HTR: On-Chip Hardware Task Relocation for Partially Reconfigurable FPGAs

Aurelio Morales-Villanueva and Ann Gordon-Ross

NSF Center for High-Performance Reconfigurable Computing (CHREC)
Dept. of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611
{morales,ann}@chrec.org

Abstract. Partial reconfiguration (PR) enables shared FPGA systems to non-intrusively time multiplex hardware tasks in partially reconfigurable regions (PRRs). To fully exploit PR, higher priority tasks should preempt lower priority tasks and preempted tasks should resume execution in any PRR. This preemption/resumption requires saving/restoring the preempted task's execution context and relocating the task to another PRR, however, prior works only provide partial solutions and impose limitations and/or overheads. We propose on-chip hardware task relocation (HTR) software, which enables a task's execution state to be saved, relocated to, and restored in *any* PRR with sufficient resources. The HTR software executes on a soft-core processor in the FPGA's static region, and is thus portable across any system/application. Experimental results evaluate HTR execution times, enabling designers to tradeoff task/PRR granularity and HTR execution times based on application requirements.

1 Introduction

Partial reconfiguration (PR) of FPGAs improves a shared system's functionality and performance via enhanced, fine-grained device reconfigurability and hardware multiplexing. The FPGA's fabric is partitioned into one static region and multiple partially reconfigurable regions (PRRs). Hardware tasks can be *scheduled* to execute in any PRR with sufficient resources—any *candidate* PRR—and if the *scheduled* PRR is executing a lower priority task, task preemption/resumption enables the lower priority task's execution state—*context*—to be paused (i.e., context save (CS)) and resumed (i.e., context restore (CR)) in another PRR. CS reads the task's execution state and saves the context to a CS bitstream, and CR merges the CS bitstream with the task's initial partial bitstream (created at synthesis) using bitstream manipulations and reconfigures the scheduled PRR with this merged bitstream.

There exists little prior work on CS and CR—context save and restore (CSR), collectively—to the same PRR [8][9], which forces a preempted task to resume execution in *only* the task's originally scheduled PRR, rather than *any* candidate PRR. Hardware task relocation (HTR) enables preempted tasks to be relocated and resumed in any candidate PRR, which can improve system performance, task throughput, and maximizes device resource utilization for application domains such as target tracking,

P. Brisk, J.G. de Figueiredo Coutinho, P.C. Diniz (Eds.): ARC 2013, LNCS 7806, pp. 185–196, 2013.

dynamic load balancing, shared servers, etc. Since HTR is more challenging than CSR and must consider the task's physical relocation on the fabric, prior CSR work is not directly applicable. HTR is relatively easy between homogenous PRRs—PRRs with the same size, shape, and resources, but different fabric locations—requiring simple bitstream manipulations to specify the new fabric location [11][14]. HTR between heterogeneous PRRs—PRRs with different sizes, shapes, resources, and/or fabric locations—is more challenging, requiring complex bitstream manipulations to specify the new fabric location and relocate the task's functionality to this location's resources and/or resource layout. Similar to HTR, bitstream (core/module) relocation (BR) [1][2][3][4][5][7][16][18] enables a task to be relocated to any PRR, however, BR does not save/restore/resume the task's execution state, thus requiring the task to be restarted, which may incur seconds/minutes/hours of re-execution.

HTR can be implemented either off- or on-chip. In off-chip HTR, an attached CPU executes HTR software, which incurs significant overhead due to lengthy communication delays between the CPU and FPGA. Alternatively, on-chip HTR hardware can eliminate off-chip communication overhead, but introduces device resource overhead, lacks task/system portability, and reduces the tasks' maximum operating frequencies. To alleviate these overheads, we propose on-chip HTR software for heterogeneous PRRs that executes on a soft-core processor in the FPGA's static region, which, as compared to prior work, eliminates off-chip communication overhead and PRR overhead/constraints, is application/system independent, and does not alter the application/system design flow. Our HTR software uses the FPGA's internal configuration access port (ICAP) for reconfiguration. We detail HTR constructs and methodologies, which enables designers to incorporate HTR into their systems, and present implementation results for a Virtex-5 LX110T with a MicroBlaze (we note that the fundamentals of our HTR software is portable to newer Xilinx device families). Results show that HTR execution times are on the order of milliseconds, and vary based on the tasks'/PRRs' sizes. These analyses enable designers to tradeoff HTR execution times and task/PRR granularity based on application requirements.

2 Related Work

There exists little prior work in CSR, of which few leverage PR. Landaker et al. [15] and Simmler et al. [17] presented off-chip CSR software but since these works did not leverage PR, CSR reconfigured the entire FPGA. Joswik et al. [9] presented off-chip CSR software for PR FPGAs and reduced CSR times using direct memory access (DMA) for the ICAP, but this work only performed CSR to the same PRR. Kalte et al. [11] and Koester et al. [14] augmented the off-chip CSR software to include on-chip custom hardware for relocating tasks to different, homogeneous PRRs, however, both methods were for one-dimensional PR on older Xilinx devices, and are not applicable to newer Xilinx devices that support two-dimensional PR.

Koch et al. [13] and Jovanovic et al. [8] eliminated off-chip communication overhead with on-chip CSR hardware for both non-PR [13] and PR FPGAs [8], and reduced CSR times using different versions of scan-path chains of flip-flops (FFs), which is a technique used in design for testability (DFT) for very large scale integrated (VLSI) circuits. However, the CSR hardware incurred device overhead, lacked

portability, reduced the system's maximum operating frequency, and required changes in the design tool flow. On-chip CSR software would alleviate these drawbacks, but would not include task relocation.

BR enables task relocation, but prior works did not relocate the task's context. Horta et al. [7] and Blodget et al. [3] presented off-chip BR software and Kalte et al. [10][12] presented on-chip BR hardware for homogeneous PRRs, however, these methods still incurred the same drawbacks as off- and on-chip CSR, respectively.

Becker et al. [1][2] and Carver et al. [4] presented on-chip BR software for heterogeneous and homogeneous PRRs, respectively, however, these methods constrained the static region's logic routing from passing through the PRRs. Where as this constraint reduced the number of partial bitstreams to one per task, as opposed to one partial bitstream for each task-to-PRR mapping, the constraint introduced area and performance overheads [4][6]. Corbetta et al. [5], Sudarsanam et al. [18], and Santambrogio et al. [16] presented custom on-chip BR hardware for homogeneous PRRs, which was orchestrated using an on-chip soft-core processor.

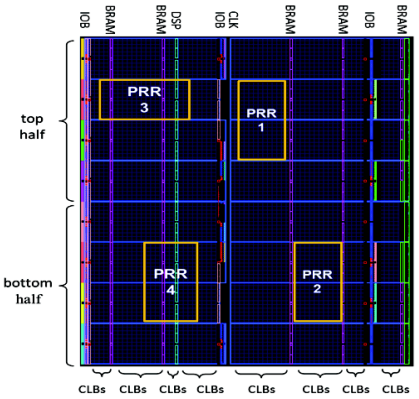

Fig. 1. Virtex-5 LX110T FPGA fabric layout

3 Virtex-5 FPGA Architecture

Since CSR and HTR are complex processes that require detailed device knowledge, we review the Xilinx Virtex-5 FPGA architecture (complete details are available in [19]), which will assist designers in incorporating HTR into their systems.

3.1 Device Architecture

Fig. 1 depicts the Virtex-5 LX110T fabric layout, the device used in our experiments, with four sample PRRs: PRR1 and PRR2 are homogeneous and PRR3 and PRR4 are heterogeneous. The device supports two-dimensional PR, which allows PRRs to occupy a rectangular fabric area. Device resources (CLBs, BRAMs, IOBs, DSPs, CLK)

are distributed in a row/column organization. The device is logically divided into two halves—top and bottom—and each half contains four rows and each row contains the same number of columns. Columns contain groups of frames and the number of frames per column depends on the type of resource in that column. A frame is the minimum unit of information used to write/read to/from the device, and a Virtex-5 frame contains 41 32-bit words.

3.2 Device Configuration

The Virtex-5 can be configured using external interfaces, such as JTAG (serial) or SelectMAP (parallel), or the internal ICAP interface (parallel). Full or partial bitstreams configure the entire device or 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 a particular column's resource configuration and the routing information to access the resources. CLB, BRAM, DSP, IOB, and CLK columns have 36, 30, 28, 54, and 4 configuration frames, respectively [19].

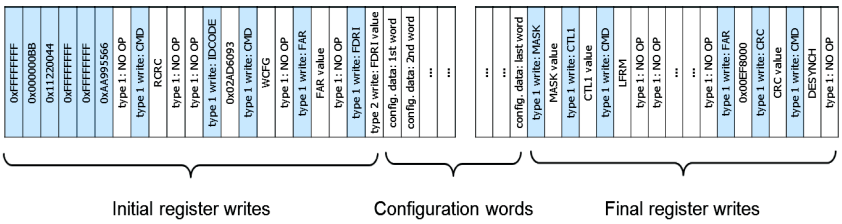

Fig. 2. Initial partial bitstream used in HTR for Virtex-5 FPGAs

Since HTR uses the ICAP, all partial bitstreams must be 32-bit word aligned. Fig. 2 depicts the initial partial bitstream structure used in HTR for the Virtex-5, which is the same as the bitstream generated by the Xilinx tools except that the initial comments (the name of the native circuit description file (*.ncd) from which the bitstream was generated and the bitstream creation date) are removed, resulting in a 32-bit word aligned file that can be used with the ICAP. The initial partial bitstream consists of a sequence of initial register writes, including the bus width words (0x000000BB and 0x11220044), the synchronization word (0xAA995566), RCRC, IDCODE (0x02AD6093), WCFG, FAR (specifies the first frame address of a PRR), and FDRI, followed by the configuration words (number of which is specified by the FDRI), and ending with the final register writes, which include MASK, CTL1, LFRM, CRC, and DESYNCH. [19] contains a complete description of these commands and special words. Note that the FAR included in the final register writes (0x00EF8000) is not associated with any PRR and is specific for the Virtex-5 LX110T.

For CS, the type 1 registers COR0 and CMD GCAPTURE are sent to the device via the ICAP to capture the FFs' values on a single edge transition of the main clock. After capturing the PRR's FFs' values, CMD RCAP is sent via the ICAP to enable future CSs [19]. CR requires initializing the PRR's FFs' values with the saved FFs'

values without interrupting the static region or the other PRRs' execution. In order to initialize a PRR with new FF values, the internal global set reset (GSR) signal in the Xilinx user primitive STARTUP_VIRTEX5 [19] must be toggled, which forces the startup sequence [19]. Since this toggle 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 [19]. Protecting the entire FPGA only needs to be done once, while unprotection/protection of the PRRs is required for each CR.

4 On-Chip Hardware Task Relocation (HTR) Software

Since the main contribution of our work is the HTR software, we assume that prior to execution, the applications have already been synthesized and partitioned into hardware tasks, the PRRs 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, including all task and candidate PRR combinations, and necessary files have been generated. We refer to a task executing in a PRR as a PR module (PRM). Even though a PRR may contain a mixture of resources, we detail HTR for PRMs that use CLBs only, however, our HTR is fundamentally applicable to heterogeneous PRRs that contain BRAMs, DSPs, and/or IOBs not in use by the PRM.

4.1 HTR Overview

We explain HTR using two heterogeneous PRRs and three PRMs: PRR1 is a candidate PRR for PRM1 and PRM2, and PRR2 is a candidate PRR for PRM2 and PRM3. Fig. 3 depicts the CSR and HTR flows (for resumption to the same or different PRR, respectively) assuming 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.

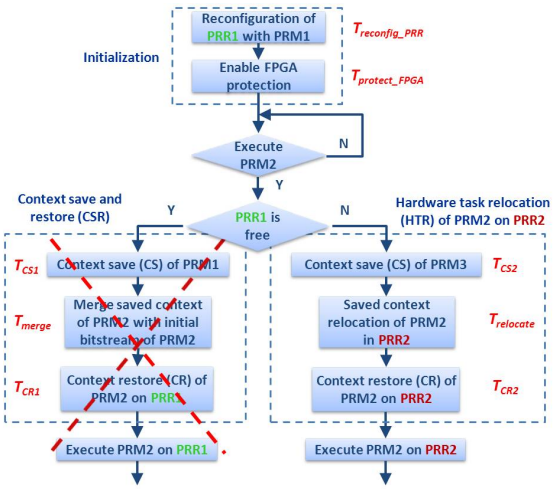

Fig. 3. On-chip context save and restore (CSR) and hardware task relocation (HTR) flows

Initialization reconfigures PRR1 with PRM1 and enables FPGA protection to prevent re-initialization of the static region's and PRRs' FFs and BRAMs (Section 3.2).

When PRM2 is ready to resume execution, PRM2 can either be resumed in PRR1 or relocated to PRR2. Since CSR is faster than HTR, PRM2 will first attempt to resume execution in PRR1. For example, if PRR1 is free or PRR1 is executing a lower priority task and can be preempted by PRM2 (i.e., PRM1 is lower priority than PRM2), CSR will resume PRM2 in PRR1 by: 1) CS of PRM1; 2) merging PRM2's saved context (CS bitstream) with PRM2's initial partial bitstream to create the merged 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), and PRR2 is available or executing a lower priority task (i.e., PRM3 is lower priority than PRM2), 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. Since CSR is not a contribution of this paper, the following subsections detail HTR only.

4.2 Context Save (CS)

Before reading a PRM's FFs' values, the PRR's clock is stopped to avoid potential setup/hold violations. Next, the capture process is initiated (T_{pre_CS}) and an HTR software loop captures/reads the PRM's FFs' values on a frame-by-frame basis (T_{CS_ICAP}), releases the ICAP (T_{post_CS}), and saves these values (i.e., the PRM's context) in the CS bitstream ($T_{CS\_bitstream}$). The CS bitstream size in 32-bit words is *1+N+N*41*, where *N* is the number of frames read and that contain the PRM's FFs' values and relative position inside the frame. 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 FFs' values, and the final *N*41* words are the contents of the *N* frames. Thus, the total execution time required for CS is: $T_{CS} = T_{pre\_CS} + T_{CS\_ICAP} + T_{post\_CS} + T_{CS\_bitstream}$.

Fig. 4. Bitstream manipulations for context relocation (HTR)

4.3 Saved Context Relocation—HTR

HTR's bitstream manipulations are similar to CSR's merge except that HTR must update the PRM's FFs' values in the scheduled PRR with the PRM's FFs' values from the CS bitstream. Fig. 4 depicts the HTR bitstream manipulations, which merge the CS and initial partial bitstreams at the 32-bit word level based on whether a single or multiple FF values need to be updated. Fig. 4 a) and b) show the update of a single or multiple FF values for HTR, respectively. All examples have been reduced to five bits for clarity. A single FF update for CSR's merge can be expressed as $f = cap * msk$

+ $ini*(/msk)$, where *cap* is the captured value, *ini* is the FF's 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. However, *f* cannot be used for HTR because HTR requires two *msk*'s: one for the saved context and the other for the initial partial bitstream in the scheduled PRR.

HTR's context relocation is expressed as $g = cap*ms + inid*(/md)$, where *cap* is the captured value, *ms* denotes if the bit is part of the saved context, *inid* is the FF's value in the scheduled PRR's initial partial bitstream, and *md* is the bit to be updated in the merged bitstream. *md* = 1 updates *inid* with *cap*, provided that *ms* = 1, and *md* = 0 retains *inid*'s value. In Fig. 4 a) and b), *bitms* and *bitmd* denote the bit position of *ms* and *md* and the expression (*bitms* - *bitmd*) denotes the bit-distance between these bits' positions in a 32-bit word. If (*bitms* - *bitmd* $\geq$ 0), $cap*ms$ is right-shifted (*bitms* - *bitmd*) bit positions using $shr(cap*ms)$, else *cap*ms* is left-shifted (*bitmd* - *bitms*) bit positions using $shl(cap*ms)$. Updating multiple FFs in a word boundary in the merged bitstream (Fig. 4 b)) is done sequentially, and each update does not have the same *cap* and *ms* words as shown. An HTR software loop executes this merge and relocation process and saves the merged bitstream to a file with a total execution time denoted by $T_{relocate}$.

4.4 Context Restore (CR)

Before CR, the scheduled PRR must be unprotected ($T_{unprotect\_PRR}$) to allow the PRR's FFs to be initialized with the new values in the merged bitstream, but the rest of the FPGA must remain protected. Next, the scheduled PRR is reconfigured (T_{update_PRR}) by interleaving the initial register writes (Fig. 2), the merged bitstream (Section 4.3), and the final register writes (Fig. 2). After the scheduled PRR is reconfigured with the PRM's relocated context ($T_{startup}$), the scheduled PRR is protected ($T_{protect\_PRR}$) to prevent future startup sequence phases for another PRR from re-initializing the scheduled PRR's FFs' values. Thus, the total execution time required for CR is: $T_{CR} = T_{unprotect\_PRR} + T_{update\_PRR} + T_{startup} + T_{protect\_PRR}$.

5 Experimental Results

5.1 Experimental Setup

We used the Xilinx XUPV5-LX110T board and the Xilinx ISE 12.4, XPS 12.4, and PlanAhead 12.4 tools. We partitioned the fabric into two heterogeneous PRRs and the static region executed a 100 MHz MicroBlaze soft-core processor running a Linux-like OS 2.6.37 based on BusyBox. We generated the executable binaries for the MicroBlaze using the GNU tools. A XPS HWICAP interfaced the MicroBlaze and the ICAP, the SDRAM provided external storage for the bitstreams, binaries, and the HTR files. The XPS timer was used to measure the T_x execution times and we averaged the execution times over five 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 (e.g., instruction and data cache parameters), the XPS HWICAP's configuration, and the memory controller used to

access the SDRAM files introduce overheads that affect the results, however, these components' configurations do not impact HTR's functionality, and in our analysis we note the impacts of different component configurations and hardware overheads on the results' trends.

We verified HTR's correct operation using two interfaces per PRR: one connected to the MicroBlaze and one in the PRM for transferring the PRM's FFs' values to the MicroBlaze. For testing purposes, PRM1, PRM2, and PRM3 implemented a 32-bit up counter, down counter, and pipelined adder/accumulator, respectively. We tested HTR using the flow in Fig. 3, verifying that the first value of each register in PRM2 after CR on a different PRR (i.e., the task was relocated) corresponded to the last value of each register in PRM2 prior to CS.

In order to generate thorough results for various 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 and analyses. We created a project with two small heterogeneous PRRs containing CLBs and selected two empty areas (areas with no CLBs and routing resources in use) on the fabric. In these empty areas, we created *pseudo-PRRs*, *pseudo* initial partial bitstreams, and *pseudo* logic location files (*.ll) for *pseudo-PRM*s.

Our experiments evaluate small-to-large and 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. The pseudo-PRRs' sizes contained one row and multiple columns ranging from one to twelve, which is the largest number of contiguous CLB columns on the Virtex-5 LX110T. 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 had 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 FFs across the CLB columns, which simulated the effects of the Xilinx tool's FF distribution done during placement and provided realistic execution times.

5.2 Execution Times

Table 1 through Table 3 show the execution times in milliseconds for the significant HTR steps. The tables contain two ranges of number of PRM FFs: a fine-grained range spanning 20 to 160 FFs in a single CLB column in 20 FF increments, and a coarse-grained range increasing the number of CLB columns, resulting in 160 FF increments. Fig. 5 plots the coarse-grained range tables' results, where each point is identified by a box with the number of rows, columns, and PRR/PRM frames depending on the graph's reported execution time.

Table 1 and Fig. 5 a) summarize $T_{reconfig\_PRR}$, which depends on the number of PRR frames (36 per CLB column). In the fine-grained range, $T_{reconfig\_PRR}$ is constant (there is only one CLB column). In the coarse-grained range, $T_{reconfig\_PRR}$ shows a linear behavior up to 960 PRM FFs. We discuss the trend above 960 FFs later in this section.

The execution time for $T_{protect\_FPGA}$ is constant and depends on the number of rows and columns in the device, which is 67.72 ms for the test device.

Table 1. Execution times (ms) for $T_{reconfig\_PRR}$

rows	1														
columns	1								2	3	4	5	6	8	12
PRR frames	36								72	108	144	180	216	288	432
PRM flip-flops	20	40	60	80	100	120	140	160	320	480	640	800	960	1280	1920
$T_{reconfig\_PRR}$	0.974	0.978	0.983	0.980	0.978	0.983	0.987	0.979	1.518	2.047	2.671	3.150	3.706	4.841	7.412

Table 2. Execution times (ms) for CS (T_{CS}), context relocation ($T_{relocate}$), and CR (T_{CR}) for small-to-large HTR

src PRR	rows	1												
	columns	1								2	3	4	5	6
	PRM frames	1							2	4	6	8	10	12
dst PRR	rows	1												
	columns	2								4	6	8	10	12
	PRM frames	2				4				8	12	16	20	24
PRM flip-flops		20	40	60	80	100	120	140	160	320	480	640	800	960
T_{CS}		4.79	4.79	4.79	4.79	4.79	4.79	4.79	5.17	5.85	6.50	7.20	7.83	8.48
$T_{relocate}$		11.53	14.96	18.46	21.38	24.99	28.89	31.92	36.02	76.79	144.81	213.34	297.81	393.81
T_{CR}		6.25	6.25	6.23	6.24	6.23	6.23	6.25	6.24	8.01	9.60	11.18	12.83	14.74

Table 3. Execution times (ms) for CS (T_{CS}), context relocation ($T_{relocate}$), and CR (T_{CR}) for large-to-small HTR

src PRR	rows	1												
	columns	2								4	6	8	10	12
	PRM frames	2				4				8	12	16	20	24
dst PRR	rows	1												
	columns	1								2	3	4	5	6
	PRM frames	1							2	4	6	8	10	12
PRM flip-flops		20	40	60	80	100	120	140	160	320	480	640	800	960
T_{CS}		5.21	5.21	5.22	5.22	5.94	5.95	5.94	5.94	7.24	8.60	10.39	11.64	13.05
$T_{relocate}$		10.90	14.43	17.86	20.62	24.38	27.73	31.42	35.79	71.77	120.18	176.31	239.15	309.52
T_{CR}		5.47	5.47	5.47	5.48	5.49	5.48	5.48	5.46	6.31	7.11	8.07	8.79	9.68

Table 2 and Table 3 summarize the execution times for T_{CS}, $T_{relocate}$, and T_{CR} for small-to-large and large-to-small HTR, respectively, and Fig. 5 b), c), and d) plot these execution times. For brevity, we omit the detailed breakdown of T_{CS} and T_{CR}, which depends on the number of PRM frames that contain used FFs in the source pseudo-PRR and the number of PRR frames in the destination PRR, respectively.

Capturing and saving the context in a small PRR shows a nearly linear increase in T_{CS}. T_{pre_CS} and T_{post_CS} for both small-to-large and large-to-small HTR are 0.54 and 1.39 ms, respectively. For small-to-large HTR, T_{CS_ICAP} ranges from 0.85 to 3.53 ms and $T_{CS\_bitstream}$ ranges from 2.01 to 3.02 ms, and for large-to-small HTR, these values range from 1.15 to 6.91 ms and from 2.13 to 4.21 ms, respectively. $T_{relocate}$ depends on the number of PRM FFs used in the PRR. The CS and merged bistreams are randomly accessed, resulting in high data cache miss rates and overheads for accessing SDRAM, which explains $T_{relocate}$'s non-linear behavior above 160 PRM FFs. Finally, T_{CR} depends on the interleaved creation of the new initial partial bitstream (Fig. 2 and Section 4.4) and sequential reconfiguration, thus T_{CR} is larger than $T_{reconfig\_PRR}$ for the same number of PRM FFs and PRR frames. $T_{startup}$ is fixed and is 0.70 ms. For small-to-large HTR, $T_{unprotect\_PRR}$, T_{update_PRR}, and $T_{protect\_PRR}$ ranges from 2.01 to 3.36 ms, 2.07 to 7.87 ms, and 1.47 to 2.81 ms, respectively, and for large-to-small HTR, these values range from 1.87 to 2.63 ms, 1.54 to 4.34 ms, and 1.36 to 2.01 ms, respectively. These results also reveal that large-to-small HTR is faster than small-to-large HTR

(i.e., $T_{relocate}$ and T_{CR} are faster). Even though T_{CS} is slower for large-to-small HTR as compared to small-to-large HTR, $T_{relocate}$ is slower than T_{CS} and T_{CR}.

The resources required by the static region, including the MicroBlaze, XPS HWICAP and GPIOs, and SDRAM controller are 12,898, 44, and 4 FFs, 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.

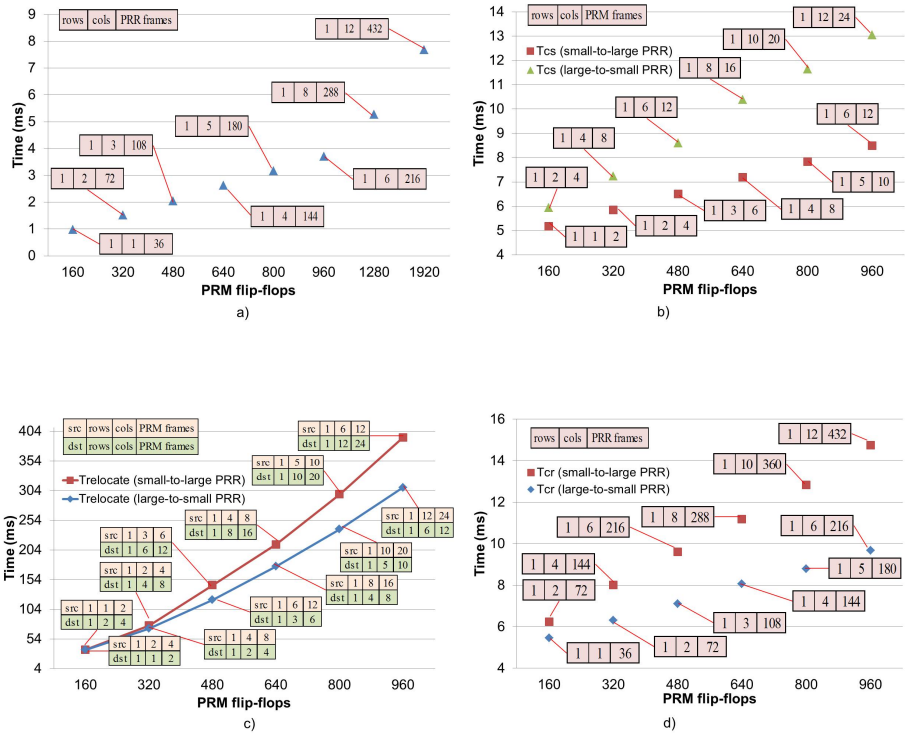

Fig. 5. Execution times (ms) for a) $T_{reconfig\_PRR}$ b) CS (T_{CS}), c) context relocation ($T_{relocate}$), and d) CR (T_{CR}) with respect to the number of PRM FFs. The adjacent rectangles indicate the number of rows, columns, and PRR/PRM frames, respectively.

Increasing the PRR's number of rows and reducing the number of columns while maintaining the same number of PRM FFs would reveal similar results as shown in the tables and figures. However, for PRRs using more than 960 PRM FFs, high data cache miss rates, SDRAM overheads when accessing the bitstreams, and the XPS HWICAP's configuration introduce a non-linear increase in the growth rate of these execution times. All HTR times may be improved by adding a custom DMA and enlarging the internal storage to the XPS HWICAP, saving the CS and initial partial bitstreams in BRAMs, or increasing/modifying the data cache size/configuration. However, BRAMs are limited and these options incur hardware overhead that may

affect the system's performance, and some of these modifications would not be portable to other systems. 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.

6 Conclusions and Future Work

In this paper, we introduced the first, to the best of our knowledge, on-chip hardware task relocation (HTR) software for two-dimensional relocation between heterogeneous PRRs, which has no off-chip communication overhead, imposes no design/system constraints, is application/system independent, and does not require changes to the design tool flow. Our HTR maximizes shared resource utilization, performance, and throughput via task preemption and resumption between heterogeneous PRRs, which preserves the task's execution state and eliminates seconds/minutes/hours/days of re-execution time. Experimental results analyze HTR execution time, which enables system designers to guide application task granularity and partitioning decisions based on application requirements. Our future work will extend HTR's functionalities to include DSP, BRAM, and IOB resources.

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

References

1. Becker, T., Koester, M., Luk, W.: Automated placement of reconfigurable regions for relocatable modules. In: Proc. of the 2010 IEEE Intl. Symp. on Circuits and Systems (ISCAS 2010), Paris, France, pp. 3341–3344 (2010)
2. Becker, T., Luk, W., Cheung, P.Y.K.: Enhancing Relocatability of Partial Bitstreams for Run-Time Reconfiguration. In: 15th Annual IEEE Symp. on Field-Programmable Custom Machines (FCCM 2007), Napa, California, pp. 35–44 (2007)
3. Blodget, B., James-Roxby, P., Keller, E., McMillan, S., Sundararajan, P.: A Self-Reconfiguring Platform. In: Cheung, P.Y.K., Constantinides, G.A. (eds.) FPL 2003. LNCS, vol. 2778, pp. 565–574. Springer, Heidelberg (2003)
4. Carver, J., Pittman, N., Forin, A.: Relocation and Automatic Floor-planning of FPGA Partial Configuration Bitstreams. Microsoft Research, WA. Technical Report MSR-TR-2008-111, Redmond, Washington (2008)
5. Corbetta, S., Morandi, M., Novati, M., Santambrogio, M.D., Sciuto, D., Spoletini, P.: Internal and External Bitstream Relocation for Partial Dynamic Reconfiguration. IEEE Trans. on Very Large Scale Integration (VLSI) Systems 17(11), 1650–1654 (2009)

6. Flynn, A., Gordon-Ross, A., George, A.D.: Bitstream Relocation with Local Clock Domains for Partially Reconfigurable FPGAs. In: Design, Automation & Test in Europe Conference & Exhibition 2009 (DATE 2009), Nice, France, pp. 300–303 (2009)
7. Horta, E.L., Lockwood, J.W.: PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). Technical Report WUCS-01-13, Washington University, St. Louis, Missouri (2001)
8. Jovanovic, S., Tanougast, C., Weber, S.: A Hardware Preemptive Multitasking Mechanism Based on Scan-path Register Structure for FPGA-based Reconfigurable Systems. In: 2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS 2007), Edinburgh, United Kingdom, pp. 358–364 (2007)
9. Jozwik, K., Tomiyama, H., Honda, S., Takada, H.: A Novel Mechanism for Effective Hardware Task Preemption in Dynamically Reconfigurable Systems. In: Intl. Conf. on Field Programmable Logic and Applications (FPL 2010), Milano, Italy, pp. 352–255 (2010)
10. Kalte, H., Lee, G., Porrmann, M., Rückert, U.: REPLICA: A Bitstream Manipulation Filter for Module Relocation in Partial Reconfigurable Systems. In: Proc. of the 19th IEEE Intl. Parallel and Distributed Processing Symp. (IPDPS 2005), Denver, Colorado (2005)
11. Kalte, H., Porrmann, M.: Context Saving and Restoring for Multitasking in Reconfigurable Systems. In: Intl. Conf. on Field Programmable Logic and Applications (FPL 2005), Tampere, Finland, pp. 223–228 (2005)
12. Kalte, H., Porrmann, M.: REPLICA2Pro: Task Relocation by Bitstream Manipulation in Virtex-II/Pro FPGAs. In: Proc. of the 3rd Conf. on Computing Frontiers (CF 2006), pp. 403–412. ACM, New York (2006)
13. Koch, D., Haubelt, C., Teich, J.: Efficient Hardware Checkpointing: Concepts, Overhead Analysis, and Implementation. In: Proc. of the ACM/SIGDA 15th Intl. Symp. on Field Programmable Gate Arrays (FPGA 2007), pp. 188–196. ACM, New York (2007)
14. Koester, M., Porrmann, M., Kalte, H.: Relocation and Defragmentation for Heterogeneous Reconfigurable Systems. In: Proc. of Intl. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA 2006), pp. 70–76. Las Vegas, Nevada (2006)
15. Landaker, W.J., Wirthlin, M.J., Hutchings, B.L.: Multitasking Hardware on the SLAAC1-V Reconfigurable Computing System. In: Glesner, M., Zipf, P., Renovell, M. (eds.) FPL 2002. LNCS, vol. 2438, pp. 806–815. Springer, Heidelberg (2002)
16. Santambrogio, M.D., Cancare, F., Cattaneo, R., Bhandari, S., Sciuto, D.: An Enhanced Relocation Manager to Speedup Core Allocation in FPGA-based Reconfigurable Systems. In: 2012 IEEE 26th International Symposium on Parallel & Distributed Processing, Workshop and PhD Forum (IPDPSW 2012), Shangai, China, pp. 336–343 (2012)
17. Simmler, H., Levinson, L., Männer, R.: Multitasking on FPGA Coprocessors. In: Grünbacher, H., Hartenstein, R.W. (eds.) FPL 2000. LNCS, vol. 1896, pp. 121–130. Springer, Heidelberg (2000)
18. Sudarsanam, A., Kallam, R., Dasu, A.: PRR-PRR Dynamic Relocation. IEEE Computer Architecture Letters 8(2), 44–47 (2009)
19. Xilinx, Inc.: Virtex-5 FPGA Configuration User Guide v3.10 (UG191) (November 18, 2011)