\text{-}1)$$

3.3 Merge of Initial Bitstream and Saved Context

The merge process is the second major step in CSR. The merge process in CSR updates a PRM flip-flop/LUTRAM/BRAM bit in the scheduled PRR with the PRM flip-flop/LUTRAM/BRAM bit from the CS bitstream. Figure 3-2 depicts these bitstream manipulations, which merge the CS and initial bitstreams at the 32-bit word level updating multiple flip-flop/LUTRAM/BRAM bits, and only on those words within a frame where context bits are involved. The example in Figure 3-2 has been reduced to eight bits for clarity.

To understand the bitstream manipulations in CSR, we use Table 3-1, that shows the truth table for CSR's merge process, expressed as $f = (cap \wedge msk) \vee (ini \wedge /msk)$,

where *cap* is the captured value, *ini* is the flip-flop/LUTRAM/BRAM bit value in the initial partial bitstream, and *msk* denotes if the bit is part of the saved context where *msk = 1* updates *ini* with *cap* and *msk = 0* retains *ini*'s value.

For the example in Figure 3-2, only the bit positions with *msk = 1* will be updated with the value of *cap* in the same bit positions. Therefore, bits 7, 5, 3, and 1 will be updated producing a final *f* word (in binary) equal to 00111100.

The merge is executed as a program loop in the soft-core processor, where the loop's execution time is dictated by the number of words that contains the task's context in the CS bitstream. After the merge, the merged partial bitstream is saved to a file. T_{merge} denotes the execution time for the merge process, including the time to save the file.

3.4 Saved Context Relocation of Hardware Tasks (HTR)

The saved context relocation is the second major step in HTR. The bitstream manipulations required for relocating the saved context (i.e., HTR) is similar to the merge process in CSR [35]. HTR updates a PRM flip-flop bit in the scheduled PRR with the PRM flip-flop bit from the CS bitstream. Figure 3-3 and Figure 3-4 depict these bitstream manipulations, which merge the CS and initial bitstreams at the 32-bit word level based on two cases: update a single flip-flop or multiple flip-flops. Figure 3-3 shows the update of a single flip-flop value for HTR, and Figure 3-4 shows multiple updates of flip-flop values for HTR. All examples have been reduced to five bits for clarity.

HTR cannot use the expression *f* used in CSR (Section 3.3) for context relocation. HTR requires two *msk*'s, one for the saved context and the other for the initial partial bitstream in the scheduled PRR. In HTR, the context relocation for flip-flops

is expressed as $g = (cap \wedge ms) \vee (inid \wedge /md)$, where *cap* is the flip-flop's captured value, *ms* denotes if the flip-flop is part of the saved context, *inid* is the flip-flop's value in the initial partial bitstream for the scheduled PRR, and *md* denotes if the flip-flop in the merged bitstream needs to be updated. $md = 1$ updates *inid* with *cap*, provided that $ms = 1$, and $md = 0$ retains *inid*'s value. In Figure 3-3 and Figure 3-4, *bitms* and *bitmd* denote the bit position of *ms* and *md* and the expression $bitms - bitmd$ denotes the distance between these bits' positions in a 32-bit word. If $bitms - bitmd \geq 0$, $(cap \wedge ms)$ is right-shifted $bitms - bitmd$ bit positions using $shr(cap \wedge ms)$, else, $(cap \wedge ms)$ is left-shifted $bitmd - bitms$ bit positions using $shl(cap \wedge ms)$. Updating multiple flip-flops in a word boundary in the merged bitstream (Figure 3-4) is done sequentially, and each update does not necessarily have the same *cap* and *ms* words as shown.

Context relocation for LUTRAMs and BRAMs follows the same procedure as the merge process in CSR (Section 3.3), but using the expression *g*, updating multiple bits in a word boundary and only on those words within a frame where context bits are involved without performing the shifting of $(cap \wedge ms)$. Since the LUTRAM and BRAM bits for the PRMs being relocated to different PRRs have the same bit positions within the words, there is no need to calculate the distance between the bit's positions or shift $(cap \wedge ms)$ as is required for flip-flops. Also, the words' positions within a frame that include LUTRAM and BRAM bits are grouped consecutively, and the LUTRAMs and BRAMs' bits are saved in consecutive frames. This layout allows us to reduce the relocation time when LUTRAMs and BRAMs are included in a PRR, since alternatively performing context relocation for LUTRAMs and BRAMs using the expression *g*, but

updating bit-by-bit (as done with flip-flops) would incur prohibitively high context relocation execution times (on the order of seconds).

The context relocation is executed as a program loop in the soft-core processor, where the loop's execution time is dictated by the number of PRM flip-flops in the CS bitstream. After the relocation, the merged partial bitstream is saved to a file. $T_{relocate}$ denotes the execution time for the context relocation (HTR), including the time to save the file.

3.5 Context Restore (CR) of Hardware Tasks

CR is the last major step in both CSR and HTR. Before CR, the scheduled PRR must be unprotected to allow initialization of the PRR's flip-flops with the new values in the merged partial bitstream, but the remainder of the FPGA must remain protected. T_{pre_cr} denotes the execution time of the sequence of commands sent via the ICAP before unprotecting the scheduled PRR and $T_{unprotect_prr_cr}$ denotes the execution time to unprotect the scheduled PRR. Next, the scheduled PRR is reconfigured via the ICAP by sending the merged partial bitstream, which has the same structure as the initial partial bitstream as shown in Figure 2-2, with the exception that no CRC is generated, and is replaced by the RCRC command. T_{update_prr} denotes the execution time to transfer this new partial bitstream to the scheduled PRR.

Even though omitting the CRC introduces the possibility of reconfiguration errors, it is a valid solution for small partial bitstreams and when the FPGA is not exposed to a harsh environment, such as high radiation levels. Toggling GSR is more time efficient than generating the CRC on the fly in software, and generating the CRC with custom hardware incurs hardware overhead (1,218 flip-flops and 5 BRAMs for the Virtex-4 [24]), which may decrease the system's maximum operating frequency. Since there is no

CRC in the final words (Figure 2-2), the only way to initialize the PRR's flip-flops' and BRAMs' values with the PRM's saved context is to force the execution of the startup sequence phase by toggling the GSR signal (Section 2.2.2). $T_{startup}$ denotes the execution time of the startup sequence phase, which includes the time to toggle GSR and the elapsed time until the signal EOS is asserted. After the startup sequence phase, the scheduled PRR is reconfigured with the PRM's relocated context and is ready for execution.

Finally, it is necessary to protect the scheduled PRR to prevent future startup sequence phases to other PRRs from re-initializing the scheduled PRR's LUTRAMs, BRAMs, and flip-flops' values. $T_{protect_prr_cr}$ denotes the execution time for this step.

Thus, the total execution time for CR (denoted as T_{cr}) is:

$$T_{cr} = T_{pre_cr} + T_{unprotect_prr_cr} + T_{update_prr} + T_{startup} + T_{protect_prr_cr} \quad (3\text{-}2)$$

3.6 CSR and HTR Portability across FPGA Device Families

No prior work addressed CSR/HTR portability across different FPGA device families, especially for 2-D different-sized heterogeneous PRRs, since prior works on on-chip BR/HTR and off-chip HTR have limitations or impose constraints that prevent implementation of HTR that can be easily ported to other FPGA device families with minimum changes.

Prior BR works [9][10][25][36][39][41][42] using on-chip custom hardware for 2-D homogeneous PRRs are complex and not portable because using custom hardware requires special constraints. These constraints require specific physical positions of inputs and outputs with respect to PRRs, no signals from static region must be routed through PRRs, and a communication bus between the custom hardware and all PRRs. Additionally, using custom hardware for on-chip BR works only for custom PRR sizes

and PRR resource organizations, and is not applicable to arbitrary PRR size/resource organization.

Prior off-chip HTR works [26][29] are not portable across FPGA device families, because off-chip HTR requires the FPGA to be attached to a host CPU with a dedicated host CPU-FPGA communication bus. Even though some FPGAs provide high speed communication bus signals to reduce the host CPU-FPGA communication overhead, this solution is expensive, requiring additional hardware (e.g., small form-factor pluggable (SFP) transceivers), and the high speed communication bus are not available in all FPGA device families. Also, using a dedicated host CPU-FPGA communication bus requires a special driver for the host CPU-FPGA communication, which in turn limits the implementation of HTR in an autonomous system on FPGAs. Additionally, using a bare-metal solution (no operating system on FPGA, and attached to a host CPU) limits HTR portability across different FPGA device families, because using a bare-metal solution would not be able to accomplish the bitstream manipulations (Section 3.3 and Section 3.4) efficiently, limiting the implementation of CSR and HTR.

Prior on-chip HTR works from Iturbe et al. [17][18][19] for 2-D heterogeneous PRRs also are not portable across FPGA device families. Iturbe's works used BRAMs (which are location-specific in the FPGA's fabric and limited in number) as buffers to hold input and output values for each hardware task, and PRRs must to be placed near to each other in the FPGA's fabric in order to copy the output values (from the output buffer) from one task to another task, using a custom hardware DRT block.

We note that a fixed on-chip custom hardware-based HTR between 2-D heterogeneous PRRs is extremely complex, because a large variety of PRR resource

distribution combinations in the PRR columns prevents the implementation of a fixed on-chip hardware-based HTR. A custom hardware implementation for establishing the mapping of flip-flops', LUTRAM bits', and BRAM bits' positions (for CR purposes, Section 3.4) between 2-D heterogeneous PRRs would be extremely difficult to implement (if not, impossible) with fixed hardware, resulting in excess area overhead and affecting the tasks' maximum operating frequencies. Also, the on-chip custom hardware for HTR may not be portable across same/different FPGA device families.

We note that our HTR approach for 2-D heterogeneous PRRs is a simpler, but not necessarily more efficient with respect to execution time, and more complete than prior works [17][18][19]. These works copy only the results computed by a hardware task (saved in an output buffer implemented with BRAMs), using a DRT block [19], to the input buffer (implemented with BRAMs) of the relocated task, while our approach relocates the entire task's context (i.e., the saved flip-flop, LUTRAM, and BRAM values required by the saved hardware task (Sections 3.2 to 3.5).

Our on-chip software-based CSR/HTR on 2-D heterogeneous PRRs is also portable across FPGA device families. We enable CSR/HTR portability by having a design specification that does not use special constraints (i.e., there is no special communication bus between the reconfiguration controller and the PRRs, there is no host CPU-FPGA communication bus, and there are no limitations with respect to PRR size/location/resource organization). Also, CSR/HTR portability is ensured by using a portable language for the CSR/HTR software application to be used with different FPGA device families. For our CSR/HTR application, we use the standard C language, and we

use configuration files selected for a device's specific architecture, where these configuration files are selected at compilation time.

The configuration files (*.far) used in our CSR/HTR application specify the FAR addresses for each resource column (CLB, DSP, etc.) per row in the specific FPGA device. The bram.far, clb.far, clk.far, dsp.far, and iob.far used in our HTR software specify the FAR addresses for all of the BRAM, CLB, CLK, DSP, and IOB columns in the FPGA, respectively. For example, the Xilinx Virtex-5 LX110T (Figure 2-1) has eight rows and 5, 54, 1, 1, and 3 BRAM, CLB, CLK, DSP, and IOB columns per row, respectively, generating 40, 432, 8, 8, and 24 32-bit words for the bram.far, clb.far, clk.far, dsp.far, and iob.far files, respectively.

Also, we use a C header file (htr.h) to define all of the ICAP commands and the commands' codings using the #define directive for the Virtex-5 FPGA device family. The htr.h file also includes the sequence of ICAP commands used in CSR and HTR for FPGA/PRR protection and PRR unprotection, CS, and CR, where all of these sequences are organized as arrays, and the definition of the number of configuration frames per each resource type (CLB, DSP, etc.) for the Virtex-5 FPGA device family. For HTR compatibility across FPGA device families, changes to htr.h are needed to include the definition of the ICAP commands and the commands' codings, the sequence of ICAP commands for CSR/HTR, and the number of configuration frames for each type of resource for different FPGA device families. Then, at compilation time, these definitions will be selected for the FPGA device family where CSR/HTR will be executed.

Additionally, before CSR/HTR begins execution, pre-processing extracts the information for each PRM/PRR mapping using the *.far files, the partial bitstreams, and the logic location file (*.ll) generated when the partial bitstreams are created. Each line in the plain text logic location file generated by the Xilinx tools contain the frame addresses, the LUTRAM bits', BRAM bits' and flip-flops' positions in the frames, and the LUTRAMs', BRAMs', and flip-flops' net names for the entire project, and the lines in the *.ll file were ordered by frame (Xilinx's default). Then, the pre-processing obtains the size and position of the PRRs, the column type (CLB, etc.) order inside PRRs, all FAR addresses involved in the PRRs, and the position of all of the flip-flop, LUTRAM, and BRAM bits inside the partial bitstreams for each PRM.

3.7 Summary

In this chapter, we have presented on-chip CSR and HTR software, which correspond to the first and second phase of this research, respectively. CSR enables hardware multitasking in the same 2-D heterogeneous PRR, and is able to preserve the context of preempted hardware tasks upon tasks resumption, without disrupting operations in the static region and other PRRs. HTR leverages CSR and enables the relocation of hardware tasks' contexts between different-sized 2-D heterogeneous PRRs in order to maximize PRRs utilization in the same FPGA and maximize tasks throughputs by providing more candidate PRRs for hardware tasks to resume operations.

Our on-chip software-based CSR/HTR for 2-D heterogeneous PRRs executes in an FPGA as an autonomous system (i.e., not using a host CPU-FPGA communication bus), using the MicroBlaze soft-core processor as a reconfiguration controller.

Our on-chip CSR/HTR is portable across FPGA device families with minimum changes (to file htr.h), by not using specific constraints in the design specification, not using a dedicated communication bus between the static region and PRRs or between PRRs, using a portable language (standard C), using the *.far files for the selected FPGA device at compilation time, and using the *.ll file and partial bitstreams generated by the Xilinx tools in order to obtain all PRMs/PRRs mapping information, such as the positions of PRRs in the FPGA fabric, size of all PRRs, column type order inside each PRR, and position of all PRMs' flip-flop, LUTRAM, and BRAM bits inside the partial bitstreams, which is all the information a system designer needs for CSR/HTR.

Next, Chapter 4 continues on by extending our CSR/HTR software in order to enable tasks' context relocation across multiple physically-distributed FPGAs which are networked-interconnected, providing additional improvements in resource usage per FPGA and task throughput due to more candidate PRRs per task.

Table 3-1. Truth table for the CSR merge process

ini	cap	msk	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

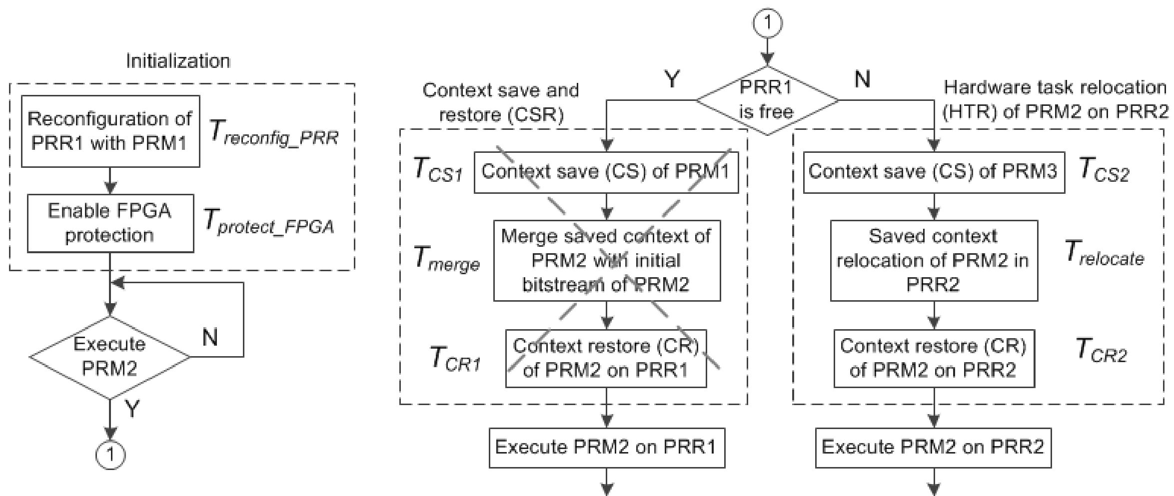

Figure 3-1. On-chip context save and restore (CSR) and hardware task relocation (HTR) flows

bit	7	6	5	4	3	2	1	0
msk =	1	0	1	0	1	0	1	0
$/msk$ =	0	1	0	1	0	1	0	1
cap =	0	0	1	0	1	0	0	0
ini =	0	0	1	1	0	1	1	0
$cap \wedge msk$ =	0	0	1	0	1	0	0	0
$ini \wedge (/msk)$ =	0	0	0	1	0	1	0	0
f =	0	0	1	1	1	1	0	0

Figure 3-2. Multiple flip-flop updates for CSR merge process

bit	4	3	2	1	0
ms =	0	0	1	0	0
cap =	1	0	1	1	0
$inid$ =	1	0	1	0	0
md =	0	0	0	0	1
$/md$ =	1	1	1	1	0
$cap \wedge ms$ =	0	0	1	0	0

$bitms = 2$

$bitmd = 0$

$bitms - bitmd = 2$

$shr(cap \wedge ms)$ =	0	0	0	0	1
$inid \wedge (/md)$ =	1	0	1	0	0
g =	1	0	1	0	1

Figure 3-3. Single flip-flop update for context relocation (HTR) process

bit 4 3 2 1 0	*bitms* = 0	*bit* 4 3 2 1 0	*bitms* = 1
ms = 0 0 0 0 1	*bitmd* = 2	*ms* = 0 0 0 1 0	*bitmd* = 3
cap1 = 0 1 0 1 1	*bitms* − *bitmd* = -2	*cap2* = 1 1 0 0 1	*bitms* − *bitmd* = -2
inid = 0 1 0 0 1		*inid* = 0 1 1 0 1	
md = 0 0 1 0 0	*shl*(*cap1* ∧ *ms*) = 0 0 1 0 0	*md* = 0 1 0 0 0	*shl*(*cap2* ∧ *ms*) = 0 0 0 0 0
/md = 1 1 0 1 1	*inid* ∧ (*/md*) = 0 1 0 0 1	*/md* = 1 0 1 1 1	*inid* ∧ (*/md*) = 0 0 1 0 1
cap1 ∧ *ms* = 0 0 0 0 1	*g* = 0 1 1 0 1	*cap2* ∧ *ms* = 0 0 0 0 0	*g* = 0 0 1 0 1

Figure 3-4. Multiple flip-flop updates in a word boundary for context relocation (HTR) process

CHAPTER 4
ON-CHIP DISTRIBUTED DYNAMIC RESOURCE MANAGEMENT SOFTWARE

In this chapter we give a detailed description of how to extend our on-chip HTR software for 2-D heterogeneous PRRs to enable dynamic context relocation of hardware tasks across FPGAs interconnected in a network (local wired, wireless network, etc.), in order to improve task throughput, reduce task idle times while waiting for a candidate PRR, and maximize the resource usage per FPGA. To accomplish CR across networked FPGAs, we present on-chip distributed dynamic resource management (DDRM) software, which is the third phase of this research.

Our on-chip DDRM leverages our HTR for 2-D heterogeneous PRRs, and executes in software on a soft-core processor in the FPGA's static region in each of the FPGAs in the network. For DDRM execution on each FPGA, we can assume that prior to DDRM execution, the applications have already been synthesized and partitioned into multiple hardware tasks (PRMs), each PRM has been assigned a priority execution level, and the PRRs and soft-core processors for all of the FPGAs have been created. Also, each autonomous system (each FPGA's soft-core processor and PRRs) contains a scheduler that maps and schedules incoming PRMs to PRRs, the mapping of local (to the same FPGA) or remote (to a different FPGA) CR (lHTR and rHTR, respectively) have been defined for all PRMs, and all full and initial partial bitstreams and necessary files have been generated for all FPGAs in the network.

Even though the network of FPGAs may contain a mixture of different types of FPGA devices, we describe DDRM for a network with identical FPGAs. However our DDRM is equally applicable to any network of different FPGA devices, and using the information from Section 3.6, we show how DDRM works with different types of FPGAs.

Additionally, in this section we assume that all FPGAs are interconnected over a local wired Ethernet network, but any interconnection network could be used.

The remainder of this chapter is organized as follows. Section 4.1 introduces the necessary definitions and concepts for understanding context relocation of hardware tasks between networked FPGAs using DDRM. Section 4.2 then gives an overview of DDRM. Next, Section 4.3 describes in detail the integration of CSR and HTR into the DDRM operations. Finally, conclusions from the third phase of this research are summarized in Section 4.4.

4.1 DDRM Concepts and Definitions

Before detailing DDRM operation, this section introduces the necessary foundational definitions and concepts used in DDRM, including the following terms: node, PRM status, PRM priority execution level, PRR availability, free PRR, busy PRR, PRR lock, and node consistency.

A node is an autonomous FPGA system that executes HTR/DDRM. PRM status defines the execution status of a PRM, where the PRM can be in only one of the following states: *executing* (currently running in a PRR), *preempted* (paused and the CS bitstream has been generated), *completed* (finished execution in a PRR), *scheduled* (assigned to a PRR to start execution), and *not assigned* (not scheduled). We use the following numerical coding to represent the PRM status: *executing* = 0, *preempted* = 1, *completed* = 2, *scheduled* = 3, and *not assigned* = 4.

The PRM's priority execution level defines the execution priority of the PRM. For DDRM, we consider four priority execution levels (0 to 3), where “3” is the highest priority execution level and “0” is the lowest priority execution level (we note that using four levels serves as only an example, and the number of distinct priority levels is

arbitrary). PRR availability defines if either a PRR is free (coded as ‘1’) or busy (coded as ‘0’). A PRR is busy if there is a PRM currently executing in the PRR. A PRR is free if no PRM is currently executing in the PRR or a PRM was already preempted or completed execution in the PRR. PRR lock flags if HTR/DDRM is currently operating in a PRR that affects the PRM status. A PRR can be locked (coded as ‘1’) or unlocked (coded as ‘0’). A locked PRR temporarily prevents higher-priority PRMs from being scheduled to that PRR while there is an operation in process (such as CS or CR). An unlocked PRR enables a PRM to be scheduled to that PRR, if the PRR is free or if the PRR is executing a lower priority PRM.

Due to DDRM’s distributed nature, inconsistencies in the network of DDRM nodes must be avoided. These inconsistencies include: executing lHTR in a local candidate PRR in a node, and later another node executes rHTR for a lower priority execution level PRM to the same PRR and node without being notified that the PRR was already busy; executing CS in a PRR without locking the PRR, and another PRM tries to start execution in the same PRR without being notified that the PRR is already locked; preempting a PRM in a PRR and later another node cannot execute rHTR to that PRR because the node was not notified that the remote PRR is free; etc.

To avoid these inconsistencies, *node consistency* is ensured by replicating data across the network’s nodes. DDRM broadcasts all PRM and PRR status changes to all of the nodes every time there is a change in a node, such as when a PRM starts execution, a PRM is preempted, lHTR or rHTR is executed, etc. Thus, all nodes have the same information regarding the mapping of all PRMs to nodes and PRRs, all PRMs’ status, and which nodes/PRRs the PRMs are currently assigned to.

Each node in the network uses two table data structures to maintain node consistency. The first table—the global table—is common across all nodes and specifies the local and remote PRM-to-PRR mappings on all nodes in the network. The global table also specifies the priority execution level and status of each PRM, and if a PRM can be locally/remotely relocated (lHTR/rHTR) to the same/different node if the local/remote PRR has sufficient resources. The global table in all nodes is dynamically updated every time there is change in a node (e.g., preemption and CS of a PRM, lHTR or rHTR execution, etc.). The second table—the local table—is specific to each node and specifies the node's current PRM-to-PRR mappings. The local table also specifies the local PRRs' statuses, including PRR lock and PRR availability, which PRM is currently assigned to a PRR, and the PRM's priority execution levels. This local table is dynamically updated in each node every time a new task is executing, when CS is currently in progress for a preempted task, etc.

Table 4-1 shows an example of the global table for a small network of two nodes, where each node has two PRRs, and three PRMs are mapped to each PRR. The table's fields' values are assigned by the system designer after the applications in all nodes have been synthesized and partitioned into multiple PRMs, and include: *fpga*, which specifies the node number (each node has a unique identification number); *prr*, which specifies the PRR number in the node (each PRR has a unique identification number per node); *prmx*, which specifies an index (ranging from 0 to the maximum number of PRMs mapped to a PRR); *prm*, which specifies the unique PRM number in the network; *pri*, which specifies the PRM's priority execution level; *stat*, which specifies the PRM's status; *rloc,* which specifies if the PRM can be locally relocated (yes = '1', no

= ‘0’) within the same node; *rrem*, which specifies if the PRM can be remotely relocated (yes = ‘1’, no = ‘0’) to another node; *rlprr*, which specifies the PRR number for local relocation, if *rloc* = ‘1’; *rlprmx*, which specifies the index (ranging from 0 to the maximum number of PRMs mapped to a PRR) for local relocation; *rlprm*, which specifies the new PRM number if the PRM is being locally relocated; *rrnode*, which specifies the node number for remote PRM relocation, if *rrem* = ‘1’; *rrprr*, which specifies the PRR number of the remote node when a PRM is remotely relocated; *rrprmx*, which specifies the index (ranging from 0 to the maximum number of PRMs mapped to a PRR) for the remote PRR; and *rrprm*, which specifies the new PRM number for remote relocation. The “X” values in the table represent “don’t cares” (e.g., if field *rloc* = ‘0’, then fields *rlprr*, *rlprmx*, and *rlprm* are don’t care, or if field *rrem* = ‘0’, then fields *rrnode*, *rrprr*, *rrprmx*, and *rrprm* are don’t care).

Since the global table (Table 4-1) establishes all of the possible PRM-to-PRR mappings on all of the nodes in the network, which are fixed, the only field in Table 4-1 that is dynamically updated and broadcasted to the other nodes due to changes in PRMs in the network (e.g., a PRM starts execution, a PRM is preempted, etc.) is *stat*.

To show the purpose and functioning of this table, we discuss examples of remote and local relocations using Table 4-1. For remote relocation, consider PRM “0” (item 1 in Table 4-1, mapped to node “0”, PRR “0”, index “0”, and priority execution level “1”), which can be remotely relocated to node “1”, PRR “1”, index “0” as PRM “100” (item 10 in Table 4-1). Also, PRM “100” in node “1” can be remotely relocated as PRM “0” in node “0”. For local relocation, consider PRM “4” (item 2 in Table 4-1, mapped to node “0”, PRR “0”, PRM index “1”, and priority execution level “2”), which can be locally

relocated to node “0”, PRR “1”, PRM index “1” as PRM “14” (item 5 in Table 4-1). Also, PRM “14” in node “0” can be locally relocated as PRM “4” in the same node. Once a PRM is locally/remotely relocated, the status (field *stat* in Table 4-1) of the old PRM changes to “4” (*not assigned*), and the relocated PRM’s status changes to “0” (*executing*), and these changes are broadcasted to all of the other nodes.

Table 4-2 shows an example of the local table used for node “1” from Table 4-1, which updates dynamically the PRR availability, PRR lock, PRM assignment to PRR, and PRM’s execution priority. The fields in this data structure are: *node_id*, which specifies the node number; *prr_id*, which specifies the PRR number; *prmx_id*, which specifies the index (ranging from 0 to the maximum number of PRMs mapped to *prr_id*); *avail_id*, which specifies the PRR availability; *lock_id*, which specifies the PRR lock; *prm_id*, which specifies the PRM number currently executing in the PRR; and *pri_id*, which specifies the PRM’s execution priority.

Since the local table (Table 4-2) establishes the local PRM-to-PRR mappings in a node, the node and PRR numbers (fields *node_id* and *prr_id*, respectively) are fixed, and the only fields that are updated in the local table (due to a change in the node, such as execution of CS, lHTR, rHTR, start execution of a PRM, etc.) are *prmx_id*, *avail_id*, *lock_id*, *prm_id*, and *pri_id*, however these fields’ values are not broadcasted to the other nodes in the network. Broadcasting these fields’ values may involve node inconsistencies, especially for rHTR. A node may use the broadcasted copy of these fields to decide whether or not to perform rHTR, and when attempting to perform rHTR, the remote PRR may become locked, or busy with a higher execution level PRM, and the rHTR will fail. Alternatively, these fields will be checked (by the local or remote

node) before any change is attempted in the node, and these fields will be updated after the change has been made, then these fields will always have the latest values.

In order to perform the context relocation of PRMs across nodes (using the information from Table 4-1 and Table 4-2 in all nodes), DDRM uses the client/server model for remote procedure call (RPC). RPC [21][43] is a widely-used model for communication in distributed systems and using RPC in DDRM enables programs in one local node to remotely execute code in a remote node. RPC uses the following terms: *client*, *server*, *client stub*, and *server stub*. A client is a procedure call on a node requesting a procedure to be executed on a server (local or remote), the server is the procedure in the local/remote node accepting the code execution request from the client, the client stub is the portion of code in the procedure call that packs/unpacks parameters (e.g., node number, PRR number/PRR lock, PRR availability, etc.) to be sent/received to/from client/server, and the server stub is the portion of code in the procedure that unpacks/packs parameters (e.g., node number, PRR number/PRR lock, PRR availability, respectively, etc.) to be received/sent from/to client/server. Each node in DDRM includes the client, server and stubs (client and server stubs).

RPC performs the following sequentially executed steps between the local and remote nodes for remote code execution [43]: a) the client procedure calls the client stub; b) the client stub builds a message with parameters (e.g., node number, PRR number, etc.) and calls the local operating system (OS) to send the message to the remote node; c) the client's OS sends a message with the parameters to the remote node's OS; d) the remote node's OS relays the message to the server stub; e) the server stub unpacks the parameters from the message and calls the server; f) the

server executes the procedure and returns the procedure's result(s) (e.g., PRR lock, PRR availability, etc.) to the server stub; g) the server stub packs the result(s) into a message and calls the server's OS to send the message to the client; h) the server's OS sends the message to the client's OS; i) the client's OS relays the message to the client stub; and j) the client stub unpacks the result(s) and returns to the client.

DDRM performs these RPC steps each time a client in a node executes a request to a server (in a local or remote node), such as checking for PRR availability, obtaining priority execution levels of PRMs, checking for a PRR lock, locking/unlocking PRRs, updating PRM status, broadcasting PRM status to other nodes, etc. The parameters in the transmitted messages from the client to the server include the node number, PRR number, etc., where all these parameters are packed in a single message. The parameters in the transmitted messages from the server to the client include the PRR lock value, PRM status value, PRR availability value, etc., where only one single parameter is packed in the message.

Packing/unpacking parameters in RPC messages and sending messages between nodes generate delays. Also, every client's RPC request to a server causes the client to wait until a message (with the results) from the server is received, which introduces more delays. However, RPC has better performance compared to other methods used in distributed systems, such as message passing interface (MPI) and Java remote machine invocation (RMI) [38].

4.2 DDRM Overview

We explain DDRM using a network of N nodes, where i represents the node number ($0 \leq i < N$), node i contains P_i PRRs, where prr_{ij} denotes PRR j on node i ($0 \leq j < P_i$), and prr_{ij} can time multiplex up to M PRMs where prm_{ijk} denotes PRM with index k for

PRR j in node i ($0 \le k < M$). In DDRM, each prm_{ijk} can be executed in any local candidate PRR in node i (using lHTR) or remote candidate PRR in other nodes (using rHTR), where the PRRs must have sufficient resources to execute the PRM. For an initial version of DDRM, we assume that each prm_{ijk} can be executed in at most two PRRs in node i (a predefined prr_{ij}, and one local candidate PRR for local relocation), and at most one remote candidate node's PRR, but this assumption can be extended to support more local and remote candidate PRRs by modifying the data structure in Table 4-1.

We denote the PRR availability of prr_{ij} using the function avail(prr_{ij}), where avail(prr_{ij}) = 0 means prr_{ij} is busy and avail(prr_{ij}) = 1 means prr_{ij} is free. avail(prr_{ij}) reads the value from field *avail_id* in Table 4-2 for prr_{ij}. A given PRR prr_{ij} with avail(prr_{ij}) = 1 on node i may receive any prm_{ijk}, and when prm_{ijk} starts execution, DDRM changes avail(prr_{ij}) to 0 (busy).

We denote the priority execution level l_{ijk} of prm_{ijk} as l_{ijk} = prio(prm_{ijk}), where $0 \le l_{ijk} < L$ and L is the maximum number of priority execution levels. prio(prm_{ijk}) reads the value from field *pri* in Table 4-1 for prm_{ijk} and copies this value to field *pri_id* for prm_{ijk} in Table 4-2. A given prm_{ijk} with priority execution level l_{ijk} that executes on PRR prr_{ij} in node i can be preempted by any prm_{ijq} with priority execution level l_{ijq}, if and only if $l_{ijk} < l_{ijq}$.

We denote the PRR lock of prr_{ij} as lock(prr_{ij}), where lock(prr_{ij}) = 1 locks prr_{ij}, and lock(prr_{ij}) = 0 unclocks prr_{ij}. lock(prr_{ij}) writes the value to the field *lock_id* for prr_{ij} in Table 4-2. Before performing an operation on prr_{ij} (e.g., CS or CR for prm_{ijk}, downloading a

partial bitstream to prr_{ij}, etc.), DDRM locks prr_{ij} which prevents requests from the same node or other node for prr_{ij}, and after the operation is completed, DDRM unlocks prr_{ij}.

When prm_{ijk} is scheduled to start/resume execution, DDRM checks if the predefined PRR for prr_{ij} is locked. If prr_{ij} is unlocked, DDRM checks for prr_{ij} availability, and if prr_{ij} is available (free), prm_{ijk} starts/resumes execution in prr_{ij}. If prr_{ij} is unavailable (busy), DDRM checks if prr_{ij} is executing a lower priority execution level PRM, and if the PRM is lower priority, DDRM preempts the lower priority PRM, generates the PRM's corresponding CS bitstream, and starts/resumes execution of prm_{ijk}. If prr_{ij} is locked or executing a higher priority execution level task, DDRM checks if prm_{ijk} can be locally relocated in a local candidate PPR in node *i*, checking the candidate PRR's lock, the availability, and if the candidate PRR is executing a lower priority execution level task. If these checks successfully pass, DDRM performs lHTR and starts/resumes execution of prm_{ijk} in the local candidate PRR. If any of these checks fail, the local relocation of prm_{ijk} is not possible, and DDRM checks if prm_{ijk} can be remotely relocated to other node's PRR using a RPC. If this check is successful, DDRM performs rHTR and starts/resumes execution of prm_{ijk} in the remote node. If this check also fails, prm_{ijk} must wait to be scheduled.

4.3 DDRM Operations

To understand the interaction between clients and servers in DDRM, we explain how DDRM works using an example system with the following specifications: two nodes, a local node *i* and a remote node *m*; node *i* has two PRRs prr_{ij} and prr_{it}; node *m* has one PRR prr_{mn}; node *i* has three PRMs prm_{ijk}, prm_{ijq}, and prm_{itu}; and node *m* has one PRM prm_{mnp}. Since DDRM uses RPC's client/server, a portion of the DDRM code executes exclusively on the client side in local node *i*, a portion of the DDRM code is

initiated (called from) the client side of node *i* and is executed on the server side of node *i*, and a portion of the DDRM code is initiated (called from) the client side of node *i* and is executed on the server side of remote node *m*.

Figure 4-1 through Figure 4-6 depict DDRM's flow of operation. Since the entire flow is too large to be displayed in a single figure, each figure shows a portion of the entire flow with references showing the transitions to/from other figures. Figure 4-2 shows additional details from Figure 4-1, and Figure 4-4 shows additional details from Figure 4-3. Unshaded rectangular and decision boxes are exclusively executed on the client side of local node *i*. Rectangular and decision boxes with horizontal lines are initiated (called from) the client side of node *i* and executed on the server side of node *i*. Light gray rectangular and decision boxes are initiated (called from) the client side of node *i* and executed on the server side of remote node *m*. Boxes with cross lines are initiated (called from) the client side of node *i* and executed sequentially (nodes *0*, *1*, … , *N-1*) on the server side of the remote nodes. Boxes with vertical lines are for when a PRM starts execution for the first time or resumes execution in the predefined PRR *j* on node *i* (prr_{ij}). Boxes with horizontal and vertical lines are for when a PRM starts execution for the first time or resumes execution in a locally-relocated PRR. Dark gray boxes are for remote relocation of a task. Since all RPC steps are initiated from the client side of a node *i* (e.g., checking for PRR availability, obtaining priority execution levels of PRMs, checking for a PRR lock, updating PRM status, etc.), all execution times T_x included in the figures are measured from the client side of node *i*.

Figure 4-1 shows the DDRM flow for executing prm_{ijq} for the first time (or resuming) in the predefined prr_{ij}, where the boxes with vertical lines contain the

necessary CSR for these operations. First, DDRM checks if prr_{ij} is unlocked. If prr_{ij} is unlocked, DDRM checks if prr_{ij} is available (free), and if prr_{ij} is free, DDRM locks prr_{ij}, starts executing (or resumes) prm_{ijq} in prr_{ij}, changes prr_{ij} availability to '0' (busy), and unlocks prr_{ij}. If prr_{ij} is busy executing a lower priority prm_{ijk} with priority execution level $l_{ijk} < l_{ijq}$, DDRM locks prr_{ij}, preempts prm_{ijk} and executes CS for prm_{ijk}, broadcasts the status of the preempted prm_{ijk} to all nodes, starts executing (or resumes) prm_{ijq}, and unlocks prr_{ij}.

Figure 4-2 shows the CSR details of Figure 4-1's boxes with vertical lines for executing prm_{ijq} for the first time (or resuming) in the predefined prr_{ij}. If there is no CS bitstream generated for prm_{ijq}, prm_{ijq} is executed for the first time in prr_{ij} by downloading the initial partial bitstream for prm_{ijq} to prr_{ij}. If prm_{ijq} was previously preempted, prm_{ijq} has a CS bitstream, and the merge process generates a new merged partial bitstream (using the CS and initial bitstreams) for prm_{ijq} to execute CR, and prm_{ijq} resumes execution in prr_{ij}. After prm_{ijq} starts/resumes execution, DDRM updates the *stat* field in the global table (Table 4-1), and the *prmx_id*, *prm_id*, and *pri_id* fields in the local table (Table 4-2) for prm_{ijq} in node *i*. Finally, DDRM broadcasts the new status of prm_{ijq} to all nodes.

Using the T_x from Figure 4-1 and Figure 4-2, the total execution time for a PRM to start execution for the first time in a free and unlocked predefined PRR (denoted as T_{exe1}) is:

$$T_{exe1} = T_{chk_lprr_lock} + T_{lprr_lock} + T_{launch_pend_task} + T_{update_node} + T_{prop_act_task} + T_{lprr_busy} + T_{lprr_unlock} \tag{4-1}$$

where $T_{chk_lprr_lock}$, T_{lprr_lock}, $T_{launch_pend_task}$, T_{update_node}, $T_{prop_act_task}$, T_{lprr_busy}, and T_{lprr_unlock} denote the execution times to check if the predefined PRR is locked, lock predefined PRR, download the initial partial bitstream and enable the execution of the PRM, update the table structures in the node for the PRM, broadcast the status of the PRM to all nodes, change the PRR availability to busy, and unlock the predefined PRR, respectively.

Similar to T_{exe1}, the total execution time for a PRM to resume execution in a free and unlocked predefined PRR (denoted as T_{res1}) is:

$$T_{res1} = T_{chk_lprr_lock} + T_{lprr_lock} + T_{resume_pend_task} + T_{update_node} + T_{prop_act_task} + T_{lprr_busy} + T_{lprr_unlock} \quad (4\text{-}2)$$

where $T_{resume_pend_task}$ denotes the execution time for resuming the preempted task (merge and CR steps involved), and all other terms in (4-2) are the same as in T_{exe1}.

Similar to T_{exe1}, the total execution time for a PRM to start execution for the first time in a busy and unlocked predefined PRR that is executing a lower priority execution level PRM (denoted as T_{exe2}) is:

$$T_{exe2} = T_{chk_lprr_lock} + T_{chk_lprm_prr} + T_{lprr_lock} + T_{cs_preempted_task} + T_{prop_preempted_task} + T_{launch_pend_task} + T_{update_node} + T_{prop_act_task} + T_{lprr_unlock} \quad (4\text{-}3)$$

where $T_{chk_lprm_prr}$, $T_{cs_preempted_task}$, and $T_{prop_preempted_task}$ denote the execution times for checking the priority execution level of the PRM currently executing in the PRR, preempting and executing CS for the preempted PRM, and broadcasting the status of the preempted PRM to all nodes, respectively, and all other terms in (4-3) are the same as in T_{exe1}.

Similar to T_{exe2}, the total execution time for a PRM to resume execution in a busy and unlocked predefined PRR that is executing a lower priority execution level PRM (denoted as T_{res2}) is:

$$T_{res2} = T_{chk_lprr_lock} + T_{chk_lprm_prr} + T_{lprr_lock} + T_{cs_preempted_task} + T_{prop_preempted_task} + T_{resume_pend_task} + T_{update_node} + T_{prop_act_task} + T_{lprr_unlock} \quad (4\text{-}4)$$

where all terms in (4-4) have previously been defined.

In Figure 4-1, if prm_{ijq} cannot start/resume execution in the predefined prr_{ij} because either prr_{ij} is locked or executing a higher priority execution level PRM, DRRM checks if prm_{ijq} is mapped to a local candidate PRR by checking if the *rloc* field in the global table (Table 4-1) is a '1' value. If prm_{ijq} is mapped to a local candidate PRR, DDRM will try to execute lHTR (Figure 4-3). If prm_{ijq} is not mapped to a local candidate PRR, DDRM will try to execute rHTR (Figure 4-5).

Figure 4-3 shows the DDRM flow to execute lHTR of prm_{ijq} (first time execution or resumption) to a local candidate prr_{it}, where the boxes with horizontal and vertical lines in Figure 4-3 contain the HTR steps for starting execution of (or resuming) prm_{ijq} as locally relocated prm_{itq}. First, DDRM checks if the local candidate prr_{it} is unlocked. If prr_{it} is unlocked, then DDRM checks if prr_{it} is free. If prr_{it} is free, prr_{it} can execute prm_{ijq} as prm_{itq}. To start/resume execution of prm_{ijq} in the free and unlocked local prr_{it}, DDRM locks prr_{it}, executes lHTR for prm_{ijq} in prr_{it} (Figure 4-4), changes the availability of prr_{it} to '0' (busy), and unlocks prr_{it}.

In Figure 4-3, if prr_{it} is busy, DDRM checks the priority execution level of prm_{itu} executing in prr_{it}. If the priority execution level of prm_{itu} is lower than the priority execution level of prm_{ijq} ($l_{itu} < l_{ijq}$), DDRM locks prr_{it}, preempts prm_{itu}, performs CS for

prm_{itu} (Section 3.2), and broadcasts the status of the preempted task prm_{itu} to all nodes. Then, DDRM execute lHTR of prm_{ijq} in prr_{it} (Figure 4-4), and finally DDRM unlocks prr_{it}.

Figure 4-4 shows the details of Figure 4-3's boxes with horizontal and vertical lines for the execution of lHTR when prm_{ijq} executes for first time (or resumes) as prm_{itq} in the local candidate prr_{it}. For first time execution of prm_{ijq} as prm_{itq}, DDRM downloads the initial bitstream of prm_{itq} in to prr_{it}. Alternatively, if prm_{ijq} resumes execution, a CS bitstream for prm_{ijq} already exists, DDRM performs HTR by merging the initial partial bitstream of prm_{itq} with the CS bitstream of prm_{ijq} (Section 3.4) to generate a new merged partial bitstream for prm_{itq}, and then executes CR for prm_{itq} (Section 3.5). After prm_{ijq} starts/resumes execution as prm_{itq}, DDRM updates the *stat* field in the global table (Table 4-1), and the *prmx_id*, *prm_id*, and *pri_id* fields in the local table (Table 4-2) for prm_{itq} in node *i*. Finally, DDRM broadcasts the new status of prm_{itq} (executing) to all nodes.

Using the T_x from Figure 4-1, Figure 4-3, and Figure 4-4, the total execution time for a locally relocated PRM to start execution for the first time in a free and unlocked local candidate PRR (denoted as T_{exe3}) is:

$$T_{exe3} = T_{chk_lprr_lock} + T_{chk_lprm_prr} + T_{chk_lrprr_lock} + T_{lrprr_lock} + T_{launch_pend_rtask} + T_{update_node} + T_{prop_act_task} + T_{lrprr_busy} + T_{lrprr_unlock} \quad (4\text{-}5)$$

where $T_{chk_lrprr_lock}$ denotes the execution time for checking if the local candidate PRR is unlocked and available. T_{lrprr_lock}, $T_{launch_pend_rtask}$, T_{lrprr_busy}, and T_{lrprr_unlock} have similar definitions as T_{lprr_lock}, $T_{launch_pend_task}$, T_{lprr_busy}, and T_{lprr_unlock}, respectively, but are applied to the local candidate PRR where lHTR is executed.

Similar to T_{exe3}, the total execution time for a locally relocated PRM to resume execution in a free and unlocked local candidate PRR (denoted as T_{res3}) is:

$$T_{res3} = T_{chk_lprr_lock} + T_{chk_lprm_prr} + T_{chk_lrprr_lock} + T_{lrprr_lock} + T_{resume_pend_rtask} + T_{update_node} + T_{prop_act_task} + T_{lrprr_busy} + T_{lrprr_unlock} \quad (4\text{-}6)$$

where $T_{resume_pend_rtask}$ has similar definition as $T_{resume_pend_task}$, but applied to the local candidate PRR where IHTR is executed, and all other terms in (4-6) have previously been defined.

Similar to T_{exe3}, the total execution time for a locally relocated PRM to start execution for the first time in a busy and unlocked local candidate PRR that is executing a lower priority execution level PRM (denoted as T_{exe4}) is:

$$T_{exe4} = T_{chk_lprr_lock} + T_{chk_lprm_prr} + T_{chk_lrprr_lock} + T_{chk_lrprm_lrprr} + T_{lrprr_lock} + T_{cs_preempted_task} + T_{prop_preempted_task} + T_{launch_pend_rtask} + T_{update_node} + T_{prop_act_task} + T_{lrprr_unlock} \quad (4\text{-}7)$$

where $T_{chk_lrprm_lrprr}$ has similar definition as $T_{chk_lprm_prr}$, but applied to checking the priority execution level of the PRM currently executing in the local candidate PRR, and all other terms in (4-7) have previously been defined.

Similar to T_{exe4}, the total execution time for a locally relocated PRM to resume execution in a busy and unlocked local candidate PRR that is executing a lower priority execution level PRM (denoted as T_{res4}) is:

$$T_{res4} = T_{chk_lprr_lock} + T_{chk_lprm_prr} + T_{chk_lrprr_lock} + T_{chk_lrprm_lrprr} + T_{lrprr_lock} + T_{cs_preempted_task} + T_{prop_preempted_task} + T_{resume_pend_rtask} + T_{update_node} + T_{prop_act_task} + T_{lrprr_unlock} \quad (4\text{-}8)$$

where all terms in (4-8) have previously been defined.

In Figure 4-3, if the local candidate PRR is locked or executing a higher priority execution level PRM, DDRM will attempt to perform a remote relocation (rHTR) of prm_{ijq} to a remote candidate PRR in a remote node (Figure 4-5).

Figure 4-5 shows the DDRM flow to execute rHTR of prm_{ijq} (first time execution or resumption) in the remote candidate prr_{mn} of the remote node m, where the dark gray box in Figure 4-5 contains the HTR steps for executing for the first time (or resuming) prm_{ijq} as remotely relocated prm_{mnq}.

In Figure 4-5, DDRM first checks the global table (Table 4-1) to determine if prm_{ijq} is mapped to remote node m for remote execution. If prm_{ijq} can be remotely executed in remote candidate PRR prr_{mn} on node m, DDRM checks if prr_{mn} is unlocked and available. If prr_{mn} is unlocked and available, DDRM checks if prm_{ijq} was previously preempted (the CS bitstream for prm_{ijq} exists). If CS bitstream for prm_{ijq} exists, DDRM transfers the CS bitstream for prm_{ijq} to the remote node m using the file transfer protocol (FTP). After transferring the CS bitstream for prm_{ijq} to node m, DDRM executes rHTR for prm_{ijq} in the remote prr_{mn} of node m, and broadcasts the status of prm_{mnq} to all nodes.

If the remote candidate prr_{mn} is unlocked but not available, DDRM checks if prr_{mn} is executing a lower priority execution level prm_{mnp}. If the priority execution level of prm_{mnp} is lower than the priority execution level of prm_{ijq} ($l_{mnp} < l_{ijq}$), DDRM locks prr_{mn} and preempts prm_{mnp} to generate the CS bitstream for prm_{mnp}. Then, DDRM broadcasts the status of the preempted prm_{mnp} to all nodes and checks if a CS bitstream for prm_{ijq} exists. If there is a CS bitstream, DDRM transfers the CS bitstream for prm_{ijq} to node m,

executes rHTR for prm_{ijq} and broadcasts the status of the remotely relocated PRM to all nodes.

The HTR steps for rHTR in Figure 4-5 (dark gray box) are shown in Figure 4-6. For first time rHTR execution of prm_{ijq}, DDRM downloads the initial bitstream of prm_{mnq} to prr_{mn} to start execution of prm_{mnq}. Alternatively, for resumption of prm_{ijq} in remote node m, there is a CS bitstream for prm_{ijq}, which was already transferred from node i to node m via FTP. DDRM executes rHTR by merging the initial bitstream of prm_{mnq} with the CS bitstream for prm_{ijq} to produce a new merged partial bitstream for prm_{mnq} (Section 3.4). Finally, DDRM executes CR (Section 3.5) for prm_{mnq} to start execution of the relocated PRM.

After prm_{ijq} (now as prm_{mnq}) starts/resumes execution in the remote candidate prr_{mn} of node m, DDRM updates the *stat* field in the global table (Table 4-1) and the *prmx_id*, *prm_id*, and *pri_id* fields in the local table (Table 4-2) for prm_{mnq} in node m, changes the prr_{mn} availability to '0' (busy), and unlocks the prr_{mn}.

Using the T_x from Figure 4-1, Figure 4-3, and Figure 4-5, the largest total execution time for a remotely relocated PRM to start execution for the first time (or resume execution) in a free and unlocked remote candidate PRR (denoted as $T_{exeres1}$) occurs when the scheduled task cannot preempt a currently executing higher priority execution level PRM in the predefined and local candidate PRRs, and is expressed as:

$$T_{exeres1} = T_{chk_lprr_lock} + T_{chk_lprm_prr} + T_{chk_lrprr_lock} + T_{chk_lrprm_lrprr} + T_{chk_rrprr_lock} + T_{rrprr_lock} + T_{ftp_csbitstream} + T_{remote_htr} + T_{prop_rhtr_status} \quad (4\text{-}9)$$

where $T_{chk_rrprr_lock}$ and T_{rrprr_lock} have similar definitions as $T_{chk_lrprr_lock}$ and T_{lrprr_lock},, respectively, but applied to the remote candidate PRR where rHTR is executed.

$T_{ftp_csbitstream}$, T_{remote_htr}, and $T_{prop_rhtr_status}$ denote the execution times for transferring the CS bitstream from the local node to the remote node, executing rHTR, and broadcasting the status of the relocated task to all nodes, respectively. All other terms in (4-9) have previously been defined.

Similar to $T_{exeres1}$, the largest total execution time for a remotely relocated PRM to start execution for the first time (or resume execution) in a busy and unlocked remote candidate PRR executing a lower priority execution level task (denoted as $T_{exeres2}$) occurs when the scheduled task cannot preempt a currently executing higher priority execution level PRM in the predefined and local candidate PRRs, and is expressed as:

$$T_{exeres2} = T_{chk_lprr_lock} + T_{chk_lprm_prr} + T_{chk_lrprr_lock} + T_{chk_lrprm_lrprr} + T_{chk_rrprr_lock} + T_{chk_rprm_rprr} + T_{rrprr_lock} + T_{cs_preempted_rprm} + T_{prop_preempted_rtask} + T_{ftp_csbitstream} + T_{remote_htr} + T_{prop_rhtr_status} \quad (4\text{-}10)$$

where $T_{chk_rprm_prr}$, $T_{cs_preempted_rprm}$, and $T_{prop_preempted_rtask}$ denote the execution times for checking if the remote candidate PRR is executing a lower priority execution level PRM, preempting and executing CS on that preempted PRM, and broadcasting the status of preempted PRM to all nodes, respectively, and all other terms in (4-10) have previously been defined.

T_{remote_htr} in (4-9) and (4-10) for first time execution of the remotely relocated PRM in remote node is expressed as:

$$T_{remote_htr} = T_{launch_pend_rrtask} + T_{update_rnode} + T_{rrprr_busy} + T_{rrprr_unlock} \quad (4\text{-}11)$$

while T_{remote_htr} in (4-9) and (4-10) for resumption of the remotely relocated PRM in remote node is expressed as:

$$T_{remote_htr} = T_{resume_pend_rrtask} + T_{update_rnode} + T_{rrprr_busy} + T_{rrprr_unlock} \quad (4\text{-}12)$$

From (4-11) and (4-12), $T_{launch_pend_rrtask}$, $T_{resume_pend_rrtask}$, T_{update_rnode}, T_{rrprr_busy}, and $T_{rrprr_unlcock}$ denote the execution times to download the initial bitstream of the relocated PRM to the remote candidate PRR, resume the relocated PRM in the remote candidate PRR (rHTR and CR are executed), update the status of the remotely relocated PRM in the remote candidate PRR, change the availability of remote candidate PRR to busy, and to unlock the remote candidate PRR, respectively.

Since all T_x included in Figure 4-1 to Figure 4-5 are measured from the point in time when a client in the local node initiates a request for starting/resuming execution of a PRM, the T_x in (4-11) and (4-12) cannot be individually measured because all steps in the dark gray box (Figure 4-6) are remotely executed in the remote node. Thus, we can only measure T_{remote_htr} as a whole in (4-11) and (4-12) and use T_{remote_htr} in (4-9) and (4-10), respectively.

4.4 Summary

In this chapter, we have presented on-chip DDRM software, which correspond to the third phase of this research. DDRM extends our HTR software to function across multiple networked-interconnected PR FPGAs, in order to enable context relocation of hardware tasks (i.e., PRMs) across multiple FPGAs. Each FPGA (i.e., node) in the network (local wired, wireless network, etc.) is an autonomous system that executes DDRM on a soft-core processor in the FPGA's static region.

Since our DDRM leverages on-chip HTR, DDRM is able to preserve the context of preempted PRMs upon PRMs resumption in different nodes in the DDRM network, maximizing PRMs throughputs and maximizing the shared resources (i.e., PRRs) usage per node by providing more candidate PRRs for PRMs to resume operations.

Due to DDRM's distributed nature, inconsistencies in the DDRM network, due to changes in the nodes (e.g., preemption of PRMs, relocation of PRMs, etc.), must be avoided, and these changes are handled in DDRM by executing the node consistency, thus each node in the network has the same information about all of the other nodes.

Node consistency in DDRM is ensured by broadcasting data across the network's nodes every time there is a change in a node, due to a change in the status of the PRMs (e.g., executing, preempted, etc.). Each node in the DDRM network uses two tables (global and local) for node consistency, where the global table, which is common across all nodes, specifies all the local and remote PRM-to-PRR mappings on all nodes in the network; and the local table, which is specific to each node, and specifies the node's current PRM-to-PRR mapping. While the local tables are dynamically updated in each node without broadcasting the changes, changes in the global tables are broadcasted to all nodes.

In order to perform the node consistency and context relocation of PRMs across nodes (using the global and local tables in all nodes), DDRM uses the client/server model of RPC, which enables programs in a node to remotely execute code in another node, and in this chapter we have given a detailed explanation of the DDRM flow using RPC to accomplish node consistency and PRMs' context relocation across nodes.

Also, in this chapter we have shown the integration of our on-chip CSR and HTR software to our DDRM, and the DDRM operation, giving a detailed description of the execution times in DDRM for several cases of starting/resuming execution of PRMs, considering local relocation (lHTR) of PRMs' contexts in the same node, or remote relocation (rHTR) of PRMs' contexts in different nodes.

Next, Chapter 5 continues on by presenting and analyzing the experimental results for our on-chip CSR, HTR, and DDRM software for varying PRR sizes and organizations. We evaluate the execution times for the major steps in CSR and HTR, and the execution times in DDRM for several cases of starting or resuming execution of PRMs under different conditions on each node in the DDRM network, which may involve lHTR or rHTR.

Table 4-1. Global table for local and remote PRM relocation for node consistency in DDRM with two nodes

item	*fpga*	*prr*	*prmx*	*prm*	*pri*	*stat*	*rloc*	*rrem*	*rlprr*	*rlprmx*	*rlprm*	*rrnode*	*rrprr*	*rrprmx*	*rrprm*
1	0	0	0	0	1	3	0	1	X	X	X	1	1	0	100
2	0	0	1	4	2	0	1	0	1	1	14	X	X	X	X
3	0	0	2	9	3	4	0	1	X	X	X	1	1	2	109
4	0	1	0	1	2	0	0	1	X	X	X	1	0	0	101
5	0	1	1	14	2	4	1	0	0	1	4	X	X	X	X
6	0	1	2	105	1	4	0	1	X	X	X	1	0	2	5
7	1	0	0	101	2	4	0	1	X	X	X	0	1	0	1
8	1	0	1	16	2	4	1	0	1	1	6	X	X	X	X
9	1	0	2	5	1	0	0	1	X	X	X	0	1	2	105
10	1	1	0	100	1	4	0	1	X	X	X	0	0	0	0
11	1	1	1	6	2	1	1	0	0	1	16	X	X	X	X
12	1	1	2	109	3	0	0	1	X	X	X	0	0	2	9

Table 4-2. Local table for node consistency in DDRM showing the currently assigned PRMs in node "1" from Table 4-1

node_id	*prr_id*	*prmx_id*	*avail_id*	*lock_id*	*prm_id*	*pri_id*
1	0	2	0	0	5	1
1	1	2	0	0	109	3

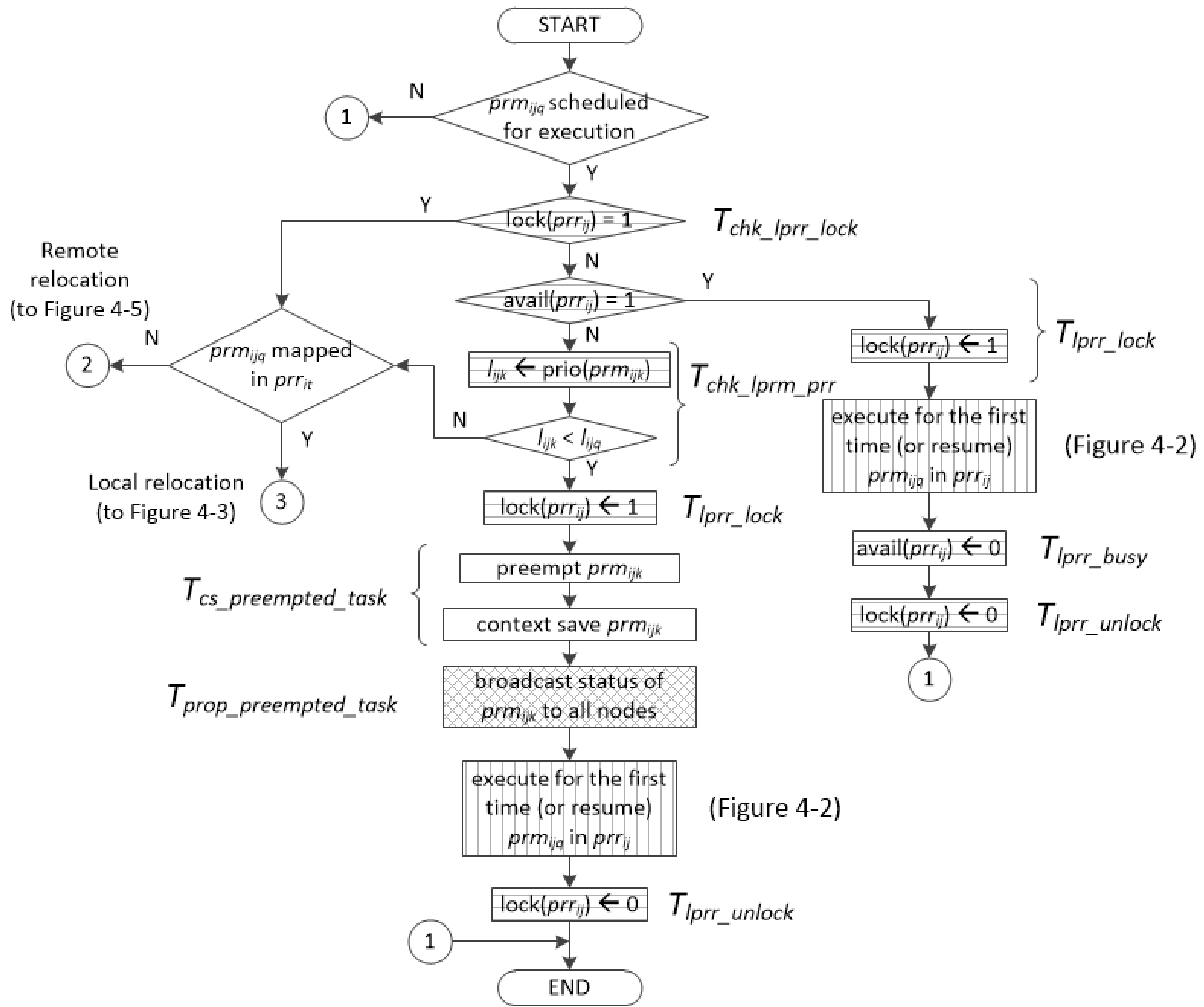

Figure 4-1. Portion of the distributed dynamic resource management (DDRM) flow showing the first time execution (or resumption) of a PRM prm_{ijq} in a predefined PRR prr_{ij}. Figure 4-2 depicts the details of the boxes with vertical lines.

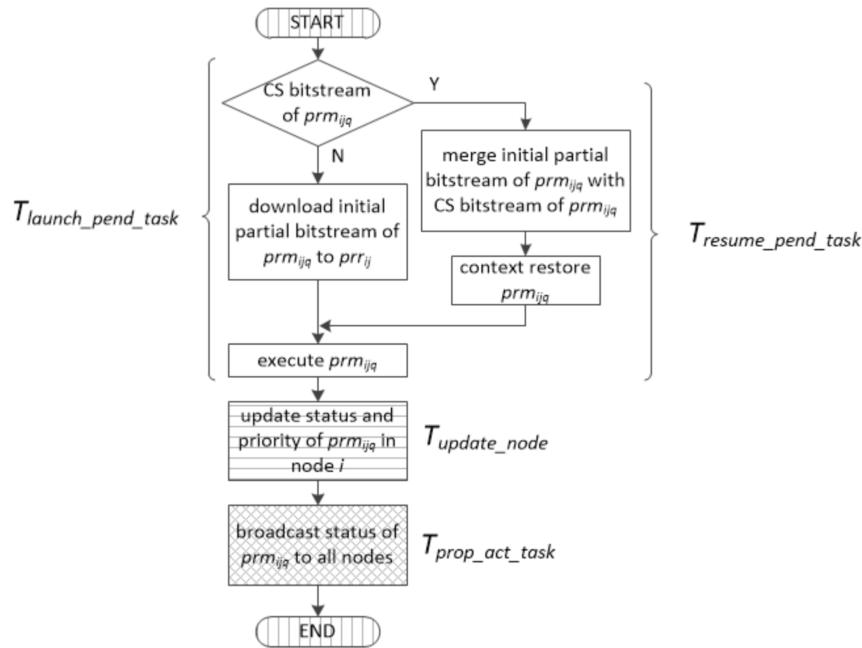

Figure 4-2. Details of the DDRM flow for the steps performed in the boxes with vertical lines in Figure 4-1 for the first time execution (or resumption) of prm_{ijq} in predefined prr_{ij}.

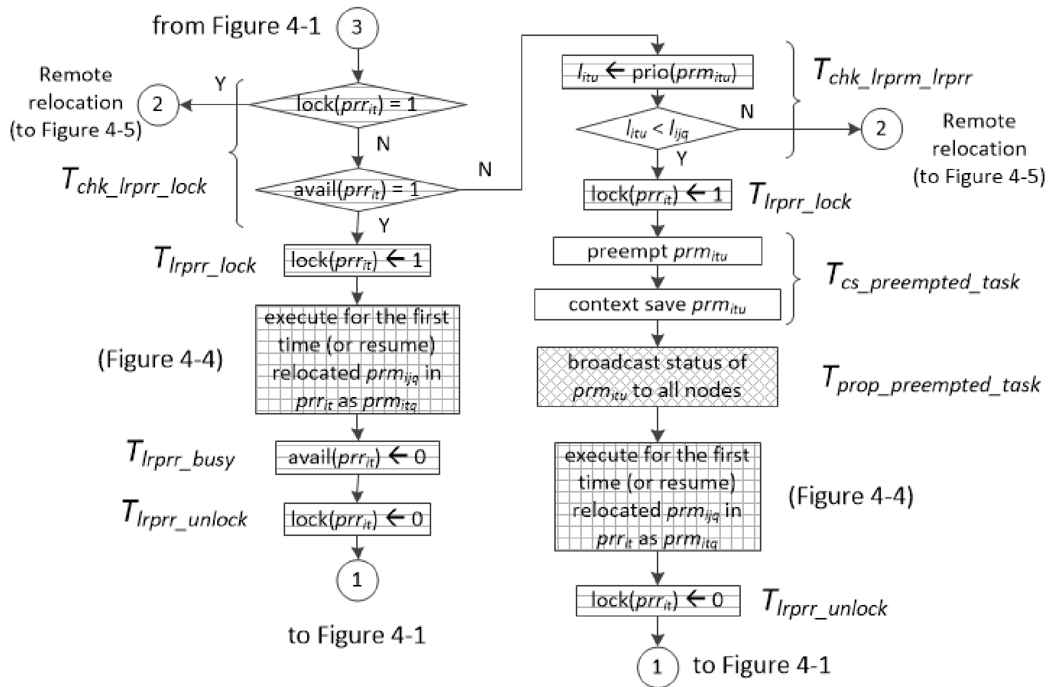

Figure 4-3. DDRM flow continued, showing the first time execution (or resumption) of a locally relocated prm_{ijq} in a local candidate prr_{it} as prm_{itq}. Figure 4-4 shows the details for the boxes with horizontal and vertical lines.

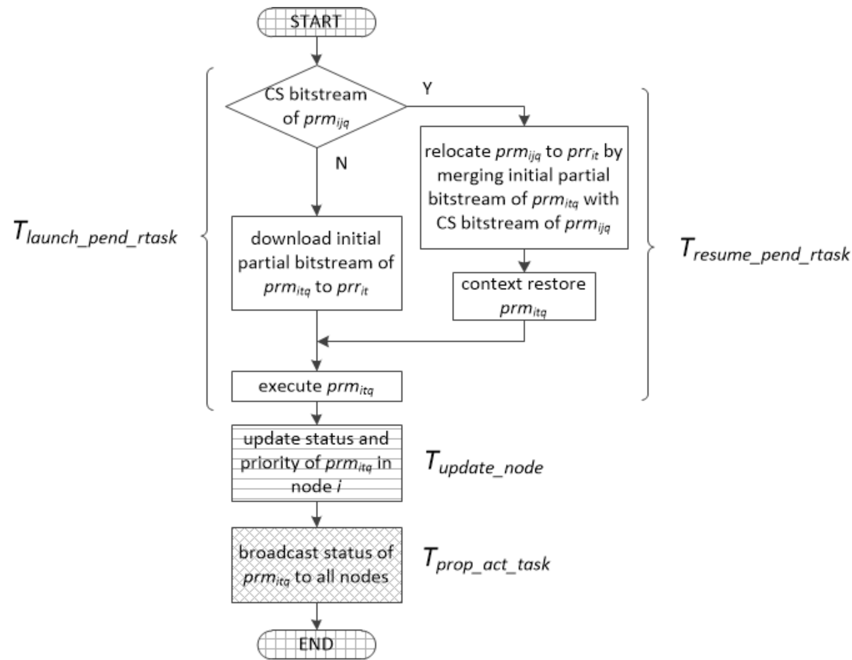

Figure 4-4. Detailed steps of the DDRM flow showing the steps performed in the boxes with horizontal and vertical lines from Figure 4-3 for the first time execution (or resumption) of a locally relocated prm_{ijq} in a local candidate prr_{it} as prm_{itq}.

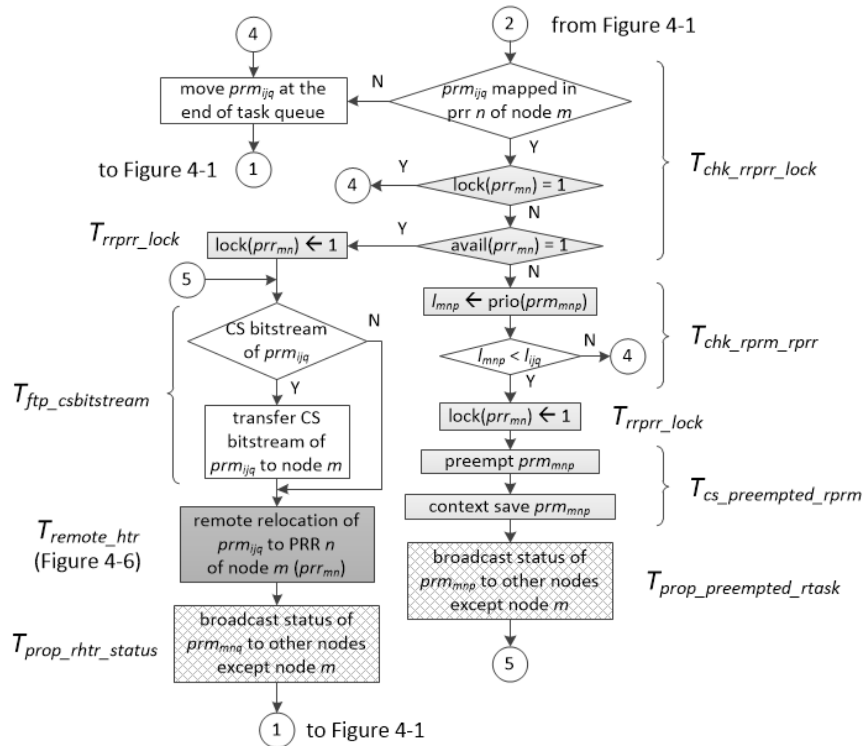

Figure 4-5. DDRM flow continued, showing the execution of the remote relocation of prm_{ijq} to a remote candidate PRR n in a node m (prr_{mn}) as prm_{mnq}.

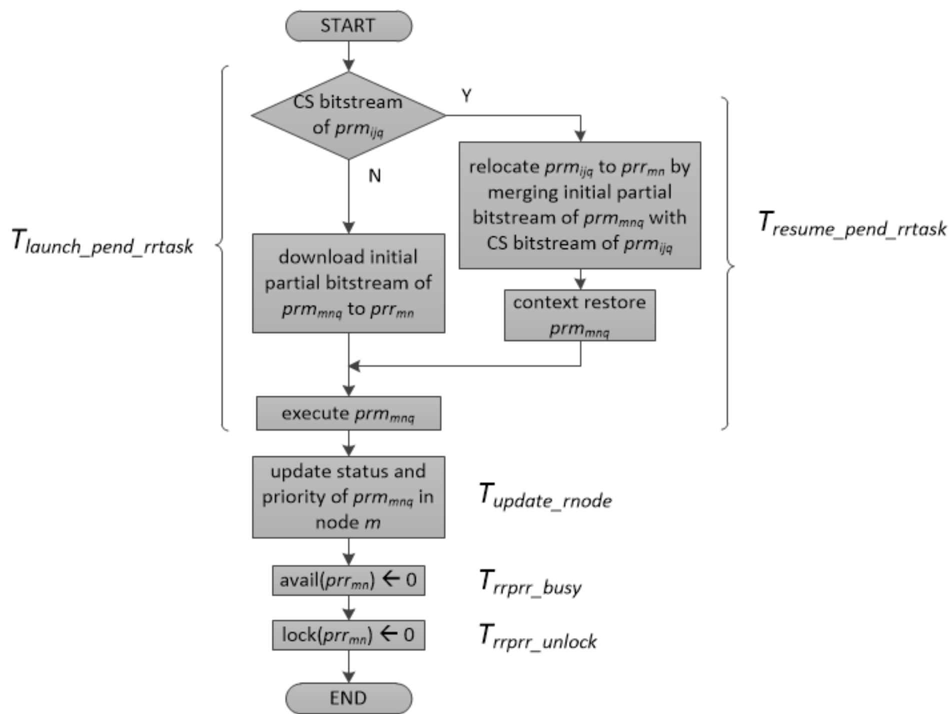

Figure 4-6. DRRM flow continued, showing the steps performed in the dark gray box in Figure 4-5 for execution of the remote relocation of prm_{ijq} to a remote candidate PRR n in node m (prr_{mn}) as prm_{mnq}.

CHAPTER 5
EXPERIMENTAL RESULTS

In this chapter we present the results of the implementation of our on-chip software-based CSR, HTR, and DDRM for 2-D heterogeneous PRRs on PR FPGAs. First, Section 5.1 presents our experimental setup used to measure the CSR, HTR, and DDRM execution times. Next, Section 5.2 and Section 5.3 evaluate the execution times for the major steps involved in CSR and HTR, respectively, for varying PRR sizes and organizations. Finally, Section 5.4 evaluates the DDRM execution times for several cases of starting/resuming PRM execution for locally (lHTR) and remotely (rHTR) relocated PRMs to the same or different FPGA, respectively, for varying PRR sizes and organizations. Even though our CSR/HTR/DDRM execution times evaluate trends for a particular FPGA device, these trends are not device specific and are indicative of general trends that can be expected on any PR FPGA that uses the ICAP and a soft-core processor as a reconfiguration controller.

5.1 Experimental setup

For our experiments, we used the Xilinx Virtex-5 XUPV5-LX110T board [56] and the Xilinx ISE 12.4, XPS 12.4 and PlanAhead 12.4 [47] tools. We partitioned the fabric into two heterogeneous PRRs and the static region executed a 100 MHz MicroBlaze [46] soft-core processor running a Linux-like OS 2.6.37 [55] based on BusyBox [6]. The executable binaries for the MicroBlaze were generated using GNU tools [54]. A XPS HWICAP interfaced the MicroBlaze and the ICAP, and SDRAM provided external storage for the bitstreams, binaries, and the CSR/HTR/DDRM files. The XPS timer was used to measure the T_x execution times for CSR, HTR, and DDRM, and we averaged the execution times over five experimental executions. Two XPS GPIOs provided

parallel interfaces between the MicroBlaze and the two PRRs (one XPS GPIO per PRR).

We note that the MicroBlaze's configuration [46] (e.g., instruction and data cache parameters), the XPS HWICAP's configuration [45], the Linux driver limitations for the HWICAP (maximum of 4 KB per readback), and the memory controller used to access files in the SDRAM introduce overheads that affect the results, however, these components' configurations do not impact the trends and analysis of our presented results, and in our discussions we note the impacts of different component configurations and hardware overheads. The MicroBlaze's instruction and data caches were direct-mapped, 16 KB caches, with eight and four 32-bit words per cache line, respectively. The data cache used a write-through policy, which affects multiple writes to contiguous data memory elements in the SDRAM. The XPS HWICAP was configured for 32-bit word data transfers with two first-in first-out (FIFO) registers, one for the read operations and the other for the write operations, each with a depth of 256 words) and operated at 100 MHz. This configuration limits CSR/HTR's performance during reconfiguration, especially for large data transfers, and when reading/writing, especially to random addresses, to large files in the SDRAM.

There are several prior works that we can compare to for ascertaining our CSR/HTR execution time improvements. Liu et al. [32] studied and enhanced the XPS HWICAP by adding DMA and BRAM to reduce PR times when the bitstreams were read from SDRAM, and reported a maximum reconfiguration throughput of 371.4 MB/s. Papadimitriou et al. [37] evaluated the effects of the XPS HWICAP and DDR SDRAM on PR and developed a cost model for PR reconfiguration time in terms of bitstream

size, and HWICAP and DDR bandwidths, reporting a maximum reconfiguration throughput of 924 KB/s. Vipin and Fahmy [44] proposed a custom ICAP controller for the Virtex-6 using two FIFOs and a DMA controller, achieving a 400 MB/s reconfiguration throughput, however no readback capability was considered. Duhem et al. [12] introduced FaRM (fast reconfiguration manager) for the Virtex-5, which used a DMA controller and two FIFOs for read and write access. FaRM achieved a reconfiguration throughput of 800 MB/s by overclocking the ICAP to 200 MHz and pre-loading the partial bitstream in a FIFO, while the readback reached 128 MB/s. Bonamy et al. [5] proposed UPaRC (ultra-fast power-aware reconfiguration controller) for the Virtex-5, which used a 256 KB BRAM for storing partial bitstreams and a custom reconfiguration controller with a DMA. UPaRC achieved a reconfiguration throughput of 1433 MB/s by overclocking the ICAP to 362.5 MHz using the dynamic reconfiguration port (DRP) feature in the Virtex-5, however, no readback capability was considered. Hansen et al. [15] presented an enhanced ICAP hard macro (ICAP_64) for the Virtex-5 using a 64 KB FIFO for storing partial bitstreams, a digital clock manager (DCM), and a phase locked loop (PLL). ICAP_64 achieved a reconfiguration throughput of 2200 MB/s by overclocking the ICAP to 550 MHz, however the readback was performed with a 100 MHz ICAP clock.

Even though we did not leverage some of the methods used in prior work (e.g., using BRAMs (which are limited in number), using a custom DMA controller, overclocking the ICAP, etc.), these methods could be incorporated into our CSR/HTR to improve CSR/HTR's performance with some tradeoff with respect to hardware overhead. We did not include these methods since our primary goal was to implement a

CSR/HTR solution that was portable across different Xilinx PR FPGA architectures without including custom hardware. This custom hardware could hinder portability, even across devices of the same family but with different speed grades, or could impose custom, non-portable placement and routing [15][44].

We verified CSR/HTR's correct operation using two interfaces per PRR, one connected to the MicroBlaze and one within the PRM, for transferring the PRM's LUTRAMs, BRAMs, and flip-flops' values to the MicroBlaze. For testing purposes, PRM1, PRM2, and PRM3 implemented a 32-bit up counter, down counter, and pipelined adder/accumulator, respectively. The counters and accumulator provided memory addresses to one LUTRAM (RAM32M) and one BRAM (RAMB36) [49]. All PRMs included a frequency divider using a 32-bit up counter and a finite state machine to transfer the LUTRAMs', BRAMs', and registers' values to the MicroBlaze. We tested the CSR operations using the flow in Figure 3-1, verifying that the values of each register, LUTRAM, and BRAM in PRM2 were successfully saved after the CS was executed, and later restored with the execution of the merge process and CR, in the same PRR. We also tested the HTR operations using the flow in Figure 3-1, verifying that the first value of each register, LUTRAM, and BRAM in PRM2 after CR on a different PRR (i.e., the task was relocated) corresponded to the last value of each register, LUTRAM, and BRAM in PRM2 prior to CS.

In order to generate thorough HTR results for many different PRR sizes in a timely manner (manual creation and testing for our experiments would have required an exorbitant amount of time) we used the following process, which did not affect the validity of our results. We created a project with two small heterogeneous PRRs

containing CLB flip-flops, one LUTRAM (RAM32M), and one BRAM (RAMB36), and selected two empty areas (areas with no CLB/DSP/IOB/BRAM columns and routing resources already in use) on the device fabric. In these empty areas, we created *pseudo PRRs*, *pseudo* initial partial bitstreams, and *pseudo* logic location files (*.ll) for *pseudo PRMs*. For HTR we copied and pasted the lines in the *.ll file that corresponded to the PRRs into several new *.ll files (one file per PRM-PRR pair) and we ordered the lines in these new files by net name for the CLB flip-flops, followed by the LUTRAM and BRAM bits ordered by frame by default (Section 3.4). For CSR, there is no need to perform any re-ordering in the generated *.ll files.

For thorough results with the pseudo PRRs, the entries in the pseudo logic location files were randomly ordered for the CLB flip-flops, simulating the effect of having the lines in the logic location files ordered by net name, followed by one pseudo LUTRAM (RAM32M) for each pseudo CLB column that contained a SLICEM [51], and followed by one pseudo BRAM (RAMB36). The pseudo PRRs sizes contained one row, one BRAM column, and multiple CLB columns ranging from one to twelve, which is the largest number of contiguous CLB columns on the Virtex-5 LX110T. The pseudo initial partial bitstreams initialized the pseudo PRR's flip-flops, LUTRAMs, and BRAMs to 0.

Table 5-1 shows the pseudo partial bitstream sizes (using our cost model for partial bitstream size, Section 2.2.3) and pseudo CS bitstream sizes (Section 3.2) used in the CSR, HTR, and DDRM experiments, for single-row pseudo PRRs containing a varying number of pseudo CLB columns, where all pseudo PRRs include one pseudo BRAM column.

Our HTR experiments evaluated context relocation from a small PRR to a large PRR (small-to-large PRR HTR) and from a large PRR to a small PRR (large-to-small PRR HTR). We denoted the pseudo PRR with the PRM's context as the *source* PRR and the pseudo PRR with the relocated context as the *destination* PRR. Since the number of experimental combinations given our pseudo PRR sizes is 144, we subset the results to show the 12 combinations where the small pseudo PRR has half the number of columns as the large pseudo PRR, which is sufficient to show the execution times' trends. In the large pseudo PRRs, we evenly distributed the PRM's flip-flops across the CLB columns, which simulated the effects of the Xilinx tool's flip-flops distribution done during placement and provided realistic execution times. For example, if the small pseudo PRR has two CLB columns and uses 320 flip-flops, the large pseudo PRR has four columns and the 320 flip-flops are distributed across all four columns (80 flip-flops per column).

We verified DDRM's correct operation using four Xilinx XUPV5-LX110T boards interconnected in a FastEthernet network, where each FPGA in the network had the same setup (static region and PRRs, and PRMs/PRRs mapping) and Linux-like OS as for the HTR setup. We created the global and local tables (Table 4-1 and Table 4-2, respectively) in each FPGA in the network for DDRM node consistency, defined priority execution levels for each PRM for lHTR (same FPGA) or rHTR (between FPGAs) CR, and verified the DDRM flow (Figure 4-1 to Figure 4-6).

Our DDRM experiments evaluated PRMs starting execution for the first time and resuming a preempted PRM under several conditions (e.g., starting/resuming the PRM execution in a free predefined PRR, starting/resuming the PRM execution in a busy

predefined PRR currently executing a lower/higher priority execution level PRM, starting/resuming the PRM execution in a remote node, etc.) in order to verify lHTR, rHTR, broadcasting of the tasks' statuses to all nodes for node consistency, etc. We followed the same procedure as for our HTR experiments, using pseudo PRRs, pseudo initial partial bitstreams, and pseudo logic location files (*.ll) for pseudo PRMs, however, we restrict lHTR and rHTR experiments for small-to-large PRR lHTR and rHTR in order to show the worst case scenario impact of lHTR and rHTR in DDRM execution times.

All DDRM execution times (Section 5.4) consider one predefined PRR, one local candidate PRR, and one remote candidate PRR for local or remote context relocation of PRMs (lHTR or rHTR, respectively), as defined in the global table (Table 4-1) for each node in the network. Adding more local or remote candidate PRRs for context relocation enhances the flexibility for context relocation, and maximizes PRR usage, however adding more local or remote PRR candidates will negatively impact DDRM execution times. Therefore, at design time, a system designer needs to make appropriate tradeoffs between PRR granularity, lHTR or rHTR or no HTR for each PRM, and HTR/DDRM execution times when partitioning the application into PRMs based on the application's requirements.

5.2 CSR Experimental Results

Table 5-2 through Table 5-5 show the execution times in milliseconds for the significant CSR steps for heterogeneous PRRs that contain one BRAM column and multiple CLB columns. Figure 5-1 through Figure 5-4 plot the results from Table 5-2 through Table 5-5.

Table 5-2 and Figure 5-1 summarize $T_{reconfig_prr}$, which depends on the number of PRR frames (36 configuration frames per CLB column, and 30 configuration frames and

128 data frames per BRAM column) in the partial bitstream. $T_{reconfig_prr}$ shows a linear behavior up to 1,600 PRM flip-flops. However, for PRRs using more than 1,600 PRM flip-flops, high data cache miss rates, and SDRAM overheads when accessing large bitstreams introduce a small non-linear increase in the growth rate of $T_{reconfig_prr}$.

The execution time for $T_{protect_fpga}$ (Figure 3-1) is constant and depends on the number of rows and columns in the device, which is 67.72 ms for the Virtex-5 LX110T.

Table 5-3 and Figure 5-2 summarize the execution times for T_{cs}. $T_{unprotect_prr_cs}$ and $T_{protect_prr_cs}$ depend on the number of rows and columns of the PRR to unprotect/protect, respectively, and show a linear increase in the growth rate. $T_{cs_bitstream}$ depends on the number of PRM frames where flip-flop, LUTRAM, and BRAM bits are part of the saved context, and shows a linear increase in the growth rate. Since every other CLB column in the PRRs contains LUTRAMs, the behavior of T_{cs_prr} affects T_{cs}, and shows a nearly linear increase.

Table 5-4 and Figure 5-3 summarize the execution times for T_{merge}. T_{merge} depends on the number of PRM frames where flip-flop, LUTRAM, and BRAM bits are part of the saved context, and shows a linear increase.

Table 5-5 and Figure 5-4 summarize the execution times for T_{cr}. $T_{unprotect_prr_cr}$ and $T_{protect_prr_cr}$ depend on the number of rows and columns of the PRR to unprotect/protect, respectively, and show a linear increase in the growth rate. $T_{startup}$ is the execution time to re-initialize the PRM flip-flops, LUTRAMs and BRAMs, and is independent of the PRR size. T_{update_prr} depends on the number of PRR frames in the merged bitstream affecting the behavior of T_{cr}, and shows a linear increase.

With the experimental setup explained in Section 5.1, and from the experimental results given in Table 5-2 through Table 5-5, we can derive the execution times in milliseconds for $T_{reconfig_prr}$, T_{cs}, T_{merge}, and T_{cr} for PRRs with H rows, one BRAM and W_{CLB} CLB columns for Virtex-5 devices, using a linear polynomial curve fitting [33], under the assumption that the PRR sizes do not affect the PR system's maximum operating frequency, which in our case is 100 MHz. This assumption does not necessarily hold true for large PRRs, because large PRRs impose longer routing delays and cause a reduction in the system's maximum operating frequency, and as a consequence cause an increase in the CSR execution times, but this formulation keeps the CSR execution times' trends and gives a lower bound of the execution times a system designer expects when implementing a PR system for hardware multitasking using a soft-core processor.

Then, assuming that the PRR sizes do not affect the PR system's maximum operating frequency, $T_{reconfig_prr}$, T_{cs}, T_{merge}, and T_{cr} (in milliseconds) for PRRs with H rows, one BRAM and W_{CLB} CLB columns, are expressed as follows:

$$T_{reconfig_prr} = H \text{ x } \{0.75 \text{ x } W_{CLB} + 2.71\} \quad (5\text{-}1)$$

$$T_{cs} = H \text{ x } \{1.65 \text{ x } W_{CLB} + 16.76\} + 2.28 \quad (5\text{-}2)$$

$$T_{merge} = H \text{ x } \{1.44 \text{ x } W_{CLB} + 21.95\} \quad (5\text{-}3)$$

$$T_{cr} = H \text{ x } \{W_{CLB} + 4.71\} + 0.33 \quad (5\text{-}4)$$

where $W_{CLB} = 1$ means one PRM's CLB column that uses a total of one to 160 CLB flip-flops, $W_{CLB} = 2$ means two PRM's CLB columns that use a total of 161 to 320 CLB flip-flops, etc.

In the case of the Virtex-5 LX110T used in our experiments, and under the assumption that large PRRs do not affect the system's maximum operating frequency, (5-1) through (5-4) would be valid for PRRs with one to 8 rows, and one to 22 CLB columns, which is the maximum number of CLB columns a PRR may have with only one BRAM column.

5.3 HTR Experimental Results

Table 5-6 and Table 5-7 show the execution times in milliseconds for the significant HTR steps for heterogeneous PRRs that contain one BRAM column and multiple CLB columns. Figure 5-5 through Figure 5-7 plot the results from Table 5-6 through Table 5-7, where each point is identified by a box with the number of rows, columns, and PRR or PRM frames, which depends on the graph's reported execution time.

Table 5-6 and Table 5-7 summarize the execution times for T_{cs}, $T_{relocate}$ and T_{cr} (for small-to-large PRR HTR and for large-to-small PRR HTR, respectively). Figure 5-5 depicts the T_{cs} results from Table 5-6 and Table 5-7. For brevity, we omit the detailed breakdown of T_{cs}, which depends on the number of PRM frames (where PRM flip-flops, LUTRAMs, and BRAMs are used) in the source pseudo PRR. Capturing and saving the context in a small PRR shows a nearly linear behavior in T_{cs}.

Figure 5-6 depicts the $T_{relocate}$ results from Table 5-6 and Table 5-7. $T_{relocate}$ depends on the number of PRM flip-flop, LUTRAM, and BRAM bits used in the PRM's context to be relocated. Accesses to the CS and merged bitstreams are random for CLB flip-flops since the logic location files for the PRM's CLBs' flip-flops in the source and destination PRRs are ordered by net name and not in ascending order by frame. Alternatively, accesses to LUTRAM and BRAM bits are sequential (by frame) in the CS

and the merged bitstreams. Random accesses to the CS and merged bitstreams for CLB flip-flops produces high data cache miss rates and overheads in SDRAM accesses, which affect $T_{relocate}$'s non-linear behavior.

Figure 5-7 depicts the T_{cr} results from Table 5-6 and Table 5-7. For brevity, we omit the detailed breakdown of T_{cr}, which depends on the number of PRR frames in the destination PRR. Since the destination PRR is unprotected and then re-protected (before and after CR, respectively), the execution times for T_{cr} are larger than $T_{reconfig_prr}$ for the same number of PRM flip-flops and PRR frames in Figure 5-1.

Based on the results depicted in Figure 5-5 through Figure 5-7, it is more efficient to relocate a task from a large to a small PRR since $T_{relocate}$ and T_{cr} are faster than relocating from a small to a large PRR. Even though T_{cs} is longer for relocating a task from a large to a small PRR as compared to relocating from a small to a large PRR, $T_{relocate}$ is much longer than T_{cs} and T_{cr}.

We note that the execution times reported here for $T_{relocate}$ are improved as compared to our prior work [34] due to an additional pre-processing step that extracts information from the *.ll files. This pre-processing generates smaller binary files to operate on instead of directly using the *.ll files, which may be excessively large when using BRAMs. Each BRAM (RAMB36) requires 36K entries in the *.ll file and accessing a large *.ll file from SDRAM severely impacts $T_{relocate}$.

The resources required by the static region, including the MicroBlaze, XPS HWICAP and GPIOs, and SDRAM controller are 12,898, 44, and 4 flip-flops, BRAMs, and DSPs, respectively, which represent 19%, 30%, and 6%, respectively, of the test

device. We note that this area overhead is reduced for devices with a dedicated on-chip hardcore processor.

Increasing the number of rows in the PRRs and reducing the number of columns while maintaining the same number of PRM flip-flops would reveal similar results as shown in Table 5-2, and Table 5-6 through Table 5-7, and in Figure 5-1, and Figure 5-5 through Figure 5-7. However, for PRRs using more than 1,600 PRM flip-flops, high data cache miss rates, SDRAM overheads when accessing the large bitstreams, and the configuration of the XPS HWICAP introduces a non-linear increase in the growth rate of the HTR execution times. All HTR execution times may be improved by adding a custom DMA and enlarging the XPS HWICAP's internal storage, saving the CS and initial partial bitstreams in BRAMs, overclocking the ICAP, or increasing/modifying the MicroBlaze's data cache size/configuration. However, BRAMs are limited in number and size, and these options incur hardware overhead that may affect the performance of the system, and some of these modifications would not be portable across other systems [15][44]. Therefore, at design time, a system designer can consider these factors and make appropriate tradeoffs between PRR granularity, hardware overhead, and HTR execution times when partitioning the application into tasks based on the application's requirements.

Similarly to the CSR experimental results (Section 5.2), and from the experimental results in HTR given in Table 5-6 and Table 5-7, we can derive the execution times in milliseconds for T_{cs}, $T_{relocate}$ and T_{cr} for small-to-large and large-to-small PRR HTR cases and for PRRs with H rows, one BRAM and W_{CLB} CLB columns for Virtex-5 devices, using a linear, quadratic, or exponential polynomial curve fitting [33]

(depending on the HTR execution times' trends), and under the assumption that the PRR sizes do not affect the PR system's maximum operating frequency. This assumption does not necessarily hold true for large PRRs, and especially for $T_{relocate}$, since the access to CLB flip-flops in the CS and initial partial bitstreams (for the source and destination PRRs, respectively) is random, and large PRRs would negatively impact the access time to those flip-flops when HTR generates the merged bitstream for the destination PRR. However, the system designer would get a lower bound estimation of the HTR execution times for a PR system given the experimental setup in Section 5.1.

Then, assuming that the PRR sizes do not affect the PR system's maximum operating frequency, $T_{cs_small\text{-}to\text{-}large}$, $T_{relocate_small\text{-}to\text{-}large}$, and $T_{cr_small\text{-}to\text{-}large}$ (in milliseconds) denote the HTR execution times for T_{cs}, $T_{relocate}$, and T_{cr}, respectively, for small-to-large PRR HTR, and are expressed as follows:

$$T_{cs_small\text{-}to\text{-}large} = H \times \{1.6 \times W_{CLB} + 17.71\} + 3.09 \tag{5-5}$$

$$T_{relocate_small\text{-}to\text{-}large} = H \times \{6.27 \times (W_{CLB})^2 + 6.32 \times W_{CLB} + 35.68\} \tag{5-6}$$

$$T_{cr_small\text{-}to\text{-}large} = H \times \{1.55 \times W_{CLB} + 7\} + 0.83 \tag{5-7}$$

Since our HTR experiments considered that in the small-to-large PRR HTR the small PRR (i.e., the source PRR) has half the number of CLB columns as the large PRR (i.e., the destination PRR), W_{CLB} will only change from one to 11 CLB columns (for the source PRR) in (5-5) to (5-7). Also, as explained in Section 5.1, the CLB flip-flops in the large PRR are evenly distributed across all the PRR's CLB columns.

Also, assuming that the PRR sizes do not affect the PR system's maximum operating frequency, $T_{cs_large\text{-}to\text{-}small}$, $T_{relocate_large\text{-}to\text{-}small}$, and $T_{cr_large\text{-}to\text{-}small}$ (in milliseconds)

denote the HTR execution times for T_{cs}, $T_{relocate}$, and T_{cr}, respectively, for large-to-small PRR HTR, and are expressed as follows:

$$T_{cs_large\text{-}to\text{-}small} = H \times \{2.21 \times W_{CLB} + 18.47\} + 3.09 \quad (5\text{-}8)$$

$$T_{relocate_large\text{-}to\text{-}small} = 35.8 \times e^{(0.3 \times W_{CLB})} \quad (5\text{-}9)$$

$$T_{cr_large\text{-}to\text{-}small} = H \times \{0.84 \times W_{CLB} + 6.34\} + 0.83 \quad (5\text{-}10)$$

Since our HTR experiments considered that in the large-to-small PRR HTR the large PRR (i.e., the source PRR) has double the number of CLB columns as the small PRR (i.e., the destination PRR), W_{CLB} will change from one to 11 CLB columns (for the destination PRR) in (5-8) to (5-10). Also, as in small-to-large PRR HTR, the CLB flip-flops in the large PRR are evenly distributed across all the PRR's CLB columns.

5.4 DDRM Experimental Results

Table 5-8 through Table 5-17 show execution times for T_{exe1}, T_{res1}, T_{exe2}, T_{res3}, T_{exe4}, and $T_{exeres2}$ (Section 4.3) in milliseconds for the significant DDRM steps for 2-D heterogeneous PRRs that contain one BRAM column and multiple CLB columns. Figure 5-8 through Figure 5-13 plot the results from Table 5-8 through Table 5-17, respectively. For the execution times that involve lHTR (T_{res3} and T_{exe4}) and rHTR ($T_{exeres2}$), the destination PRRs have double the number of CLB columns as compared to the source PRRs, which reveals the worst case scenario impact of lHTR and rHTR on DDRM execution times, having similar trend results as the execution times produced for small-to-large PRR HTR experiments (Section 5.3).

All DDRM execution times are measured from the point in time when a client in the local node initiates a request for starting/resuming execution of a PRM. We note that the DDRM execution times are affected when remote nodes are involved (e.g., broadcasting the status of PRMs, transferring the CS bitstream to a remote node,

checking PRR availability on remote nodes, executing rHTR, etc.) due to delays in accessing the remote nodes. These delays depend on the physical medium and maximum transfer rate of the network used to interconnect the nodes for DDRM. For our initial version of DDRM, these delays are related to accessing remote nodes in a FastEthernet network.

Table 5-8 and Figure 5-8 summarize the execution times for T_{exe1} (starting execution of a PRM for the first time in a predefined local available PRR). T_{exe1} shows a linear behavior that depends on $T_{launch_pend_task}$, which depends on $T_{reconfig_prr}$. All other execution times in Table 5-8 are nearly constant. We note that the execution time for broadcasting the status of the executing PRM to all four nodes ($T_{prop_act_task}$) in the FastEthernet network contributes an average of 44 ms to T_{exe1}. We note that 60% of $T_{prop_act_task}$ represents overhead, which is dictated by the delays in accessing the remote nodes using a FastEthernet network.

Table 5-9 and Figure 5-9 summarize the execution times for T_{res1} (PRM resumption in a predefined local available PRR). T_{res1} depends on the behavior of $T_{resume_pend_task}$, which depends on T_{merge} and T_{cr}. All other execution times in Table 5-9 are nearly constant. T_{merge} depends on the number of BRAM, LUTRAM, and flip-flop bits contained in the CS bitstream and T_{cr} depends on the size of the merged bitstream. Since every other CLB column in the PRRs contains LUTRAMs, the behavior of T_{merge} affects T_{res1}, and shows a nearly linear increase. Similarly to T_{exe1}, $T_{prop_act_task}$ contributes to T_{res1} with an average of 44 ms.

Table 5-10, Table 5-11, and Figure 5-10 summarize the execution times for T_{exe2} (starting execution of a PRM for the first time in a predefined local busy PRR that is

executing a lower priority execution level PRM). T_{exe2} depends on the behavior of $T_{cs_preempted_task}$ and $T_{launch_pend_task}$, and all other execution times in Table 5-10 and Table 5-11 are nearly constant. $T_{cs_preempted_task}$ depends on the number of frames that contain LUTRAMs, BRAMs and flip-flops and are part of the PRM's context, which in turn affects T_{cs}. Since every other CLB column in the PRRs contains LUTRAMs, $T_{cs_preempted_task}$ shows a nearly linear behavior. Similarly to T_{exe1}, $T_{launch_pend_task}$ depends on the behavior of $T_{reconfig_prr}$. The combination of $T_{cs_preempted_task}$ and $T_{launch_pend_task}$ affects T_{exe2}, and shows a nearly linear behavior. Broadcasting the status of the preempted PRM ($T_{prop_preempted_task}$) and the status of the resumed for execution PRM ($T_{prop_act_task}$) are both on average 44 ms.

Table 5-12, Table 5-13, and Figure 5-11 summarize the execution times for T_{res3} (PRM resumption using IHTR in a free and unlocked local candidate PRR). T_{res3} depends on the behavior of $T_{resume_pend_rtask}$, which depends on $T_{relocate}$ and T_{cr}. All other execution times in Table 5-12 and Table 5-13 are nearly constant. Since we are executing IHTR and the destination PRR has twice as many CLB columns as compared to the source PRR, T_{res3} shows similar non-linear behavior as the small-to-large PRR HTR (Figure 5-6).

Table 5-14, Table 5-15, and Figure 5-12 summarize the execution times for T_{exe4} (starting execution of a PRM for the first time in busy local candidate PRR that is executing a lower priority execution level PRM). T_{exe4} depends on the behavior of $T_{cs_preempted_task}$ and $T_{launch_pend_rtask}$. All other execution times in Table 5-14 and Table 5-15 are nearly constant. The combination of $T_{cs_preempted_task}$ and $T_{launch_pend_rtask}$ affects T_{exe4}, and shows a nearly linear behavior, similar to T_{exe2}.

Table 5-16, Table 5-17, and Figure 5-13 summarize the execution times for $T_{exeres2}$ (the largest execution time for PRM resumption using rHTR in a remote candidate PRR that is executing a lower priority execution level PRM). $T_{exeres2}$ depends on the behavior of $T_{cs_preempted_rprm}$ and T_{remote_htr}. All other execution times in Table 5-16 and Table 5-17 are nearly constant. $T_{cs_preempted_rprm}$ depends on the number of frames that contain LUTRAMs, BRAMs, and flip-flops and are part of the remote PRM's context, and shows a nearly linear behavior. Since we are executing rHTR and the destination PRR (in the remote node) has twice as many CLB columns as compared to the source PRR (in the local node), T_{remote_htr} shows a similar non-linear behavior as the small-to-large PRR HTR (Section 5.3, Figure 5-6), which affects $T_{exeres2}$ and shows a non-linear behavior.

From Table 5-17 and Figure 5-13 we note that T_{remote_htr} updates the status (executing) of the remotely relocated PRM in the remote node (T_{update_rnode} in Figure 4-6), and the broadcasting of this status to the rest of nodes in the FastEthernet network is $T_{prop_rhtr_status}$ with an average of 27.4 ms, where 52% of this time corresponds to overhead due to accessing to the rest of nodes.

Also, from Table 5-17 we note that the execution time to transfer the CS bitstream from local node to remote node ($T_{ftp_csbitstream}$) is 412 ms in average, which is independent of the CS bitstream size. This result cannot be taken as a general rule, and may be due to the small range variation in the CS bitstream size (from 22.0 KB to 28.9 KB, Table 5-1), the physical medium and maximum transfer rate in the FastEthernet network used for our initial version of DDRM. We note that 99% of $T_{ftp_cbitstream}$ represents overheads, including accessing the CS bitstream in external SDRAM in the

local node, establishing FTP connection to the remote node, transferring the CS bitstream using a FastEthernet network, and saving the transferred CS bitstream in external SDRAM in the remote node.

Similarly to the HTR experimental results (Section 5.3), and from the DDRM experimental results given in Table 5-8 through Table 5-17, we can derive the execution times in milliseconds for T_{exe1}, T_{res1}, T_{exe2}, T_{res3}, T_{exe4}, and $T_{exeres2}$ for PRRs with H rows, one BRAM and W_{CLB} CLB columns for identical Virtex-5 FPGA devices in the DDRM network, using a linear, or quadratic polynomial curve fitting [33] (depending on the DDRM execution times' trends), under the assumption that the PRR sizes do not affect the node's maximum operating frequency. As in Section 5.2 and Section 5.3, this assumption does not necessarily hold true for large PRRs, and especially for T_{res3} and $T_{exeres2}$, where small-to-large PRR lHTR and rHTR are involved, respectively, due to random accesses to CLB flip-flops in the CS and initial partial bitstreams (in the source and destination PRRs, respectively) in order to generate the merged bitstream in the destination PRR. However, the system designer would get a lower bound estimation of the DDRM execution times given the experimental setup in Section 5.1.

With this assumption, T_{exe1}, T_{res1}, T_{exe2}, T_{res3}, T_{exe4}, and $T_{exeres2}$ (in milliseconds) for PRRs with H rows, one BRAM and W_{CLB} CLB columns, are expressed as follows:

$$T_{exe1} = H \times \{0.82 \times W_{CLB} + 6.95\} + 73.56 \quad (5\text{-}11)$$

$$T_{res1} = H \times \{2.86 \times W_{CLB} + 29.7\} + 74.44 \quad (5\text{-}12)$$

$$T_{exe2} = H \times \{2.38 \times W_{CLB} + 26.56\} + 129.32 \quad (5\text{-}13)$$

$$T_{res3} = H \times \{6.17 \times (W_{CLB})^2 + 9.14 \times W_{CLB} + 41.91\} + 96.55 \quad (5\text{-}14)$$

$$T_{exe4} = H \times \{4.24 \times W_{CLB} + 26.06\} + 152.37 \quad (5\text{-}15)$$

$$T_{exeres2} = H \times \{6.4 \times (W_{CLB})^2 + 10.24 \times W_{CLB} + 89.87\} + 573.77 \quad (5\text{-}16)$$

Since T_{res3} and $T_{exeres2}$ in our DDRM experiments considered the small-to-large PRR lHTR and rHTR, respectively, the small PRR (i.e., the source PRR) has half the number of CLB columns as the large PRR (i.e., the destination PRR), then, for the Virtex-5 LX110T device used in our experiments, W_{CLB} will only change from one to 11 CLB columns (source PRR) in (5-11) to (5-16).

Table 5-1. CS bitstream and partial bitstream sizes (in KB) used in the CSR, HTR, and DDRM experiments

CLB columns	Flip-flops frames in CS	LUTRAM frames in CS	BRAM frames in CS	Total frames in CS	CS bitstream size (KB)	Partial bitstream size (KB)
1	2	4	128	134	22.0	31.9
2	4	4	128	136	22.3	37.7
3	6	8	128	142	23.3	43.5
4	8	8	128	144	23.6	49.2
5	10	12	128	150	24.6	55.0
6	12	12	128	152	24.9	60.8
7	14	16	128	158	25.9	66.5
8	16	16	128	160	26.2	72.3
9	18	20	128	166	27.2	78.1
10	20	20	128	168	27.6	83.8
11	22	24	128	174	28.5	89.6
12	24	24	128	176	28.9	95.4

Note: All partial bitstreams include one BRAM column, and all PRRs have one row.

Table 5-2. Execution times (ms) for $T_{reconfig_prr}$

CLB columns	CLB frames in PRR	BRAM configuration frames in PRR	BRAM data frames in PRR	Total frames in PRR	PRM flip-flops in the PRR	$T_{reconfig_prr}$
1	36	30	128	194	160	3.44
2	72	30	128	230	320	4.26
3	108	30	128	266	480	5.00
4	144	30	128	302	640	5.79
5	180	30	128	338	800	6.45
6	216	30	128	374	960	7.27
7	252	30	128	410	1120	7.95
8	288	30	128	446	1280	8.76
9	324	30	128	482	1440	9.45
10	360	30	128	518	1600	10.27
11	396	30	128	554	1760	11.10
12	432	30	128	590	1920	11.81

Note: All PRRs have one row and include one BRAM column.

Table 5-3. Execution times (ms) for CS (T_{cs}) in CSR

CLB columns	PRM flip-flops	T_{pre_cs}	$T_{unprotect_prr_cs}$	T_{cs_prr}	T_{post_cs}	$T_{protect_prr_cs}$	$T_{desynch}$	$T_{cs_bitstream}$	T_{cs}
1	160	0.31	1.12	8.75	0.20	1.24	1.76	7.36	20.74
2	320	0.31	1.26	9.29	0.20	1.36	1.76	7.52	21.70
3	480	0.31	1.38	11.10	0.20	1.51	1.76	8.06	24.32
4	640	0.31	1.54	11.56	0.20	1.68	1.76	8.33	25.38
5	800	0.31	1.70	13.40	0.20	1.79	1.76	8.84	28.00
6	960	0.31	1.76	14.08	0.20	1.93	1.76	9.08	29.12
7	1120	0.31	1.92	15.09	0.20	2.05	1.76	9.40	30.73
8	1280	0.31	2.07	15.65	0.20	2.17	1.76	9.55	31.71
9	1440	0.31	2.19	17.43	0.20	2.29	1.76	10.02	34.20
10	1600	0.31	2.33	18.02	0.20	2.44	1.76	10.24	35.30
11	1760	0.31	2.46	19.16	0.20	2.63	1.76	10.68	37.20
12	1920	0.31	2.59	20.31	0.20	2.73	1.76	11.05	38.95

Note: All PRRs have one row and include one BRAM column.

Table 5-4. Execution times (ms) for the merge process (T_{merge}) in CSR

CLB columns	PRM frames	PRM flip-flops	T_{merge}
1	134	160	23.22
2	136	320	24.71
3	142	480	26.40
4	144	640	27.80
5	150	800	29.16
6	152	960	30.60
7	158	1120	32.18
8	160	1280	33.40
9	166	1440	35.02
10	168	1600	36.21
11	174	1760	37.60
12	176	1920	39.06

Note: All PRRs have one row and include one BRAM column.

Table 5-5. Execution times (ms) for CR (T_{cr}) in CSR

CLB columns	PRM flip-flops	T_{pre_cr}	$T_{unprotect_prr_cr}$	T_{update_prr}	$T_{startup}$	$T_{protect_prr_cr}$	T_{cr}
1	160	0.30	1.23	3.25	0.03	1.31	6.12
2	320	0.30	1.40	3.85	0.03	1.46	7.04
3	480	0.30	1.52	4.52	0.03	1.58	7.95
4	640	0.30	1.65	5.36	0.03	1.81	9.15
5	800	0.30	1.77	6.04	0.03	1.86	10.00
6	960	0.30	1.87	6.73	0.03	1.95	10.88
7	1120	0.30	1.99	7.46	0.03	2.29	12.07
8	1280	0.30	2.13	8.11	0.03	2.45	13.02
9	1440	0.30	2.25	8.78	0.03	2.63	13.99
10	1600	0.30	2.37	9.44	0.03	2.84	14.98
11	1760	0.30	2.53	10.21	0.03	3.07	16.14
12	1920	0.30	2.63	10.96	0.03	3.18	17.10

Note: All PRRs have one row and include one BRAM column.

Table 5-6. Execution times (ms) for CS (T_{cs}), context relocation ($T_{relocate}$), and CR (T_{cr}) for small-to-large PRR HTR

Source PRR CLB columns	Source PRR PRM frames	Destination PRR CLB columns	Destination PRR PRM frames	PRM flip-flops	T_{cs}	$T_{relocate}$	T_{cr}
1	134	2	136	160	22.29	47.93	9.31
2	136	4	140	320	23.91	74.28	10.98
3	142	6	148	480	26.07	110.35	12.56
4	144	8	152	640	27.06	161.35	14.10
5	150	10	160	800	29.28	223.98	15.58
6	152	12	164	960	30.20	299.30	17.13

Note: The source and destination PRRs have one row, and all PRRs include one BRAM column.

Table 5-7. Execution times (ms) for CS (T_{cs}), context relocation ($T_{relocate}$), and CR (T_{cr}) for large-to-small PRR HTR

Source PRR CLB columns	Source PRR PRM frames	Destination PRR CLB columns	Destination PRR PRM frames	PRM flip-flops	T_{cs}	$T_{relocate}$	T_{cr}
2	136	1	134	160	23.70	45.52	7.99
4	140	2	136	320	25.68	62.15	8.82
6	148	3	142	480	28.49	88.37	9.72
8	152	4	144	640	30.24	121.36	10.55
10	160	5	150	800	33.02	169.49	11.39
12	164	6	152	960	34.30	214.24	12.15

Note: The source and destination PRRs have one row, and all PRRs include one BRAM column.

Table 5-8. DDRM execution times (ms) for T_{exe1} with respect to the number of PRM flip-flops. All PRRs have one row and include one BRAM column.

PRM flip-flops	$T_{chk_lprr_lock}$	T_{lprr_lock}	$T_{launch_pend_task}$	T_{update_node}	$T_{prop_act_task}$	T_{lprr_busy}	T_{lprr_unlock}	T_{exe1}
160	6.0	6.1	7.9	5.9	44.1	5.9	5.7	81.6
320	6.0	6.1	8.5	5.9	44.0	5.9	5.8	82.2
480	6.0	6.1	9.4	5.8	44.0	5.9	5.8	83.0
640	5.9	6.1	10.1	5.8	43.6	5.8	5.7	83.0
800	5.9	6.1	11.0	5.7	43.8	5.7	5.8	84.0
960	6.0	6.1	11.8	5.8	44.0	5.8	5.9	85.4
1120	6.1	6.2	12.7	5.8	44.1	5.8	5.9	86.6
1280	6.0	6.1	13.3	5.7	44.2	5.7	5.9	86.9
1440	6.0	6.1	14.5	5.8	44.0	5.8	5.8	88.0
1600	5.9	6.1	15.3	5.8	43.7	5.8	5.7	88.3
1760	6.0	6.1	15.8	5.8	44.3	5.8	5.7	89.5
1920	6.0	6.2	16.6	5.9	43.5	5.9	5.8	89.9

Table 5-9. DDRM execution times (ms) for T_{res1} with respect to the number of PRM flip-flops. All PRRs have one row and include one BRAM column.

PRM flip-flops	$T_{chk_lprr_lock}$	T_{lprr_lock}	$T_{resume_pend_task}$	T_{update_node}	$T_{prop_act_task}$	T_{lprr_busy}	T_{lprr_unlock}	T_{res1}
160	6.0	6.1	34.4	5.8	43.9	5.8	5.8	107.8
320	5.9	6.0	34.5	5.8	46.3	5.7	5.8	110.0
480	5.9	6.1	38.3	5.9	45.2	5.8	5.7	112.9
640	5.9	6.1	41.3	5.8	43.9	5.8	5.8	114.6
800	5.9	6.0	43.2	5.7	43.7	5.8	5.8	116.1
960	6.0	6.2	45.2	5.9	45.1	5.9	5.8	120.1
1120	6.1	6.2	49.6	5.9	44.2	5.8	5.9	123.7
1280	6.0	6.1	53.5	5.8	43.9	5.8	5.8	126.9
1440	6.0	6.1	55.9	5.9	44.0	5.9	5.7	129.5
1600	5.9	6.1	57.9	5.8	43.8	5.8	5.7	131.0
1760	6.0	6.1	61.2	5.7	44.0	5.7	5.8	134.5
1920	6.0	6.2	64.5	5.9	44.1	5.9	5.8	138.4

Table 5-10. DDRM execution times (ms) for T_{exe2} with respect to the number of PRM flip-flops (part 1 of 2). All PRRs have one row and include one BRAM column.

PRM flip-flops	$T_{chk_lprr_lock}$	$T_{chk_lprm_prr}$	T_{lprr_lock}	$T_{cs_preempted_task}$	$T_{prop_preempted_task}$
160	6.0	17.8	5.8	21.4	44.2
320	5.9	17.6	5.7	22.2	43.9
480	6.0	17.8	5.8	24.8	44.3
640	5.9	17.5	5.7	25.3	45.1
800	6.0	17.7	5.8	28.0	44.1
960	6.0	17.7	5.8	28.5	44.2
1120	6.1	17.8	5.8	31.0	44.2
1280	6.0	17.6	5.7	32.0	44.0
1440	6.0	17.6	5.8	34.5	44.0
1600	5.9	18.0	5.7	35.4	44.0
1760	6.0	17.6	5.7	36.3	44.1
1920	6.0	17.7	5.8	38.4	44.2

Table 5-11. DDRM execution times (ms) for T_{exe2} with respect to the number of PRM flip-flops (part 2 of 2). All PRRs have one row and include one BRAM column.

PRM flip-flops	$T_{launch_pend_task}$	T_{update_node}	$T_{prop_act_task}$	T_{lprr_unlock}	T_{exe2}
160	7.9	5.8	44.0	5.8	158.7
320	8.5	5.7	43.8	5.7	159.0
480	9.5	5.9	44.1	5.9	164.1
640	10.0	5.8	43.9	5.8	165.0
800	11.2	5.8	44.0	5.8	168.4
960	11.7	5.8	44.0	5.8	169.5
1120	12.8	5.9	44.1	5.9	173.6
1280	13.3	5.8	43.9	5.8	174.1
1440	14.3	5.8	43.9	5.8	177.7
1600	15.1	5.7	43.9	5.7	179.4
1760	16.0	5.8	43.9	5.7	181.1
1920	16.9	5.9	44.0	5.9	184.8

Table 5-12. DDRM execution times (ms) for T_{res3} with respect to the number of PRM flip-flops (part 1 of 2). All PRRs have one row and include one BRAM column.

PRM flip-flops	$T_{chk_lprr_lock}$	$T_{chk_lprm_prr}$	$T_{chk_lrprr_lock}$	T_{lrprr_lock}	$T_{resume_pend_rtask}$
160	6.0	17.7	5.8	5.8	57.6
320	5.9	17.5	5.7	5.7	84.8
480	6.0	17.8	5.8	5.8	124.3
640	6.0	17.5	5.7	5.7	175.7
800	6.0	17.8	5.8	5.8	245.2
960	6.0	17.8	5.8	5.8	317.6
1120	5.9	17.6	5.7	5.7	408.4
1280	6.0	17.7	5.7	5.7	510.2
1440	6.0	17.7	5.7	5.7	624.3
1600	6.0	17.6	5.7	5.8	750.7
1760	5.9	17.7	5.7	5.8	889.5
1920	6.0	17.7	5.7	5.8	1,040.7

Table 5-13. DDRM execution times (ms) for T_{res3} with respect to the number of PRM flip-flops (part 2 of 2). All PRRs have one row and include one BRAM column.

PRM flip-flops	T_{update_node}	$T_{prop_act_task}$	T_{lrprr_busy}	T_{lrprr_unlock}	T_{res3}
160	5.9	44.0	5.9	5.8	154.5
320	5.7	44.0	5.7	5.8	180.8
480	5.9	44.3	5.8	5.9	221.6
640	5.8	43.9	5.7	5.7	271.7
800	5.9	44.3	5.8	5.8	342.4
960	5.9	44.3	5.8	5.8	414.8
1120	5.8	44.1	5.8	5.7	504.7
1280	5.8	44.2	5.8	5.8	606.9
1440	5.8	44.1	5.8	5.8	720.9
1600	5.9	44.1	5.8	5.8	847.4
1760	5.8	44.1	5.8	5.8	986.1
1920	5.8	44.2	5.8	5.8	1,137.5

Table 5-14. DDRM execution times (ms) for T_{exe4} with respect to the number of PRM flip-flops (part 1 of 2). All PRRs have one row and include one BRAM column.

PRM flip-flops	$T_{chk_lprr_lock}$	$T_{chk_lprm_prr}$	$T_{chk_lrprr_lock}$	$T_{chk_lrprm_lrprr}$	T_{lrprr_lock}	$T_{cs_preempted_task}$
160	6.0	17.8	5.7	17.4	5.7	22.1
320	5.9	17.5	5.7	17.0	5.8	23.7
480	6.0	17.8	5.8	17.5	5.8	27.4
640	5.9	17.5	5.7	17.2	5.7	28.9
800	6.0	17.8	5.8	17.5	5.7	32.6
960	6.0	17.7	5.8	17.4	5.8	33.9
1120	5.9	17.7	5.7	17.3	5.7	37.9
1280	6.0	17.6	5.7	17.4	5.8	39.0
1440	6.0	17.7	5.7	17.3	5.8	43.2
1600	6.0	17.7	5.7	17.4	5.7	44.1
1760	6.0	17.6	5.7	17.3	5.8	48.4
1920	6.0	17.7	5.7	17.3	5.8	49.2

Table 5-15. DDRM execution times (ms) for T_{exe4} with respect to the number of PRM flip-flops (part 2 of 2). All PRRs have one row and include one BRAM column.

PRM flip-flops	$T_{prop_preempted_task}$	$T_{launch_pend_rtask}$	T_{update_node}	$T_{prop_act_task}$	T_{lrprr_unlock}	T_{exe4}
160	44.2	8.6	5.7	44.5	5.8	183.5
320	43.9	10.1	5.7	43.9	5.7	184.9
480	44.3	11.8	5.8	44.4	5.8	192.4
640	44.0	13.4	5.7	43.8	5.7	193.5
800	44.3	15.2	5.9	44.2	5.9	200.9
960	44.2	16.9	5.8	44.0	5.8	203.3
1120	44.2	18.5	5.8	44.0	5.8	208.6
1280	44.2	20.2	5.7	44.0	5.7	211.3
1440	44.1	21.9	5.8	44.4	5.8	217.7
1600	44.2	23.6	5.8	44.0	5.8	220.0
1760	44.1	25.2	5.8	44.2	5.8	225.9
1920	44.2	27.0	5.8	44.1	5.8	228.6

Table 5-16. DDRM execution times (ms) for $T_{exeres2}$ with respect to the number of PRM flip-flops (part 1 of 2). All PRRs have one row and include one BRAM column.

PRM flip-flops	$T_{chk_lprr_lock}$	$T_{chk_lprm_prr}$	$T_{chk_lrprr_lock}$	$T_{chk_lrprm_lrprr}$	$T_{chk_rrprr_lock}$	$T_{chk_rprm_rprr}$	T_{rrprr_lock}
160	6.1	17.9	5.7	17.3	16.4	32.8	7.3
320	6.0	17.6	5.7	17.4	16.4	32.8	7.4
480	6.1	17.9	5.8	17.4	16.4	32.8	7.4
640	6.0	17.6	5.7	17.4	16.4	32.7	7.3
800	6.0	17.8	5.7	17.3	16.4	32.8	7.3
960	6.0	17.7	5.8	17.4	16.4	32.9	7.3
1120	6.0	17.8	5.7	17.4	16.5	32.8	7.3
1280	6.0	17.8	5.7	17.3	16.4	32.8	7.3
1440	6.0	17.7	5.7	17.4	16.4	32.8	7.4
1600	6.1	17.7	5.7	17.4	16.4	32.8	7.3
1760	6.0	17.7	5.7	17.4	16.4	32.8	7.3
1920	6.0	17.7	5.8	17.4	16.4	32.8	7.3

Table 5-17. DDRM execution times (ms) for $T_{exeres2}$ with respect to the number of PRM flip-flops (part 2 of 2). All PRRs have one row and include one BRAM column.

PRM flip-flops	$T_{cs_preempted_rprm}$	$T_{prop_preempted_rtask}$	$T_{ftp_csbitstream}$	T_{remote_htr}	$T_{prop_rhtr_status}$	$T_{exeres2}$
160	31.6	29.1	416.2	75.2	27.5	683.1
320	33.4	28.7	412.7	102.6	27.3	708.0
480	36.5	28.7	408.8	141.6	27.5	746.9
640	38.0	27.4	407.7	193.5	27.3	797.0
800	41.9	28.8	414.4	261.5	27.5	877.4
960	43.5	27.5	414.0	337.3	27.6	953.4
1120	47.0	27.5	414.4	428.9	27.4	1,048.7
1280	48.3	30.3	406.2	532.7	27.4	1,148.2
1440	52.1	28.8	414.7	649.2	27.4	1,275.6
1600	53.3	28.6	411.7	778.5	27.4	1,402.9
1760	57.3	27.5	412.6	920.7	27.4	1,548.8
1920	58.3	28.6	411.9	1,075.7	27.4	1,705.3

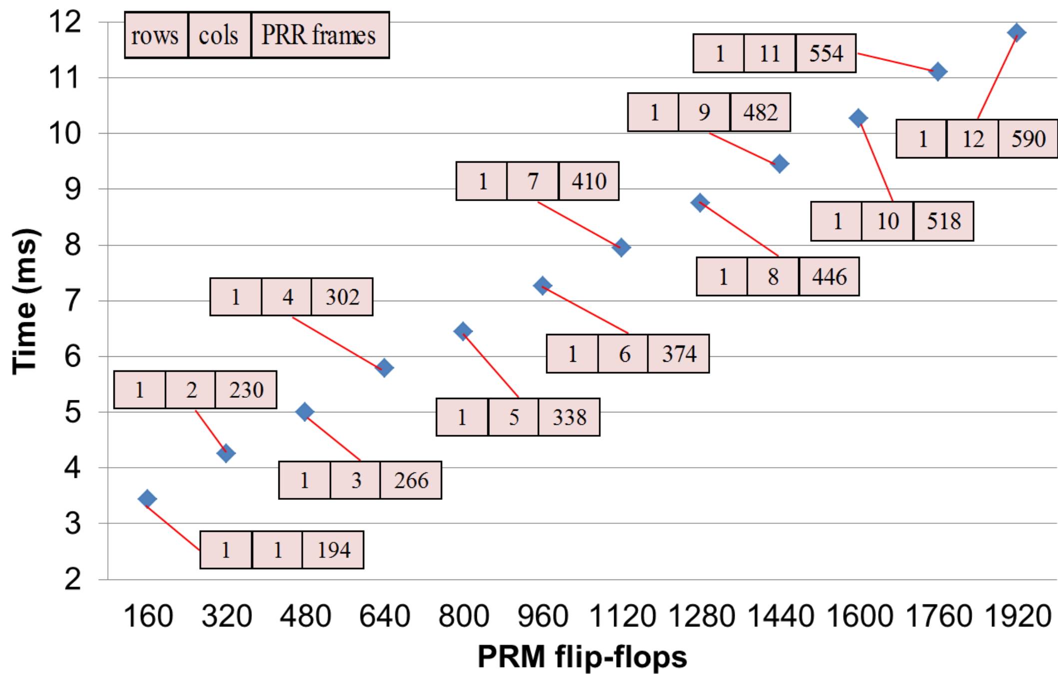

Figure 5-1. Execution times (ms) for $T_{reconfig_prr}$ with respect to the number of PRM flip-flops. The adjacent rectangles indicate the number of PRR rows, CLB columns, and PRR frames. All PRRs include one BRAM column.

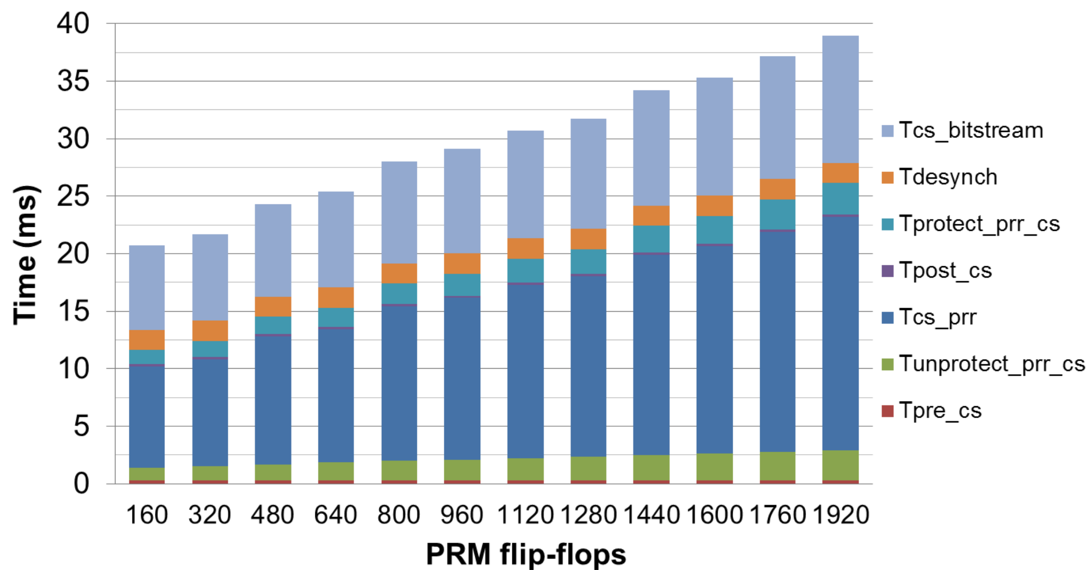

Figure 5-2. Execution times (ms) for CS (T_{cs}) in CSR with respect to the number of PRM flip-flops. All PRRs include one BRAM column.

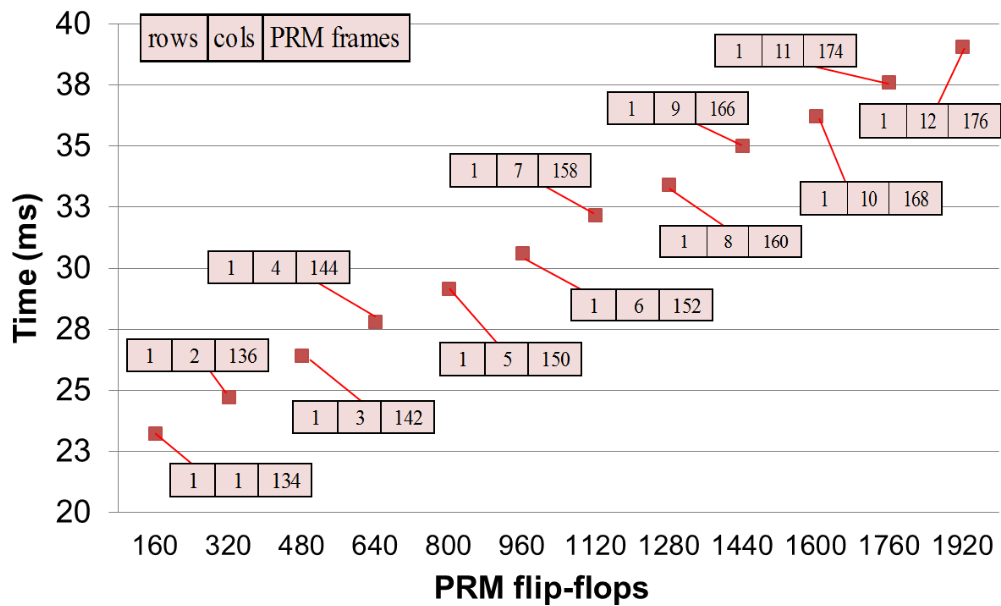

Figure 5-3. Execution times (ms) for the merge process (T_{merge}) in CSR with respect to the number of PRM flip-flops. The adjacent rectangles indicate the number of PRR rows, CLB columns, and PRM frames. All PRRs include one BRAM column.

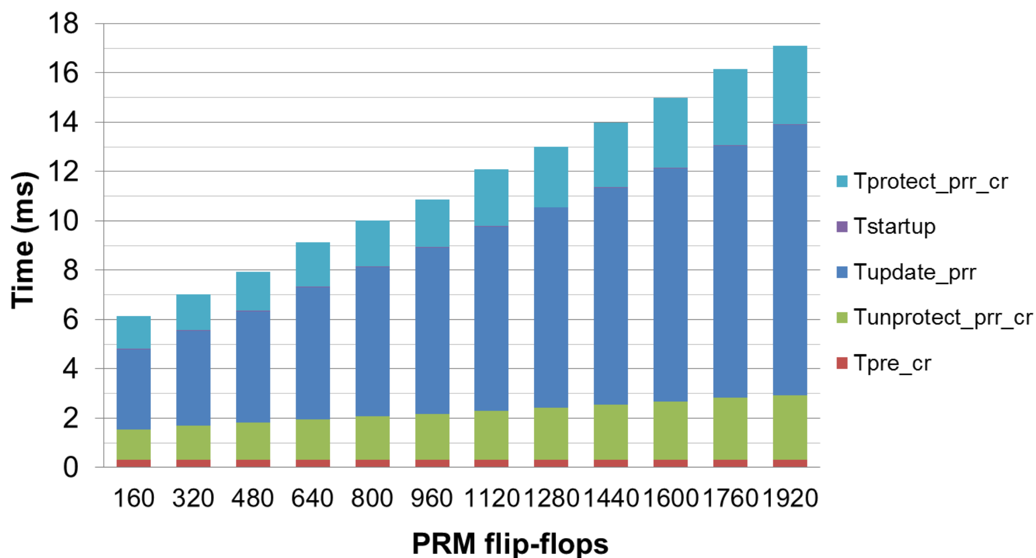

Figure 5-4. Execution times (ms) for CR (T_{cr}) in CSR with respect to the number of PRM flip-flops. All PRRs include one BRAM column.

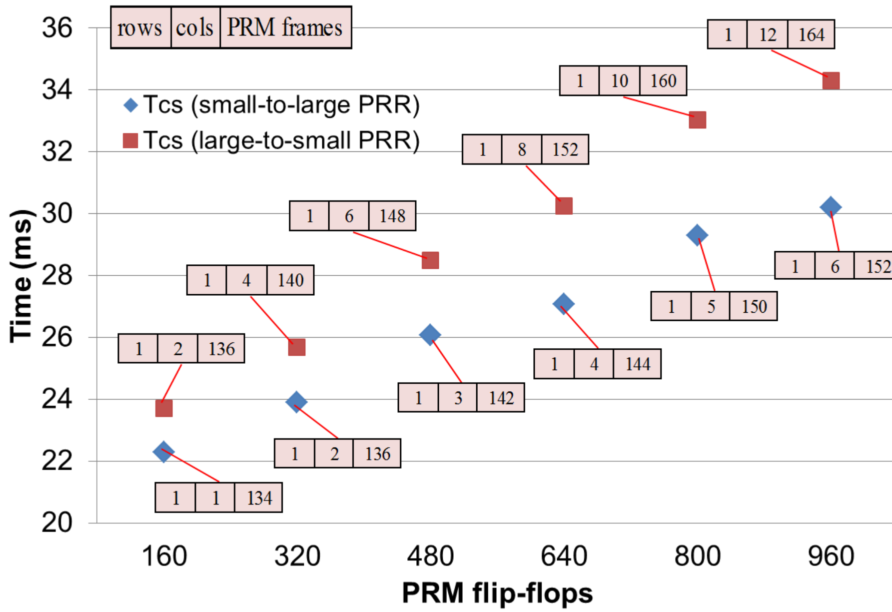

Figure 5-5. Execution times (ms) for CS (T_{cs}) in HTR with respect to the number of PRM flip-flops. The adjacent rectangles indicate the number of PRR rows, CLB columns, and PRM frames in the source PRR. All PRRs include one BRAM column.

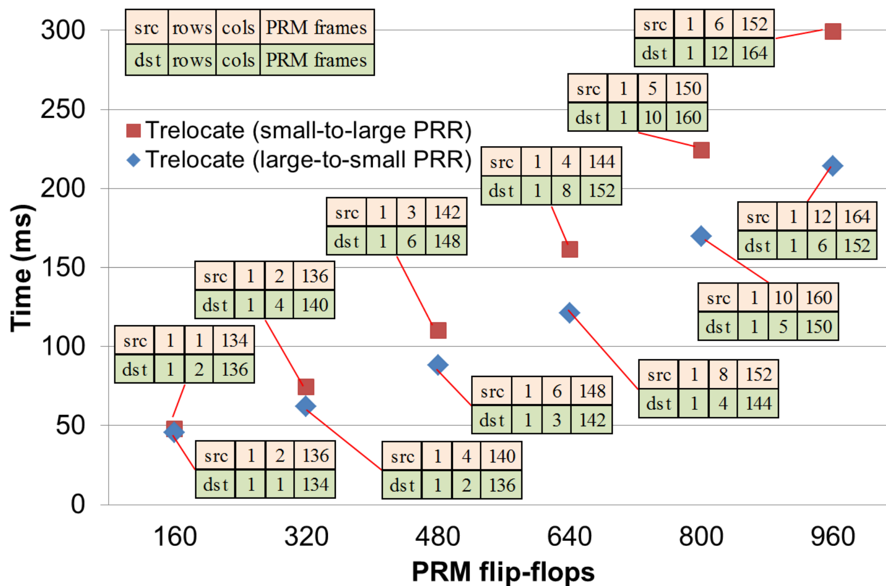

Figure 5-6. Execution times (ms) for context relocation ($T_{relocate}$) in HTR with respect to the number of PRM flip-flops. The adjacent rectangles indicate the number of PRR rows, CLB columns, and PRM frames of the source (src) and destination (dst) PRRs. All PRRs include one BRAM column.

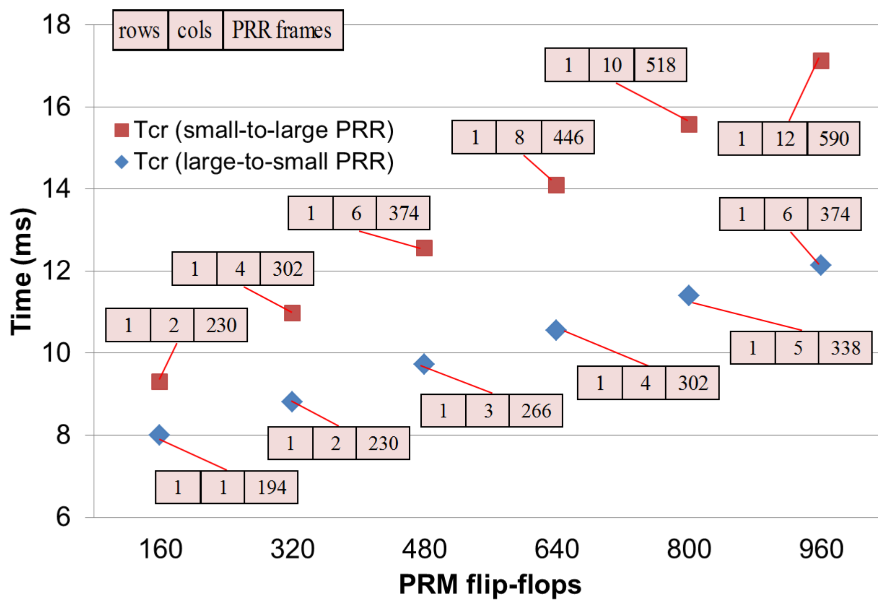

Figure 5-7. Execution times (ms) for CR (T_{cr}) in HTR with respect to the number of PRM flip-flops. The adjacent rectangles indicate the number of PRR rows, CLB columns, and PRR frames in the destination PRR. All PRRs include one BRAM column.

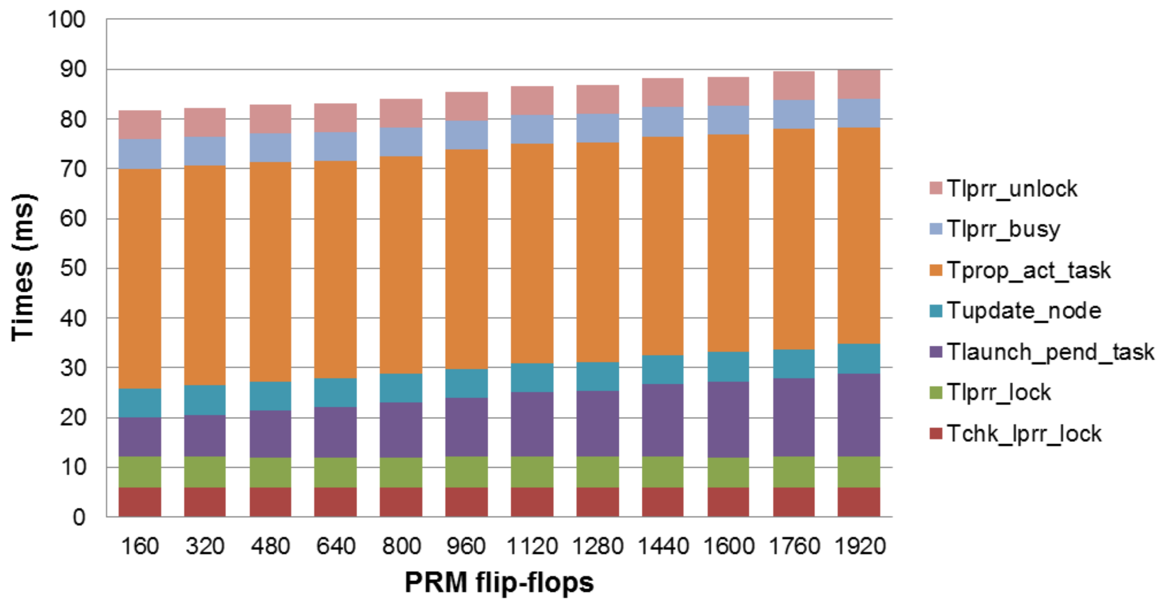

Figure 5-8. DDRM execution times (ms) for T_{exe1} with respect to the number of PRM flip-flops. All PRRs include one BRAM column.

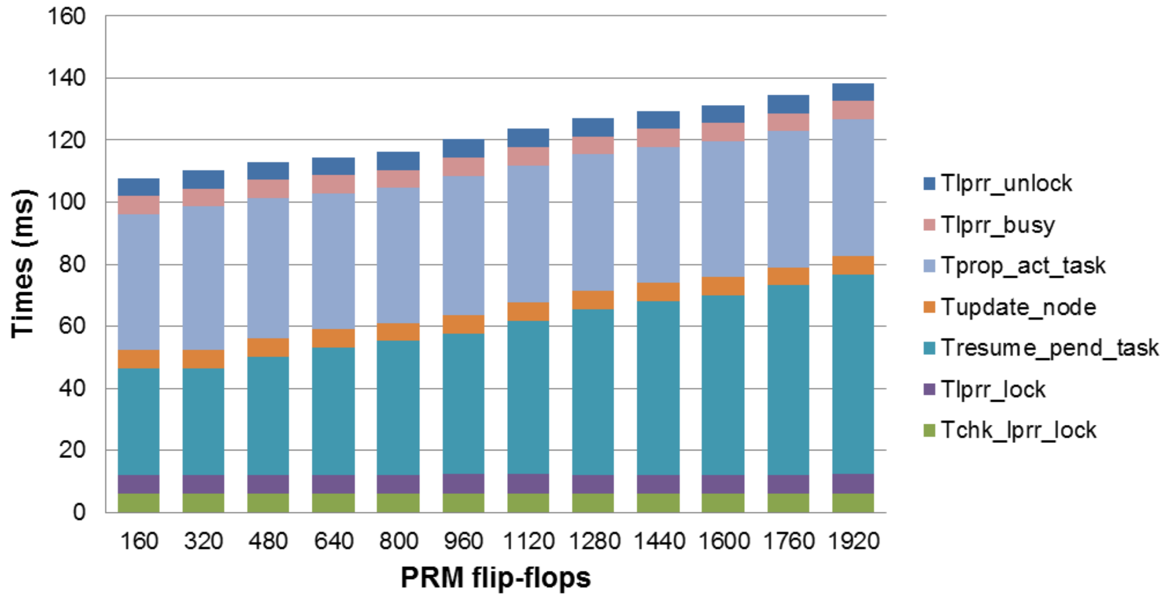

Figure 5-9. DDRM execution times (ms) for T_{res1} with respect to the number of PRM flip-flops. All PRRs include one BRAM column.

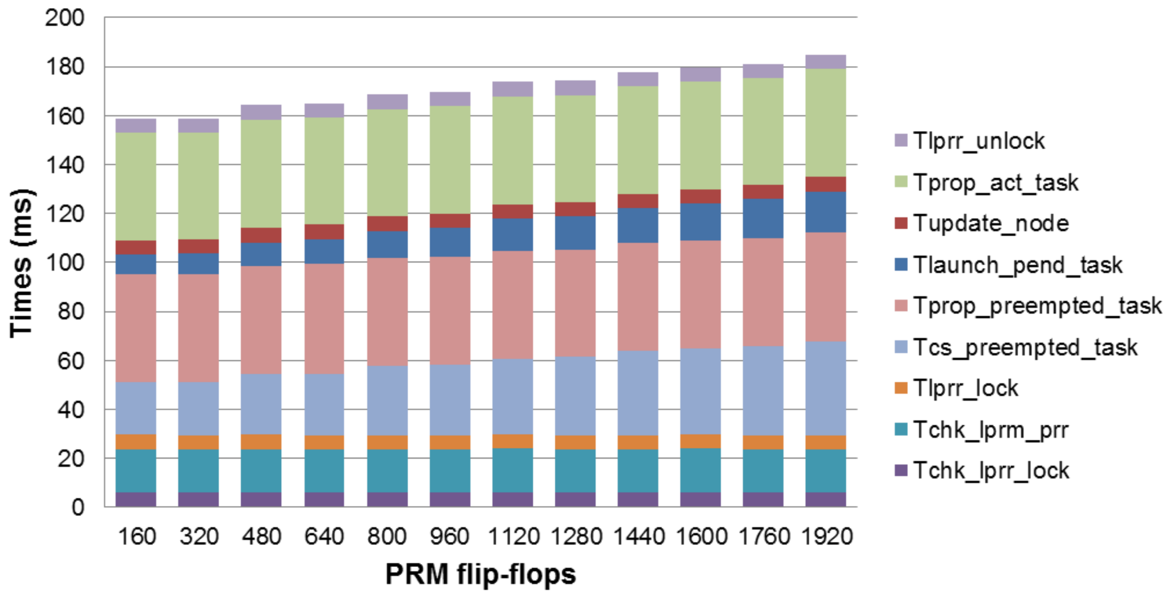

Figure 5-10. DDRM execution times (ms) for T_{exe2} with respect to the number of PRM flip-flops. All PRRs include one BRAM column.

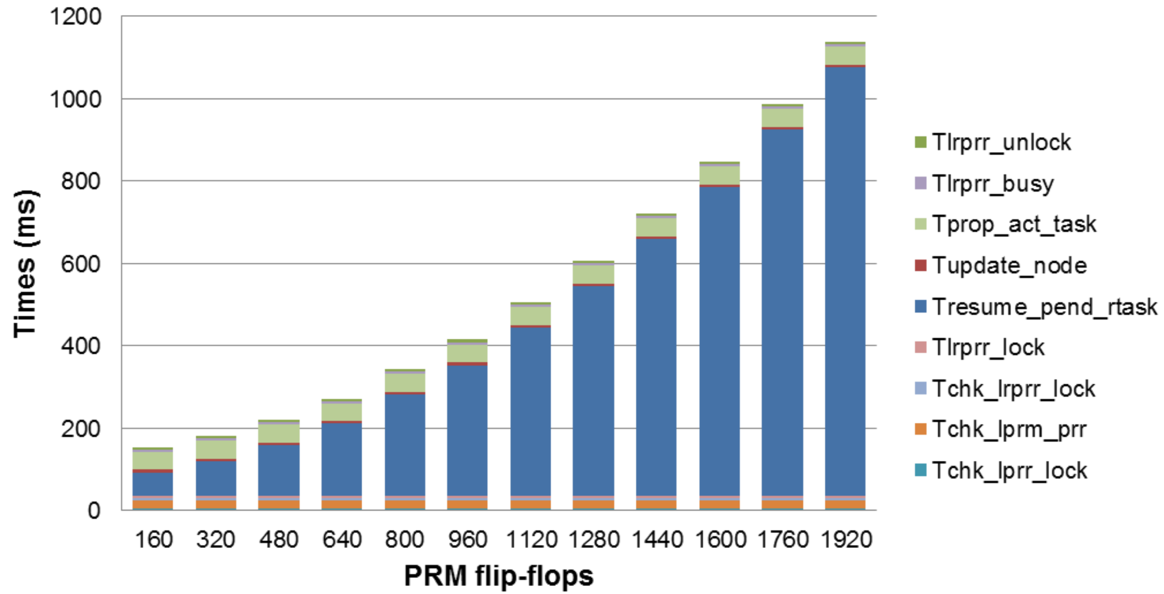

Figure 5-11. DDRM execution times (ms) for T_{res3} with respect to the number of PRM flip-flops. All PRRs include one BRAM column.

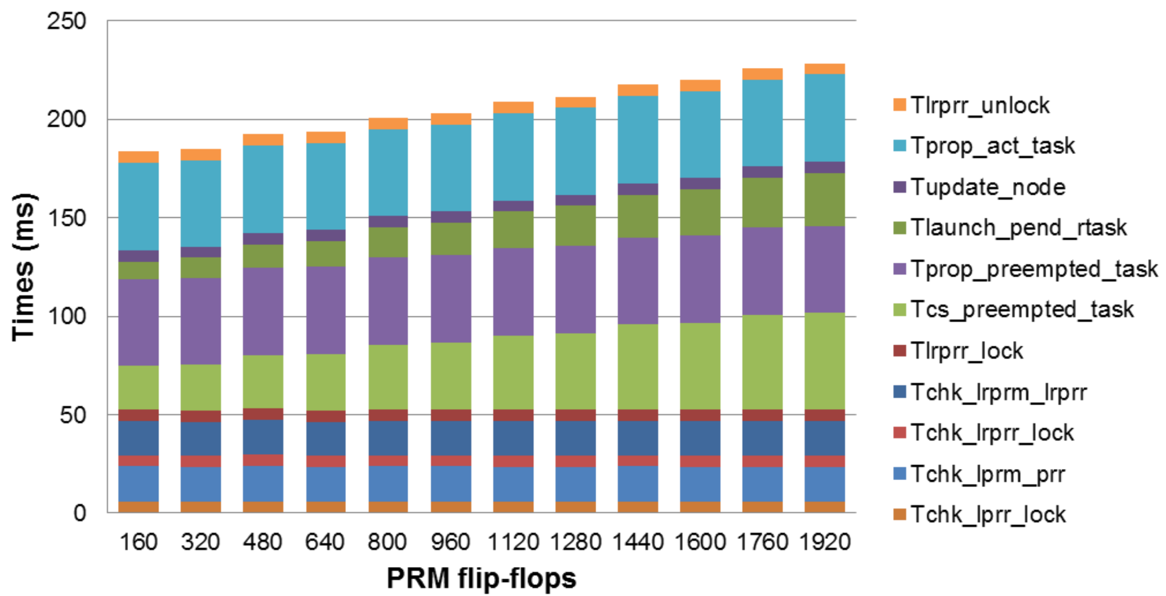

Figure 5-12. DDRM execution times (ms) for T_{exe4} with respect to the number of PRM flip-flops. All PRRs include one BRAM column.

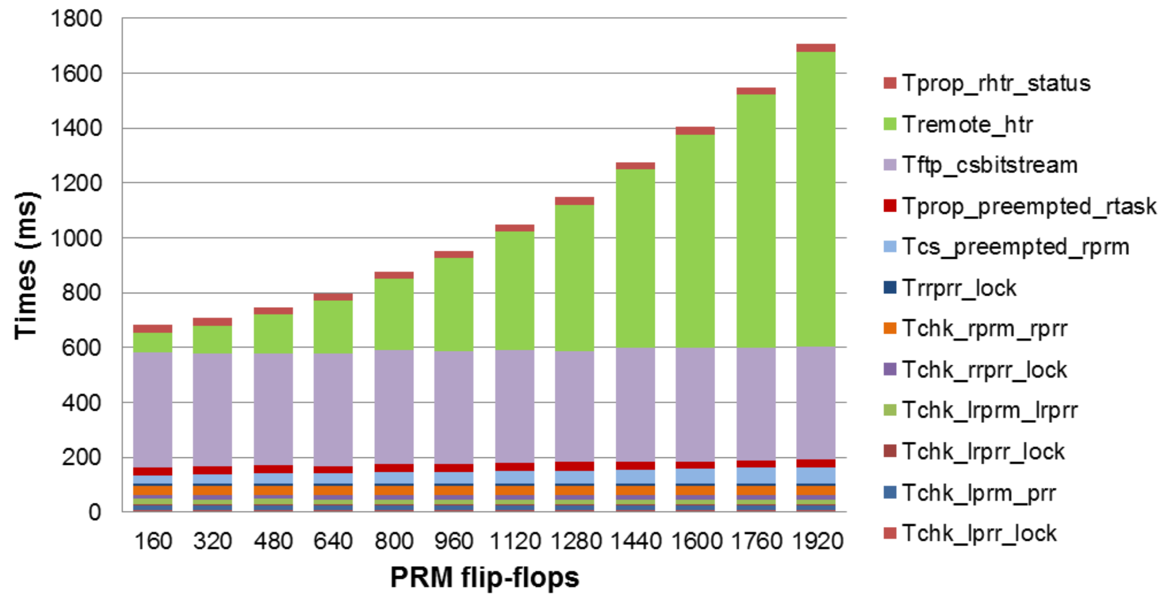

Figure 5-13. DDRM execution times (ms) for $T_{exeres2}$ with respect to the number of PRM flip-flops. All PRRs include one BRAM column.

CHAPTER 6
CONCLUSIONS

In this research we have developed on-chip software tools for hardware multitasking in partially reconfigurable (PR) field-programmable gate arrays (FPGAs). PR FPGAs are partitioned into one static region and multiple PR regions (PRRs), where PRRs are reconfigured with partial bitstreams which affords faster reconfiguration times. This partitioning enables hardware multitasking, where hardware tasks should be able to execute in PRRs, be preempted, replaced by higher priority hardware tasks, and later resume execution in the same or different PRR (i.e., candidate PRR) with sufficient resources in the same FPGA, or even in different physical FPGAs on an interconnected network, as if the tasks were not previously preempted. The ability of hardware tasks to be able to resume execution in the same or different FPGAs is addressed by the three research phases presented in this document.

In Phase 1, we leverage PR FPGAs for hardware multitasking on the same PRR, and introduce on-chip context save and restore (CSR) software for any heterogeneous PRR (i.e., a PRR with different resource types in the FPGA fabric). CSR is able to preempt and save the task's execution state (i.e., context), replace the preempted task with another task, and later resume the preempted task in the same PRR. The CSR software executes on a soft-core processor in the FPGA's static region without disrupting operations of the static region and other tasks executing in other PRRs in the FPGA.

In Phase 2, we extend our CSR software and introduce on-chip hardware task relocation (HTR) software to enable context relocation of hardware tasks between different-sized heterogeneous PRRs (i.e., PRRs with different locations in the FPGA

fabric, different shape, and different resource type distribution inside the PRRs). With our HTR, preempted hardware tasks should be able to resume operations in any heterogeneous candidate PRR with sufficient resources. As in CSR, The HTR software executes on a soft-core processor in the FPGA's static region without disrupting operations of the static region and other tasks executing in other PRRs in the FPGA.

The contributions of Phase 1 and Phase 2 are summarized as follows. Our CSR/HTR maximizes the use of PRRs, enabling the execution, preemption, and resumption of hardware tasks between heterogeneous PRRs without losing previous task's execution state, which eliminates seconds/minutes of re-execution time and may reduce the pending time of scheduled tasks to be resumed. Our CSR/HTR executes on autonomous FPGA systems, which does not incur off-chip communication overhead, does not introduce device overhead, does not impact the tasks' maximum operating frequencies, has no special constraints on the PRRs, is application independent, is portable across different systems (with minimum changes, Section 3.6), and does not require tool flow changes. The CSR/HTR fundamentals are applicable to newer Xilinx device families, such as the Virtex-6, the 7 series, and the Zynq-7000. Additionally, to the best of our knowledge, our HTR is the first solution that enables context relocation of hardware tasks between different-sized heterogeneous PRRs, which was not addressed in prior works.

In Phase 3, we extend our HTR software to work with multiple FPGAs in a network to enable a dynamic context relocation of hardware tasks between different FPGAs and introduce on-chip distributed dynamic resource management (DDRM)

software. Our DDRM executes on each autonomous FPGA system that is part of the DDRM network, which can be any local wired or wireless network.

The contributions of Phase 3 are summarized as follows. DDRM enables the execution, preemption, and resumption of hardware tasks across FPGAs in a network, without losing previous task's execution state, no matter where (which PRR of an FPGA in the network) the task last executed. DDRM provides additional task throughput improvements, reduces task idle time while waiting for execution, and improves shared resource usage per FPGA, by having more candidate PRRs per task, which enables application domains such as dynamic load balancing of hardware tasks across FPGAs, distributed fault tolerant systems, etc. To the best of our knowledge, no prior work proposes such a flexible solution for context relocation of hardware tasks in a distributed FPGA network.

Our results show the growth rate of CSR, HTR, and DDRM execution times as the PRR size increases, revealing that CSR/HTR/DDRM is most attractive for smaller PRRs, allowing the system designer to determine the application partitioning granularity (number of tasks/modules) and the task-to-PRR mappings (considering PRR sizes and CSR/HTR/DDRM times), according to the application requirements.

Our future work will focus on reducing CSR/HTR execution times, extending HTR and DDRM to support different FPGA architectures and incorporating HTR/DDRM with a run-time reconfiguration scheduler for hardware multitasking on multiple PR FPGAs interconnected in a network.

REFERENCES

[1] T. Becker, M. Koester, and W. Luk, "Automated Placement of Reconfigurable Regions for Relocatable Modules," *Proc. IEEE Int'l Symp. Circuits and Systems (ISCAS)*, pp. 3341-3344, 2010.

[2] T. Becker, W. Luk, and P. Y.K. Cheung, "Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration," *Proc. 15th Ann. IEEE Int'l Symp. Field-Programmable Custom Computing Machines (FCCM)*, pp. 35-44, 2007.

[3] C. Beckhoff, D. Koch, and J. Torresen, "Portable Module Relocation and Bitstream Compresion for Xilinx FPGAs," *Proc. 24th Int'l Conf. Field Programmable Logic and Applications (FPL)*, pp. 1-8, 2014.

[4] B. Blodget, P. James-Roxby, E. Keller, S. McMillan, and P. Sundararajan, "A Self-Reconfiguring Platform," *Proc. 13th Int'l Conf. Field Programmable Logic and Application (FPL)*, pp. 565-574, 2003.

[5] R. Bonamy, H. Pham, S. Pillement, and D. Chillet, "UPaRC - Ultra-Fast Power-aware Reconfiguration Controller," *Proc. Design, Automation and Test in Europe Conf. and Exhibition (DATE)*, pp. 1373-1378, 2012.

[6] BusyBox, http://www.busybox.net/, 2015.

[7] J. Carver, N. Pittman, and A. Forin, "Relocation and Automatic Floor-planning of FPGA Partial Configuration Bit-Streams," Technical Report MSR-TR-2008-111. Microsoft Research, Redmond, WA, 2008.

[8] S. Corbetta, F. Ferrandi, M. Morandi, M. Novati, M.D. Santambrogio, and D. Sciuto, "Two Novel Approaches to Online Partial Bitstream Relocation in a Dynamically Reconfigurable System," *Proc. IEEE Computer Society Annual Symp. on VLSI (ISVLSI)*, pp. 457-458, 2007.

[9] S. Corbetta, M. Morandi, M. Novati, M.D. Santambrogio, D. Sciuto, and P. Spoletini, "Internal and External Bitstream Relocation for Partial Dynamic Reconfiguration," *IEEE Trans. Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 11, pp. 1650-1654, Nov. 2009.

[10] A. Dasu and R. Kallam, "Accelerated Relocation Circuit," United States Patent 20110082994, Apr. 7, 2011.

[11] T. Drahonovsky, M. Rozkovec, and O. Novak, "A Highly Flexible Reconfigurable System on a Xilinx FPGA," *Proc. Int'l Conf. on Reconfigurable Computing and FPGAs (ReConFig)*, pp. 1-6, 2014.

[12] F. Duhem, F. Muller, and P. Lorenzini, "FaRM: Fast Reconfiguration Manager for Reducing Reconfiguration Time Overhead on FPGA," *Proc. 7th Int'l Symp. Applied Reconfigurable Computing (ARC)*, pp. 253-260, 2011.

[13] A. Flynn, A. Gordon-Ross, and A.D. George, "Bitstream Relocation with Local Clock Domains for Partially Reconfigurable FPGAs," *Proc. Design, Automation and Test in Europe Conf. and Exhibition (DATE)*, pp. 300-303, 2009.

[14] S. Garcia, B. Granado, and J.C. Prevotet, "Hardware Task Context Management for Fine Grained Dynamically Reconfigurable Architecture," *Proc. Workshop on Design and Architectures for Signal and Image Processing*, 2007.

[15] S.G. Hansen, D. Koch, and J. Torresen, "High Speed Partial Run-Time Reconfiguration Using Enhanced ICAP Hard Macro," *Proc. 25th IEEE Int'l Symp. Parallel and Distributed Processing Workshop and PhD Forum (IPDPSW)*, pp. 174-180, 2011.

[16] E.L. Horta and J.W. Lockwood, "PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs)," Technical Report WUCS-01-13. Washington University, Saint Louis, MO, 2001.

[17] X. Iturbe, K. Benkrid, T. Arslan, R. Torrego, and I. Martinez, "Methods and Mechanisms for Hardware Multitasking: Executing and Synchronizing Fully Relocatable Hardware Tasks in Xilinx FPGAs," *Proc. 21st Int'l Conf. Field Programmable Logic and Applications (FPL)*, pp. 295-300, 2011.

[18] X. Iturbe, K. Benkrid, A. Ebrahim, C. Hong, T. Arslan, and I. Martinez, "Snake: An Efficient Strategy for the Reuse of Circuitry and Partial Computation Results in High-Performance Reconfigurable Computing," *Proc. Int'l Conf. Reconfigurable Computing and FPGAs (ReConFig)*, pp. 182-189, 2011.

[19] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, and T. Arslan, "Microkernel Architecture and Hardware Abstraction Layer of a Reliable Reconfigurable Real-Time Operating System (R3TOS)," *ACM Trans. Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 1, pp. 5:1-5:35, Feb. 2015.

[20] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, I. Martinez, T. Arslan, and J. Perez, "R3TOS: A Novel Reliable Reconfigurable Real-Time Operating System for Highly Adaptive, Efficient, and Dependable Computing on FPGAs," *IEEE Trans. Computers*, vol. 62, no. 8, pp. 1542-1556, Aug. 2013.

[21] W. Jia and W. Zhou, *Distributed Network Systems: From Concepts to Implementations*. Boston, MA: Springer Science and Business Media, Inc., 2005.

[22] S. Jovanovic, C. Tanougast, and S. Weber, "A Hardware Preemptive Multitasking Mechanism Based on Scan-path Register Structure for FPGA-based Reconfigurable Systems," *Proc. 2nd NASA/ESA Conf. Adaptive Hardware and Systems (AHS)*, pp. 358-364, 2007.

[23] K. Jozwik, H. Tomiyama, M. Edahiro, S. Honda, and H. Takada, "Comparison of Preemption Schemes for Partially Reconfigurable FPGAs," *IEEE Embedded Systems Letters*, vol. 4, no. 2, pp. 45-48, Jun. 2012.

[24] K. Jozwik, H. Tomiyama, S. Honda, and H. Takada, "A Novel Mechanism for Effective Hardware Task Preemption in Dynamically Reconfigurable Systems," *Proc. 20th Int'l Conf. Field Programmable Logic and Applications (FPL)*, pp. 352-255, 2010.

[25] H. Kalte, G. Lee, M. Porrmann, and U. Rückert, "REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems," *Proc. 19th IEEE Int'l Symp. Parallel and Distributed Processing (IPDPS)*, 2005.

[26] H. Kalte and M. Porrmann, "Context Saving and Restoring for Multitasking in Reconfigurable Systems," *Proc. 15th Int'l Conf. Field Programmable Logic and Applications (FPL)*, pp. 223-228, 2005.

[27] H. Kalte and M. Porrmann, "REPLICA2Pro: Task Relocation by Bitstream Manipulation in Virtex-II/Pro FPGAs," *Proc. 3rd Conf. Computing Frontiers (CF)*, pp. 403-412, 2006.

[28] D. Koch, C. Haubelt, and J. Teich, "Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation," *Proc. ACM/SIGDA 15th Int'l Symp. Field Programmable Gate Arrays (FPGA)*, pp. 188-196, 2007.

[29] M. Koester, M. Porrmann, and H. Kalte, "Relocation and Defragmentation for Heterogeneous Reconfigurable Systems," *Proc. Int'l Conf. Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pp. 70-76, 2006.

[30] Y.E. Krasteva, A.B. Jimeno, E. de la Torre, and T. Riesgo, "Straight Method for Reallocation of Complex Cores by Dynamic Reconfiguration in Virtex II FPGAs," *Proc. 16th IEEE Int'l Workshop on Rapid System Prototyping (RSP)*, pp. 77-83, 2005.

[31] W.J. Landaker, M.J. Wirthlin, and B.L. Hutchings, "Multitasking Hardware on the SLAAC1-V Reconfigurable Computing System," *Proc. of 12th Int'l Conf. Field Programmable Logic and Applications (FPL)*, pp. 806-815, 2002.

[32] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-time Partial Reconfiguration Speed Investigation and Architectural Design Space Exploration," *Proc. 19th Int'l Conf. Field Programmable Logic an Applications (FPL)*, pp. 498-502, 2009.

[33] MathWorks, MATLAB Curve Fitting Toolbox User's Guide (Release R2015a), http://www.mathworks.com/help/releases/R2015a/pdf_doc/curvefit/curvefit.pdf, 2015.

[34] A. Morales-Villanueva and A. Gordon-Ross, "HTR: On-Chip Hardware Task Relocation for Partially Reconfigurable FPGAs," *Proc. 9th Int'l Symp. Applied Reconfigurable Computing (ARC)*, pp. 185-196, 2013.

[35] A. Morales-Villanueva and A. Gordon-Ross, "On-Chip Context Save and Restore of Hardware Tasks on Partially Reconfigurable FPGAs," *Proc. 21st Ann. IEEE Int'l Symp. Field-Programmable Custom Computing Machines (FCCM)*, pp. 61-64, 2013.

[36] M. Morandi, M. Novati, M.D. Santambrogio, and D. Sciuto, "Core Allocation and Relocation Management for a Self Dynamically Reconfigurable Architecture," *Proc. IEEE Computer Society Annual Symp. on VLSI (ISVLSI)*, pp. 286-291, 2008.

[37] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model," *ACM Trans. Reconfigurable Technology and Systems (TRETS)*, vol. 4, no. 4, pp. 36:1-36:24, Dec. 2011.

[38] K. Qureshi and H. Rashid, "A Performance Evaluation of RPC, Java RMI, MPI and PVM," *Malaysian J. Computer Science*, vol. 18, no. 2, pp. 38-44, Dec. 2005.

[39] M.D. Santambrogio, F. Cancare, R. Cattaneo, S. Bhandari, and D. Sciuto, "An Enhanced Relocation Manager to Speedup Core Allocation in FPGA-based Reconfigurable Systems," *Proc. 26th IEEE Int'l Symp. Parallel and Distributed Processing Workshop and PhD Forum (IPDPSW)*, pp. 336-343, 2012.

[40] H. Simmler, L. Levinson, and R. Männer, "Multitasking on FPGA Coprocessors," *Proc. 10th Int'l Conf. Field Programmable Logic and Applications (FPL)*, pp. 121-130, 2000.

[41] A. Sreeramareddy, R. Kallam, A.R. Dasu, and A. Akoglu, "Self-configurable Architecture for Reusable Systems with Accelerated Relocation Circuits (SCARS-ARC)," *Proc. 24th IEEE Int'l Symp. Parallel and Distributed Processing Workshop and PhD Forum (IPDPSW)*, pp. 1-4, 2010.

[42] A. Sudarsanam, R. Kallam, and A. Dasu, "PRR-PRR Dynamic Relocation," *IEEE Computer Architecture Letters*, vol. 8, no. 2, pp. 44-47, Dec. 2009.

[43] A.S. Tanenbaum and M. Van Steen, *Distributed Systems: Principles and Paradigms*, 2nd ed. Upper Saddle River, NJ: Pearson Education Inc., 2007.

[44] K. Vipin and S.A. Fahmy, "A High Speed Open Source Controller for FPGA Partial Reconfiguration," *Proc. Int'l Conf. Field-Programmable Technology (FPT)*, pp. 61-66, 2012.

[45] Xilinx, LogiCORE IP XPS HWICAP Product Specification (DS586 v5.00a), 2010.

[46] Xilinx, MicroBlaze Processor Reference Guide - Embedded Development Kit EDK 12.3 (UG081 v11.2), 2010.

[47] Xilinx, Partial Reconfiguration User Guide (UG702 v12.3), 2010.

[48] Xilinx, PlanAhead Software Tutorial - Overview of the Partial Reconfiguration Flow (UG743 v12.3), 2010.

[49] Xilinx, Virtex-5 Libraries Guide for HDL Designs (UG621 v12.4), 2010.

[50] Xilinx, Virtex-5 FPGA Configuration User Guide (UG191 v3.10), 2011.

[51] Xilinx, Virtex-5 FPGA User Guide (UG190 v5.4), 2012.

[52] Xilinx, Virtex-5 FPGA XtremeDSP Design Considerations User Guide (UG193 v3.5), 2012.

[53] Xilinx, Partial Reconfiguration User Guide (UG702 v14.3), 2012.

[54] Xilinx, Microblaze GNU Tools - Xilinx Open Source Wiki, http://xilinx.wikidot.com/mb-gnu-tools, 2015.

[55] Xilinx, MicroBlaze Linux (General) - Xilinx Open Source Wiki, http://xilinx.wikidot.com/microblaze-linux, 2015.

[56] Xilinx, Xilinx University Program XUPV5-LX110T Development System, http://www.xilinx.com/univ/xupv5-lx110t.htm, 2015.

BIOGRAPHICAL SKETCH

Aurelio Morales was born in Lima, Perú in 1961. He received his Bachelor of Science degree and the Master of Science degree in electronics engineering from the Universidad Nacional de Ingeniería (UNI) in Lima, in 1985, and 1991, respectively. He received a Fulbright Fellowship in 1992 and enrolled in a M.S. program at State University of New York, Buffalo, and earned the M.S. degree in electrical engineering in 1994. Back to Perú, he joined UNI as a part time associate professor while working at Telefónica del Perú.

Aurelio enrolled in the Ph.D. program in the Department of Electrical and Computer Engineering at the University of Florida in the fall of 2009, as a recipient of the Unidad Coordinadora del Programa de Ciencia y Tecnología (FINCyT) Fellowship, and went on a leave of absence at UNI. While pursuing his degree, Aurelio participated as a research volunteer in the NSF Center for High-Performance Reconfigurable Computing (CHREC).

He received his Ph.D. from the University of Florida in the summer of 2015 and went back to Perú, where he is currently a tenure-track full time professor of the Faculty of Electrical and Electronics Engineering at UNI. His current research interests include FPGA dynamic partial reconfiguration, computer architecture, reconfigurable computing, and embedded systems.