This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2017.2714844, IEEE Embedded Systems Letters

CoRQ: Enabling Runtime Reconfiguration under WCET Guarantees for Real-Time Systems

Marvin Damschen, *Student Member, IEEE,* Lars Bauer, *Member, IEEE,* and Jörg Henkel, *Fellow, IEEE*

***Abstract*— Real-time systems have an increasing demand for *predictable* performance. Only recently novel models and analyses were proposed that make the performance benefits of runtime-reconfigurable architectures accessible for optimized worst-case execution time guarantees. However, the implicit assumption in these works is that the process of reconfiguration itself complies with execution time guarantees. The realization of a reconfiguration controller that fulfills these assumptions and that is amenable to worst-case execution time guarantees is so far unavailable. In this letter we detail the challenges of runtime reconfiguration in real-time systems and show that conflicts while accessing a shared main memory during reconfiguration can lead to a slowdown of more than $21\times$ in reconfiguration bandwidth. We present concepts that enable runtime reconfiguration under worst-case execution time guarantees and release our implementation of these concepts as an open-source project.**

***Index Terms*—Real-time systems, worst-case execution time (WCET), timing analysis, reconfigurable computing, accelerators.**

I. Introduction

Failing to meet a deadline can lead to severe malfunctions and in extreme cases even threaten life in safety-critical domains like automotive, avionics or the rapidly growing domain of smart/autonomous machines (e.g., robotics or automated driving). Therefore, the maximum execution time under any possible input, the so called *worst-case execution time* (WCET) [1], needs to be guaranteed for each task that should run on a real-time system.

Real-time systems have an increasing demand for *predictable* performance, e.g., automated driving requires vast amounts of sensor data to be processed under timing constraints. The demand cannot be fulfilled by modern system on chips (SoCs) that optimize for average-case performance, because they are no longer analyzable for WCET guarantees. Modern processors have increasingly complex microarchitectures with out-of-order scheduling pipelines, several hardware threads and multiple (shared) cache layers that render analysis for WCET guarantees virtually infeasible [2].

One way to achieve timing-analyzable performance is to design hardware accelerators that speed up the tasks' most compute-intensive parts, so called *computational kernels* (also known as hotspots) that are comprised of one or more nested loops [3, 4]. When implementing these accelerators as application-specific integrated circuits, the system would lack flexibility with respect to revised standards or new algorithms. Instead, using a runtime reconfigurable architecture maintains a high flexibility and even allows for reconfiguring the accelerators at runtime, thereby increasing the performance as well as the computing efficiency (compared to a static set of accelerators) at the cost of a more complex timing analysis.

The authors are with the Chair for Embedded Systems (CES), Karlsruhe Institute of Technology, Karlsruhe 76131, Germany (e-mail: marvin.damschen@kit.edu; lars.bauer@kit.edu; henkel@kit.edu).

The authors want to thank Artur Eckhart for his work on prototyping CoRQ. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR89).

While runtime reconfiguration was previously investigated with respect to real-time scheduling [5, 6], only recently novel models and analyses were proposed that make the benefits of runtime-reconfigurable architectures accessible for WCET guarantees in uniprocessor systems. In [3] it was shown that –in addition to a considerable speedup– the overestimation of a task's static WCET guarantee can be reduced by providing WCET guarantees for kernels in which calculations are one by one moved from software to hardware accelerators by runtime reconfiguration. Accelerators typically provide functionality that corresponds to several hundred instructions when executed on the CPU pipeline, possibly including conditional branches and other control flow. Analyzing instructions for worst-case latency introduces pessimism due to, e.g., pipeline hazards or instruction cache misses that need to be accounted for when the CPU behavior can not exactly be determined statically. The latency of the hardware accelerators that are executed on the reconfigurable fabric is under direct control of the application designer and often precisely known (e.g., by leveraging High Level Synthesis tools). In addition to benefits for WCET analysis of single tasks, research on real-time task scheduling showed that runtime reconfiguration is a powerful tool to raise the utilization of a real-time system that requires scheduling guarantees [7].

Existing work on runtime reconfiguration in the context of real-time systems implicitly *assumes* that the process of reconfiguration itself complies with timing guarantees [3–7], e.g., the time it takes to configure a hardware accelerator on the reconfigurable fabric (*reconfiguration delay*) is assumed constant and free from conflicts with other system components that could affect WCET guarantees. The realization of a runtime reconfiguration controller that fulfills these assumptions and that is amenable to WCET guarantees is so far unavailable. However, guaranteed reconfiguration delays are crucial to realize runtime-reconfigurable real-time systems.

Our novel contributions in this letter are:

- A runtime reconfiguration controller called "Command-based Reconfiguration Queue" (CoRQ) that provides guaranteed latencies for its operations and supports timing analysis for WCET guarantees. We release it as an open-source project, including examples and benchmarks[1].
- We show that conflicts while accessing a shared main memory during reconfiguration can lead to a slowdown of more than $21\times$ in reconfiguration bandwidth. In contrast, CoRQ guarantees constant reconfiguration delays even under heavy system bus load.

II. Runtime Reconfiguration under WCET Guarantees

A straight-forward approach of improving WCET guarantees of a kernel using runtime reconfiguration with the constraints of timing-analyzability and reasonable implementation effort is the *stalling* approach [3]. Software-only execution, i.e., without any accelerators, is shown in Fig. 1 (top). As shown in Fig. 1 (middle), a task that reconfigures an accelerator using stalling, stalls

[1] Available at: https://git.scc.kit.edu/CES/corq

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2017.2714844, IEEE Embedded Systems Letters

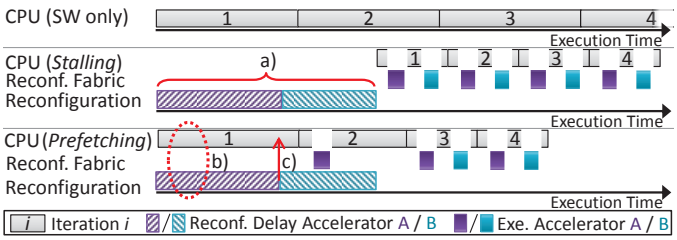

Fig. 1: Timelines of executing a Kernel using Software only, Stalling and Prefetching

its execution for the whole reconfiguration delay. At most one reconfiguration can be performed by the reconfiguration port at any time. Once all reconfigurations have completed (at the end of the reconfiguration delay, Fig. 1 a)), the task proceeds execution in software and executes the reconfigured hardware accelerators in every iteration of the kernel. A task that requests reconfiguration of accelerators using stalling can be analyzed for WCET guarantees using established timing analysis techniques by adding the reconfiguration delay (see Fig. 1 a)) to the WCET of the basic block that requests the reconfiguration. The assumption is that the reconfiguration delay can be determined statically, which is reasonable for the stalling approach because the task's memory accesses and reconfiguration cannot interfere on main memory or a shared system bus. However, stalling is not state-of-the-art in reconfigurable systems, because the CPU remains idle during reconfiguration.

An approach that enables the CPU to perform useful operations in parallel to reconfiguration is *prefetching*, i.e., accelerators are configured as early as possible in the control flow graph (CFG) so that a considerable amount of reconfiguration delay has already passed at the point in time when the accelerators are actually needed (see Fig. 1 (bottom)). Prefetching is an established technique in average-case optimizing reconfigurable systems, because it provides considerable performance improvements. For real-time systems, however, prefetching poses new challenges:

- As memory transfers can be initiated simultaneously by the reconfiguration of accelerators and by tasks running on the CPU (see Fig. 1 b)), it needs to be ensured that assumptions about memory access delays during static timing analysis of the guaranteed WCET bound capture potential conflicts on main memory or a shared system bus.
- Even when the reconfiguration delay of an accelerator would be a statically-known constant value, the worst-case state of the task's execution on the CPU is unclear: how far did the task proceed (in the worst-case) during the reconfiguration delay? In other words, from what point is it safe to assume during static timing analysis that, e.g., Accelerator A is readily configured on the reconfigurable fabric and available to be invoked (see Fig. 1 c), details in [3])?
- If program execution is faster or takes a different path than the worst-case path, a prefetch could become obsolete because the prefetched accelerator will not be executed anymore (see Fig. 2). In real-time systems, the possibility of an already occupied reconfiguration port can lead to delays that are hard to analyze and therefore introduce pessimism in the resulting WCET bound. Therefore, it is crucial to be able to abort reconfigurations such that one can guarantee for each prefetch that the reconfiguration port is unoccupied.

It might seem that the stalling approach is the favorable way to perform reconfiguration in real-time systems due to the potentially complex analysis of prefetching. However, stalling poses similar challenges for timing analysis when scheduling

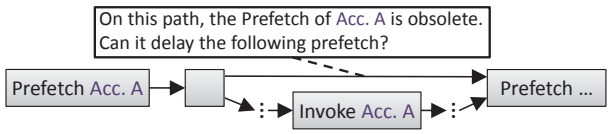

Fig. 2: Control-Flow Graph that shows how one Prefetch can delay another Prefetch, thus impairing Timing Analysis

TABLE I: CoRQ Commands with Cycles spent in EXE State

Command	**Im**mediate/**Qu**eueable	latency$_{EXE}$[1]
`clearQ`	Im, Qu	5
`stopQ`	Im, Qu	0
`resumeQ`	Im	0
`abortReconf`	Im	5
`setBaseAddr`	Qu	1
`configBitsExt`[2]	Qu	—
`configBitsInt`[2]	Qu	$6 + \lceil B/4 \rceil$
`stallCPU`	Qu	1
`unstallCPU`	Qu	1
`sendGPIO`	Qu	1
`sendIRQ`	Qu	1

[1] discussed in Section III-A
[2] detailed in Section III-B, B - size of bitstream [byte]

multiple real-time tasks, even on a uniprocessor system: when a task that requests a reconfiguration is stalled, another task could be executed in parallel to the reconfiguration delay (see Fig. 1 a) and [7]). Concerning the resulting WCET bound by analyzing either approach, it was shown in [3] that prefetching always provides a considerable speedup at runtime, but there are cases where additional WCET overestimation compared to stalling diminishes the speedup on the WCET guarantee. In the following section, we will introduce our reconfiguration controller *CoRQ*, and explain how it addresses the challenges that we observed and supports the stalling approach as well as prefetching in a predictable way.

III. Enabling Runtime Reconfiguration in Real-Time Systems with CoRQ

The focus of our reconfiguration controller *Command-based Reconfiguration Queue* (CoRQ) is to enable the CPU to issue sequences of reconfiguration requests, provide guaranteed reconfiguration delays and relieve the CPU from managing accelerator availability. CoRQ provides commands to inform the CPU of finished reconfigurations in a predictable way; the CPU never has to poll or be interrupted to obtain the information that an accelerator has become available (following a reconfiguration). CoRQ processes 32-bit commands and can be instantiated with an internal memory to store *bitstreams* (configuration data for the reconfigurable fabric). Commands are issued by the CPU using load/stores over the system bus (see Fig. 3). They are either executed immediately or enqueued in an internal FIFO queue (denoted as *immediate* or *queueable* commands in the following). Table I shows all 11 currently supported commands grouped by category (immediate or queueable). The immediate commands are used to control CoRQ itself (stop/resume processing enqueued commands, clear queue, reset) and abort a running reconfiguration. Queueable commands relieve the CPU from managing reconfigurations, i.e., they configure bitstreams (from internal or external memory), provide information about available accelerators through a general-purpose interface (or send an interrupt to the CPU), and can even stall/unstall the CPU to implement the stalling approach. In the following we illustrate how stalling and prefetching can be realized with CoRQ.

Reconfiguring a single accelerator in the stalling approach (see Section II, Fig. 1 upper timeline) can be performed using the following sequence of commands:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2017.2714844, IEEE Embedded Systems Letters


```
1 stopQ          (Im: Stop processing commands from queue)
2 stallCPU       (Qu: Stall CPU)
3 setBaseAddr    (Qu: Set main memory base address)
4 configBitsExt  (Qu: Reconfigure from main memory)
5 sendGPIO       (Qu: Reset accelerators)
6 unstallCPU     (Qu: Unstall CPU)
7 resumeQ        (Im: Process enqueued commands)
```

Listing 1: CoRQ Commands for the Stalling Approach

First, processing commands from the queue is stopped (immediately), otherwise the following command (Line 2) would stall the CPU before the `unstallCPU` command could be enqueued. All following commands (including stall CPU) are queueable. Assuming that the main memory is idle while stalling the CPU, one can use it to configure bitstreams even under timing guarantees. First, the base address of the bitstream is set and then `configBitsExt` instructs CoRQ to configure a bitstreams relative to this base address (Lines 3 and 4). This way, the whole 32-bit address space can be addressed using 32-bit wide commands. Afterwards, `sendGPIO` is executed to trigger CoRQ's GPIOs, e.g., to reset the configured accelerator and ensure it is in a sane state before using it. Once these commands are processed, the CPU is resumed. Finally, `resumeQ` (Line 7) is used to start processing the enqueued commands.

A reconfiguration of two accelerators using prefetching (executing software in parallel, see bottom timeline of Fig. 1) can be performed using the following commands:

```
1 clearQ          (Im: Ensure command queue is empty)
2 abortReconf     (Im: Ensure free reconfiguration port)
3 configBitsInt   (Qu: Reconfigure from internal memory)
4 sendGPIO        (Qu: Store info 'Accelerator A available')
5 configBitsInt   (Qu: Reconfigure from internal memory)
6 sendGPIO        (Qu: Store info 'Accelerator B available')
```

Listing 2: CoRQ Commands for Prefetching

In this case, neither processing queued commands is stopped nor the CPU is stalled. Therefore, the CPU proceeds executing software after issuing the commands to CoRQ. In this example we assume that a previous prefetch could still occupy the reconfiguration port and obsolete commands could be in the queue (see Fig. 2). To be able to guarantee the reconfiguration delay, we need to ensure that no earlier reconfiguration requests are still pending. Therefore, first all remaining commands are cleared and reconfiguration (if any) is aborted (Lines 1 and 2). Afterwards, a bitstream from internal memory is configured. This way, loading the bitstream does not conflict with memory accesses from the CPU to main memory. Once reconfiguration completes, `sendGPIO` is executed (Line 4) to notify the CPU that the first accelerator has become available. This can be done by writing into a memory-mapped register that the CPU can read or by writing to a lookup table that is automatically queried before executing an accelerator. This enables the CPU to use each configured accelerator immediately once it is configured (see Fig. 1 (bottom)), without waiting for the whole set of commands to have finished processing by CoRQ. Afterwards, a second accelerator is configured (Lines 5 and 6).

These two examples illustrate that stalling as well as prefetching can be realized by using CoRQ with simple sequences of commands issued by the CPU.

A. Command Execution

CoRQ processes commands using a finite state machine (FSM) consisting of three states: Fetch from queue (FE), Decode

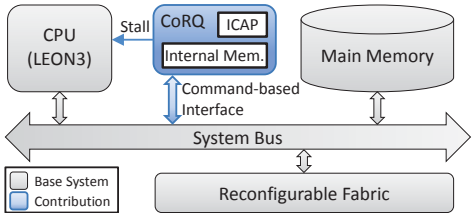

Fig. 3: System on Chip Overview

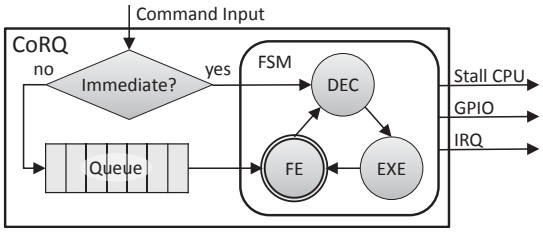

Fig. 4: High-Level View of how CoRQ processes Commands

(DEC) and Execute (EXE) (see Fig. 4). Fetching a command takes a single cycle, the DEC state takes two cycles and the latency of EXE depends on the command (see Table I). Immediate commands control CoRQ itself, and thus have priority over commands from the queue. After being identified as immediate (which takes one cycle), these commands reset the FSM (potentially aborting a queueable command in EXE) and directly enter DEC. In sum, executing either an immediate or a queueable command takes $3 + \text{latency}_{\text{EXE}}$ cycles.

Enqueueing a command takes 2 cycles for identifying it as queueable and writing it to the queue. Commands can simultaneously be enqueued to and fetched from the queue. The realization of this simultaneous access (with a double-ported FIFO) incurs an additional delay of 2 cycles for commands to become visible to the FSM if the FIFO was empty.

B. Guaranteed Reconfiguration Delay

It is possible to load bitstreams from arbitrary addresses, however, accessing the system bus and a shared main memory (especially DDR) can incur memory access delays that are hard to bound for WCET guarantees. Therefore, guaranteeing reconfiguration delays when using CoRQ-external memory (`configBitsExt`) is outside the scope of this letter. Reconfiguration delays are guaranteed when the CoRQ-internal memory is used (`configBitsInt`). The CoRQ-internal memory is implemented using SRAM (so-called Block RAMs on Xilinx FPGAs). This way, the `configBitsInt` command can feed one word of the bitstream in each cycle to the reconfiguration port and utilize its full bandwidth (see Section IV). Additionally, the `configBitsInt` command requires 5 setup cycles and a single cycle at completion. Thus, $\text{latency}_{\text{EXE}} = 6 + \lceil B/4 \rceil$ cycles, with B being the size of the bitstream in bytes (see Table I). Including the latency of FE and DEC, *configuring a single bitstream from CoRQ-internal memory (`configBitsInt`) is guaranteed to take exactly* $9 + \lceil B/4 \rceil$ *cycles*.

C. Analyzing Sequences of Commands

In the examples of Section III, the latency of command sequences is simply the sum of the latencies of the queueable commands: configuring a single bitstream from main memory using stalling (see Listing 1 and Fig. 1 (middle)) results in a latency of $t_{\texttt{stallCPU}} + t_{\texttt{setBaseAddr}} + t_{\texttt{configBitsExt}} + t_{\texttt{sendGPIO}} + t_{\texttt{unstallCPU}} + t_{\texttt{resumeQ}} = 4 + 4 + t_{\texttt{configBitsExt}} + 4 + 4 + 3 =$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/LES.2017.2714844, IEEE Embedded Systems Letters

TABLE II: Ressource Utilization

	LUTs	FlipFlops	BRAM
LEON3 CPU (standard config.)	8,144	3,450	14
CoRQ	398	546	1
Internal Mem. of CoRQ (384 KB)	233	6	96
Available on VC707	303,600	607,200	1,030

$19 + t_{\texttt{configBitsExt}}$ cycles. This is the latency after `resumeQ` reaches CoRQ. At this point the immediate command `stopQ` was already executed (taking 3 cycles) in parallel to filling the queue with commands, and therefore it does not add to this latency. As mentioned before, we do not provide guarantees for `configBitsExt`.

Configuring two bitstreams using prefetching (see Listing 2 and Fig. 1 (bottom)) results in a latency of $t_{\texttt{clearQ}} + t_{\texttt{abortReconf}} + t_{\texttt{configBitsInt}} + t_{\texttt{sendGPIO}} + t_{\texttt{configBitsInt}} + t_{\texttt{sendGPIO}} = 8 + 8 + (9 + \lceil B_1/4 \rceil) + 4 + (9 + \lceil B_2/4 \rceil) + 4 = 42 + \lceil B_1/4 \rceil + \lceil B_2/4 \rceil$. This latency starts once the immediate command `clearQ` reaches CoRQ and is running in parallel to the CPU that sends the commands following `clearQ` to CoRQ. Executing previous commands always takes at least as long as the delay for enqueueing the current command, therefore enqueueing the commands does not add to the delay. If it can be guaranteed that there are no pending reconfigurations, `clearQ` and `abortReconf` can be omitted. In this case, enqueueing the first command to the empty queue would incur an additional latency of 4 cycles, resulting in a total delay of $30 + \lceil B_1/4 \rceil + \lceil B_2/4 \rceil$ (see Section III-A).

IV. Experimental Evaluation

We implemented a synthesis flow for partial reconfiguration and evaluated CoRQ based on a Gaisler LEON3 design (GRLIB GPL 1.4.1, also see Fig. 3) targeting the Xilinx VC707 board (Virtex-7 FPGA)[2]. We used a LEON3 design provided by Gaisler that instantiates a single LEON3, uses the DDR3 on the VC707 as main memory and runs at 100 MHz. CoRQ was added to the AHB system bus and a signal was connected to the LEON3 to enable stalling, no further changes were made to the SoC. For evaluation purposes, we simply reconfigure patterns of flashing the VC707's LEDs. The resource utilization is shown in Table II.

The reconfiguration port (called *Internal Configuration Access Port* (ICAP) in Xilinx devices) can process 4 byte each cycle at maximum 100 MHz on the VC707. Therefore, the theoretical maximum reconfiguration bandwidth is 381.47 MB/s[3]. We reconfigure 25 partial bitstreams of $B = 57,248$ bytes each, which together takes a minimum of 357,800 cycles when assuming the theoretical maximum reconfiguration bandwidth without overheads. Using CoRQ, these reconfigurations take 358,036 cycles[4] which corresponds to a reconfiguration bandwidth of 381.22 MB/s. This means that CoRQ is only 0.066 % (or 236 cycles) slower than the theoretical maximum.

In the following we evaluate the impact of system bus conflicts on the reconfiguration bandwidth. Figure 5 shows the reconfiguration bandwidth results as measured by the CPU. Note that measuring itself adds an overhead, therefore, the measured reconfiguration bandwidth is always lower than the analytical bandwidth of CoRQ. The results were obtained for reconfigurations using the CoRQ-internal memory (Int. Mem.), as well as main memory over the shared AHB system bus (Main. Mem.). 'Stalling' leaves the CPU idle during reconfiguration,

[2]Project incl. benchmarks available at: https://git.scc.kit.edu/CES/corq

[3]More precisely: $(4 \cdot 1024^{-2})/10^{-8} = 381.4697265625$ MB/s

[4]Sum of latencies of the individual commands (see Section III): $4 + 25 \cdot (9 + \lceil 57,248/4 \rceil) + 4 + 3$ cycles

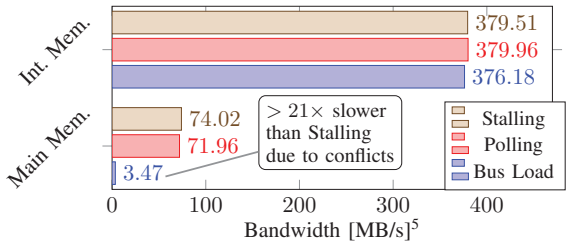

Fig. 5: Reconfiguration bandwidth measured by the CPU, revealing a high variance when using main memory

whereas 'Polling' means that the CPU repeatedly reads CoRQ's status register to check whether reconfiguration has completed (producing traffic on the AHB). 'Bus Load' uses a simple DMA unit that repeatedly initiates maximum length (256 words) AHB burst transactions to provoke system bus and main memory conflicts during reconfiguration. The small variance in measurements when using CoRQ-internal memory ($< 1\%$) stems from the overhead of measuring. The CPU's bus accesses (e.g., for fetching instructions and reading the timers) conflict with the DMA. CoRQ's commands itself always have exactly the same latency when using internal memory.

When reconfiguring over main memory, accesses from CPU, the DMA and CoRQ are in conflict. We can observe a strong variance in reconfiguration bandwidth between the measurements. The measurement under DMA bus load reports only 4.69% of the Stalling bandwidth. This shows that reconfiguration controller design is crucial in runtime-reconfigurable real-time systems. Simply utilizing a shared memory for reconfiguration can lead to a slowdown of more than $21\times$ in reconfiguration bandwidth.

V. Conclusion and Future Work

We have discussed challenges for timing-analyzable runtime reconfiguration in systems that require WCET guarantees. We presented how these challenges can be addressed, and introduced CoRQ: a reconfiguration controller for real-time systems that provides guaranteed reconfiguration delays for the stalling and prefetching approaches for a uniprocessor system.

In future work we will advance our research on timing-analyzable runtime reconfiguration towards multi-core and mixed-criticality systems, and investigate how a reconfigurable fabric can be shared between tasks that require WCET guarantees and tasks that are executed in a best-effort manner.

References

[1] R. Wilhelm, J. Engblom, A. Ermedahl *et al.*, "The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 36:1–36:53, May 2008.

[2] P. Axer, R. Ernst, H. Falk *et al.*, "Building timing predictable embedded systems," *ACM Trans. on Embed. Comput. Syst.*, vol. 13, no. 4, pp. 82:1–82:37, 2014.

[3] M. Damschen, L. Bauer, and J. Henkel, "Timing Analysis of Tasks on Runtime Reconfigurable Processors," *IEEE Trans. on Very Large Scale Integration Syst.*, vol. 25, no. 1, pp. 294–307, Jan. 2017.

[4] ——, "Extending the WCET Problem to Optimize for Runtime-Reconfigurable Processors," *ACM Trans. on Archit. and Code Optim.*, vol. 13, no. 4, pp. 45:1–45:24, Dec. 2016.

[5] C. Steiger, H. Walder, M. Platzner, and L. Thiele, "Online scheduling and placement of real-time tasks to partially reconfigurable devices," in *Proc. of Real-Time Syst. Symp.* IEEE, 2003, pp. 224–235.

[6] F. Dittmann and S. Frank, "Hard real-time reconfiguration port scheduling," in *Proc. of the Conf. on Design, Automation and Test in Europe*. IEEE, 2007, pp. 123–128.

[7] A. Biondi, A. Balsini, M. Pagani *et al.*, "A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs," in *Proc. of Real-Time Syst. Symp.* IEEE, 2016, pp. 1–12.

[5]Average of 50 measures, maximum error $< 0.01\%$