Preemption of the Partial Reconfiguration Process to Enable Real-Time Computing With FPGAs

ENRICO ROSSI, Scuola Superiore Sant'Anna
MARVIN DAMSCHEN and LARS BAUER, Karlsruhe Institute of Technology
GIORGIO BUTTAZZO, Scuola Superiore Sant'Anna
JÖRG HENKEL, Karlsruhe Institute of Technology

To improve computing performance in real-time applications, modern embedded platforms comprise hardware accelerators that speed up the task's most compute-intensive parts. A recent trend in the design of real-time embedded systems is to integrate field-programmable gate arrays (FPGA) that are reconfigured with different accelerators at runtime, to cope with dynamic workloads that are subject to timing constraints. One of the major limitations when dealing with partial FPGA reconfiguration in real-time systems is that the reconfiguration port can only perform one reconfiguration at a time: if a high-priority task issues a reconfiguration request while the reconfiguration port is already occupied by a lower-priority task, the high-priority task has to wait until the current reconfiguration is completed (a phenomenon known as *priority inversion*), unless the current reconfiguration is aborted (introducing unbounded delays in low-priority tasks, a phenomenon known as *starvation*). This article shows how priority inversion and starvation can be solved by making the reconfiguration process *preemptive*—that is, allowing it to be interrupted at any time and resumed at a later time without restarting it from scratch. Such a feature is crucial for the design of runtime reconfigurable real-time systems but not yet available in today's platforms. Furthermore, the trade-off of achieving a guaranteed bound on the reconfiguration delay for low-priority tasks and the maximum delay induced for high-priority tasks when preempting an ongoing reconfiguration has been identified and analyzed. Experimental results on the Xilinx Zynq-7000 platform show that the proposed implementation of preemptive reconfiguration introduces a low runtime overhead, thus effectively solving priority inversion and starvation.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; **Real-time systems**; • **Hardware** → *Communication hardware, interfaces and storage*; *Emerging technologies*;

Additional Key Words and Phrases: Field programmable gate array (FPGA), dynamic partial reconfiguration, real-time reconfiguration, preemptive reconfiguration

ACM Reference format:
Enrico Rossi, Marvin Damschen, Lars Bauer, Giorgio Buttazzo, and Jörg Henkel. 2018. Preemption of the Partial Reconfiguration Process to Enable Real-Time Computing With FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 11, 2, Article 10 (July 2018), 24 pages.
https://doi.org/10.1145/3182183

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89).
Authors' addresses: E. Rossi and G. Buttazzo, Scuola Universitaria Superiore Sant'Anna, Piazza Martiri della Libertá 33, 56127 Pisa, Italy; emails: {enrico.rossi, giorgio.buttazzo}@gantannapisa.it; M. Damschen, L. Bauer, and J. Henkel, Karlsruhe Institute of Technology, Kaiserstraße 12, 76131 Karlsruhe, Germany; emails: {damschen, lars.bauer, henkel}@kit.edu.

https://doi.org/10.1145/3182183

1 INTRODUCTION

Real-time systems are ubiquitous in our everyday life, such as in safety-critical domains like automotive, avionics, or the rapidly growing domain of smart/autonomous machines (e.g., robotics or automated driving). In contrast to general-purpose computing systems, the correctness of a real-time system depends not only on the results of its computations but also on the *time* at which outputs are produced. Delivering a result after a predetermined deadline may lead to malfunctions that can jeopardize the entire system. Therefore, for many safety-critical systems it is essential to *guarantee* that all time-sensitive computations are able to complete their execution within their deadlines. To improve the performance of real-time systems, modern embedded platforms comprise hardware accelerators that speed up the task's most compute-intensive parts.

A recent trend in the design of real-time embedded systems is to integrate field-programmable gate arrays (FPGAs) that are reconfigured with different accelerators at runtime, to cope with dynamic workloads, like in signal processing or computer vision applications [2, 6, 8, 11, 15]. For instance, platforms like the Xilinx Zynq or Altera SoC combine general-purpose CPUs with an FPGA on a single chip to enable the development of application-specific accelerators that can run on the FPGA in parallel with the software executing on the CPU. Low-latency channels enable communication between accelerators and software. In addition, the possibility of reconfiguring specific portions of the FPGA at speeds of 400MB/s enables the adoption of runtime virtualization techniques to share the FPGA among multiple tasks in different time windows, so extending the number of functions that can be accelerated. Such a virtualization technique for the FPGA has been proven to be effective to achieve a significant speedup in real-time applications [2, 12]. Virtualization is generally realized using *partial* FPGA reconfiguration, where parts of the FPGA are reconfigured while the remaining configuration remains fully functional.

One of the major limitations when dealing with partial FPGA reconfiguration is that the reconfiguration port can only perform one reconfiguration at a time. This means that when multiple tasks issue simultaneous requests to reconfigure different portions of the FPGA fabric, such requests must be sequentialized according to a given scheduling algorithm. In current implementations, the reconfiguration port of the FPGA does not allow preemptions; that is, once a reconfiguration process is started, it can either be completed or aborted. In other words, if a high-priority task issues a reconfiguration request while the reconfiguration port is already occupied by a lower-priority task, the high-priority task has to wait until the current reconfiguration is completed, unless the current reconfiguration is aborted to be restarted later from the beginning (see Section 3). Unfortunately, considering the relatively long reconfiguration delays of current platforms[1] (e.g., compared to context switching), both such solutions are not suitable for real-time applications. In fact, while non-preemptive reconfigurations introduce long delays in high-priority tasks (a phenomenon known as *priority inversion*), aborting them may introduce unbounded delays in low-priority tasks (a phenomenon known as *starvation*). Both problems can be avoided by making the reconfiguration process *preemptive*, allowing it to be interrupted at any time and *resumed* at a later time from the point of interruption. Such a feature is not yet available in today's platforms.

This article presents a solution to preempt the partial reconfiguration process of hardware accelerators onto an FPGA, preventing both *priority inversion* and *starvation* phenomena. We demonstrate the feasibility of preempting partial reconfiguration, analyze its worst-case latencies, and verify the findings in an experimental evaluation.

[1]For example, on the Xilinx Zynq-7010 platform (lower end in terms of reconfigurable resources), a maximum reconfiguration bandwidth of 400MB/s is supported; however, reconfiguring 25% of the total resources still takes more than 2ms.

In particular, the novel contributions of this work can be summarized as follows:

- To our knowledge, this is the first realization of a preemptive FPGA reconfiguration.[2] Our approach does not require any changes to the bitstreams that are reconfigured.
- Using the proposed preemptive reconfiguration mechanism, the reconfiguration delay is bounded analytically for low-priority tasks that are subject to preemptions.
- A trade-off analysis is presented to balance the reconfiguration delay experienced by the low-priority tasks and the maximum delay induced in high-priority tasks when preempting an ongoing reconfiguration.

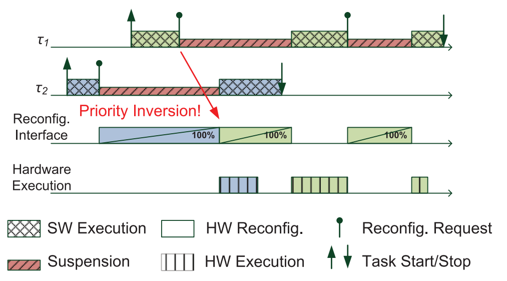
2 RELATED WORK

Research on operating systems for reconfigurable hardware has mainly focused on providing the same abstractions and multitasking capabilities for accelerated functions as are common for software tasks, so that they can be scheduled as "hardware tasks." Accordingly, research on real-time scheduling using runtime-reconfigurable hardware originally considered software and hardware tasks to be separate entities [8, 10, 11, 14]. In early work, the reconfiguration delay was assumed to be negligible [8, 14]. However, reconfiguration delays on current FPGAs are in the range of milliseconds, whereas context switches are in the range of microseconds. Pellizzoni and Caccamo [11] accounted for reconfiguration delays in their schedulability analysis of both software and hardware tasks. Software tasks can be migrated to hardware to maximize the number of dynamic tasks that can be admitted in the real-time schedule. Danne and Platzner [8] presented a schedulability analysis for periodic hardware tasks that considers the possibility of preempting their execution (but not their reconfiguration). A realization of how the execution of hardware tasks on the FPGA can be preempted was presented by Happe et al. [10]. Note that storing and restoring the execution state of preempted hardware tasks causes frequent reconfigurations, which stresses the importance of considering the reconfiguration delay.

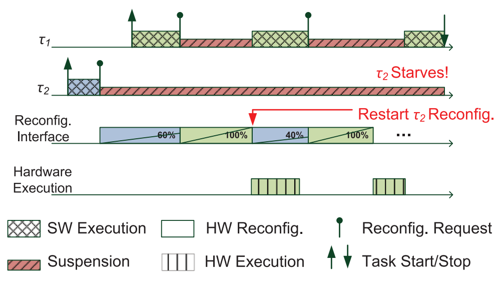
In embedded systems, FPGAs are typically used to accelerate compute-intensive functions that are triggered by software tasks [15]. Input and output data are exchanged using instruction set extensions or the on-chip bus [9]. Recently, Biondi et al. [2] presented a scheduling framework and a related analysis for periodic real-time tasks that exploit runtime reconfiguration of hardware accelerators. It was shown that the proposed framework allows admitting more tasks to the system compared to a full software implementation or a static FPGA allocation. Furthermore, preemption was identified as a desirable property for the reconfiguration port, especially when the execution time of the accelerator is short compared to its reconfiguration delay (e.g., in systems that use instruction set extensions) [5, 9].

Preempting the reconfiguration port requires extensive knowledge about the format and operations contained in the data sent to it. Existing work in this context (e.g., by Happe et al. [10]) utilizes the reconfiguration port to read back the state of configured accelerators. Thus, enabling preemptive execution of configured accelerators, but not preemption for the reconfiguration process itself. In another work, Benz et al. [1] investigated reverse-engineering of hardware circuits in the form of configuration data to netlists, using information obtained from synthesis tools about the structure of the targeted FPGA. The data sent to the reconfiguration port not only consists of the configuration data itself, but additionally includes operations for the reconfiguration port that change the state of the whole FPGA (detailed in Section 6). For preempting reconfigurations, these operations need to be considered.

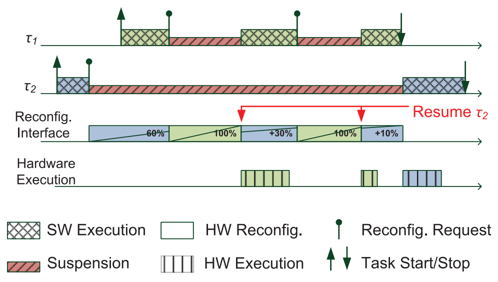
[2]The hardware and software to achieve this property will be released as an open-source project upon publication.



(a) Configure-to-completion leads to priority inversion: When an active reconfiguration cannot be stopped, then τ_1 has to wait for τ_2 to complete its reconfiguration even if τ_1 has a higher priority.



(b) Abort leads to starvation: τ_1 repeatedly requests reconfigurations, aborting τ_2's reconfiguration (that needs to be restarted from the beginning).



(c) Preemptive reconfiguration solves priority inversion and starvation: Lower-priority reconfigurations can be preempted by a higher-priority task. The preempted reconfiguration can be resumed afterward.

Fig. 1. Multi-tasking systems can experience priority inversion and starvation problems due to a non-preemptive shared resource. Those problems are solved in this work for reconfigurations by making the shared reconfiguration port preemptive.

In summary, the existing work on hardware tasks is still not sufficient to provide a complete abstraction that enables preemptive multitasking for software as well as hardware tasks, and while preemptive reconfiguration has been assumed for hardware accelerators in real-time task sets [2], it was so far not realized. In the remainder of this article, the term *task* always refers to software tasks that have access to hardware accelerators.

3 MOTIVATIONAL EXAMPLE

When a task issues a reconfiguration request in the described model (which follows the commonly used model detailed in Biondi et al. [2]), it self-suspends to wait for the reconfiguration to be completed, allowing the other tasks to execute on the CPU. Consider, for instance, a single-core system running two tasks, τ_1 and τ_2, with τ_1 having higher priority than τ_2. The following three approaches can be adopted for managing reconfiguration requests:

(a) *Configure-to-completion*: An active reconfiguration cannot be preempted or aborted once started, but occupies the reconfiguration port until completed. This case is shown in Figure 1(a), where τ_2 starts executing and takes control over the reconfiguration port. When τ_1 starts executing and requests a reconfiguration, it must wait for τ_2 to complete its reconfiguration—that is, tasks are not executed according to their priority order (*priority inversion*).

(b) *Abort*: The reconfiguration process cannot be suspended and resumed, but it can be aborted. Every time a reconfiguration is aborted, it needs to be restarted from the beginning. While this policy avoids priority inversion, it can lead to starvation of τ_2, as shown in Figure 1(b). τ_2 starts executing and takes control over the reconfiguration port, but its reconfiguration is aborted once τ_1 requests a reconfiguration. When higher-priority reconfiguration requests abort τ_2's reconfiguration frequently, then τ_2 suffers from starvation.

(c) *Preemptive reconfiguration*: The reconfiguration requested by a task can be preempted in favor of another higher-priority reconfiguration. As shown in Figure 1(c), τ_2 starts executing and takes control over the reconfiguration port, its reconfiguration is preempted once τ_1 requests a reconfiguration. When the higher-priority reconfiguration is completed, the suspended reconfiguration of τ_2 is resumed from the last feasible point (where it is safe to resume the reconfiguration, details in Section 6). Despite the frequent reconfigurations of τ_1, τ_2 will complete its reconfiguration eventually, because (in reasonable environments) τ_1 does not generate reconfiguration requests at a rate that would constantly occupy the reconfiguration port (details in Section 8).

More realistic scenarios involving multiple tasks and large and more complex hardware accelerators (that require larger reconfiguration time) would lead to a more complex contention of the reconfiguration port, increasing delays and making the problems of priority inversion and starvation even more severe. To solve those problems, this article presents the realization of preemptive reconfiguration. Using the proposed approach the reconfiguration delay for lower-priority tasks is bounded, while higher-priority tasks do not experience priority inversion, thus allowing the development of multi-priority and mixed-criticality systems that benefit from runtime reconfiguration.

The following section introduces the required background for realizing preemptive reconfiguration.

4 BACKGROUND FOR PREEMPTIVE RECONFIGURATION

Reconfiguring an FPGA is more complex than a simple transfer of the configuration data from main memory to the configuration memory of the FPGA. Especially when reconfiguring an FPGA partially, it needs to be ensured that the FPGA remains in a consistent state and the not-reconfigured parts remain functional all the time. A *partial bitstream* contains all the information for the area that should be reconfigured, such as the address in the configuration memory that corresponds to the area of the FPGA where the design should be placed. The state of the FPGA is controlled by a finite state machine (FSM) that is part of the reconfiguration port. This FSM executes *operations* that are part of the bitstream besides the configuration data. This section briefly summarizes the information about the reconfiguration port and bitstreams that are openly available for Xilinx FPGAs [17].

4.1 Xilinx Bitstreams and the Reconfiguration Port

This work focuses on the reconfiguration port of Xilinx FPGAs, called the *internal configuration access port* (ICAP). The information contained in the bitstreams for these FPGAs can be divided into two categories: operations executed by the reconfiguration port FSM and the actual configuration data that is transferred to the FPGA configuration memory. The smallest addressable segments of the configuration memory are called *frames*, and all operations act upon one or more frames. In Xilinx 7 Series FPGAs, each frame consists of 101 32-bit words [17]. The relevant operations for preemptive reconfiguration that are executed by the reconfiguration port FSM are summarized in the following.

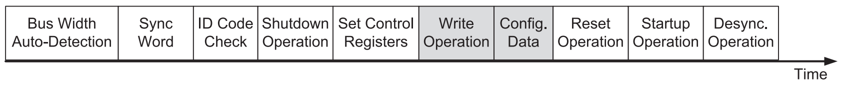


Fig. 2. Generic sequence of operations performed in a partial bitstream. White blocks are operations common to all partial bitstreams; gray blocks are specific operations that depend both on the size and physical position of the reconfigurable area inside the FPGA and on the used FPGA device. The gray blocks constitute the actual configuration data, and the largest part of the partial bitstream by far.

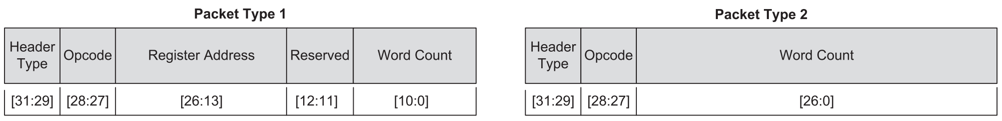


Fig. 3. The FPGA bitstream consists of two packet types: Type 1 and Type 2. The Type 1 packet is used for register reads and writes; the Type 2 packet, which must follow a Type 1 packet, is used to write long blocks.

Figure 2 shows the chronological sequence of the most relevant operations performed by partial bitstreams (a more detailed description can be found in Section 5). Partial bitstreams start with a *Bus Width Auto-Detection* operation that is used to automatically detect the word width that is sent to the reconfiguration port (1, 2, or 4 bytes are possible values). After that, the *Synchronization Word* initializes the reconfiguration port to accept configuration data, followed by the *ID Code Check*, which ensures that the bitstream target device matches the FPGA that is being reconfigured. The *Shutdown Operation* safely shuts the area that is going to be reconfigured down and the *Set Control Register Operation* configures the available reconfiguration features. *Write Operations* transfer the actual configuration data, specifying the starting frame address and the amount of words (multiples of whole frames) to write.

After all configuration data has been loaded into the FPGA configuration memory, a *Reset Operation* is performed to initialize the logic inside the reconfigured region. Partial bitstreams end with a *Startup Operation*, where the device activates I/Os and the logic belonging to the reconfigured area, and a *Desynchronization Operation* that de-synchronizes the reconfiguration port (inverse to the Synchronization Word). After de-synchronization, the reconfiguration port ignores any following data on its inputs until the next synchronization.

Two additional operations are used in partial bitstreams:

- *No-operation*: Decoded by the state machine without producing any actions. Since each operation is executed by the reconfiguration port FSM, each operation has a defined latency. This operation is used, when required, to introduce clock-cycles delays to wait for the completion of the running operation.
- *Null operation*: This is a write operation that writes zeros to a specific FPGA register. Some operation as the *Shutdown Operation* and the *Startup Operation* need to be activated after being issued to the reconfiguration port FSM. The activation is done by a Null operation.

Each operation can be sent to the reconfiguration port in packets of two possible formats: "Type 1" or "Type 2." Type 1 packets are used when small amount of data words (to be read or written) are required by the operation. Type 2 packets are used to write long segments of data into the FPGA configuration memory. The address within the configuration memory needs to be supplied by a preceding Type 1 packet [17].

As Figure 3 shows, both packet types have a *Word Count* field that contains the exact number of data words following the actual operation. It instructs the reconfiguration port FSM to directly write that amount of words to the configuration registers or memory instead of decoding them.

Two FPGA-internal registers are central for the purpose of preemptive reconfiguration:

- *Frame Address Register* (FAR). This register contains the address of the next frame in configuration memory to be written.
- *Frame Data Input Register* (FDRI). This register contains the number of data frames that have to be written to the FPGA configuration memory, starting from the frame address specified by the FAR.

These registers are the reconfiguration state that needs to be restored when a reconfiguration is resumed.

The information summarized in this section has been gathered from openly available Xilinx documentation [17], whereas the information in the following section was gathered by manually decoding partial and full bitstreams.

5 DECODING PARTIAL RECONFIGURATION

As mentioned in Section 4, there is no openly documented structure of partial bitstreams provided by Xilinx. Such a structure is required for preemptive partial reconfiguration to determine points in the bitstream at which reconfiguration can safely be preempted and resumed. Therefore, a general structure for partial bitstreams is defined in the following, based on information gathered from decoding numerous bitstreams. There are two types of bitstreams that have been utilized to obtain the required information: *standard* bitstreams and *debug* bitstreams. Standard bitstreams configure multiple frames after a single write to the FAR and the FDRI, which increment automatically at the end of each frame. Debug bitstreams configure each frame individually, writing the FAR and the FDRI after each frame, and thus providing information about FPGA-specific configuration memory addressing.

With information gathered by decoding both bitstream types, it was possible to group operations in partial bitstreams into *sequences* that fulfill specific purposes. Each operation sequence can optionally contain configuration data organized in *data chunks*. We further group sequences in the bitstream into *sections* (consisting of one or more sequences). Configuration data within a sequence of operations contains the description of the hardware that will be reconfigured inside the pre-defined reconfigurable area, called the *reconfigurable slot*, inside the FPGA. The resulting general structure of sections for partial bitstreams in for Xilinx FPGAs is shown in Figure 4.

A partial bitstream can be structured into three main sections: Common Header, Reconfigurable Slot Data Section, and Common Trailer.

Common Header. This first section is common to all partial bitstreams targeting the same FPGA and it is the only device-specific section. Its main function is to synchronize the reconfiguration port and prepare the device to receive the bitstream. It consists of the following four sequences: *Synchronization Header*, *Write Initialization* followed by its data chunk (which contains the data used to initialize special configurable blocks inside the FPGA), *Shutdown*, and *Set Control Register*. These sequences are made of operations that initialize the reconfiguration port to receive data (Synchronization Word), set its bus width (Bus Width Auto-Detection), and perform the ID code check (ID Code Check). Furthermore, during the Write Initialization sequence, an FPGA-specific number of frames is sent to the configuration memory to configure particular resources called CFG_CLB used to define which part of the FPGA itself will be reset or reconfigured [13]. This section ends with a Shutdown sequence, which safely disables the area that is going to be reconfigured, and a Set Control Register sequence, which configures the FPGA device [17].

Reconfigurable Slot Data Section. This section of the bitstream depends on the slot that is reconfigured. It contains the slot's configuration data, which describe the user-logic that will be written

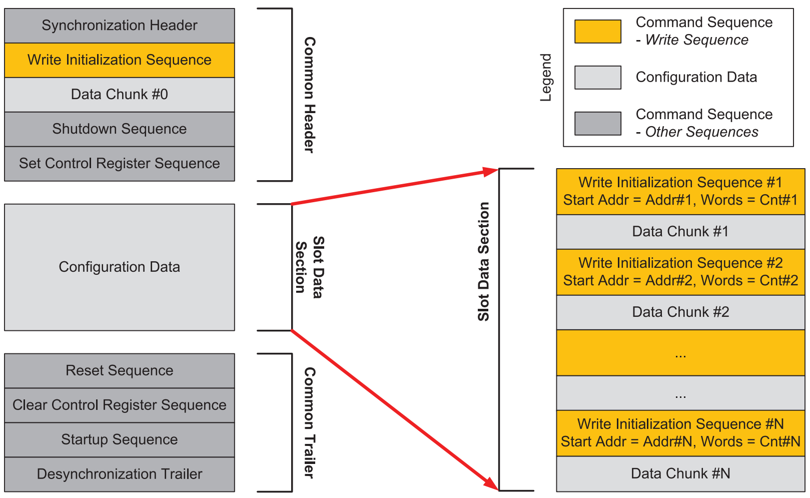

Synchronization Header
Write Initialization Sequence
Data Chunk #0
Shutdown Sequence
Set Control Register Sequence
Common Header
Configuration Data
Slot Data Section
Reset Sequence
Clear Control Register Sequence
Startup Sequence
Desynchronization Trailer
Common Trailer
Legend
Command Sequence - Write Sequence
Configuration Data
Command Sequence - Other Sequences
Slot Data Section
Write Initialization Sequence #1 Start Addr = Addr#1, Words = Cnt#1
Data Chunk #1
Write Initialization Sequence #2 Start Addr = Addr#2, Words = Cnt#2
Data Chunk #2
...
...
Write Initialization Sequence #N Start Addr = Addr#N, Words = Cnt#N
Data Chunk #N

Fig. 4. Structure of a partial bitstream. The left-hand side shows a partial bitstream where three main sections are identified: Common Header, Reconfigurable Slot Data Section, and Common Trailer. Darker sections are common to all partial bitstreams, whereas the Configuration Data section depends on the area that is reconfigured. The right-hand side shows the structure of the Reconfigurable Slot Data Section.

to the FPGA configuration memory. Configuration data is divided into an even number of data chunks consisting of numerous frames to be written into the configuration memory.

Common Trailer. The last section is common to all partial bitstreams targeting the same FPGA. The function of this section is to reset the programmed logic and to de-synchronize the reconfiguration port. Four main sequences can be identified: *Reset*, *Clear Control Register*, *Startup*, and *De-Synchronization Trailer*. These sequences consist of operations used to initialize and activate the reconfigured logic and safely de-synchronize the reconfiguration port.

As explained in Section 4, partial bitstreams contain information for the area to be reconfigured and operations that control the reconfiguration FSM. Therefore, to enable preemptable reconfiguration, the knowledge of partial bitstream structures is essential, because a reconfiguration can be aborted at any time but can only be resumed from specific points in the bitstream. Section 6 explains where in the bitstream reconfigurations can be safely resumed.

6 FINDING VALID RESUMPTION POINTS

A main challenge in realizing a preemptive reconfiguration is to obtain valid *resumption points*, each consisting of a *resumption offset* in the bitstream and its associated FAR and FDRI (see Section 4.1) from which a preempted reconfiguration can safely be resumed. This section describes different types of resumption points and how they are obtained from a partial bitstream.

As detailed in the previous section, configuration data are transferred into the FPGA configuration memory by data chunks that consist of numerous frames. A frame is the smallest transferable amount of configuration data. It is possible to preempt, but not start or resume, a reconfiguration in the middle of a frame. Furthermore, it would be unsafe to resume a reconfiguration in the middle of an operation sequence that instructs the reconfiguration port FSM to write FPGA-internal configuration registers, since the registers that had been written before the preemption could have been changed by the preempting configuration.

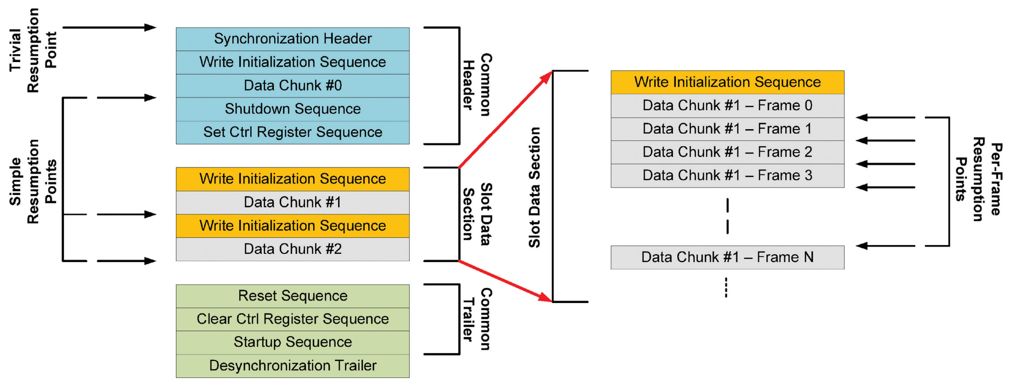


Fig. 5. Position of Simple resumption points and Per-Frame resumption points inside a partial bitstream. A Per-Frame resumption point is fixed at the end of each frame within a data chunk. The resumption point related to the last frame is a Simple resumption point.

Depending on the offset where a reconfiguration is preempted, different procedures are necessary to resume it afterward. Therefore, we define three types of resumption points that require different sequences of operations to resume a preempted reconfiguration:

(i) *Trivial resumption point*: Beginning of a bitstream.
(ii) *Simple resumption point*: The resumption offset points to the end of a data chunk in the bitstream. In this case, reconfiguration can be resumed at the offset after sending the Synchronization Header sequence that is part of the Common Header (see Section 5).
(iii) *Per-Frame resumption point*: The resumption offset points to the end of a frame within a data chunk. Resuming a reconfiguration from Per-Frame resumption points requires determining the correct FAR and FDRI values as well as sending the correct Synchronization Header and Write Initialization sequences based on these values.

Figure 5 shows the placement of different resumption points within the partial bitstream. Simple resumption points require less information to be obtained from the bitstream and to be stored at runtime, but result in considerably higher worst-case reconfiguration delay bounds than Per-Frame resumption points (as detailed in Section 8). Per-Frame resumption points enable the resumption of reconfigurations at every frame (the smallest entity of configuration data that Xilinx FPGAs can process), and thus it is not possible to improve their position by bitstream manipulation. In the following, we will detail how all types of resumption points can be found within a bitstream.

6.1 Simple Resumption Points

This type of resumption point can be found inside a partial bitstream by searching for FDRI operations that are part of Write Initialization sequences as shown in Figure 6. When skipping the following configuration data chunk using the word count of the operation (see Section 4.1), the resulting offset points to a Simple resumption point.

When a reconfiguration is preempted, it needs to be resumed at the previous resumption point. Therefore, the amount of data between two resumption points directly contributes to the additional delay for the preempted reconfiguration. To guarantee a minimum progress for the preempted lower-priority task (in Section 8.2), this overhead should be as low as possible. Simple resumption points impair the performance of the system, because a great amount of progress can be lost for the

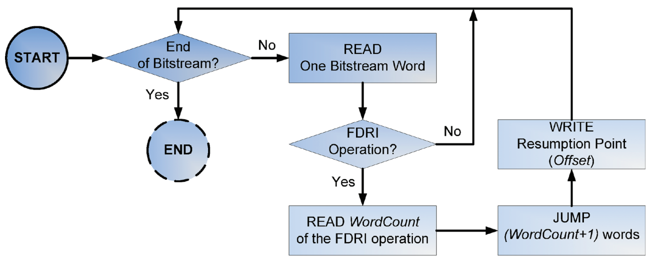

START
End of Bitstream?
No
READ One Bitstream Word
Yes
END
FDRI Operation?
No
WRITE Resumption Point (*Offset*)
Yes
READ *WordCount* of the FDRI operation
JUMP (*WordCount+1*) words

Fig. 6. Flow-chart diagram to find Simple resumption points inside partial bitstreams.

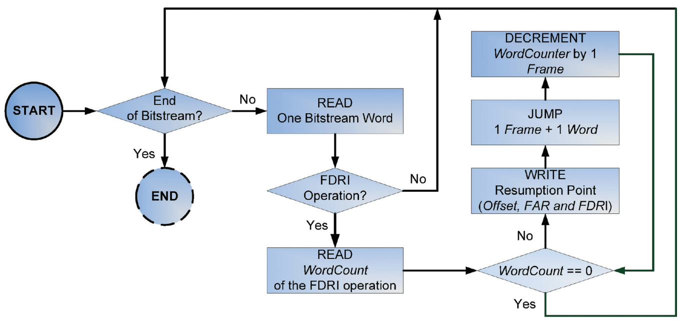

START
End of Bitstream?
No
READ One Bitstream Word
Yes
END
FDRI Operation?
No
Yes
READ *WordCount* of the FDRI operation
DECREMENT *WordCounter* by 1 *Frame*
JUMP 1 *Frame* + 1 *Word*
WRITE Resumption Point (*Offset, FAR and FDRI*)
No
WordCount == 0
Yes

Fig. 7. Flow-chart diagram that shows how to find Per-Frame resumption points inside the Slot Data Section of a partial bitstream. The FAR increment in this section is 0x01.

lower-priority tasks. The distance between Simple resumption points depends on the size of the area that is reconfigured: in a minimum-size slot on a Zynq-7z010 device (800 LUTs), the average distance between Simple resumption points is 6,640 words (ca. 25% of the bitstream) and the biggest is 11,744 words (about 44% of the bitstream, i.e., the worst case that needs to be considered in the analysis in Section 8.2). Therefore, a more fine-granular distribution of resumption points in the bitstream is beneficial.

6.2 Per-Frame Resumption Points Within the Slot Data Section

The Slot Data Section is by far the biggest section of a partial bitstream (multiple tens of thousands of words). Therefore, we focus on finding Per-Frame resumption points in that section first (without considering the data chunk within the Common Header). In standard bitstreams, the FAR is initialized in the Write Initialization sequence and then incremented automatically by the reconfiguration port FSM after each frame, while writing the data chunk. Searching for Per-Frame resumption points requires knowing the FAR value increment after each frame, to determine the FAR value (FPGA-internal configuration memory address) that corresponds to a certain resumption point (offset within the bitstream). This information can be taken from debug bitstreams: inside the Slot Data Section (but not in the data chunk within the Common Header), a FAR value increment of 0x01 (*frame increment*) is applied after each frame.

Figure 7 shows how to find Per-Frame resumption points. The first step is to find the beginning of each data chunk (FDRI-write operation inside the Write Initialization sequence) and its starting FAR value. Then, a resumption point can be found at every frame of configuration data, associating its offset with the calculated FAR value and an FDRI value. The correct FAR value for Per-Frame

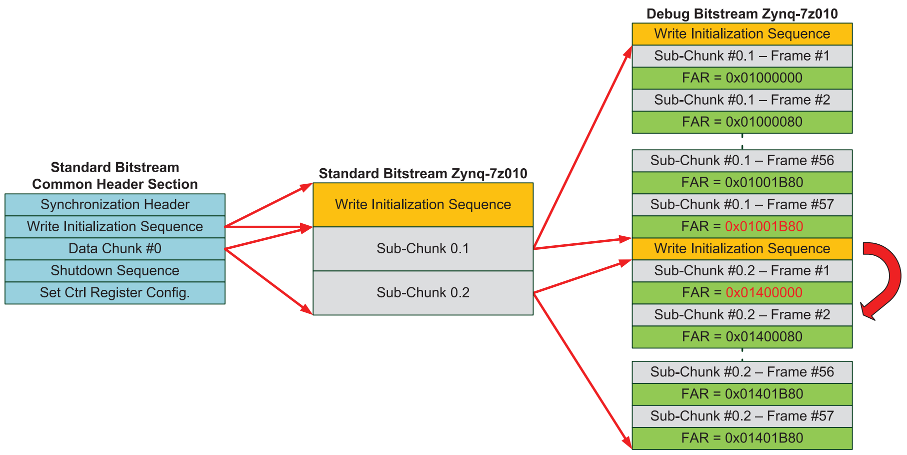

Standard Bitstream Common Header Section
Synchronization Header
Write Initialization Sequence
Data Chunk #0
Shutdown Sequence
Set Ctrl Register Config.
Standard Bitstream Zynq-7z010
Write Initialization Sequence
Sub-Chunk 0.1
Sub-Chunk 0.2
Debug Bitstream Zynq-7z010
Write Initialization Sequence
Sub-Chunk #0.1 – Frame #1
FAR = 0x01000000
Sub-Chunk #0.1 – Frame #2
FAR = 0x01000080
Sub-Chunk #0.1 – Frame #56
FAR = 0x01001B80
Sub-Chunk #0.1 – Frame #57
FAR = 0x01001B80
Write Initialization Sequence
Sub-Chunk #0.2 – Frame #1
FAR = 0x01400000
Sub-Chunk #0.2 – Frame #2
FAR = 0x01400080
Sub-Chunk #0.2 – Frame #56
FAR = 0x01401B80
Sub-Chunk #0.2 – Frame #57
FAR = 0x01401B80

Fig. 8. Detailed structure of Data Chunk #0. The FAR increment is 0x80 (column increment), and each sub-chunk has its starting and ending FAR value. Highlighted in red the discontinuity of the FAR value between two different sub-chunks.

resumption points within the Slot Data section is calculated by adding a frame increment for each frame to the FAR starting value. The FDRI value is fixed to the number of words in a frame (101).

Using Per-Frame resumption points in addition to Simple resumption points can reduce the distance between resumption points drastically. However, the worst-case distance between resumption points (that needs to be considered for real-time analyses in Section 8.2) is now determined by Data Chunk #0 within the Common Header. This data chunk requires additional steps to determine the FAR increments, as detailed in the following. As mentioned in Section 5, the size of Data Chunk #0 is device dependent and grows proportionally with the size of the FPGA.

6.3 Per-Frame Resumption Points Within the Common Header

Searching for Per-Frame resumption points inside Data Chunk #0 within the Common Header requires the knowledge of its internal device-dependent structure. In contrast to all other data chunks, Data Chunk #0 is divided into smaller *sub-chunks* and each sub-chunk has starting and ending FAR values that are not observable in a standard bitstream. The number of sub-chunks and their starting and ending FAR values depend on the targeted FPGA only, but not on the resources and dimensions of the reconfigurable slot that should be configured.

The structure, FAR values, and FAR increments of Data Chunk #0 can be obtained by analyzing a debug bitstream for the targeted device. As discussed in Section 5, the debug bitstream loads each frame individually, writing the FAR and the FDRI value after each frame. Therefore, the internal partitioning of Data Chunk #0 into several sub-chunks can be inferred by inspecting the FAR values in the debug bitstream: a discontinuity in FAR values identifies the beginning of a new sub-chunk. Within the same sub-chunk, FAR values are contiguous. The first FAR write identifies the beginning of the first sub-chunk and every FAR discontinuity provides the ending value of the current sub-chunk and the starting value of the next sub-chunk. Figure 8 shows the structure of Data Chunk #0 for a partial bitstream targeting the Xilinx Zynq-7010 FPGA.

The process for finding Per-Frame resumption points in Data Chunk #0 is similar to finding Per-Frame resumption points in the Slot Data section (see Section 6.2 and Figure 7). However, the FAR increment after each frame is a *column increment* (0x80) and not a single-frame increment

(0x01). The information about the end of each sub-chunk and the correct FAR values are read from the debug bitstream.

By introducing Per-Frame resumption points in Data Chunk #0, the maximum gap between two resumption points can be reduced to a single frame (101 words), and hence at most the configuration progress of a single frame is lost when a reconfiguration is preempted.

7 PREEMPTIVE RECONFIGURATION

In the following, the software interface, the custom reconfiguration controller, and the way they interact to realize preemptive reconfiguration under real-time constraints by utilizing resumption points (as detailed in the previous section) are described.

7.1 Software Interface for Preemptive Reconfiguration

Each task that wants to reconfigure a reconfigurable slot sends a request to the reconfiguration driver, which provides a unified software interface for runtime reconfiguration for all tasks. Then, the driver handles the reconfiguration request by sending *commands* to the *reconfiguration controller* that translates these commands into signals for the reconfiguration port, and it initiates the transfer of configuration data from main memory (or controller-internal SRAM) to the reconfiguration port. The purpose of the reconfiguration controller is to alleviate the CPU from managing the reconfiguration port and keeping track of ongoing reconfiguration requests. A reconfiguration request to the driver specifies the requesting task's priority, ID, and bitstream's memory address. The task priority is used by the driver to determine whether the request has priority over a running reconfiguration (if any). The driver compares the task priority with the priority of the current reconfiguration (stored in a hardware register of the reconfiguration controller) to decide whether to preempt or not. If the requesting task has the same priority as the current reconfiguration, then the driver sends the reconfiguration request only if the requested slot is currently unoccupied. In case the requested slot is busy, the driver returns an error to the user task, which can decide to proceed in software or ask again for hardware acceleration. The task ID from the reconfiguration request is used to resume the respective task when the reconfiguration has finished. As soon as a task sends a reconfiguration request, it self-suspends. Additional information can be provided to execute the reconfiguration request. For example, the reconfiguration controller can be configured to send an interrupt to the CPU in case of a reconfiguration error.

Tasks are unaware of reconfiguration requests from other tasks. The driver translates reconfiguration requests into commands to the reconfiguration controller, while possibly preempting and resuming reconfigurations such that the reconfiguration request with the highest priority is being processed at each point in time. This leads to the following cases that the driver has to handle:

(i) *No ongoing reconfiguration*: In case no reconfiguration is currently being performed, the driver translates the reconfiguration request into commands that are sent to the reconfiguration controller and then waits for a new request.

(ii) *Ongoing reconfiguration*: In case there is a reconfiguration currently being performed, the driver compares the priority of the running reconfiguration request and the new reconfiguration request to determine whether to abort the current reconfiguration to start processing the new request, or to enqueue the new request after the running reconfiguration request. When a task of higher priority requests a reconfiguration while a task of lower priority currently occupies the reconfiguration port, the lower-priority reconfiguration is preempted. To preempt a reconfiguration, the driver aborts the current reconfiguration and determines how far the lower-priority reconfiguration has proceeded (as an offset in the bitstream). After that, the resumption point for the lower-priority reconfiguration

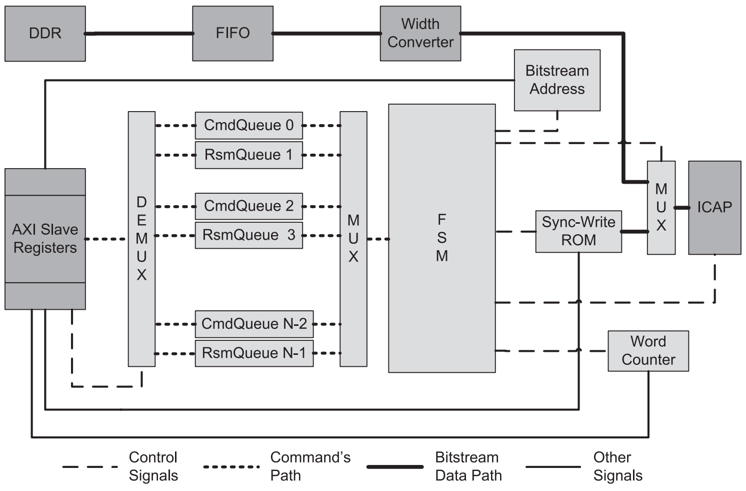


Fig. 9. Detailed block diagram of the custom reconfiguration controller.

request that is closest to the determined offset (and that was already passed during reconfiguration) is searched in a list of all resumption points for the respective bitstream. Then, the higher-priority request is enqueued. Afterward, depending on the resumption point, the correct synchronization (see Section 6) and the lower-priority reconfiguration are enqueued to proceed from the resumption point. Once the higher-priority reconfiguration has finished, the reconfiguration controller automatically resumes the lower-priority reconfiguration (assuming no other higher-priority task requests a reconfiguration), as we will detail in the following section.

7.2 Preemptive Reconfiguration Controller

The reconfiguration driver handles the tasks' reconfiguration requests by translating them into *commands* that are sent to the custom reconfiguration controller over the system bus (ARM AMBA AXI on the Xilinx Zynq devices). The reconfiguration controller processes commands internally using an FSM, based on the command-based reconfiguration queue (CoRQ) [7] that guarantees a fixed latency for each command (but main memory transfers), and was extended by commands that enable the reconfiguration preemption. The goal of the designed reconfiguration controller is to provide a high-level interface to perform preemptive reconfigurations for tasks with different priorities.

Figure 9 shows an overview of the reconfiguration controller. It communicates to the system using AXI interfaces: one master interface is used by the controller to fetch bitstream data from main memory and one slave interface is used to connect the controller to an AXI bus from which the controller can accept commands (e.g., sent by the ARM CPU on Zynq devices). In particular, the AXI slave interface allows to control the input demux to write into a single queue, while the internal logic controls the output mux to let the FSM process commands from the highest priority, non-empty queue. It integrates the ICAP as the reconfiguration port on Xilinx devices.

In addition to the FSM, the reconfiguration controller provides a configurable number of FIFO queues. For each task priority supported by the system, two queues are instantiated: (i) a *command queue* that accepts any FSM commands from tasks of the respective priority and (ii) a *resume queue* that only stores FSM commands to resume reconfigurations from resumption points

Table 1. Reconfiguration Controller Commands

Command	**Im**mediate/**Qu**eueable	Latency
`configureBitstreamExt`	Qu	—
`configureBitstreamInt`	Qu	$9 + \lceil B/4 \rceil$
`setBaseAddress`	Qu	3
`abortReconfiguration`	Im	7
`setFAR`	Qu	3
`setFDRI`	Qu	3
`sendSyncronization`	Qu	470
`resumeFSM`	Im	3

B, size of bitstream [byte].

(see Section 6). The priority of the queues is strictly ordered: first by the task priorities, and within a single task priority the resume queue has a higher priority than the command queue (i.e., there are two queue priorities for each task priority). In other words, preempted reconfigurations have a higher priority than the following reconfiguration requests from tasks of the same task priority. The reconfiguration controller is able to manage up to 8 priority levels (parameterized at design time), with each queue storing up 128 commands.

The FSM is the core of the reconfiguration controller and it fetches, decodes, and executes commands from the highest-priority non-empty queue (either command or resume queue). Based on the commands from the queue, the FSM controls the FPGA reconfiguration port. Commands are either executed immediately or enqueued into one of the internal FIFO queues (denoted as *immediate* or *queueable* commands). Immediate commands are used to control the FSM itself (e.g., pause/resume processing enqueued commands) and abort a running reconfiguration. Queueable commands relieve the CPU from managing reconfigurations—for instance, they configure bitstreams (from internal or external memory) and notify the CPU once reconfiguration have finished. The commands that are utilized for preemptive reconfiguration are listed in the following and Table 1 shows their latencies (in clock cycles).

- `setBaseAddress`: Commands can have, at most, 27-bit for data, but some of them require 32-bit addresses, such as, `configureBitstream`, `setFAR`, and `setFDRI` (detailed below). `setBaseAddress` is used to set an offset that is combined with the address bits supplied by other commands to obtain 32-bit addresses.
- `configureBitstreamInt`/`configureBitstreamExt`: Starts transfer of the bitstream to the reconfiguration port from the memory address specified by 20 bits inside the command combined with 12 bits set prior via the `setBaseAddress` command. It is possible to decide the storage source of the bitstream: it can be fetched from the controller-internal memory, which guarantees a fixed bandwidth, or from the main memory.
- `abortReconfiguration`: This command will abort the ongoing reconfiguration, stopping the data transfer and resetting the reconfiguration port. A flag allows to choose whether processing of further commands should be paused afterward or not.
- `setFAR`: Updates an operation sequence with a new FAR value that reflects a previously selected resumption point. The sequence to update is stored in the reconfiguration controller and sent to the reconfiguration port to resume a preempted reconfiguration (Synchronization Header and a Write Initialization sequence, see Section 5). This command must be preceded by `setBaseAddress`.

- `setFDRI`: Similar to `setFAR`, this command updates the operation sequence that is sent to the reconfiguration port to resume a preempted reconfiguration with a new FDRI value. This command must be preceded by `setBaseAddress`.
- `sendSyncronization`: This command sends the operation sequence to resume a preempted reconfiguration. A flag is used to determine whether the whole sequence should be sent (for resuming Per-Frame resumption points, see Section 6) or the Synchronization Header only (for resuming Simple resumption points). Before issuing this command, FAR and FDRI must have been set using `setFAR` and `setFDRI`.
- `resumeFSM`: If the FSM was paused (e.g., by `abortReconfiguration`), this command resumes it.

The reconfiguration controller records the state of an ongoing reconfiguration. *Bitstream Address* and *Word Counter* are registers that are used to store the base address of the currently reconfigured bitstream and determine the offset of the currently transferred bitstream word. These two values are used during preemption to determine the resumption point. In particular, the bitstream base address is used as a bitstream ID, to target the right set of resumption points, whereas the Word Counter value is used as an offset and compared with all pre-calculated resumption offsets to find the closest accomplished resumption point (details are provided in Section 7.3).

As explained in Section 4, resuming a reconfiguration means to resume the transfer of configuration data to the configuration memory of the FPGA, it does not mean any modifications on the bitstream. Therefore, each time a reconfiguration should be resumed from a specific resumption point, the reconfiguration port needs to be synchronized first using the Synchronization Header (and possibly the Write Initialization). The reconfiguration controller includes a *Sync-Write ROM*, a memory that contains the Synchronization Header and Write Initialization sequences. Two words inside the Write Initialization sequence are programmable: the FAR and FDRI values that define a specific resumption point (set using `setFAR` and `setFDRI`). Depending on the type of resumption point, only the Synchronization Header (Simple resumption points) or both Synchronization Header and Write Initialization sequences (Per-Frame resumption points) are sent to the reconfiguration interface.

7.3 Hardware/Software Integration

To trigger a reconfiguration, the reconfiguration driver enqueues a `setBaseAddress` and a `configureBitstream` command into the command queue that is determined by the task priority. As soon as one of the queues signals that it is non-empty, the reconfiguration controller FSM starts fetching and executing its commands.

To preempt an ongoing reconfiguration, the reconfiguration driver sends an (immediate) `abortReconfiguration` command and reads the state of the aborted reconfiguration to find the correct resumption point. In particular, Word Counter and Bitstream Address values are read from the hardware controller. Since the application could have multiple reconfigurable slots and partial bitstreams, a method to address the respective set of resumption points and the specific resumption point of the preempted reconfiguration is required. Figure 10 shows the data structure used to select the correct resumption point. The address from which the bitstream was fetched in the preempted reconfiguration (pointing to main memory or controller-internal memory) is used as an identification method to target the set of resumption points related to the preempted reconfiguration. Moreover, the Word Counter value is used as an offset and compared with all resumption offsets within each resumption point, to find the nearest and already passed one. Finally, resuming a reconfiguration requires different commands, depending on the type of resumption point (Figure 11):

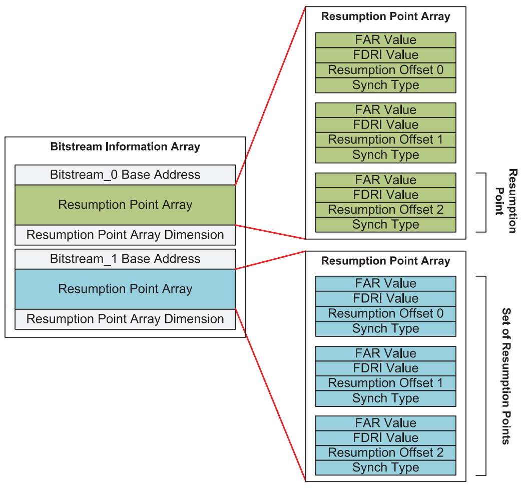


Fig. 10. Data structure for two bitstreams with three resumption points each. The *Bitstream Information Array* is indexed using the Bitstream Address value read from the hardware, whereas the Word Counter value is used to identify the correct resumption point within the *Resumption Point Array*. The *Synch Type* field specifies the type of resumption point.

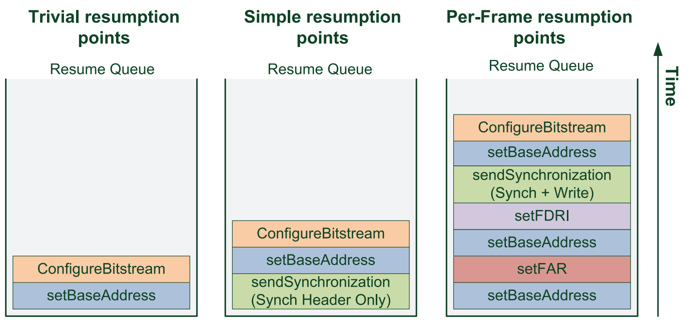


Fig. 11. Command sequence to resume a reconfiguration from Trivial, Simple, and Per-Frame resumption points.

- *Trivial resumption points*: In this case, the reconfiguration driver enqueues a `setBaseAddress` and a `configureBitstream` command. There is no need for the reconfiguration controller to send a Synchronization Header sequence, as the reconfiguration simply resumes from the beginning of the bitstream.
- *Simple resumption points*: The reconfiguration driver enqueues a `sendSyncronization` command specifying to send a Synchronization Header sequence only, as this type of resumption points already contain a Write Initialization sequence. Then, a `setBaseAddress` and a `configureBitstream` command are enqueued to resume the reconfiguration.
- *Per-Frame resumption points*: In addition to the Synchronization Header sequence, this type of resumption point needs a Write Initialization sequence. The reconfiguration driver enqueues a `setFAR` and a `setFDRI` command (both have to be preceded by a `setBaseAddress`) to set the FAR and FDRI value taken from the selected resumption point. Then a `sendSyncronization` command (that specifies to send both Synchronization

Header and Write Initialization sequence), a `setBaseAddress` and a `configureBitstream` command are enqueued.

This section completes the description of how preemptive reconfiguration was realized. In the following section, we analyze this property for real-time constraints.

8 WORST-CASE LATENCY ANALYSIS OF PREEMPTIVE RECONFIGURATION

The realization of preemptive reconfiguration is based on CoRQ [7], a reconfiguration controller that guarantees worst-case latency bounds on the commands it processes, even on the reconfiguration itself when a memory bandwidth is guaranteed (e.g., when using FPGA-internal SRAM). Commands that have been implemented for preempting and resuming reconfigurations have guaranteed latencies, as listed in Table 1.

This allows us to apply response time analysis techniques that are standard to determine the finish time of real-time tasks [3] to determine the finish time of preemptive reconfigurations under real-time constraints. The focus of this section is to determine worst-case bounds on the latency overhead that a higher-priority task experiences, when preempting a lower-priority task (WCET_{hp}). Furthermore, we determine an upper bound on the reconfiguration delay for lower-priority reconfiguration requests for a given worst-case interval of reconfiguration preemptions (rdelay_{lp}), and discuss under which circumstances preemption enables us to guarantee a minimum progress for these reconfigurations.

8.1 Overhead for Preempting an Ongoing Reconfiguration

When a higher-priority task sends a reconfiguration request to the reconfiguration driver, the driver needs to preempt the ongoing reconfiguration (if any, as detailed in Section 7.3). Preemption entails operations performed by the driver itself (in software), as well as commands that are sent to the reconfiguration controller. The first command that is sent to the reconfiguration controller aborts the ongoing reconfiguration (while keeping information about its progress) and suspends processing of further commands (enqueued commands are kept in the queues). This way, operations performed by the driver and commands sent to the reconfiguration controller to perform the preemption are executed in sequence and therefore analyzed separately.

After `abortReconfiguration`, the driver determines the queue containing the commands that were being processed before the abort (the highest-priority non-empty queue) and then reads the last state of the aborted reconfiguration. Based on this state, the correct resumption point for the preempted reconfiguration is determined using binary search on the table of resumption points available for the respective bitstream. The corresponding commands to resume from this resumption point are then enqueued into the resume queue for the lower-priority task in the reconfiguration controller (see Section 7.3). Afterward, the `configureBitstream` command for the higher-priority task is enqueued into its standard queue. We know that it was empty before, otherwise it would not be possible that we are preempting the reconfiguration of a lower-priority task. Therefore, the reconfiguration request of the higher-priority task is immediately processed, when the driver finally resumes processing of commands by the reconfiguration controller FSM using `resumeFSM`. We implemented the reconfiguration driver as a bare metal binary (without task handling, a full-featured implementation based on FreeRTOS is evaluated in Section 9) to get an estimate on the worst-case execution time (WCET) of the driver's operations when preempting an ongoing reconfiguration. The WCET estimate is obtained using the commercial WCET analyzer AbsInt aiT,[3] based on an ARM Cortex R5F real-time CPU running at 600MHz that is available in

[3]https://www.absint.com/.

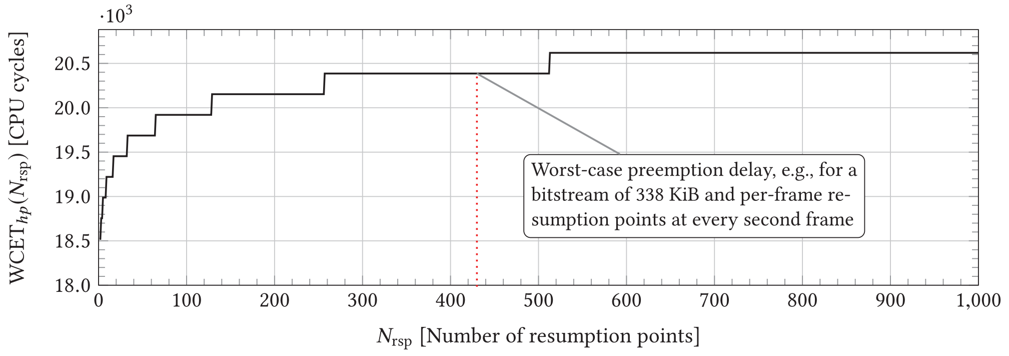


Fig. 12. WCET for preempting a reconfiguration ($\mathrm{WCET}_{hp}(N_{\mathrm{rsp}})$), depending on the number of resumption points (N_{rsp}) available in the bitstream that is preempted. For example, a bitstream size of 338KiB has been reported for several image filter accelerators (each) on the Zynq-7010 in Biondi et al. [2], resulting in $N_{\mathrm{rsp}} \approx 430$ with Per-Frame resumption points at every second frame.

the recent Xilinx Zynq UltraScale+ generation. While the presented approach is not tied to a specific CPU, it is not possible to apply WCET analyzers to the ARM Cortex A9 that is available on the Xilinx Zynq-7000, because of its out-of-order pipeline. Equation (1) shows the obtained result for the operations that the driver performs in software in WCET cycles (of the CPU):

$$\mathrm{WCET}_{hp\_driver}(N_{\mathrm{rsp}}) = 18229 + \lfloor \log_2(N_{\mathrm{rsp}}) + 1 \rfloor \cdot 233. \tag{1}$$

Due to the binary search of the resumption point, the WCET depends on N_{rsp} (the number of resumption points that were determined statically), resulting in a "parametric" WCET bound [16].

In the following, we analyze the worst-case latency of the commands processed by the reconfiguration controller to perform the preemption (as sent by the reconfiguration driver). These are: `abortReconfiguration` (suspending processing further commands) and `resumeFSM` (after enqueueing the higher-priority reconfiguration). Processing these commands takes $t_{\texttt{abortReconfiguration}} = 7$ and $t_{\texttt{resumeFSM}} = 3$ cycles respectively on the reconfigurable fabric [7]. Furthermore, the frequency factor between CPU and reconfigurable fabric c_{freq} needs to be accounted for—for instance, the CPU runs at a frequency c_{freq} times higher than the reconfiguration controller. Therefore, the WCET for processing the commands for preemption in CPU cycles is

$$\mathrm{WCET}_{hp\_\mathrm{cmds}} = c_{\mathrm{freq}} \cdot (t_{\texttt{abortReconfiguration}} + t_{\texttt{resumeFSM}}) = c_{\mathrm{freq}} \cdot 10. \tag{2}$$

In the presented case, the reconfiguration controller runs at 100MHz, while the ARM Cortex R5F runs at 600MHz on the Xilinx Zynq UltraScale+ (i.e., $c_{\mathrm{freq}} = 6$). In total, the following WCET in CPU cycles that the higher-priority task experiences as an overhead for its reconfiguration requests, in case it needs to preempt an ongoing reconfiguration from a lower-priority task, is obtained:

$$\mathrm{WCET}_{hp}(N_{\mathrm{rsp}}) = \mathrm{WCET}_{hp\_\mathrm{driver}}(N_{\mathrm{rsp}}) + \mathrm{WCET}_{hp\_\mathrm{cmds}} = 18289 + \lfloor \log_2(N_{\mathrm{rsp}}) + 1 \rfloor \cdot 233. \tag{3}$$

This is the worst-case overhead that needs to be considered for reconfiguration requests in response-time analyzes like presented in Biondi et al. [2], when reconfiguration preemption should be allowed. Figure 12 shows how the overhead grows, depending on the number of resumption points available in the bitstream that is preempted. Note that the overhead is *guaranteed* to remain below 21,100 cycles when providing resumption points for every second frame (every 202 words) for a bitstream that is smaller than 2.5MiB (approximately the size of a full bitstream for the Xilinx Zynq-7010). This overhead corresponds to ca. 8% of the *observed* worst-case scheduling overhead in an ARM-based Linux system with real-time extensions [4]. Even when reconfiguring

25% of the total resources of the Zynq-7010 at maximum reconfiguration bandwidth, the designed preemption approach would increase the delay by only 1.75% in the worst case.

8.2 Reconfiguration Delay of Preempted Reconfigurations

This section focuses on providing an upper bound for the reconfiguration delay of lower-priority reconfiguration requests under reconfiguration preemptions (rdelay_{lp}). Tasks in the lower-priority level encounter preemptions from higher-priority levels only; therefore, we reason about preemptions encountered by a single lower-priority task in the following. Using `configureBitstreamInt`, the reconfiguration controller sets up a memory transfer from FPGA-internal SRAM to the reconfiguration port (taking exactly nine cycles), and then transfers one word (4 bytes) of bitstream in each cycle (full utilization of the reconfiguration port). Reconfiguration controller, reconfiguration port, and the FPGA-internal SRAM can be clocked at up to 100MHz, resulting in a maximum reconfiguration bandwidth of 400MB/s.[4] Given a bitstream of size B in bytes, a reconfiguration without any preemptions takes exactly $\text{rdelay}(B) = 9 + \lceil B/4 \rceil$ cycles on the reconfigurable fabric using the designed reconfiguration controller (details are provided in Damschen et al. [7]).

To determine the worst-case reconfiguration delay under preemptions, we need to consider the number of preemptions that can occur during the reconfiguration. Each preemption creates overhead that prolongs the reconfiguration delay, creating the possibility for additional preemptions. To bound this effect, we apply an iterative process from response time analysis [3] as follows. Assuming the minimum distance between preemptions (due to reconfiguration requests from higher-priority tasks) is t_{preempt}, we obtain the following recursive sequence:

$$\text{rdelay}_{lp}^{n+1}(B) = \text{rdelay}(B) + \underbrace{\left\lceil \frac{\text{rdelay}_{lp}^{n}(B)}{t_{\text{preempt}}} \right\rceil}_{\text{number of preemptions}} \cdot \underbrace{(t_{\text{rsp}} + t_{\text{sync}})}_{\text{overhead of being preempted}} . \tag{4}$$

The process starts with $\text{rdelay}_{lp}^{0}(B) = \text{rdelay}(B)$ and terminates when $\text{rdelay}_{lp}^{n+1}(B) = \text{rdelay}_{lp}^{n}(B)$. t_{rsp} is the maximum time that it takes to configure the difference between two resumption points (the maximum progress that is lost when a reconfiguration is preempted). For example, $t_{\text{rsp}} = 101$ cycles using Per-Frame resumption points (see Section 6.3). $t_{\text{sync}} = 470$ cycles is the latency of the `sendSyncronization` command when processed by the reconfiguration controller. Then, we obtain the reconfiguration delay of a reconfiguration request from a lower-priority task under reconfiguration preemptions as

$$\text{rdelay}_{lp}(B) = \text{rdelay}_{lp}^{n}(B) + \left\lceil \frac{\text{rdelay}_{lp}^{n}(B)}{t_{\text{preempt}}} \right\rceil \cdot \text{waiting}_{hp}, \tag{5}$$

where waiting_{hp} is the maximum time that the reconfiguration port is occupied by higher-priority tasks during a preemption. Note that t_{preempt} and waiting_{hp} need to be bounds for tasks from all higher-priority levels (than the priority level of the task that is currently preempted). For instance, for more than two priority levels, this simple analysis will be imprecise. A more precise analysis would need to be integrated into the response-time analysis of the whole task set and is out of the scope of this article, as we focus on the reconfiguration itself. In the following, we will show that a minimum progress for the lower-priority task can be guaranteed.

[4]For instance, $(4 \cdot 1024^{-2})/10^{-8} = 381.4697265625\text{MiB/s}$.

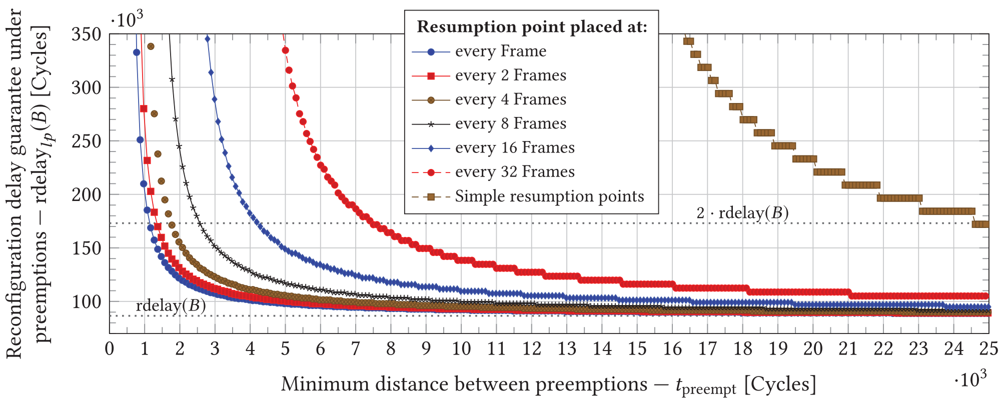


Fig. 13. Reconfiguration delay guarantee (rdelay_{lp}(338KiB), with $\text{waiting}_{hp} = 0$) for different time windows of unpreempted reconfiguration and different resumption point types (resulting in different latency overheads of being preempted due to lost progress in reconfiguration (t_{rsp})).

8.2.1 Guaranteeing Minimum Reconfiguration Progress Under Preemptions. Intuitively, a reconfiguration (that is subject to preemptions) advances, when the time windows of unpreempted reconfiguration are bigger than the latency overhead of being preempted. More formally, we show that minimum reconfiguration progress can be guaranteed when $t_{\text{preempt}} > t_{\text{rsp}} + t_{\text{sync}}$ (see Section 8.2). Minimum progress during reconfiguration is equivalent to a bound on the reconfiguration delay (see Equation (5)). Assuming waiting_{hp} is bounded, we therefore need to show that the iterative process of Equation (4) converges.

Proposition 1. $t_{\text{preempt}} > t_{\text{rsp}} + t_{\text{sync}} \Rightarrow \exists n \in \mathbb{N} : rdelay_{lp}^{n+1}(B) = rdelay_{lp}^{n}(B)$.

Proof. By induction, showing that $\text{rdelay}_{lp}^{n+1}(B) \leq (t_{\text{rsp}} + t_{\text{sync}} + 1) \cdot \text{rdelay}(B)$ and that the sequence is monotonically increasing. □

Convergence of Equation (4) immediately follows from Proposition 1. Therefore, we can guarantee a minimum progress for a reconfiguration that is subject to preemptions, when the time windows of unpreempted reconfiguration (t_{preempt}) are bigger than the latency overhead of being preempted ($t_{\text{rsp}} + t_{\text{sync}}$).

Figure 13 shows how $\text{rdelay}_{lp}(B)$ evolves for different values of t_{preempt} for a bitstream of B = 338KiB and different granularities of placing resumption points. waiting_{hp} is set to 0, because the bound of how long a higher-priority task occupies the reconfiguration port has no impact on convergence of Equation (4), and to focus on the overhead of preemption. It is observable that t_{preempt} as well as the granularity at which resumption points are provided for a certain bitstream have a considerable impact on whether the reconfiguration delay can be bounded as well as on the size of the obtainable bounds. Note that t_{preempt} is measured in cycles of the reconfiguration controller (running at 100MHz in the presented case). When using Simple resumption points only, higher-priority tasks can preempt the lower-priority reconfiguration at most every 73,290 CPU cycles ($t_{\text{preempt}} = 12,215$ cycles of the reconfiguration controller) on a 600MHz CPU so that a minimum progress can be guaranteed. This bound is more than one magnitude lower for Per-Frame resumption points. Combining the results shown in Figure 12 and Figure 13, it can further be seen that there is a trade-off between the minimum guaranteed bandwidth for reconfigurations of the lower-priority task and the WCET bound on performing preemptions for the higher-priority tasks. Reasonable trade-offs can be determined for specific use cases by choosing a suitable granularity

Table 2. FPGA Resource Utilization. (a) Resource Utilization of the Whole Hardware Design Consists of two Reconfigurable Slots and the Designed Reconfiguration Controller that are Connected to the ARM Cores on the Xilinx Zynq. (b) Resource Utilization of the Reconfiguration Controller that Enables Preemptive Reconfiguration Supports Eight Priority Levels and up to 128 Pending Commands for Each Level

Resource	Utilized	Available	% Utilized
LUT	7,252	17,600	41.20%
LUTRAM	729	6,000	12.15%
FF	8,449	35,200	24.00%
BRAM	12.50	60	20.83%
BUFG	10	32	31.25%

(a)

Resource	Utilized	Available	% Utilized
LUT	2,006	17,600	11.40%
LUTRAM	193	6,000	3.22%
FF	2,261	35,200	6.42%
BRAM	5	60	8.33%
BUFG	8	32	25.00%

(b)

Note: In both tables, percentages refer to a Zynq-7z010 chip.

of placing resumption points. This was achieved by introducing Per-Frame resumption points in Section 6.

9 EXPERIMENTAL EVALUATION

The experimental evaluation of this work has been done on a Xilinx Zynq-7z010, featuring an ARM Cortex-A9 dual-core running at 600MHz and an Artix-7 FPGA on the same SoC. The real-time operating system FreeRTOS[5] was extended to support preemptive reconfiguration: a reconfiguration driver was added for the designed custom reconfiguration controller and the task scheduler is aware of pending reconfiguration requests to wake up the respective tasks for completed reconfigurations. We utilize two reconfigurable slots with different dimension and incorporate different type of resources: the bigger slot has 2,400 LUTs, 10 BlockRAM, and 20 DSPs, and its bitstream's size is 364KB and has a reconfiguration time of 932.35μs; the smaller has 800 LUTs, 10 BlockRAM, and no DSPs, and its bitstream's size is 103KB and has a reconfiguration time of 265.59μs. In this evaluation, the reconfiguration controller contains 8 command and resume queues each, one (pair) for each FreeRTOS priority level (see Section 7.2). Each queue can store up to 128 pending commands. The reconfiguration controller's interrupt line and interrupt lines for each reconfigurable slot have been connected to the Zynq system.

The test application consists of two periodic *user tasks* and one *driver task* that handles all reconfiguration requests. Each user task has its own private reconfigurable slot that reconfigures multiple time during its execution. After a reconfiguration request, the user tasks self-suspend. The driver task is the highest priority task in the system and the only task allowed to communicate with the reconfiguration controller. It executes immediately after a reconfiguration request and checks whether the reconfiguration controller is idle or a reconfiguration is ongoing. If a lower-priority reconfiguration is ongoing, the driver handles the whole process of preempting the reconfiguration (see Section 7.3). Every time a reconfiguration is completed, the reconfiguration controller triggers an interrupt and the interrupt handler resumes the user task that requested the reconfiguration.

9.1 Experimental Results

9.1.1 Resource Utilization. Table 2(a) shows the resource utilization of the entire system, whereas Table 2(b) shows the resource utilization of the reconfiguration controller. Results in Table 2(a) do not account for any logic inside the reconfigurable slots (i.e., configured accelerators would add to the resource utilization).

[5]http://www.freertos.org/.

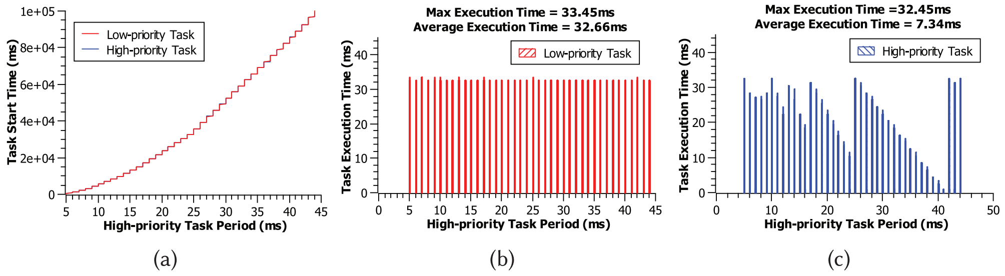


Fig. 14. Priority inversion experiment. The ongoing reconfiguration cannot be stopped and must be completed before starting another reconfiguration. (a) The start time of both tasks while the period of the high-priority task varies. (b, c) The distribution of the maximum observed execution time of user tasks for each value of the high-priority task period.

Utilization rates of the FPGA shown in Table 2(a) and (b) refer to a Zynq-7z010 device, which is the smallest Zynq-7000 device. For example, on the next-bigger Zynq-7z015, only 15.7% of LUTs and 13.9% of BRAMs would be utilized by the whole hardware design. On the biggest Zynq-7000, the Zynq-7z100 less than 3% of the resources would be utilized, respectively. Despite its constrained resources, it is showed that preemptive reconfiguration can be realized on the low-end Zynq-7z010.

9.1.2 Maximum Observed Execution Time. Three experiments were performed that reflect the motivational examples explained in Section 3 and show the benefits of preemptive reconfiguration. Due to the used FPGA device, both reconfigurable slots are quite small and have small reconfiguration time, so small that would have been difficult, with only two tasks in the system, to clearly show problems of non-preemptive and benefits of preemptive reconfiguration. For this reason, the reconfiguration time of each slot has been software extended, associating one reconfiguration request with multiple physical reconfiguration.

In all the experiments, the low-priority task has a fixed period of 50ms and a reconfiguration time of 32.63ms while the high-priority task's period is varied from 5ms to 44ms with a fixed reconfiguration time of 0.79ms.

Response time analysis in real-time systems needs to provide guarantees for the worst case—that is, that reconfiguration requests from the low-priority and high-priority tasks are in conflict and that the higher-priority task needs to preempt the lower-priority reconfiguration. To provoke this case, the tasks perform frequent reconfigurations, but no additional computations.

Figure 14 shows configure-to-completion (Section 3, item (a)), where the reconfiguration interface cannot be preempted and the ongoing reconfiguration has to be completed. In this scenario, both tasks can run concurrently, as Figure 14(a) shows, but the high-priority task is delayed due to priority inversion. As Figure 14(c) shows, the maximum execution time of the high-priority task that has been measured is 32.45ms.

Figure 15 shows the abort approach (Section 3, item (a)). This approach allows to abort the ongoing reconfiguration and restart it later from the beginning. Figure 15(a) shows the start time of both user tasks while the period of the high priority task varies. Both user tasks are periodic and the start time is the time where a task starts its routine from the beginning, and it is clearly visible that the low-priority task starves until the high-priority task's period reaches 34ms. Then, the period of the high-priority task is large enough to allow the low-priority one to reconfigure the FPGA.

Figure 16 shows the preemption approach—for instance, it is possible to preempt and later resume the ongoing reconfiguration (Section 3, item (c)) while keeping the reconfiguration progress. Both tasks benefit from preemptable reconfiguration, because the low-priority task does not suffer

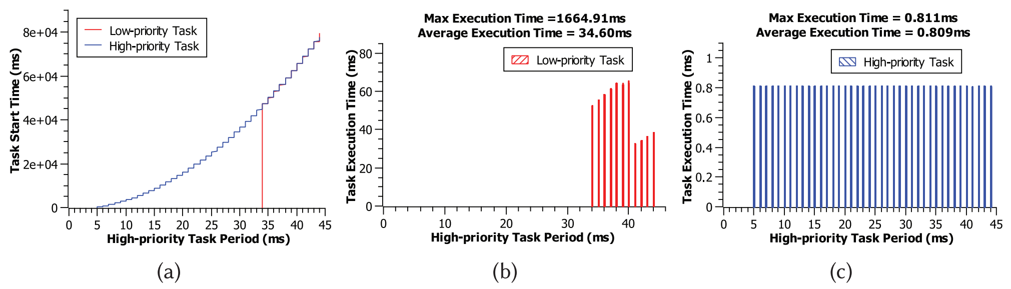


Fig. 15. Abort experiment. The ongoing reconfiguration can be aborted and later restarted from the beginning. (a) The start time of both tasks while the period of the high-priority task varies. (b, c) The distribution of the maximum observed execution time of user tasks for each value of the high-priority task period.

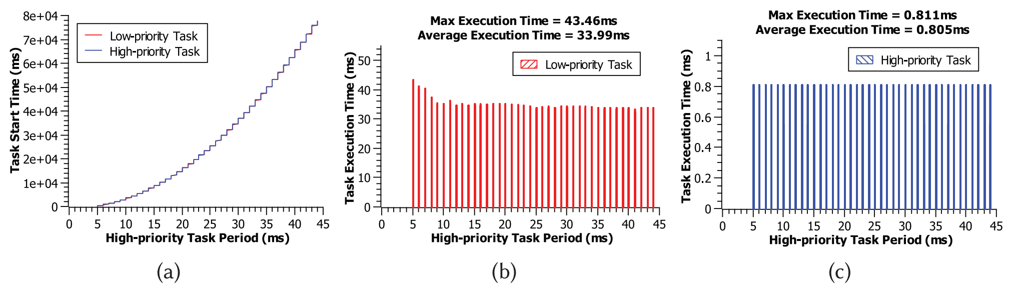


Fig. 16. Preemption experiment. The ongoing reconfiguration can be preempted to serve an higher priority reconfiguration request and later resumed. (a) The start time of both tasks while the period of the high-priority task varies. (b, c) The distribution of the maximum observed execution time of user tasks for each value of the high-priority task period.

from starvation while the high-priority task does not experience priority inversion. Therefore, as Figure 16(a) shows, both task can run concurrently on the system, reconfiguring the FPGA.

As all the experiments show, priority inversion increased the execution time of the high-priority tasks by more than 3× and starvation prevented the low-priority task to complete its execution. Preemptable reconfiguration solved these problems by avoiding priority inversion and preventing the low-priority task to starve, allowing it to improve the utilization of the system.

10 CONCLUSION

This article presented the first realization of preemptive reconfiguration and was evaluated on a Xilinx 7 series FPGA that uses the ICAP as the reconfiguration port. Our approach does not require any changes to the bitstreams that should be configured. Moreover, we have proven that our presented solution solves the problems of priority inversion and starvation caused by contention on the reconfiguration port under reasonable conditions (e.g., no adversarial high-priority tasks that constantly generate reconfiguration requests). Worst-case latency bounds were presented. To achieve preemptive reconfiguration, resumption points were defined (i.e., parts of a bitstream from which a preempted reconfiguration can safely be resumed). Furthermore, methods to determine all possible and safe resumption points were presented. Resumption points were leveraged by a combination of a custom command-based reconfiguration controller and a reconfiguration driver that manage reconfiguration requests and handle reconfiguration preemption and resumption.

Worst-case bounds on the latency overhead that a higher-priority task experiences when preempting a lower-priority task and upper bounds on the reconfiguration delay—experienced by the lower-priority task for a given worst-case interval of reconfiguration preemptions—have been determined analytically and revealed the trade-off between the minimum guaranteed bandwidth for reconfigurations of lower-priority tasks and the WCET bound on performing preemptions for higher-priority tasks. Experimental results verified that priority inversion and starvation caused by contentions on the reconfiguration port are effectively solved using the proposed realization of preemptive reconfiguration, when integrated into the industry-grade real-time operating system FreeRTOS. This was achieved even on a low-end (in terms of resources) reconfigurable device, the Xilinx Zynq-7z010. In total, it was shown that the overhead for achieving preemptive reconfiguration is considerably outweighed by the benefits it provides.

Preemptive resources are fundamental to fulfill real-time constraints and this work enables multi-priority real-time systems (including mixed-criticality systems) to benefit from runtime-reconfigurable hardware accelerators with preemptive reconfiguration capabilities.

REFERENCES

[1] Florian Benz, André Seffrin, and Sorin A. Huss. 2012. Bil: A tool-chain for bitstream reverse-engineering. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'12)*. 735–738.

[2] Alessandro Biondi, Alessio Balsini, Marco Pagani, Enrico Rossi, Mauro Marinoni, and Giorgio Buttazzo. 2016. A framework for supporting real-time applications on dynamic reconfigurable FPGAs. In *Proceedings of the Real-Time Systems Symposium (RTSS'16)*. 1–12.

[3] Alan Burns. 1993. *Preemptive Priority Based Scheduling: An Appropriate Engineering Approach*. Department of Computer Science, University of York.

[4] Gilles Chanteperdrix and Richard Cochran. 2009. The ARM fast context switch extension for Linux. In *Proceedings of the Real Time Linux Workshop*. 255–262.

[5] Marvin Damschen, Lars Bauer, and Jörg Henkel. 2016. Extending the WCET problem to optimize for runtime-reconfigurable processors. *ACM Trans. Architect. Code Optim.* 13, 4, Article 45, 24 pages. DOI: http://dx.doi.org/10.1145/3014059

[6] Marvin Damschen, Lars Bauer, and Jörg Henkel. 2016. Timing analysis of tasks on runtime reconfigurable processors. *IEEE Trans. Very Large Scale Integr. Syst.* 25, 1, 294–307. DOI: http://dx.doi.org/10.1109/TVLSI.2016.2572304

[7] Marvin Damschen, Lars Bauer, and Jörg Henkel. 2017. CoRQ: Enabling runtime reconfiguration under WCET guarantees for real-time systems. *IEEE Embed. Syst. Lett.* To appear.

[8] Klaus Danne and Marco Platzner. 2006. An EDF schedulability test for periodic tasks on reconfigurable hardware devices. *ACM SIGPLAN Notices* 41, 7, 93–102.

[9] Carlo Galuzzi and Koen Bertels. 2011. The instruction-set extension problem: A survey. *ACM Trans. Reconfigur. Technol. Syst.* 4, 2, Article 18, 28 pages.

[10] Markus Happe, Andreas Traber, and Ariane Keller. 2015. Preemptive hardware multitasking in ReconOS. In *Proceedings of the International Symposium on Applied Reconfigurable Computing (ARC'15)*. 79–90. DOI: http://dx.doi.org/10.1007/978-3-319-16214-0_7

[11] Rodolfo Pellizzoni and Marco Caccamo. 2007. Real-time management of hardware and software tasks for FPGA-based embedded systems. *IEEE Trans. Comput.* 56, 12, 1666–1680.

[12] Sangeet Saha, Arnab Sarkar, and Amlan Chakrabarti. 2015. Scheduling dynamic hard real-time task sets on fully and partially reconfigurable platforms. *IEEE Embed. Syst. Lett.* 7, 1, 23–26.

[13] Kentaro Sano, Dimitrios Soudris, Michael Hübner, and Pedro C. Diniz. 2015. In *Proceedings of the 11th International Symposium on Applied Reconfigurable Computing (ARC'15)*, Vol. 9040.

[14] Christoph Steiger, Herbert Walder, and Marco Platzner. 2004. Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks. *IEEE Trans. Comput.* 53, 11, 1393–1407.

[15] Russell Tessier, Kenneth Pocek, and Andre DeHon. 2015. Reconfigurable computing architectures. *Proc. IEEE* 103, 3, 332–354.

[16] Emilio Vivancos, Christopher Healy, Frank Mueller, and David Whalley. 2001. Parametric timing analysis. *ACM SIGPLAN Notices* 36, 8, 88–93.

[17] Xilinx Inc. 2016. *7 Series FPGAs Configuration User Guide*. UG470. Xilinx Inc.

Received September 2017; revised December 2017; accepted January 2018