Preemptive Hardware Multitasking in ReconOS

Markus Happe(✉), Andreas Traber, and Ariane Keller

Communication Systems Group, ETH Zurich, Zürich, Switzerland
{markus.happe,ariane.keller}@tik.ee.ethz.ch, atraber@student.ethz.ch

Abstract. Preemptive hardware multitasking is not supported in most reconfigurable systems-on-chip (rSoCs), which severely limits the scope of hardware scheduling techniques on these platforms. While modern field-programmable gate arrays (FPGAs) support dynamic partial reconfiguration of any region at any time, most hardware tasks cannot be preempted at arbitrary points in time, because context saving and restoring is not supported out of the box by the vendors. Although hardware task preemption techniques have been proposed in the past, they cannot be found in today's rSoCs. In this paper we therefore propose a novel methodology for preemptive hardware multitasking that does not require any changes at the task level and show that our approach can be seamlessly integrated to an established execution environment for rSoCs, called ReconOS. Our experimental results show that we can successfully capture and restore the states of all flip-flops and block RAMs in a reconfigurable region on a Xilinx Virtex-6 FPGA at arbitrary points in time. Context capturing/restoring can be performed at a bandwidth of 22-28 MB/s, which allows for context switches in the order of milliseconds.

Keywords: Preemptive hardware multitasking · Context save and restore · Partial reconfiguration · Reconfigurable system-on-chip · ICAP

1 Introduction

Dynamic partial reconfiguration (DPR) is one of the most exciting features of modern field-programmable gate arrays (FPGAs). DPR allows to reconfigure partial regions of the FPGA fabric without affecting the rest of the system. Reconfigurable systems-on-chip (rSoCs) combine processor(s) with multiple reconfigurable hardware regions (slots) on a single chip and use DPR to dynamically switch between multiple hardware tasks in a slot. For instance, rSoCs can dynamically map the most used tasks to hardware according to the current workload to increase the system performance. However, hardware tasks can not be preempted in most rSoCs and either have to run to completion or need to be terminated during execution, before they can be replaced by other hardware tasks. This severely limits the scope for hardware multitasking, where multiple hardware tasks share the same reconfigurable slots over time.

In contrast to this, software systems support preemptive multitasking where multiple software tasks share a single processing unit. A scheduler selects the next

K. Sano et al. (Eds.): ARC 2015, LNCS 9040, pp. 79–90, 2015.
DOI: 10.1007/978-3-319-16214-0_7

software tasks which should be executed on the processor by following a given scheduling algorithm. Using preemptive multitasking, a scheduler can preempt and resume all tasks at arbitrary points in time and therefore ensure fairness amongst competing software tasks, minimize starvation and improve the responsiveness of the tasks by applying smart scheduling algorithms. Unfortunately, we do not see similar benefits for most reconfigurable hardware systems to fully exploit the DPR feature on today's FPGAs, preemptive hardware multitasking should be supported in rSoCs. One major challenge of preemptive hardware multitasking is the saving/restoring of the task's context. Unlike software tasks that have a well-defined context, the context of a hardware task is stored in a large number of state-holding elements, such as flip-flops, DSP blocks, LUT-RAMs and block RAMs, which complicates context switching.

Several research projects have developed preemptive hardware multitasking techniques, which either follow a (i) `task-specific` or a (ii) `bitstream read-back` preemption technique. The task-specific techniques add dedicated hardware structures to the hardware tasks, e.g. scan-chains [2,5], such that the context of a task can be extracted/inserted. However, task-specific methods generate a considerable overhead in hardware resources and require modifications of the tasks at source code or netlist level. It would be highly preferable, if the hardware tasks do not need to be modified at all (similar to software tasks).

The second class of preemptive hardware multitasking techniques reads-back the current configuration of the slot area over the internal configuration access port (ICAP). Related work has shown that preemptive hardware multitasking is possible using bitstream read-back for Virtex-e [4], Virtex-4 [3] and Virtex-5 [8] FPGAs. Unfortunately, the bitstream read-back methods have to be tailored to the FPGA families, since the FPGA architectures and bitstream format change from one FPGA family to the next. Most proposed methods require modifications at the task-level, which complicates their integration to existing rSoCs.

Although preemptive hardware multitasking seems to be highly beneficial, no advanced execution environment for rSoCs seems to support any of these techniques. Therefore, we show in this paper that our novel preemptive hardware multitasking technique can be integrated into a multithreaded execution environment called ReconOS. To the best of our knowledge, this is the first paper that investigates hardware task preemption on Virtex-6 FPGAs using bitstream read-back over the ICAP interface.

This paper provides the following contributions:

1. We give detailed instructions how to capture and restore flip-flops and block RAMs on Virtex-6 FPGAs, revealing many information that cannot be found in the official Xilinx documentation. Our novel preemptive hardware multitasking technique does not require any changes at task level.
2. We extended the ReconOS execution environment for rSoC architectures to support our preemptive hardware multitasking technique. For this purpose, we implemented a new hardware ICAP controller that supports all required functionality for capturing/restoring a task's context.

3. Finally, we show in our experimental evaluation on a ReconOS system that we can efficiently restore the contexts of four different hardware threads on a Virtex-6 FPGA at arbitrary points in time.

The paper is structured as follows: Section 2 discusses related work and Section 3 presents our preemptive hardware multitasking methodology. In Section 4 we show how the presented multitasking methodology can be embedded to the ReconOS execution environment. Finally, Section 5 presents our experimental results and Section 6 concludes the paper.

2 Related Work

Several context save and restore approaches have been studied in the past 15 years. Simmler et. al. [9] first proposed a technique for transparent context saving and restoring by bitstream read-back and manipulation. By refining these concepts, Kalte and Porrmann [4] implemented an architecture for relocatable hardware tasks which allows the extraction of state values from an FPGA's storage elements and their injection into partial bitstreams for reconfiguration in a different location on the device. Their approach is transparent to the hardware module's designer, but was tailored to outdated Xilinx Virtex-e FPGAs.

More recent related work has demonstrated that context saving and restoring can also be performed on newer Xilinx FPGAs, such as Virtex-4/5 FPGAs, by reading back the configuration data over the ICAP interface. For instance, Jozwik et al. [3] have used a Virtex-4 FPGA to capture and restore the context of hardware tasks. However, in contrast to our approach they needed additional combinational logic inside the reconfigurable regions to be able to restore the task's context. Morales-Villanueva and Gordon-Ross [8] have presented a technique to capture and restore the context of hardware tasks over the ICAP interface on Virtex-5 FPGAs. They have also demonstrated that it is possible to relocate hardware tasks between reconfigurable regions. Our work fits well to the approaches that read-back all registers of a reconfigurable region over the ICAP interface [3,8]. Unlike related work, we focus on newer Virtex-6 FPGAs and additionally capture and restore the state of block RAMs.

As an alternative approach, Jovanovic et. al. [2] as well as Koch et al. [5] proposed linking registers together in a serial scan-chain that can be used to read or write a hardware module's context in a transparent manner. Compared to the previous preemption techniques, the time overhead for a task preemption is reduced at the cost of additional hardware. Although the scan-chain approach can be applied to all FPGA families, it requires modifications of the hardware modules. In contrast to this approach, we can preempt hardware tasks without any modification at the source code or netlist level.

Lübbers and Platzner [7] extended the multithreaded programming model provided by ReconOS to support cooperative scheduling techniques. In cooperative multitasking a hardware thread informs the operating system whether it can be preempted. At thread preemption the thread saves its context to a shared memory that can be accessed by the operating system. This approach can significantly reduce the thread context and allows for thread migrations to other reconfigurable regions. However, in contrast to our approach cooperative multitasking requires deep modifications of the hardware threads at source code level and does not allow for thread preemptions at arbitrary points in time.

3 Methodology: How To Preempt Hardware Tasks

This section introduces our novel methodology for hardware task preemption on Xilinx FPGAs at arbitrary points in time. We assume that the vendor design tools are used to generate an rSoC, which contains at least one reconfigurable region. The Xilinx design tools for partial reconfiguration [10] generate (i) a full bitstream that contains the configuration of the entire FPGA fabric and (ii) multiple partial bitstreams that contain task-specific configurations of the reconfigurable regions (slots). At system start, the full bitstream has to be downloaded to the FPGA configuration memory, which contains the entire rSoC configuration. At run-time, the rSoC can dynamically replace a hardware task in a reconfigurable slot by downloading the partial bitstream of the next hardware task over the ICAP interface. We assume that the partial bitstreams are stored in external memory, e.g. a compact flash card. It is important to note that the `RESET_AFTER_RECONFIG` attribute has to be set for all reconfigurable regions in the user constraint file (UCF), when the vendor tools are executed. Otherwise the state of the FPGA resources can not be restored for these regions.

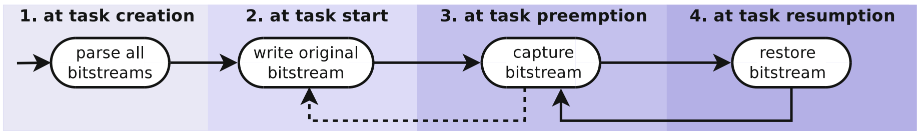

Fig. 1. Preemptive hardware multitasking approach for a reconfigurable region

The Xilinx approach for partial reconfiguration does not support saving and restoring a task's context out of the box. Therefore, we propose a novel methodology for Xilinx FPGAs which captures/restores the context of a partial region to allow for preemptive hardware multitasking. Our methodology does not require any modification of the hardware tasks, but we assume that each task has a clock and a reset signal. Our preemption methodology relies on the capabilities of the ICAP interface and the bitstream format of the FPGA family. Currently, we only support the Xilinx Virtex-6 family. Figure 1 shows the main stages of our multitasking approach for one reconfigurable region. The four stages of our methodology are described in the following subsections.

3.1 At Task Creation: Parse All Partial Bitstreams

In an initial stage, we parse all partial bitstream of a reconfigurable region to identify certain configuration metadata. The partial bitstreams of a Xilinx

Virtex-6 contains three different kinds of configuration frame blocks: (i) configurable logic blocks (CLBs), input/output blocks, clocks, (ii) BRAM contents and (iii) a `CFG_CLB` block. The `CFG_CLB` block defines which part of the FPGA needs to be reset or reconfigured. It only appears in a partial bitstream, if the `RESET_AFTER_RECONFIG` attribute has been set for this region [10,11]. Figure 2 shows an abstract overview of a partial bitstream with three frame blocks.

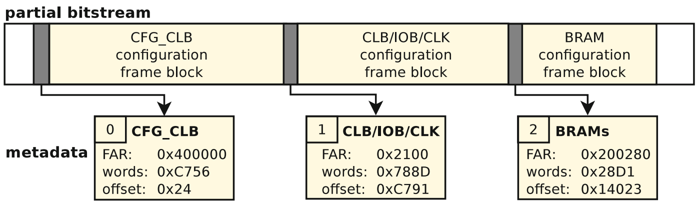

Fig. 2. Parsing metadata of a partial bitstream

We store the frame address register (FAR), the number of 32-bit words and the offset inside the partial bitstream of each frame block as metadata. This metadata is required when we want to capture the context of a hardware task. Furthermore, the rSoC creates a working copy of each original bitstream at run-time, called 'captured bitstream'. This working copy is stored in main memory. It gets updated, whenever the corresponding task is preempted, to store the captured context. It can be deleted, when the task terminates. Similar to Liu et al. [6], we replace the cyclic redundancy check (CRC) at the end of the captured bitstream with a 'no operation' command, thus we do not need to update the CRC value at task preemption. Note that this might cause reliability issues.

3.2 At Task Start: Write Original Bitstream

In the second stage, we configure a hardware task for a first time. Hence, no context needs to be restored. Therefore, we only need to write the original partial bitstream that has been generated by the Xilinx `bitgen` tool to the ICAP interface. Details on how to write (partial) bitstreams to the ICAP interface can be found in the corresponding user guide [11].

We set the reset signal of the hardware task during the reconfiguration process. This reset signal should also be connected to the static interfaces that connect the hardware task to the rest of the system. If the static interfaces to the reconfigurable region are active during reconfiguration, it might happen that data is sent accidentally from the partial region to the interfaces.

3.3 At Task Preemption: Capture Bitstream

In the third stage, a hardware thread is preempted and we need to store the contents of its state-holding elements, such as flip-flops (FFs) and block RAMs

(BRAMs). We deactivate the clock of the hardware task during the capturing process to freeze the task execution.

In a first step, we capture the current state of all flip-flops in hidden registers in the configuration memory (INIT0/INIT1) by calling the `GCAPTURE` command over the ICAP interface. The hidden registers INIT0/INIT1 contain the initial states of all FFs and are used during the initial configuration of the FPGA. The `GCAPTURE` command must be sent over ICAP to the device, which replaces the initial values of the FFs with the captured values, see [10,11] for more details. Per default this command operates on the entire FPGA fabric. Therefore, we need to define constraints for the reconfigurable region by writing the `CFG_CLB` configuration frame block of the original bitstream to the ICAP interface.

In a second step, we read back the configuration frame blocks of the reconfigurable region as defined by the metadata in Section 3.1 (with an exception for the `CFG_CLB` block). We update the captured bitstream of the hardware task by overwriting the configuration frames with the captured configuration frames. Instructions for reading back configuration frame blocks can be found in [11].

In a final step, we need to modify certain configuration bits in order to restore the BRAM contents at a later point. We believe that these bits define for each BRAM whether its memory contents should be restored. We have found out that these bits follow a certain pattern by investigating the bitstreams for different reconfigurable regions. Hence, this step can be automated. According to our observations, we have to modify a single bit of specific 32-bit words in the configuration frame blocks for BRAMs by following this equation:

$$w'_i(j) = \begin{cases} 0, & \text{if } \exists k \in \mathbb{N}_0 : i = \frac{81k+36}{8} \land j = 17 \\ w_i(j), & \text{otherwise} \end{cases}$$

where $w_i(j)$ is the j-th bit of the i-th configuration word (31 downto 0).

3.4 At Task Resumption: Restore Bitstream

In the final stage, we write a captured bitstream back to the reconfigurable region to restore the previously preempted hardware task. After this reconfiguration we need to trigger the global set/reset port of the `STARTUP_VIRTEX6` primitive. Otherwise, the states of the FFs and BRAMs will not be restored. The partial bitstreams contain a `GRESTORE` command, which is probably supposed to call this startup primitive. However, similar to [8] we have observed in our experiments that the `GRESTORE` command did not restore the states of the FFs and BRAMs. Therefore, we manually trigger the startup method over the global set/reset (GSR) port over the `STARTUP_VIRTEX6` interface after writing back the captured bitstream. Furthermore, we set the reset signal of the hardware task in order to prevent unexpected behavior of the hardware task during reconfiguration of the reconfigurable region. We unset the reset signal before we trigger the GSR event. Similar to the capturing stage, we disable the clock in this stage.

4 ReconOS Architecture For Hardware Multitasking

We have integrated our multitasking methodology to the operating system ReconOS. Although ReconOS supports the dynamic reconfiguration of hardware modules (hardware threads) out of the box, there was no support for preemptive hardware multitasking. In this section we describe the ReconOS architecture and introduce our new ReconOS ICAP hardware controller and software scheduler, which can preempt and resume hardware tasks in reconfigurable regions at arbitrary points in time.

4.1 ReconOS Multithreading Approach and Architecture

The operating system ReconOS [1] extends the multithreaded programming model to the domain of reconfigurable hardware. Instead of regarding hardware modules as passive coprocessors to the system CPU, they are treated as independent hardware threads on an equal footing with software threads running on the system. ReconOS allows hardware threads to use the same operating system (OS) services for communication and synchronization as software threads, providing a transparent programming model across the hardware/software boundary. The hardware threads are represented by delegate threads in software, which call the operating system services on behalf of the hardware threads.

ReconOS has been implemented as an extension to (embedded) operating system kernels, such as Linux or Xilkernel. ReconOS is targeted at platform FPGAs integrating microprocessors and reconfigurable logic. It takes advantage of the dynamic partial reconfiguration capabilities of Xilinx FPGAs to reconfigure hardware threads during run-time. This allows multiple hardware threads to transparently share the reconfigurable resources.

Figure 3 shows the hardware architecture of a ReconOS system that supports preemptive hardware multitasking. The architecture contains a single reconfigurable hardware region (reconfigurable slot), which can hold one hardware thread at a time. A dedicated hardware OS interface (OSIF) handles the hardware threads OS requests and forwards them to the operating system kernel running on the CPU. It also manages the low-level synchronization. Each hardware thread is connected to the memory subsystem over a memory interface (MEMIF), such that each thread can autonomously access the main memory.

In Figure 3 the hardware thread A is configured to the reconfigurable slot. However, a software scheduler can replace the currently running hardware thread with another thread (B, C, or D). The original and captured partial bitstreams of all available hardware threads (A–D) are stored in the main memory. The captured bitstreams include the captured states of the FFs and the BRAMs.

In ReconOS a hardware thread is connected to its delegate thread and to the memory subsystem over FIFO-based interfaces. The threads should not be preempted while the thread sends/receives data to/from the FIFO interfaces, since this data is currently not captured (and restored). Hence, the scheduler should wait until all FIFO interfaces of a thread are empty, before it preempts the thread. We assume that our hardware threads are computing for the majority

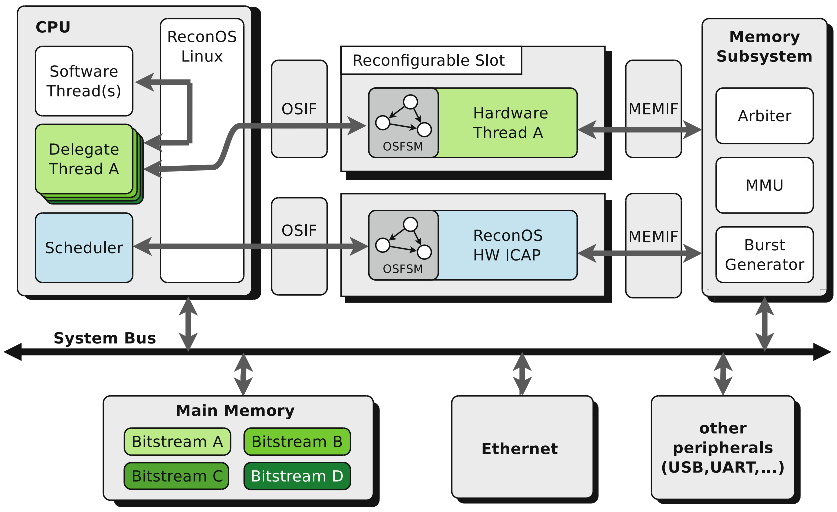

Fig. 3. ReconOS architecture with one hardware slot and four hardware threads (A–D)

of the time and only access the OSIF/MEMIF interface once in while. Hence, we believe that this restriction of the interruptibility can be neglected in practice.

4.2 ReconOS ICAP Controller

We have implemented a new hardware ICAP controller and a software scheduler, which support preemptive hardware multitasking in ReconOS. The scheduler is a Linux user-space task that controls all stages of our multitasking methodology and performs the required modifications of the captured bitstreams. The ReconOS HW ICAP controller was implemented as a ReconOS hardware thread, such that the controller has a separate interface to the main memory. Therefore, the software scheduler only needs to send read/write commands and main memory addresses to the ReconOS HW ICAP thread to read-back or write partial bitstreams. The scheduler can also set the reset signals and enable/disable the clock signals for all reconfigurable hardware slots. Furthermore, the scheduler can trigger the global set/reset port of the `STARTUP_VIRTEX6` interface, which is instantiated in the ReconOS HW ICAP thread.

Figure 4 shows the block diagram of the ReconOS HW ICAP controller. The ICAP interface is connected to a local dual-port memory which is controlled by a separate finite state machine (ICAP_FSM) that manages the transfer of the bitstream between the local memory and the ICAP interface. This local memory can be accessed by a second finite state machine (Reconos_FSM) that manages the communication with the operating system and the main memory. The local memory is not large enough to hold a complete bitstream. Thus, the ICAP controller splits the bitstreams into chunks. We use double-buffering for writing original/captured bitstreams and single-buffering for reading back the configuration frame blocks of a slot.

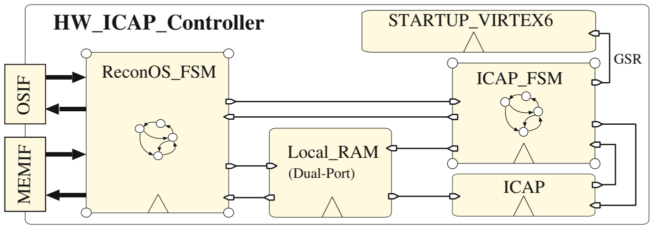

Fig. 4. ReconOS HW ICAP controller

5 Experimental Results

In this chapter we present experimental results for our preemptive hardware multitasking approach on Virtex-6 FPGAs. We have performed all measurements on a Xilinx Virtex-6 ML605 evaluation board (XC6VLX240T FPGA).

In our experiments, we have used a ReconOS design with a single reconfigurable slot as depicted in Figure 3. The processor, all hardware modules and hardware threads were clocked at 100 MHz. We have tested four reconfigurable hardware threads: `ADD`, `SUB`, `MUL`, and `LFSR`, which are described below:

1. The `ADD` thread contains three 32-bit registers R_{1-3}. The thread continuously computes $R_3 = R_1 + R_2$. The registers can be accessed by a software application over the OSIF interface.
2. The `SUB` thread is similar to the `ADD` thread, but computes $R_3 = R_1 - R_2$.
3. The `MUL` thread computes the product of R_1 and R_2 in R_1 steps. The (intermediate) result is stored in R_3. The result R_3 is computed as the addition of R_2 with itself R_1 times, i.e. $R_3 = \sum_{i=1}^{R_1} R_2$. In each step, R_1 is decremented by one and R_3 is updated to the current intermediate result. Hence, we can preempt the thread during computation and validate if the register values have been correctly captured/restored. This thread is used to validate the cycle-true state restoration of flip-flops.
4. The `LFSR` thread stores the values of several linear feedback shift registers (LFSRs) in a local memory of 8KB and continuously shifts their register values. The thread only implements a single 16-bit linear feedback shift register, which processes all LFSRs sequentially. For this purpose, the hardware thread loads the value of one LFSR at a time from the local memory, shifts its 16-bit register for one bit and stores the register value back to the local memory; and then continues with the next LFSR. Figure 5 shows the overview of the LFSR thread.
   The initial values for the LFSRs are copied from the main memory to the local BRAMs over the MEMIF interface. The `LFSR` thread is used to validate the cycle-true state restoration of flip-flops and BRAMs. In our experiments, we have stored four LFSRs in the local memory.

The `ADD`, `SUB`, and `MUL` threads also contain an 8KB local memory each, which can be accessed from software over the MEMIF interface in order to test, if the

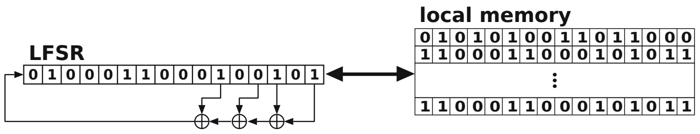

Fig. 5. `LFSR` hardware thread

BRAM entries have been captured and restored correctly. However, these threads do not alter their local memories internally during computation. Hence, we can not validate a cycle-true state restoration of the BRAMs for these threads.

In extensive experiments, we validated that all hardware threads could be successfully preempted and restored at arbitrary points in time. For the `MUL` and `LFSR` thread, we could validate that the state restoration was cycle-true for both FFs and BRAMs. Table 1 shows the results of our performance measurements for two bitstream sizes. We have randomly selected two regions on the FPGA fabric, where the first region covers about 2% of the FPGA area (bitstream size: 361 KB) and the second region covers about 4% of the FPGA area (bitstream size: 741 KB). It can be seen that context capturing takes longer than context restoring and that the execution times depend linearly on the bitstream size.

Table 1. Context capture/restore performance

bitstream size	$t_{capture}$	$t_{restore}$	t_{total}	**bandwidth**	**max swaps/s**
741 KB	16.0 ms	9.7 ms	25.7 ms	28 MB/s	38
361 KB	10.3 ms	5.5 ms	15.8 ms	22 MB/s	63

The capture time $t_{capture}$ of a partial bitstream depends on the number and size of reconfiguration frames and the overhead to trigger the readback requests over the ICAP interface. For performance reasons, successive reconfiguration frames can be combined to a single readback request, which lowers the overhead caused by the ICAP interface. For both cases, the partial bitstreams can be read back with two readback requests only, one for the CLB configurations and one for the BRAM configurations. Since the performance overhead for the two readback operations is the same for both bitstream sizes, the relative bandwidth is higher for the larger bitstream size in Table 1. However, the maximum number of task swaps per second is lower for the larger bitstream.

Scheduling algorithms for reconfigurable hardware threads are not in the scope of this paper. However, we have performed an example measurement over time for a manually-defined schedule that uses all four hardware threads, which is shown in Figure 6. In this example, all four threads are scheduled periodically in the following sequence: `ADD`→`SUB`→`LFSR`→`SUB`→`ADD`→`MUL`). The individual time slices have been predefined manually (not by a scheduling algorithm) and we assume that the threads are independent from each other. We can see that it is possible to swap several times between the hardware threads in the interval of

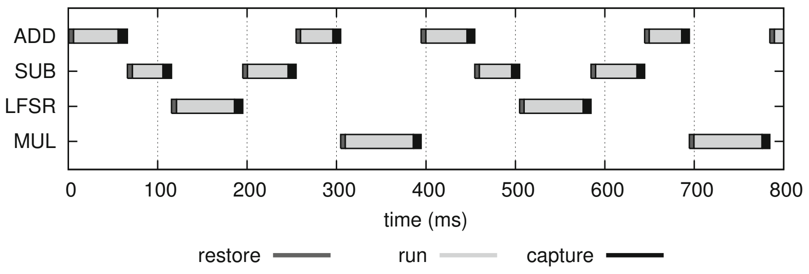

Fig. 6. Preemptive multitasking with four hardware threads

Table 2. Resource consumption

component	#FFs	#LUTs	#BRAMs
HW slot	5616	2808	7
ADD/SUB	375	603	2
MUL	474	751	2
LFSR	474	810	2
ReconOS HW ICAP	323	741	2
XPS_HWICAP	750	804	1

a single second. The partial bitstream size was 361 KB in this example, which corresponds to about 2% of the FPGA area.

Table 2 lists the resource consumption for the partial reconfigurable slot, for our reconfigurable hardware threads and for the ReconOS HW ICAP controller. It can be seen that our hardware threads only use a fraction of the actual slot area. However, we always capture and restore all flip-flops and BRAMs of the partial region. Hence, the capture and restore times for this slot are always the same. This means that we could implement more complex threads than the `ADD`,`SUB`,`MUL`,`LFSR` thread for this slot without increasing the capture/restore times. The ReconOS HW ICAP controller represents the entire hardware overhead of our multitasking approach, since we do not introduce any extra logic to the hardware threads in contrast to most related work. However the resource consumption of our HW ICAP controller is comparable to the resource consumption of the Xilinx `XPS_HWICAP` controller. Hence, we conclude that the area overhead of our approach is negligible.

6 Conclusion and Future Work

In this paper, we have shown a novel methodology that allows for preemptive hardware multitasking on Xilinx Virtex-6 FPGAs without requiring modifications at the task level. Our approach reads back the contents of all FFs and block RAMs of a predefined reconfigurable region on an FPGA fabric over the ICAP interface. We have integrated our preemptive hardware multitasking approach to the ReconOS operating system. In our experiments, we could successfully capture and restore the context of four different hardware threads at a bandwidth of 22-28 MB/s, which allows for multiple tasks swaps per second.

In future work we plan to extend our context capturing / restoring mechanisms to LUT-RAMs and DSP blocks and experiment with real-world applications. We plan to port our hardware multitasking approach to further FPGA families, such as Xilinx Virtex-7 FPGAs or Zynq SoC boards. Finally, we want to investigate how task relocation techniques can be integrated to ReconOS.

References

1. Agne, A., Happe, M., Keller, A., Lübbers, E., Plattner, B., Platzner, M., Plessl, C.: ReconOS - An Operating System Approach for Reconfigurable Computing. IEEE Micro **34**(1), 60–71 (2014)
2. Jovanovic, S., Tanougast, C., Weber, S.: A hardware preemptive multitasking mechanism based on scan-path register structure for FPGA-based reconfigurable systems. In: NASA/ESA Conf. on Adaptive Hardware and Systems (2007)
3. Jozwik, K., Tomiyama, H., Honda, S., Takada, H.: A novel mechanism for effective hardware task preemption in dynamically reconfigurable systems. In: Int. Conference on Field Programmable Logic and Applications (2010)
4. Kalte, H., Porrmann, M.: Context saving and restoring for multitasking in reconfigurable systems. In: FPL Conference. IEEE (2005)
5. Koch, D., Haubelt, C., Teich, J.: Efficient hardware checkpointing: Concepts, overhead analysis, and implementation. In: FPGA Symp. ACM (2007)
6. Liu, S., Pittman, R.N., Forin, A.: Minimizing partial reconfiguration overhead with fully streaming DMA engines and intelligent ICAP controller. In: ACM/SIGDA Int. Symposium on Field Programmable Gate Arrays (2010)
7. Lübbers, E., Platzner, M.: Cooperative multithreading in dynamically reconfigurable systems. In: FPL Conference. IEEE (2009)
8. Morales-Villanueva, A., Gordon-Ross, A.: HTR: On-Chip Hardware Task Relocation for Partially Reconfigurable FPGAs. ARC 2013. LNCS, vol. 7806, pp. 185–196. Springer, Heidelberg (2013)
9. Simmler, H., Levinson, L., Männer, R.: Multitasking on FPGA coprocessors. In: Int. Workshop on Field Programmable Logic and Applications. Springer (2000)
10. Xilinx: Partial Reconfiguration - User Guide UG702 v14.5 (2013). http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug702.pdf
11. Xilinx: Virtex-6 FPGA Configuration - User Guide UG360 v3.7 (2013). http://www.xilinx.com/support/documentation/user_guides/ug360.pdf