i-Core: A Runtime-Reconfigurable Processor Platform for Cyber-Physical Systems

Marvin Damschen, Martin Rapp, Lars Bauer, and Jörg Henkel

Abstract We provide an overview of *i*-Core (invasive Core), a processor platform with a runtime-reconfigurable instruction set. *i*-Core couples a general-purpose processor core with a reconfigurable fabric that enables the configuration of application-specific hardware accelerators at runtime. This way, *i*-Core can adapt its instruction set to choose a runtime trade-off between the resources allocated and the performance achieved for application-specific acceleration. The adaptivity of *i*-Core is leveraged to advance several research areas that are addressed in this chapter. First, we provide an overview of the *i*-Core architecture. Then, we summarize our research findings in the areas of task scheduling, reliability, and hard real-time as well as multi-core systems. The focus of this summary is on our recent findings in the context of worst-case execution time guarantees.

1 Introduction

Reconfigurable computing, i.e., performing computations using a reconfigurable fabric such as field-programmable gate arrays (FPGAs), was introduced in the early 1990s [40]. Today it is an established computing paradigm in a growing number of application domains in research and industry, not only in embedded computing (e.g., signal processing [39], computer vision [30], or encryption [29]), but also in high-performance and scientific computing (e.g., financial pricing [19] or DNA-sequencing [12]), data centers (e.g., searching [35] or database queries [20]), networks (routing [33], intrusion detection [18]), and others. In these domains, applications generally comprise several compute-intensive loops, so-called *computational kernels*, that benefit greatly from implementation as application-specific hardware accelerators in terms of performance and energy efficiency. FPGAs enable the utilization of application-specific hardware accelerators without fabricating

M. Damschen (✉) · M. Rapp · L. Bauer · J. Henkel (✉)
Karlsruhe Institute of Technology, Karlsruhe, Germany
e-mail: damschen@kit.edu; martin.rapp@kit.edu; lars.bauer@kit.edu; henkel@kit.edu

S. S. Bhattacharyya et al. (eds.), *Embedded, Cyber-Physical, and IoT Systems*,
https://doi.org/10.1007/978-3-030-16949-7_1

custom chips and they provide flexibility as well as the ability for upgrades like software.

Several alternatives exist when designing reconfigurable cyber-physical systems that combine a general-purpose CPU with a reconfigurable fabric. Generally, a tighter integration reduces communication latency between CPU and reconfigurable fabric, but requires more effort in the architectural design of the system. Numerous products are available that add an FPGA as a separate chip to an existing system by attaching it to the system's peripheral bus. However, it is crucial for cyber-physical applications to minimize (1) the communication latency between CPU and reconfigurable fabric to enable acceleration of short-running kernels (e.g., in control loops) and (2) the system's power consumption as well as (3) area footprint. Therefore, the advancing trend of processor integration has resulted in *reconfigurable SoCs* that combine FPGAs and CPUs on a single chip (e.g., Xilinx Zynq or Intel (formerly Altera) SoC FPGA), which have led to the wide adoption of reconfigurable systems in cyber-physical systems, e.g., in implementations of advanced driver assistance systems in the automotive domain. In reconfigurable SoCs, CPU and FPGA are still separate processing devices that communicate over the (internal) system bus.

In this chapter, we demonstrate that an even tighter integration of CPU and FPGA than in current reconfigurable SoCs is beneficial to target non-functional properties of cyber-physical systems based on *i*-Core. *i*-Core (invasive Core [28]) is a *reconfigurable processor* that attaches an FPGA-based reconfigurable fabric directly to the CPU pipeline. In the remainder of this chapter, we first introduce the *i*-Core architecture. Then, an overview of our recent research is given, starting with task scheduling (Sect. 2) for reconfigurable processors. We then focus on two non-functional properties that are commonly found in cyber-physical systems: reliability constraints (Sect. 3) and worst-case execution time guarantees (Sect. 4). Finally, we summarize our work on reconfigurable multi-core architectures (Sect. 5).

1.1 The i-Core Architecture

This section introduces key concepts of the *i*-Core architecture as a basis for the following sections. Details, including comparisons to other reconfigurable system designs that couple a reconfigurable area with a processor core, can be found in [8]. An overview of the *i*-Core architecture is shown in Fig. 1. *i*-Core is a reconfigurable processor, i.e., it is based on a general-purpose processor (GPP) pipeline and enables the execution of runtime-reconfigurable *custom instructions* (CIs). CIs extend the processor's core instruction set architecture (cISA) by application-specific instructions that are realized using (1) microcode and (2) reconfigurable accelerators that are detailed in the following.

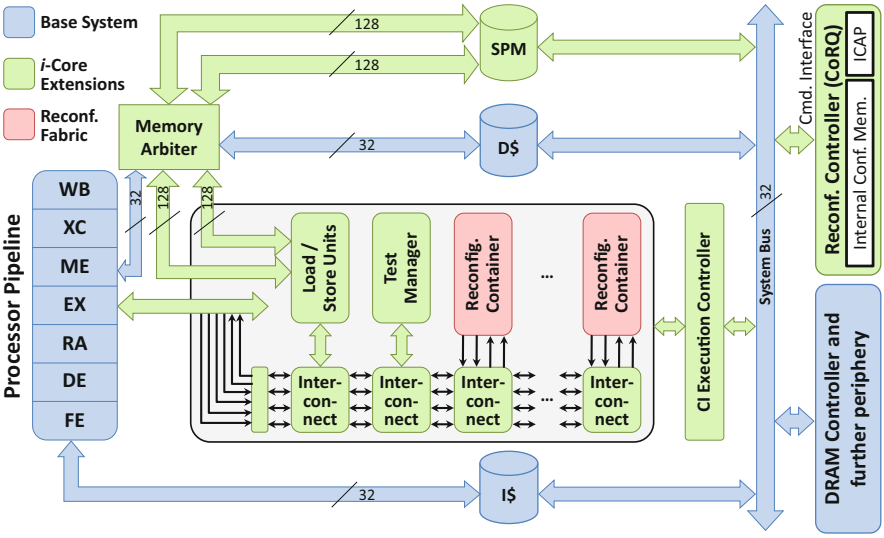

Fig. 1 The main components that the *i*-Core processor platform comprises

1.1.1 Microcoded Custom Instructions

When the processor pipeline encounters a CI in its execute (EX) stage, the pipeline stalls and initiates execution of the respective microprogram (i.e., a program written in microcode) that implements the functionality of the encountered CI on the CI execution controller. The microprogram controls all resources of the reconfigurable fabric:

- Load/store units (LSUs) enable access to the main memory (through the processor's L1 data cache (D$)) and high-bandwidth scratchpad memory (SPM) for CIs
- Reconfigurable containers (RCs), (embedded) FPGAs that provide the reconfigurable area for runtime-reconfiguration of accelerators (one accelerator per container, each of similar complexity as, e.g., floating-point multiply-accumulate or a dozen integer operations)
- Interconnects connect LSUs, RCs, and the processor's register file to a common (four word-wide segmented) bus.

Note that a single microprogram can utilize one or more accelerators. In other words, *the functionality defined by a CI is realized using one or more accelerators*. Application-specific hardware accelerators provide an important trade-off: the more area is utilized, the higher the resulting performance. At the same time, multiple accelerators compete for the constrained reconfigurable area. This trade-off is the result of instruction-level parallelism that can be exploited when more hardware resources are added to an application-specific accelerator. The main benefit of

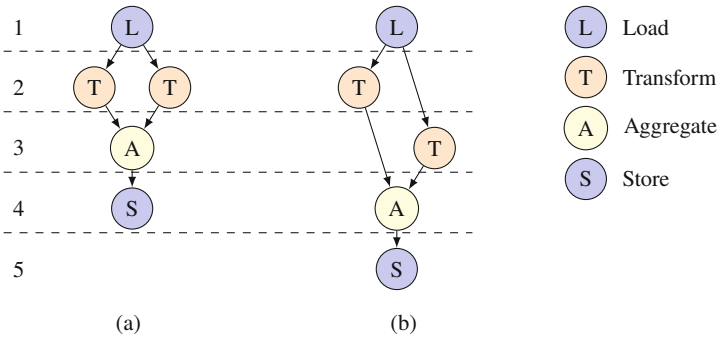

Fig. 2 CIs define computations as DFGs that can be scheduled with different amounts of accelerators, resulting in different latencies. (**a**) Two "transform" accelerators configured. (**b**) One "transform" accelerator configured

allowing CIs to utilize more than one accelerator is that this trade-off can be chosen at runtime by providing several microprograms that implement the same CI, but utilize different amounts of accelerators that each implement a part of the CIs functionality. Consequently, CIs define computations as data-flow graphs (DFG) where nodes are accelerators and load/stores. Figure 2 shows a simplified example of a CI that loads input data, performs transformations on the data, aggregates results, and finally stores them. Depending on how many "transform" accelerators are configured in the RCs at runtime, the DFG can be scheduled in four steps (see Fig. 2a) or five steps (see Fig. 2b). Each of these schedules corresponds to a microprogram for the CI execution controller, which implements the CI.[1]

The CI model employed by *i*-Core provides additional opportunities for performance increases over a model where a CI corresponds to a single accelerator:

- *Sharing*. Because the functionality of a CI is "split" in several accelerators, the computations of a single accelerator become more general. Thus, once configured accelerators create opportunities to be utilized by multiple CIs (implementing different functionality).
- *Upgrading*. Reconfiguration takes time. The *reconfiguration delay* on modern architectures is still in the range of milliseconds, depending on the configuration size. In the *i*-Core CI model, a CI can already be executed in hardware as soon as each required accelerator type is configured at least once on the reconfigurable fabric (like in Fig. 2b). The more accelerators finish reconfiguration during runtime, the more parallelism can be exploited and the lower the CI latency gets (e.g., runtime reconfiguration of an additional "transform" accelerator leads to schedule Fig. 2a instead of Fig. 2b).

[1] In the remainder of this chapter we will refer to CI implementation and CI microprogram interchangeably, depending on the focus of the respective section.

More details about such *modular* CIs, the load/store unit (see Fig. 1), and the address generation can be found in [8, 11].

1.1.2 Software Emulation

So far, we discussed how CIs are executed, assuming all accelerators required by a certain implementation are currently configured on the reconfigurable fabric. However, CIs can be *unavailable*, i.e., there exists no schedule of the CI's DFG for the accelerators that are currently configured. Two alternatives exist to handle the case that the *i*-Core attempts to execute an unavailable CI at runtime: *stalling* and *software emulation*. Figure 3 visualizes the different execution models. Software only (top) corresponds to execution of iterations of a kernel that does not utilize any CIs. *Stalling* (middle) executes CIs on the reconfigurable fabric and trying to execute an unavailable CI is an error. Therefore, CIs need to be configured before the kernel is entered and the execution is stalled until the required CIs are available. Finally, *software emulation* (bottom) triggers functionally equivalent software execution of an unavailable CI on the *i*-Core's pipeline using the base processor's cISA. It enables execution of the kernel while required CIs are still being reconfigured. Thus, progress can already be made without any CIs and as soon as reconfiguration of a CI finishes, the CI is utilized to speed up the following iteration of the kernel. While software emulation is always beneficial at runtime, it is more complex to analyze for execution time guarantees than stalling, which will be detailed in Sect. 4. Software emulation is realized using two alternative approaches in *i*-Core: an "unimplemented instruction trap" with a corresponding trap handler or CI Invocations that branch to either the CI or software emulation before trying to execute the CI. CI Invocations are beneficial for execution time guarantees and are detailed in Sect. 4.2.

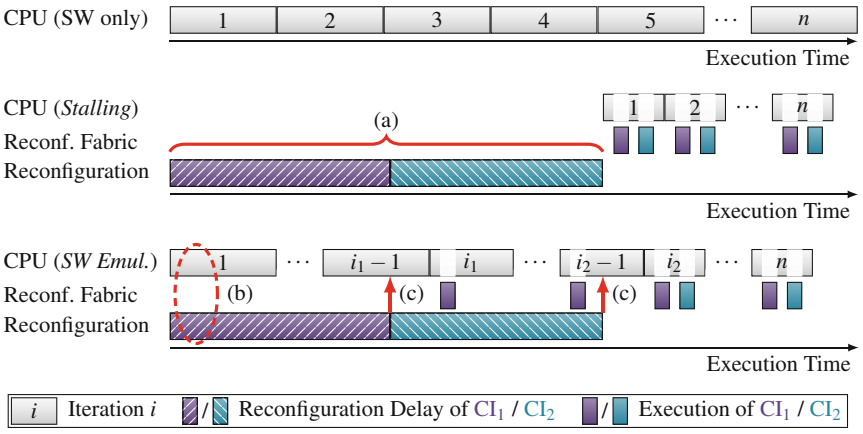

Fig. 3 Timelines of executing a kernel using software only, stalling and software emulation

This section provided an overview of the *i*-Core architecture as a background for the following sections. *i*-Core exists as a constantly evolving hardware prototype, currently it is based on the Gaisler LEON3 processor[2] and synthesizes to Xilinx Virtex-7 FPGAs. A more detailed explanation of how the architecture is realized can be found in [8, 9, 11]. Additionally, a SystemC-based cycle-accurate simulator is available [10] for early evaluation of runtime system algorithms. In the following, task scheduling on runtime-reconfigurable processors like *i*-Core is discussed.

2 Task Scheduling for Runtime-Reconfigurable Processors

While application-specific accelerators can provide significant speedup in domains such as mobile computing (e.g., 5G, cryptography) and robotics (e.g., image/audio processing, feature matching), applications in these domains are typically composed of multiple tasks. Systems for these applications are often dynamic multi-tasking systems, i.e., task arrival is not known at design time and task duration is dependent on input data (e.g., the recognized objects in a camera-based mobile robot). As long as such dynamically changing multi-tasking scenarios are not efficiently supported on reconfigurable processors, their inherent efficiency advantages are inaccessible for these demanding domains. In the following, we present how efficient multi-tasking support is enabled for reconfigurable processors by presenting two task schedulers for scenarios where it is unknown at compile time which tasks will execute at the same time (and which CIs will be required). First, we present a task scheduler that is optimized for reducing the *tardiness* (i.e., the total time by which deadlines were missed over all tasks), then a task scheduler for improving the *makespan* (i.e., the completion time of the tasks).

2.1 Task Scheduler for Optimizing Tardiness

Tardiness reduction is important for application scenarios where at least some of the tasks have soft deadlines, e.g., reducing the tardiness for a video recording and encoding task leads to a reduced number of dropped frames. In [7] we present the performance aware task scheduling (PATS) strategy that aims to reduce tardiness of all running tasks. We introduce the notion of *task efficiency* in reconfigurable processors and observe that it changes over time for a single task: A task that has just started executing a kernel (i.e., its required accelerators are not yet configured) will need to execute CIs using software emulation and thus, have a *low efficiency*. In contrast, a task that has finished reconfiguring its accelerators will perform more computations in the same amount of time and thus, have *high efficiency*. Therefore,

[2] https://www.gaisler.com/.

our scheduler favors executing tasks with high efficiency (unless it would lead to a deadline miss for a task with low efficiency). Tasks with low efficiency can perform their reconfigurations in parallel to execution of tasks with high efficiency and achieve a high efficiency until they are executed at a later time. PATS was compared with standard scheduling approaches like earliest deadline first (EDF), rate-monotonic scheduling (RMS), and the scheduler used in the Molen processor [37] on reconfigurable processors with different fabric sizes and tasksets with different deadlines. PATS achieves a 1.45× better tardiness on average (maximum 1.92×, minimum 1.14× better) compared to the other schedulers.

2.2 Task Scheduler for Optimizing Makespan

For systems without deadlines, such as in high-performance computing, an important performance metric is the makespan (i.e., the completion time) of a taskset. To improve the makespan on a reconfigurable processor, we introduced a combined task scheduling and reconfigurable area allocation approach: makespan optimization for reconfigurable processors (MORP) [24]. Using the notion of task efficiency introduced in PATS, MORP is based on the observation that task efficiency is low after an application switches from one kernel to another (as it requires reconfiguration of accelerators for the new kernel), an effect we call reconfiguration-induced cycle loss (RiCL). By reducing the RiCL of a taskset, we improve its makespan. The largest potential for RiCL reduction is in complex tasks that switch between different kernels during their execution time (e.g., video encoders). Such tasks are scheduled by our approach as *primary* tasks, which are initially assigned the full reconfigurable fabric. Using combination of offline profiling and lightweight online monitoring, the approach predicts when the primary task will switch from one kernel to another. A short time before leaving a kernel (at high efficiency), a small share of the reconfigurable area is already reallocated to a *secondary* task. While accelerators of the secondary task are reconfigured, the primary task completes its current kernel. Upon switching to its next kernel, the task efficiency of the primary task drops (as its required accelerators are not yet configured). Therefore, the system temporarily schedules the secondary task, which by this point has a higher efficiency than the primary task. The primary task reconfigures its accelerators while the secondary task is running. During its final reconfigurations (for the upcoming kernel), the primary task reacquires the reconfigurable area that was allocated to the secondary task. At this point, the MORP switches to the primary task at high efficiency. Figure 4 shows results of MORP in comparison to other schedulers for different tasksets (each containing a subset of: H.264, SUSAN, AdPCM, AES, SHA). Our approach achieves an average makespan reduction by 6.5% and 20.3%, compared to the SPT scheduler (shortest processing time, optimal for makespan minimization on a non-reconfigurable processor) and RR (round robin) scheduling, respectively. Compared to the theoretical lower bound of makespan (where RiCL is assumed to be zero for any taskset), our approach produces results that are on

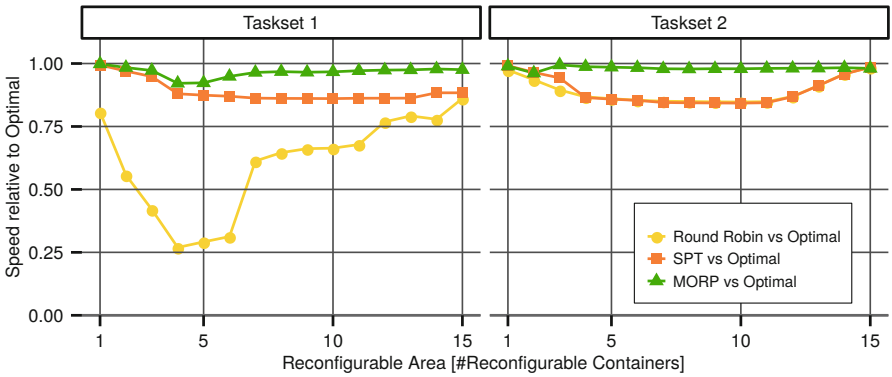

Fig. 4 Comparison of MORP to state-of-the-art scheduling approaches

average only 2.8% worse than this theoretical optimal result. The other evaluated schedulers produce schedules with makespans that are 14–20% worse than optimal.

In summary, our runtime-reconfiguration aware task schedulers can achieve 1.45× better tardiness (PATS) than comparable state-of-the-art schedulers and a makespan reduction that on average is only 2.8% worse than the theoretical lower bound. This section presented how efficient multi-tasking support is enabled on runtime-reconfigurable processors. When processors are embedded into a bigger system, like a cyber-physical system, generally non-functional requirements (e.g., real-time or reliability constraints) need to be met. Deploying runtime-reconfigurable processors in such systems provides opportunities and challenges as the following sections detail. The next section focuses on the non-functional property of reliable execution.

3 Reliable Reconfigurable Processors

Reliability concerns due to technology scaling have been a major focus of researchers and designers for several technology nodes. The reliability of reconfigurable processors is threatened not only by soft errors, but also by aging effects and latent defects. Latent defects are present in the material (unobserved) and may manifest as permanent fault during normal operation. Aging effects degrade the properties of transistors and may lead to a reduced performance (via increased transistor threshold voltage and correspondingly reduced maximal frequency) or even permanent faults (e.g., time-dependent dielectric breakdown) [27]. Soft errors instead are transient and may lead to bit flips in memory cell or logic latches [26].

In a reconfigurable processor, the non-reconfigurable components (e.g., CPU pipeline; see Fig. 1) may potentially be fabricated in an older technology node, as the system performance does not stem from high pipeline frequency, but from high

parallelism of the reconfigurable accelerators. But the reconfigurable containers (RCs) should always be implemented in cutting-edge technology, to close the area/frequency gap compared to non-reconfigurable accelerators. 2.5D stacking with a silicon interposer and stacked chiplets can be used to integrate the non-reconfigurable components with the cutting-edge reconfigurable fabric. Xilinx is using 2.5D stacking since its Virtex-7 series and also Achronix promotes it as an alternative to their commercial embedded FPGAs [3]. As the RCs are manufactured in the latest technology nodes, reliability concerns are even more serious for them. However, the inherent flexibility and adaptivity due to the reconfigurability of the RCs also provides opportunities for fault mitigation. In this section, we present an overview of our comprehensive efforts in the OTERA project (online test strategies for reliable reconfigurable architectures; based on the *i*-Core) that allow to ensure functional RCs, correct reconfiguration, tolerate faults, mitigate aging, and guarantee a target reliability for observed soft-error rates.

3.1 Testing for Structural Defects and Correct Reconfiguration

In order to test the reconfigurable containers (RCs) and their accelerators for correct functionality, we added a "test manager" in a similar way as the RCs (see Fig. 1). It supports a *pre-configuration test* (PRET) and a *post-configuration test* (PORT) [6]. PRET tests the RC-internal resources for permanent faults and PORT tests whether the accelerator was configured correctly and whether it meets the target frequency. In order to test the RC-internal resources, there needs to be a way to apply test input data to RCs and to analyze their result. Therefore, PRET cannot be performed using the application-specific accelerators, but special *test configurations* (TCs, i.e., accelerators with the purpose to test RCs) need to be used.

To test the CLB array of a RC for permanent *stuck-at* faults, we developed nine TCs that test *all* CLB features, i.e., LUTs as logic, LUTs as shift registers, LUTs as distributed RAMs, multiplexers, registers, and carry-chain logic [1]. To perform a test with PRET, at first, one of the TCs needs to be configured into a RC. Afterwards, a special CI is executed that instructs the "test manager" to create and send test patterns to the RC-under-test, let it compute, and receive and analyze its computation result. The TCs use between 6 and 320 test patterns to perform their specific test. Their lowest frequency is 154 MHz, i.e., faster than the lowest frequency of the application-specific accelerators.

To evaluate the testing overhead, we use an H.264 video encoder with 9 CIs that permanently reconfigures the RCs to adapt them to the current *processing kernel* (i.e., motion estimation, encoding, or in-loop deblocking filter). The application requests application-specific reconfigurations and after every *n*th request, PRET *inserts* a small test. The RC that the application wanted to reconfigure anyway is reconfigured to contain one of the TCs and then its test patterns are applied. After the test, the actually requested application-specific accelerator is reconfigured into this RC. Note that the application continues executing during reconfiguring the TC

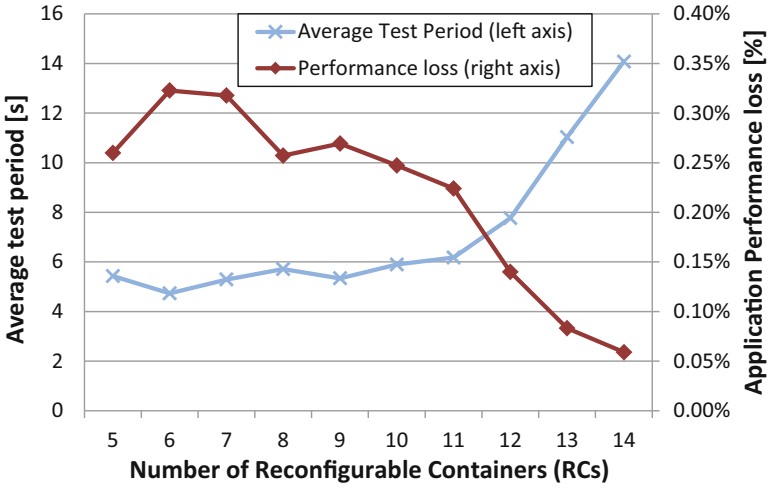

Fig. 5 Delay until all RCs are exhaustively tested by PRET vs. application performance loss due to testing

and note that many TCs are required until the entire reconfigurable fabric is tested (9 TCs per RC).

For the results shown in Fig. 5 we performed one TC after every four application-specific reconfigurations [1]. The figure analyzes the average test period and the application performance loss for reconfigurable processors with varying number of RCs. The average test period denotes the time in seconds until *all* RCs in the system are tested by *all* nine TCs each. The application performance loss denotes the additionally needed time compared to a system without any testing. Despite the very low overhead (on average 0.22%), the structural test completes very fast (on average 7.15 s). It is notable that the test period increases significantly for more than 11 RCs (while at the same time the overhead reduces). The reason is that for reconfigurable processors with so many RCs, some of the RCs are less often reconfigured by the application and thus they are tested less often and thus dominate the test period. To reduce the test period for these cases, we can use periodic testing of RCs that were not reconfigured for a longer time. For brevity, details on this extension and details on PORT (similar to PRET but without demanding extra TCs) and test scheduling are omitted and can be found in [6].

3.2 Fault Tolerance and Aging Mitigation

The pre-configuration test (PRET) described in Sect. 3.1 allows to detect permanent faults, but it provides no means to deal with them. In this section we present the idea of *diversified configurations* that (1) allows to tolerate permanent faults in RCs

and that (2) can also be used to mitigate aging, i.e., their root cause. Whenever a permanent fault manifests in a CLB of a RC (detected by PRET), accelerators that use this CLB as part of their operation will no longer function correctly in all cases. However, accelerators that do not use that CLB will not be affected by the fault. The main idea of diversified configurations is to provide different implementations (i.e., place and route results) of the same accelerator that differ in their resource usage. In [47] we present the algorithm that generates a minimal set of diversified configurations for an accelerator such that for any single CLB that is found to be faulty, a diversified configuration exists that does not use that CLB. Additionally, the algorithm can generate even more diversified configuration, such that also multi-CLB-errors can be tolerated. We use the `PROHIBIT` constraint from Xilinx to ensure that the specified CLBs are not used and then let the Xilinx P&R tools freely determine the implementation of the accelerator on the remaining CLBs. In our evaluation for several applications (H.264, ADPCM, AES, JPEG), two to seven diversified configurations are sufficient to provide fault tolerance for any single-CLB fault at a minimal frequency loss of 5.6% compared to the accelerator implementations without diversification [47].

As diversified configurations differ in their usage of CLBs—and thus also in how much *stress* they induce to individual CLBs—they can also be used to mitigate aging. For instance, the degree of hot-carrier injection (HCI) aging depends (among others) on the average toggle rate of transistors. Higher toggle rate (at otherwise identical conditions) leads to faster HCI aging, which eventually increases the transistor threshold voltage. By continuously switching between different diversified configurations, the stress can be balanced over all transistors, instead of being accumulated into few transistors that would otherwise age fastest and thus affect the performance or reliability goals of the system first. Figure 6a shows the stress that an accelerator (`alu4` from the MCNC benchmark suite) induces into its RC, when it remains configured in the same RC for the entire application execution time [46]. Each square in Fig. 6a corresponds to one CLB of the RC and the color shows the average toggle rate of the transistors in this CLB. Figure 6b shows the stress when using a maximally diversified configuration of the same accelerator and Fig. 6c shows the stress when periodically reconfiguring between these two configurations. Even though the stress is more balanced, the peak stress in Fig. 6c is not reduced significantly. Figure 6d achieves a significantly reduced peak stress by switching between four diversified configurations (the minimal number to tolerate all single-CLB faults for this accelerator). The maximum HCI stress reduction when using the minimum number of configurations for single-CLB fault tolerance ranges up to 68.9%, which increases the time to failure by 222% [46].

In addition to balancing the stress within a RC by switching between diversified configurations (intra-RC stress balancing), we also developed a stress-aware placement algorithm (STRAP) that decides for all accelerators that shall be reconfigured, which diversified configuration shall be used and into which RC it shall be reconfigured (inter-RC stress balancing) [47, 49]. This is important as some accelerators may be used significantly more often than others, which could lead to some RCs being significantly more stressed than others (even though being

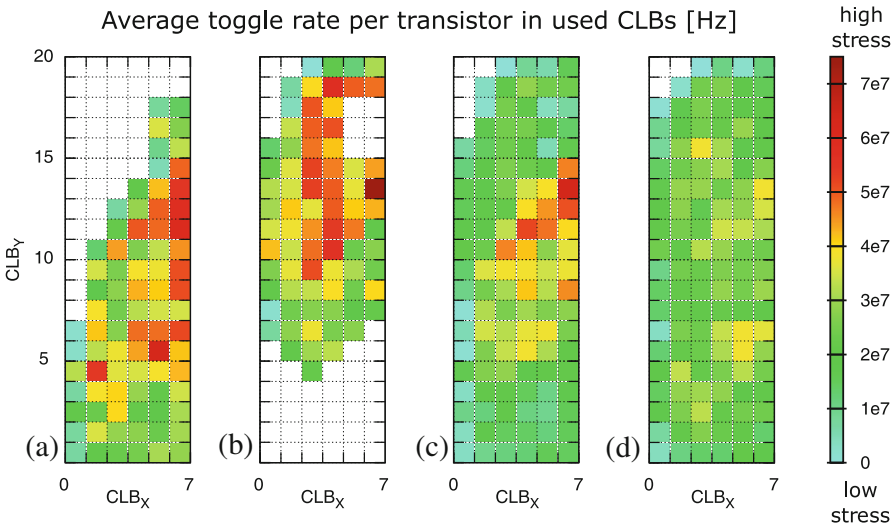

Fig. 6 The average toggle rate (stress) of the CLBs of a reconfigurable container after several executions of (**a**) an accelerator, (**b**) a maximally diversified implementation of the same accelerator, (**c**) an alternating schedule (reconfiguration) of the first two implementations, (**d**) a balanced schedule of four implementations

reasonable balanced within the RC). As the placement decision has to be made at runtime and depends on the accumulated stress due to the CI executions so far, we developed an efficient search space pruning technique to reduce the runtime overhead by calculating guaranteed lower and upper bounds of the achievable stress distribution. Details on the algorithms can be found in [47, 49]. Altogether, using our diversified configurations with our fault-tolerant and stress-balancing accelerator placement, an H.264 video encoder delivers from 1.9× up to 3.7× the performance of a baseline system in the presence of 4 to 40 faulty CLBs, while at the same time achieving up to 6.8× higher mean-time to failure (MTTF) than a baseline system for a set of benchmarks [47].

3.3 Guaranteed Reliability for Reconfigurable Processors

Besides permanent faults and aging effects, single-event upsets (SEUs) are the most-demanding challenge for reliable reconfigurable processors. SEUs manifest themselves as a temporary *bit flip* in the logic that may be captured by state-holding elements. As reconfigurable processors use SRAM-based reconfigurable fabrics to implement their RCs, they contain rather large amounts of state-holding elements and are thus especially susceptible to SEUs. Additionally, if an SEU flips a configuration bit, then this may actually alter the function of the accelerator

(defined by the configuration bits) and this fault is only corrected when the RC is reconfigured again. Instead, the pipeline of reconfigurable processors (see Fig. 1) is less susceptible to SEUs, as it is not implemented using a reconfigurable fabric. It is also not critical for the overall performance (most performance-relevant parts are implemented as CIs), so it can be implemented using an older technology node, which is inherently less susceptible for SEUs. Therefore, the software implementation of a CI has the highest reliability and can be used when the reliability of a CI implementation on the reconfigurable fabric is too low.

Complex accelerators that use more resources have more *critical configuration bits* (i.e., those bits that define the functionality of the accelerators) and therefore have a higher probability that one of their critical bits is flipped due to an SEU. Based on the number of critical bits and the SEU rate,[3] we can determine the reliability of an accelerator, i.e., the probability that it produces a correct result at a certain time. The reliability starts at 100% right after the accelerator was reconfigured and it reduces exponentially over time until it is *scrubbed*[4] or reconfigured (see Fig. 7a). To increase reliability, modular redundancy can be applied to these accelerators (see Fig. 7b). This reduce the rate at which the reliability decreases, since even in case of an SEU, the system is able to correct it. Alternatively, the scrubbing frequency can be increased (see Fig. 7c), but the bandwidth to access the configuration data is limited and a higher scrubbing frequency also affects the accelerator reconfiguration time, as scrubbing and reconfiguring both need to access the same configuration port.

Only slight changes in the hardware architecture are needed to implement modular redundancy, such that any two (three) neighboring containers can be combined to a DWC (TMR) pair. In case of TMR (triple modular redundancy), the CI may complete, but the faulty RC needs to be scrubbed soon to avoid aggregating multiple faults in the TMR pair. In case of DWC, the CI has to be aborted and has to execute in software emulation instead. Future executions of the same CI are directly sent to software and both RCs need to be scrubbed, as it is not known which of them caused the fault. However, applying modular redundancy means that less accelerators are available for parallel execution, which means that the overall application execution time increases. At the same time, also the *resident time* of accelerators increases, i.e., it takes longer until they are reconfigured by the application. The longer the resident time, the higher the probability that one of the critical configuration bits gets corrupted and the lower the reliability of the accelerator. Therefore we also have to increase the scrubbing frequency with the corresponding drawbacks.

Instead of statically choosing the degree of redundancy, reconfigurable processors can change between performance (no redundancy), DWC, and TMR at runtime and that can be decided for each accelerator independently [48]. The

[3] Determined by a soft-error monitor, e.g., using the number of ECC errors in memory, caches, etc.

[4] The configuration bits of a RC are read back and their ECC values are checked for errors with subsequent correction, if needed.

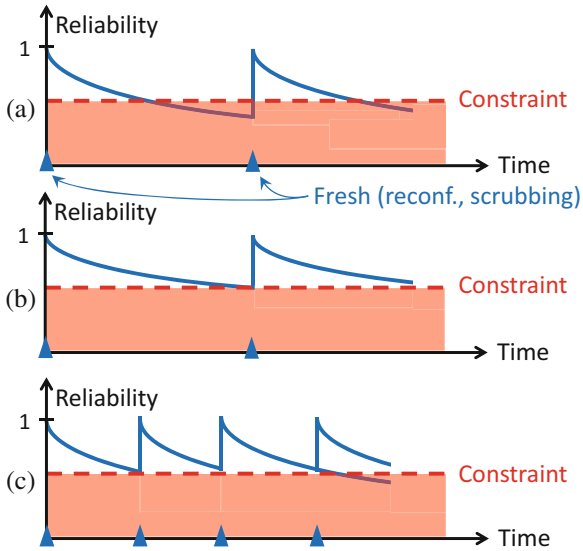

Fig. 7 The reliability of a reconfigurable container over time: (**a**) after reconfiguration or scrubbing, the reliability is known to be "1" and then decreases exponentially, potentially violating a given reliability constraint. (**b**) In order to avoid the violation, modular redundancy can be used to reduce the reliability decrease. (**c**) Alternatively (or in combination), more frequent scrubbing can be used

main challenge is to select the accelerators and their redundancy modes to satisfy application-determined reliability constraint, while at the same time maximizing the performance. We developed an algorithm that conducts the decision in multiple steps [45, 48]. The main idea is to combine all reliability impacting factors for an accelerator execution in one metric, the *effective critical bits* (ECBs), and then to distribute the ECBs among kernels and CIs. The metric is inspired by the critical bits, i.e., the configuration bits of an accelerator that define its functionality. These bits need to have the correct value in order to execute the accelerator correctly. Applying modular redundancy to an accelerator increases its reliability. That is basically the same effect as if the accelerator would have *less* critical bits, and that is exactly what we express as a reduction in ECBs of that accelerator. Similarly, increasing the scrubbing rate reduces the ECBs for all accelerators. For a CI that executes in software emulation, its ECBs basically correspond to zero, i.e., it is very reliable.

For a given target reliability and SEU rate, we calculate the number of ECBs that we can maximally tolerate [45]. Then, we partition these ECBs among the different kernels of the application, based on their resource requirements and expected execution time. In the next step, the ECBs of a kernel are partitioned among its CIs. Those CIs that have long execution time and require a large number of accelerators obtain the most budget, i.e., we assign more ECBs to complex and/or

slow CIs in order to realize them in faster hardware implementations [45]. This ECB budgeting only needs to be recomputed if the target reliability or the monitored SEU rate change. Finally, we select the minimally needed scrubbing frequency and the redundancy modes for accelerators to implement the CIs such that they do not violate the given ECB budget and maximize the application performance [48]. Thereby the approach guarantees an application-specific minimum level of reliability for the CIs and the application. Since the accelerators are not over-protected, the performance of the application is maximized for the given target reliability. We evaluated the approach using an H.264 video encoder and varying SEU rates and we compared it against threshold-based methods that reconfigure all accelerators to DWC (or TMR) when exceeding a threshold (or implement the CIs in software if too few RCs are available). Compared to these threshold-based methods, out approach guarantees the same target reliability while providing 20.0% (DWC) or 42.6% (TMR) higher application performance on average. In the best case, up to 34.8% (DWC) and 68.3% (TMR) faster execution is obtained without violating the given reliability constraint.

This concludes our discussion of reliability concerns in runtime-reconfigurable processors. Orthogonal to reliability concerns, designers of cyber-physical systems need to deal with timing worst-case execution time concerns that are discussed in the following section.

4 Worst-Case Execution Time Guarantees

In contrast to general-purpose computing systems, cyber-physical systems need to meet non-functional requirements like timing constraints. Failing to meet a given deadline can lead to severe malfunctions, therefore a *timing validation* is performed to guarantee the timing constraints [44]. As part of the timing validation, a schedulability analysis is performed to guarantee that the set of tasks that should be executed on a system can be scheduled at runtime under any circumstances. The input to the schedulability analysis is the *worst-case execution time* (WCET), which needs to be known for every task from the taskset [44].

Statically determining the WCET of a task is a complex problem. Due to the undecidability halting problem, it is in general impossible to determine the precise WCET of a task or its worst-case input [34]. WCET analysis is further complicated by the fact that modern processor design focuses on reducing the average execution time: Features like pipelining, caches, and branch prediction introduce a microarchitectural state, i.e., the latency of an instruction is dependent on the execution history. Even with recent advances in research, WCET analysis lags years behind current microarchitectures with out-of-order scheduling pipelines, several hardware threads, and multiple (shared) cache layers [38]. The real challenge for a successful timing validation is to obtain *tight bounds* of the execution time, i.e., the overestimation should be as low as possible. Therefore, performance features amenable for timing analysis are requested [4, 21, 41] to face the increasing

performance demands of real-time systems. Improving the WCET of a task is highly desirable as it may enable a cyber-physical system to meet previously infeasible timing constraints or make room for power optimizations.

This section introduces WCET analyses and optimization of tasks on runtime-reconfigurable processor designs like *i*-Core as one way to escape the scarcity of timing-analyzable performance features. First, this section introduces a reconfiguration controller that provides guaranteed reconfiguration delays to enable runtime reconfiguration in hard real-time systems. Afterwards, it is shown how WCET bounds can be obtained for processors with a runtime-reconfigurable instruction set. Finally, this section presents how WCET bounds can be optimized by selecting WCET-optimizing CIs.

4.1 Guaranteed Reconfiguration Delays

The most straight-forward approach of improving WCET guarantees a task using runtime reconfiguration with the constraints of timing-analyzability and reasonable implementation effort is to apply the *stalling* model [17] as shown in Fig. 3 (middle). Using the stalling model, a task that requests reconfigurations of accelerators can be analyzed for WCET guarantees with established timing analysis techniques by adding the reconfiguration delay (see Fig. 3(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 when the CPU is stalled and its memory accesses cannot interfere with reconfiguration on main memory or a shared system bus. However, stalling is not state of the art in reconfigurable systems, because the CPU is forced to remain idle during reconfiguration and cannot, e.g., execute computations that do not depend on the accelerators being reconfigured.

An approach that enables the CPU to perform useful operations in parallel to reconfiguration is software emulation as explained in Sect. 1.1.2 and shown in Fig. 3 (bottom). Software emulation is an established technique in average-case optimizing reconfigurable systems, because it provides considerable performance improvements. For real-time systems, however, it poses new challenges:

- Figure 3(b): Static timing analysis needs to capture potential conflicts on main memory or a shared system bus when the reconfiguration process and the CPU act on memory in parallel.
- Figure 3(c): When reconfiguration and execution on CPU execute in parallel, they need to be synchronized at some point. The main question for WCET estimates is: 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 execution sped up? This question is addressed in Sect. 4.2.

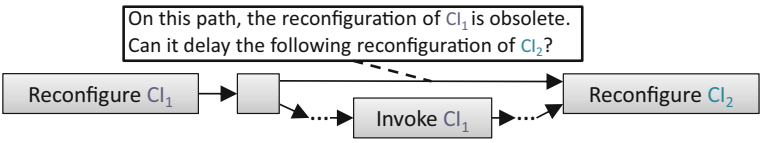

Fig. 8 Control-flow graph that shows how one reconfiguration request can delay another reconfiguration request, thus impairing timing analysis

- Figure 8: At runtime, execution could follow a faster path than the worst-case path. A new reconfiguration request could be reached, while a reconfiguration for a previous kernel is still in progress. 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. To be able to guarantee that the reconfiguration port is unoccupied for each reconfiguration request, it needs to be possible to abort reconfigurations.

Limiting tasks to the stalling model might seem as the favorable way to enable reconfiguration in real-time systems due to the potentially complex analysis of software emulation. When scheduling multiple real-time tasks, however, the stalling model poses similar challenges even on a uniprocessor system: when a task that requests a reconfiguration is stalled, another task can execute in parallel to the reconfiguration delay (see Fig. 3(a) and, e.g., [13]). In Sect. 4.2 it will be shown that software emulation 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.

4.1.1 Enabling Runtime Reconfiguration in Real-Time Systems with CoRQ

To address the challenges of runtime reconfiguration in real-time systems, we designed a reconfiguration controller *command-based reconfiguration queue* (CoRQ[5] [16]) that enables the CPU (any CPU, not necessarily the *i*-Core) to issue sequences of reconfiguration requests, provides guaranteed reconfiguration delays, and relieves the CPU from managing accelerator availability. CoRQ informs 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. 1). They are either executed immediately or enqueued in an internal FIFO queue (denoted as *immediate* or *queueable* commands, respectively,

[5]Open-source project available at: https://git.scc.kit.edu/CES/corq.

Table 1 CoRQ commands with cycles spent in EXE state

Command	**Im**mediate/**Qu**eueable	Latency$_{EXE}$ (cycles)[a]
`clearQ`	Im, Qu	5
`abortReconf`	Im	5
`configBitsInt`[b]	Qu	$6 + \lceil B/4 \rceil$
`sendGPIO`	Qu	1

B—size of bitstream (byte)
[a]Discussed in Sect. 4.1.2
[b]Detailed in Sect. 4.1.3

```
1 clearQ          (Im: Ensure command queue is empty)
2 abortReconf     (Im: Ensure free reconfiguration port)
3 configBitsInt (Qu: Configure bitstream B1 from internal memory)
4 sendGPIO        (Qu: Store info 'CI1 available')
5 configBitsInt (Qu: Configure bitstream B2 from internal memory)
6 sendGPIO        (Qu: Store info 'CI2 available')
```

Listing 1 CoRQ commands for implementing software emulation

in the following). 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 CIs through a general-purpose interface (or send an interrupt to the CPU), and can even stall/unstall the CPU to implement the stalling model. CoRQ currently supports 11 commands, the subset of 4 commands that realize software emulation is shown in Table 1. In the following we illustrate how software emulation can be realized with CoRQ, realization of the stalling model is detailed in [16].

A reconfiguration of two accelerators using software emulation (executing software in parallel, see bottom timeline of Fig. 3) is realized using commands as shown in Listing 1. The CPU proceeds executing software after issuing the commands to CoRQ, thus reconfiguration is performed in parallel to execution on the CPU. To be able to guarantee the reconfiguration delay, it needs to be ensured that no earlier reconfiguration requests are still pending and occupy the reconfiguration port (see Fig. 8). 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 CI has become available (each CI only uses one accelerator in this simplified example). Then, the CI can immediately be used once it is configured (see Fig. 3 (bottom)), without waiting for the whole set of commands to have finished processing by CoRQ or executing a software handler to manage CI availability. A second CI is configured in Lines 5 and 6 of the example.

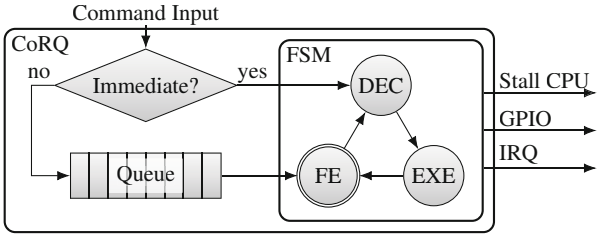

Fig. 9 High-level view of how CoRQ processes commands

The example illustrates how software emulation can be realized using CoRQ with simple sequences of commands issued by the CPU. In the following, the command execution and timing behavior of commands are detailed.

4.1.2 Command Execution

Commands are processed by CoRQ using a finite state machine (FSM) consisting of three states: fetch from queue (FE), decode (DEC), and execute (EXE) (see Fig. 9). Fetching a command takes a single cycle, the DEC state takes two cycles, and the latency of EXE depends on the command (see Table 1). Immediate commands control CoRQ itself, and thus have priority over commands from the queue. 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.

4.1.3 Guaranteed Reconfiguration Delay

CoRQ can 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. 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) such that one word of configuration data can be fed to the reconfiguration port in each cycle. Thus, CoRQ utilizes the configuration port's full bandwidth (see Sect. 4.1.4). Additionally, the `configBitsInt` command requires five setup cycles and a single cycle at completion. Let B denote the size of the bitstream in bytes (see Table 1), then $\text{latency}_{\text{EXE}} = 6 + \lceil B/4 \rceil$ cycles. 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*.

Table 2 Resource utilization

	LUTs	Flip-flops	BRAM
LEON3 CPU (standard config.)	8144	3450	14
CoRQ	398	546	1
Internal Mem. of CoRQ (384 KB)	233	6	96
Available on VC707	303,600	607,200	1030

In the example of Sect. 4.1.1, the latency of the command sequence is simply the sum of the latencies of the queueable commands: Configuring two bitstreams using software emulation (see Listing 1 and Fig. 3 (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 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.

4.1.4 Results

The resource utilization of CoRQ, when added to the default Gaisler LEON3 design (GRLIB GPL 1.4.1) targeting the Xilinx VC707 board, is shown in Table 2. The design instantiates a single LEON3, uses the DDR3 on the VC707 as main memory, and runs at 100 MHz.

The reconfiguration 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 MiB/s.[6] In the following, 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[7] which corresponds to a reconfiguration bandwidth of 381.22 MiB/s. Thus, CoRQ is only 0.066% (or 236 cycles) slower than the theoretical maximum.

Figure 10 shows the reconfiguration bandwidth results as measured by the CPU. 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, whereas "polling" means that the CPU repeatedly reads CoRQ's status register to check whether reconfiguration has completed (producing traffic on the AHB). "Bus conflicts" uses a simple DMA unit that repeatedly initiates maximum length (256 words) AHB burst transactions to provoke system bus and main memory conflicts during

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

[7] Sum of latencies of the individual commands: $4 + 25 \cdot (9 + \lceil 57{,}248/4\rceil) + 4 + 3$ cycles.

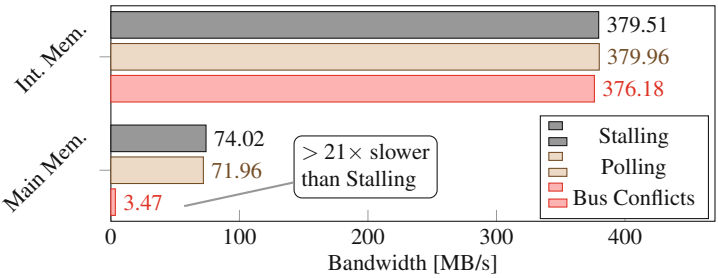

Fig. 10 A high variance in reconfiguration bandwidth is revealed when using main memory (measured by CPU, average of 50 measures, maximum error <1%)

reconfiguration. The small variance in measurements when using CoRQ-internal memory (<1%) stems from the overhead of measuring. CoRQ's commands itself always have exactly the same latency when using internal memory.

When using main memory for reconfiguration, accesses from the CPU, the DMA, and CoRQ are in conflict. This results in a strong variance in reconfiguration bandwidth between the measurements: The measurement under DMA bus conflicts 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× in reconfiguration bandwidth.

This section demonstrated how reconfiguration delay guarantees can be achieved to enable runtime reconfiguration in real-time systems using CoRQ. In another work, CoRQ formed the basis to design a reconfiguration controller that enables preemptable runtime reconfiguration in Xilinx Zynq-based multi-priority real-time systems [36]. Once reconfiguration delay guarantees are established, the following section shows how WCET estimates are obtained for tasks that leverage runtime-reconfigurable accelerators for predictable performance.

4.2 Worst-Case Execution Time Analysis

To obtain a safe worst-case execution time (WCET) estimate, timing analysis needs to be performed on the reconstructed control-flow graph (CFG) of the application binary [43]. The analysis of WCET estimates in the presence of CIs that are reconfigured using software emulation (as explained in Sect. 1.1.2, see Fig. 3) is achieved as follows. Reconfiguration of accelerators that speed up an upcoming kernel is initiated using CoRQ's reconfiguration commands (see Sect. 4.1) in a basic block immediately before entering the respective kernel. Analysis of this basic block yields the guaranteed reconfiguration delay per CI. Stalling the execution for the whole reconfiguration delay of all CIs and only then entering the kernel was mentioned in the previous section as a simple technique to perform analyzable

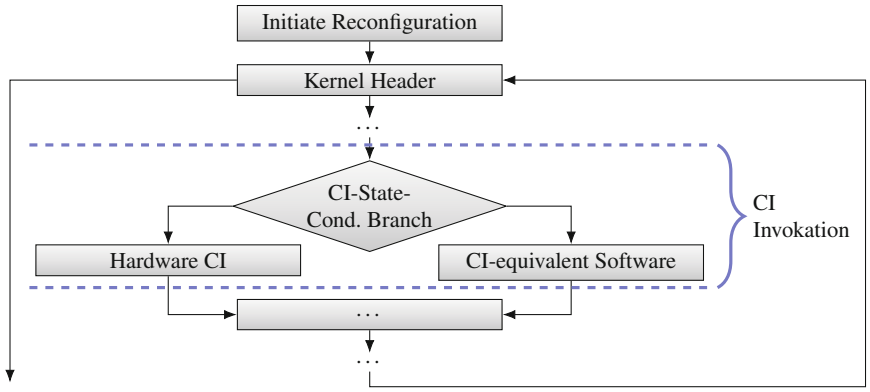

Fig. 11 *i*-Core kernel model for static timing analysis in the presence of runtime reconfiguration of custom instructions

reconfiguration. In this case, however, the reconfiguration delay (of several milliseconds) is often not amortized, especially in applications that switch between multiple kernels. Instead of stalling, this section presents a worst-case analysis that enables the RISC pipeline of the *i*-Core to be used to execute a software emulation of the CIs during the reconfiguration delay in hard real-time systems. This is achieved by (1) the reconfiguration controller presented in Sect. 4.1 and (2) by worst-case analysis of the *CI Invocation* construct shown in Fig. 11. The CI Invocation construct introduces a conditional branch that executes either the CI on the fabric or functionally equivalent cISA code on the processor pipeline.

CI Invocations realize software emulation in an analyzable way: First, execution of a kernel starts in software only (without accelerators) and at some point in time (during some iteration) accelerators finish their configuration and succeeding kernel iterations are sped up. An example of this process is shown in the timeline on Fig. 3 (bottom) for a kernel with n iterations utilizing two CIs. When entering the kernel, all CI Invocations of the first iterations are executed using cISA code. Reconfiguring the accelerators required by CI_1 finishes during iteration $i_1 - 1$. Beginning with iteration i_1, CI Invocations of CI_1 use accelerators on the fabric and benefit from a much lower runtime per iteration. In parallel, reconfiguration proceeds. During iteration $i_2 - 1$ all accelerators for CI_2 become available. Beginning with iteration i_2, the remaining iterations of the kernel are sped up even more as additionally to CI_1, all CI Invocations of CI_2 are now executed in hardware.

4.2.1 Timing Anomaly of Runtime-Reconfigurable Systems

The challenge in obtaining a precise WCET estimate is to statically determine the worst-case iteration i at which a CI can be guaranteed to be readily configured in hardware. A safe, but imprecise execution time bound can be obtained by assuming

that no CI ever finishes configuration and all CI Invocations always branch to the unaccelerated cISA code. While this would result in speedups at runtime, the WCET bound would be as high as not using accelerators at all. To obtain a safe and precise bound yielding speedups, the worst-case iteration i_k, in which reconfiguration finishes and the CI Invocation utilizes the *i*-Core fabric to execute the CI, needs to be determined statically for each CI_k. We do so by determining execution time bounds for all basic blocks in a kernel with existing timing analysis tools extended by analysis for CI latencies (we use the commercial AbsInt aiT [2] and the open-source OTAWA [5] timing analyzers in our evaluations). Once the time bounds for all basic blocks are known, the time bounds for one iteration of the whole kernel can be determined. These time bounds depend on whether specific CI Invocations branch to the CI or equivalent software. Starting with a time bound of WCET_1 for an iteration with cISA code only and a reconfiguration delay of r_1 for CI_1, Fig. 3 (bottom) seems to suggest that i_1 can be determined as: $\lceil r_1/\text{WCET}_1 \rceil + 1$, i.e., CI_1 is unavailable for $\lceil r_1/\text{WCET}_1 \rceil$ iterations and in the following iteration we can assume it to be available. It turns out, however, that *executing every iteration of the kernel in worst-case time during reconfiguration does not necessarily result in the worst-case value for* i_1 as the following example demonstrates.

The timeline in Fig. 12 shows an example for a kernel executing 6 iterations and configuring a single CI. Figure 12a shows a possible runtime behavior of the

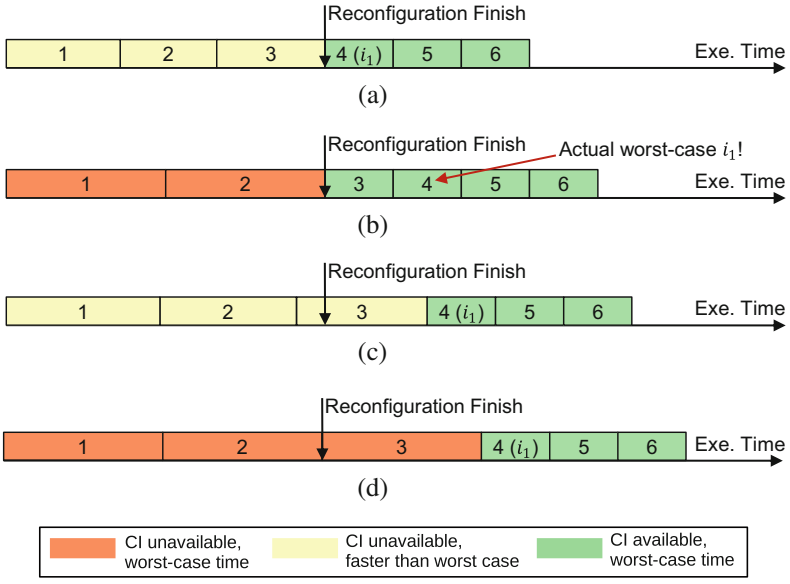

Fig. 12 Different cases for execution times of kernel iterations before the CI becomes available. (**a**) Faster than worst case, assuming no iterations overlap reconfiguration finish. (**b**) Worst case, assuming no iterations overlap reconfiguration finish. (**c**) Faster than WCET before reconfiguration finish. Iteration overlapping reconfiguration finish leads to extended execution time (*timing anomaly*). (**d**) Applied case for safe WCET bounds

kernel, where iterations 1–3 are each executed faster than worst-case time (CI not yet available) and iterations 4–6 execute in worst-case time (CI available, because reconfiguration finished). To obtain the worst-case execution for the whole kernel, it seems intuitive to assume all iterations before the reconfiguration finishes to execute in WCET_1 (software emulation of CI_1) and in WCET_2 (CI_1 available in hardware) after the reconfiguration finishes (Fig. 12b). However, when executing slightly faster iterations than WCET_1 before the reconfiguration finishes, we end up with the execution sequence of Fig. 12c: Iteration 3 cannot benefit from CI_1 and delays following iterations, because it "overlaps" the reconfiguration finish. This *timing anomaly* of runtime reconfiguration was first discovered and safely bounded in [17], it was shown that i_1 (the worst-case iteration in which CI_1 is guaranteed to be available) is determined as:

$$i_1 = \lceil r_1 / \mathrm{WCET}_1 \rceil + 2$$

That is, one additional iteration needs to be accounted for in which the reconfiguration might not yet have finished (Fig. 12d). Once i_k is determined for each CI_k, constraints are generated for the worst-case path analysis approach implicit path enumeration technique (IPET) [32]. [17] provides a detailed explanation of how constraints are generated for the IPET for multiple CIs with support for nested loops, conditional execution, and multi-context analysis (supporting, e.g., caches).

4.2.2 Results

We evaluated the timing analysis approach on *i*-Core by generating constraints for AbsInt aiT. aiT is closed-source software, thus, CI support could not be directly integrated. Instead, every CI opcode in the binary is substituted by an ADD opcode and a constraint in aiT's AIS2 constraint language to set the delay for the new ADD instruction to the delay of the specific CI. aiT outputs an XML report, which is parsed to determine every i_k for every kernel and generate the constraints described in the previous section. Our generated constraints are then used to calculate the final WCET bound using aiT.

We evaluated our analysis with an H.264 encoder application which uses 9 CIs covering the most compute-intensive kernels. In the following, we show results for the loop filter kernel for the stalling model and software emulation. Observed worst-case execution time results are obtained using our SystemC-based cycle-accurate simulator of *i*-Core.

For the analysis we use results obtained from performing timing analysis on a binary that executes the loop filter kernel of H.264 on 99 macroblocks (QCIF resolution). The loop filter is the kernel of lowest complexity in the H.264 encoder, it contains a single CI (in-loop deblocking edge filter on 4 pixels) and allows detailed analysis of worst-case CI availability. The guaranteed time bounds are compared to results obtained by executing the same binary in our simulator. f_{fabric} stays constant at 100 MHz and we choose multiples of it for f_{CPU} which resemble realistic setups

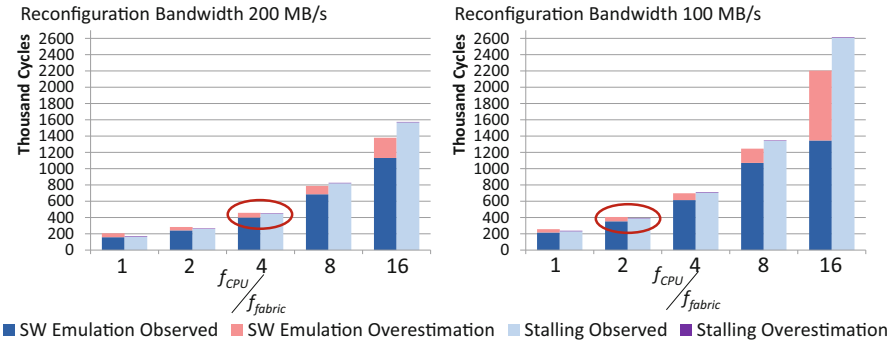

Fig. 13 Observed runtimes and guaranteed WCET bounds (*in cycles*) for loop filter. Fabric frequency $f_{\text{fabric}} = 100\text{MHz}$. Red circle marks the point from which software emulation is beneficial over stalling. Software only WCET bound is >4.5 million cycles

(rounded to the next power of two). For example, the LEON3 processor, which the *i*-Core implementation is based on, is advertised as running at 400 MHz when implemented as an ASIC, its successor the LEON4 is advertised running at 1500 MHz. The commercially available Xilinx Zynq-7000 SoC couples an ARM Cortex A9 at 866 MHz with a Xilinx 7-Series reconfigurable fabric.

All results shown in Fig. 13 are measured in *cycles of the CPU pipeline*. The left and right graph show the results for a reconfiguration bandwidth of 200MB/s and 100MB/s, respectively. When increasing the minimum evaluated CPU pipeline frequency of 100 MHz by a factor of c for a fixed f_{fabric} (x-axis), the runtime benefit of hardware CIs compared to software emulation decreases. Thus, the execution time in CPU cycles increases. In the observed runtime, software emulation is always beneficial over stalling. However, the WCET overestimation (execution time difference of WCET over observed runtime) is higher for software emulation than for stalling. As a result, stalling is beneficial over software emulation for $f_{\text{CPU}}/f_{\text{fabric}} \in \{1, 2\}$ at a reconfiguration bandwidth of 200MB/s, as well as $f_{\text{CPU}}/f_{\text{fabric}} = 1$ at a reconfiguration bandwidth of 100MB/s for obtaining a low WCET bound. In our experiments we observed that software emulation benefits from slow reconfiguration bandwidths or high CPU frequencies.

This section detailed how WCET estimates are obtained for tasks utilizing runtime-reconfigurable processors like *i*-Core for a given selection of CIs. In the following section we present how WCET-optimizing CIs are selected for a constrained reconfigurable area.

4.3 Worst-Case Execution Time Optimization

This section presents an approach of *selecting WCET-optimizing sets of CIs* for computational kernels that seamlessly integrates into state-of-the-art timing

analysis. The approach does not target the reduction of overestimation of a task's WCET bound or resolving the problem of timing anomalies, but to statically select subsets from a bigger set of possible CI implementations for a constrained reconfigurable area, with the aim of optimizing the task's WCET bound. The main problem in selecting WCET-optimizing CIs is *the instability of the worst-case path*, i.e., when reducing the latency of the worst-case path by inserting a CI, a completely different path can become the new worst-case path. Therefore, WCET bound estimation is an integral part of WCET-optimizing CI selection. The problem bears resemblance to other static optimizations targeting the worst-case path like instruction cache locking or scratchpad memory allocation of program code, but requires new models [15]. CI selection, also referred to as *instruction set selection*, is the second of the two main steps in the so-called *instruction set extension problem* [22]. The first step is the *CI generation* that is performed when compiling the application source code. In this step, kernels are identified in the application and partitioned into segments of code to execute in software and segments to execute using hardware accelerators. For the segments to execute in hardware, several alternatives that differ in resource demands as well as latencies are generated and then synthesized into configurations for the reconfigurable fabric (see Fig. 2). Which CIs are implemented in hardware instead of the original software code and how many reconfigurable containers to allocate for accelerators of a respective CI are determined by the CI selection according to an optimization goal, e.g., average-case performance. Several approaches to CI generation exist that can provide CIs and implementation alternatives as input to CI selection [22]. Different from existing CI selection approaches targeting average-case performance, WCET-optimizing selection requires the application binary, as it is the only way to be able to obtain precise WCET bound estimates (see Sect. 4.2). To obtain a finished binary with generated CIs while keeping the flexibility to execute the original software, we introduced the *CI Invocation* construct in Sect. 4.2 (see Fig. 11). CI Invocations are further extended to *CI super blocks* that allow multiple choices of hardware implementations for a CI (instead of just the binary choice between hardware and software).

4.3.1 CI Super Blocks and Optimization Goal

CI super blocks are a concept used to enable static WCET optimization. As shown in Fig. 14, CI super blocks begin with a conditional before every CI which jumps to the functionally equivalent software code when the CI is not implemented in hardware. This way, their implementation in an application behaves just like the CI Invocation construct (see Sect. 4.2). During WCET optimization, however, CI super blocks capture all the information about implementation alternatives for the respective CI that should be invoked. For each implementation j of CI k that is

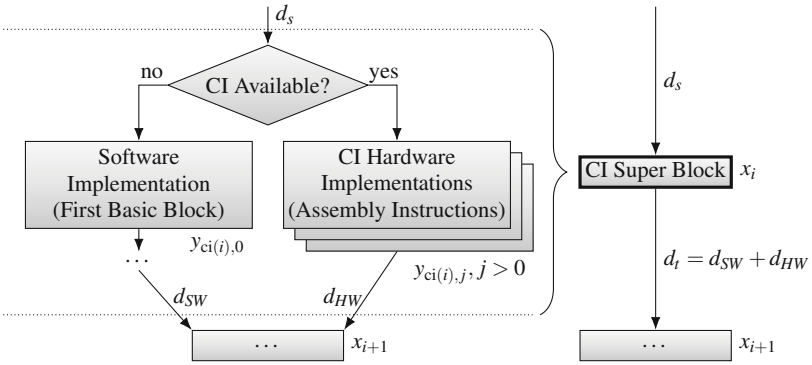

Fig. 14 CI super block as part of a CFG

invoked by a CI super block i, the following information are stored:

- reconfiguration delay $r_{k,j}$
- worst-case execution time $e_{i,j}$
- reconfigurable container demands $a_{k,j}$ (area).

By convention, $j = 0$ denotes the software implementation of a CI k, thus $r_{k,0} = 0$ and $a_{k,0} = 0$. Effectively, we obtain a CFG that is parameterized by the chosen implementation for each CI using CI super blocks. Thus, the result of the selection is a matrix $y \in \{0,1\}^{|\mathcal{CI}| \times M}$ that maps each CI $k \in |\mathcal{CI}|$ to an implementation $j \in M$, where $|\mathcal{CI}|$ is the total number of CIs and M the maximum number of implementations per CI, respectively. The worst-case path analysis IPET [32] formulates an ILP objective function over basic blocks i with constant execution times c_i that finds the execution counts x_i of the respective basic block that lead to the maximum execution time ($\max_x \sum_i c_i x_i$). On top of that, the WCET-optimizing selection needs to find the selection y that minimizes the maximum execution time:

$$\min_{y} \left(\max_{x} \left(\underbrace{\sum_{i \in \mathrm{BB}} c_i x_i}_{\text{Basic Blocks (BB)}} + \underbrace{\sum_{i \in \mathrm{SB}} \sum_{j} e_{i,j}\, y_{\mathrm{ci}(i),j}\, x_i}_{\text{CI Super Blocks (SB)}} \right) + \underbrace{\sum_{k} \sum_{j} y_{k,j} r_{k,j}}_{\text{Reconfiguration Delay}} \right) \quad (1)$$

where $\mathrm{ci}(i)$ maps a CI super block i to the CI it invokes. Similar to flow networks, IPET formulates ILP constraints on the variables (x_i) that model the control flow and capture relative execution counts of basic blocks. One of the additional constraints for WCET-optimizing CI selection is that the total number of reconfigurable containers A must not be exceeded by the selection y ($\sum_k \sum_j a_{k,j} y_{k,j} \leq A$). Even if the reconfigurable area was infinitely large, it is not necessarily beneficial to select

the highest-performance implementation for each CI: it might also incur the most reconfiguration delay.

The objective function (Eq. (1)) and its constraints formulate an NP-hard combinatorial optimization problem (already IPET is NP-hard). In the following, we discuss approaches that solve this problem optimally and heuristically.

4.3.2 Solving WCET-Optimizing CI Selection

An optimal solution to the WCET-optimizing CI selection can be obtained using a branch and bound algorithm that generates all possible selections of CI implementations and prunes branches that do not fit onto the reconfigurable area [15]. The problem with this approach is that the WCET of each possible selection needs to be evaluated. The number of possible selections grows exponentially in the number of reconfigurable containers A of the reconfigurable fabric and the number of CIs used in the task under optimization. Thus, we designed a greedy algorithm that proceeds as follows:

1. Start with an empty reconfigurable area (choose the software implementation $j = 0$ for all CIs)
2. Obtain a WCET estimate for the current selection y
3. For each CI k, calculate the profit on *current* worst-case path of "upgrading" from selected implementation j to the next implementation $j+1$ in the ascending order of reconfigurable container demand $a_{k,j}$ as:

$$\text{profit}(k, j+1, x) := \underbrace{\sum_{\substack{i \in \text{SB} \\ \text{ci}(i)=k}} (e_{i,j} - e_{i,j+1}) x_i}_{\text{latency reduction on current worst} - \text{case path x}} - \underbrace{(r_{k,j+1} - r_{k,j})}_{\text{additional reconfiguration cost}} \quad (2)$$

4. If exists, select upgrade $j + 1$ (instead of j) for CI k with highest positive profit (increasing allocated reconfigurable containers by at least one, possibly changing the worst-case path) and go to 2, terminate otherwise

Instead of requiring a number of WCET estimates that grows exponentially in the number of accelerators that fit onto the reconfigurable fabric A and the number of CIs used in the task under optimization, this greedy algorithm performs at most A WCET estimates: After each estimate (2) at least one accelerator is added to the selection (4) until all reconfigurable containers are occupied or no further candidate for upgrading exists. In the following we will show how this greedy algorithm compares to the optimal solution.

4.3.3 Results

While our timing analysis of tasks on reconfigurable processors as detailed in Sect. 4.2 could be evaluated using the commercial timing analyzer AbsInt aiT [2], we extended WCET analysis as an integral part of WCET optimization in this section. Therefore, the optimal branch and bound as well as heuristic selection algorithms were implemented "processors" within the open-source WCET estimation framework OTAWA [5]. We extended the existing analysis support for the LEON3 CPU in OTAWA to support CI opcodes, CI super blocks with configuration-dependent latency, and reconfiguration delay. In the following, results are shown for a reconfiguration bandwidth of 400 MB/s (maximum bandwidth, see Sect. 4.1), running the CPU at 400 MHz (which the LEON3 processor is advertised as running at when implemented as an ASIC) and the reconfigurable fabric at 100 MHz (which corresponds to the current *i*-Core design on Xilinx Virtex-7).

Figure 15 shows the runtime of our optimization approaches when optimizing the most complex kernel of the H.264 encoder "encode macroblock." The kernel uses six different CIs for which the algorithms need to select WCET-optimizing implementations that fit onto the reconfigurable area. Results are shown for different area sizes (number of accelerators that fit onto the reconfigurable fabric). As can be seen in the graph, even when no area is available for selecting CI implementations (and no candidates but a complete software evaluation are evaluated), the runtime is almost 5 s for both approaches. The reason is that reconstructing the CFG from the binary and analyzing all basic blocks already takes around 4.6 s. The runtime of the optimal algorithm first increases exponentially when the amount of area is increased, but then flattens out because branch and bound finds more opportunities to prune the search space. The runtime of the greedy algorithm seems to stay constant. It rises slightly, however, until an area of 10 accelerators. The amount of WCET estimates that need to be performed to find the final result grows linearly in the available area.

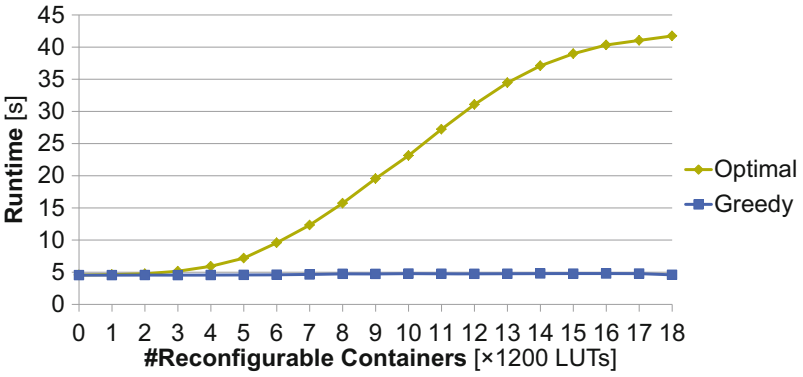

Fig. 15 Runtime of WCET-optimizing CI selection when optimizing H.264's encode macroblock kernel, which invokes six different CIs

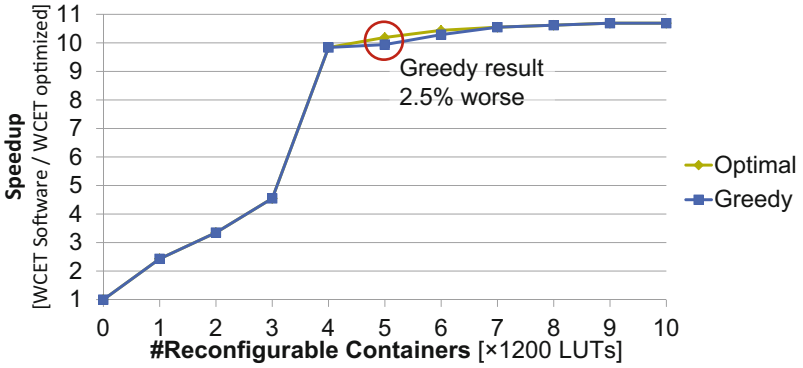

Fig. 16 Resulting speedup of WCET-optimizing CI selection when optimizing H.264's encode macroblock kernel compared to executing without acceleration

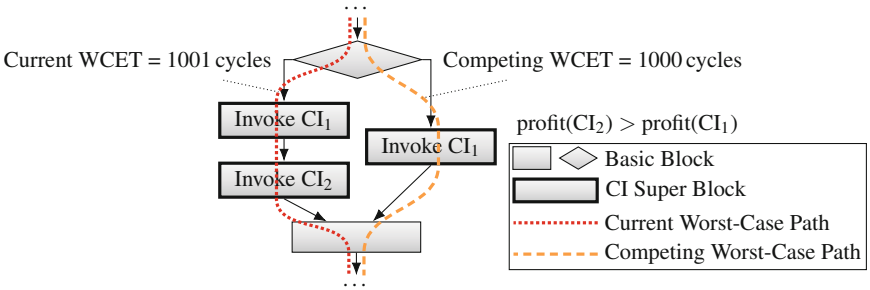

Fig. 17 Greedily selecting the CI with biggest profit on *current* worst-case path (CI_2) is not always optimal. The worst-case path changes during optimization

On average, an additional estimate adds only 40ms to the algorithm's runtime. Once the area reaches 10, there are still CIs that could be upgraded but the profit (Eq. (2)) becomes negative, because the additional reconfiguration delay increase is bigger than the latency reduction of the current worst-case path. Therefore, the runtime does not increase any further.

Figure 16 shows that the speedup results of the CI selections obtained by the greedy algorithm compared to execution without any accelerators are on par with the optimal solution for most cases. However, for an area of 5 and 6 accelerators the speedup obtained by greedy is up to 2.5% worse than the speedup of the optimal solution. Figure 17 shows a simplified example of the case in which greedy performs suboptimal. In this example, the current worst-case path (left path through the CFG) contains two CI super blocks that could be upgraded. CI_2 provides a bigger profit (see Eq. (2)) than CI_1, therefore, greedy will select CI_2. However, no matter how big the profit of CI_2 actually is on the current worst-case path the WCET will only be reduced by a single cycle, because the right path through the CFG will immediately become the new worst-case path and reduce the WCET from 1001

cycles to 1000 cycles. The optimal solution would have been to select CI_1. Even though it provides less profit for the current worst-case path, it would also reduce the competing worst-case path and thus provide a bigger benefit for the total WCET. The greedy algorithm cannot make this choice, because it is impossible to foresee the next worst-case path after optimizing the execution time of the current worst-case path.

This section presented WCET optimization approaches that selected CI implementations to optimize worst-case paths and it concludes our descriptions of execution on *i*-Core under real-time guarantees. So far, we described *i*-Core as a single core of a system on chip. In the following section, we present how the runtime reconfigurability of *i*-Core can be leveraged in a multi-core system.

5 Runtime-Reconfiguration in Multi-Core Systems

The resulting speedup from executing a CI using hardware accelerators, instead of executing the equivalent cISA implementation, depends on the inherent parallelism of the accelerated computations and the amount of reconfigurable accelerators used for their implementation. For achieving its highest performance and exploiting its full inherent parallelism, e.g., JPEG encoding requires approximately 5× as much reconfigurable area as SHA cryptographic hashing [31]. When designing a reconfigurable system such that the JPEG and SHA CIs require 100% and 20% of the fabric area, respectively, and either one or the other CI is used for the same amount of time, the system will only utilize 60% of its fabric area on average. In other words, 40% of the reconfigurable area could be utilized for other computations, e.g., executing multiple CIs *concurrently*. In the previous sections, the presented SoC contained *i*-Core as the only processor. To create opportunities to execute multiple CIs in parallel, general-purpose processors (GPPs) are added to the SoC. The non-reconfigurable GPPs are allowed to offload compute-intensive calculations onto the reconfigurable area. In the following, an overview is provided on how this can be achieved.

5.1 COREFAB: Concurrent Reconfigurable Fabric Utilization

To exploit the opportunity of increased accelerator utilization and enable acceleration for GPPs that reside in the same SoC as *i*-Core, we introduced COREFAB in [23]. COREFAB describes hardware components and a protocol that enable the GPPs in the SoC to utilize the reconfigurable fabric of the *i*-Core via the fabric access manager (FAM) as shown in Fig. 18. Most remarkably, COREFAB enables the *concurrent* execution of two (different) CIs issued by (1) the *i*-Core and (2) any of the GPPs in the SoC. In the following, CIs issued by the *i*-Core are named “primary CIs” and those from the GPPs are named “remote CIs.” From the

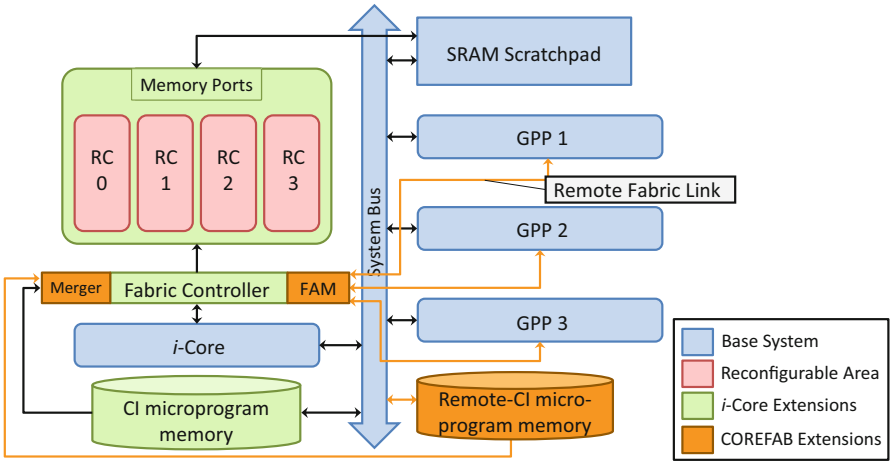

Fig. 18 COREFAB defines hardware extensions and a protocol that enable general-purpose processors to execute CIs on the *i*-Core's reconfigurable fabric

application developer's view, primary CIs and remote CIs appear identical, but the latency to issue a remote CI is higher and primary CIs are preferred over remote CIs during concurrent execution. Remote CIs use a dedicated "remote-CI microprogram memory" (to store microprograms used by GPPs) that can be accessed in parallel to the "CI microprogram memory" that stores the primary CIs (see Fig. 18). If a primary CI and a remote CI execute in parallel, then the "CI merger" analyzes whether or not the microoperations of the two CIs conflict in their resource usage (e.g., accelerators and communication lines between accelerators) in every control step. In case of no conflict, the control steps of both CIs are merged *on-the-fly* to allow parallel execution. Otherwise, the remote CI is stalled until the next control step, while the primary CI proceeds without delay.

To reduce the likelihood that primary and remote CI conflict, we developed an online binding that generates the microprograms of the primary/remote CIs in such a way that they use distinct computation and communication resources if possible [25]. In our evaluation, we compared COREFAB with the state-of-the-art reconfigurable multi-core systems. They only allow for exclusive access to the reconfigurable fabric, either by time multiplexing (CIs cannot execute in parallel) [14] or by spatially partitioning the fabric area into private regions (CIs are inflexible in that they can never use the entire reconfigurable fabric) [42]. COREFAB combines the flexibility of a CI being able to use the entire fabric and the parallel execution of primary and remote CIs. As shown in Fig. 19, COREFAB improves the performance of a SoC consisting of three GPPs and an *i*-Core by 1.3× on average compared to time multiplexing (like [14]). This improvement is achieved without reducing the performance of the *i*-Core itself. Spatial partitioning (like [42]) leads to a 2% better performance of the GPPs compared to COREFAB; however, this comes at the cost of more than 3× lower performance of the *i*-Core itself. In

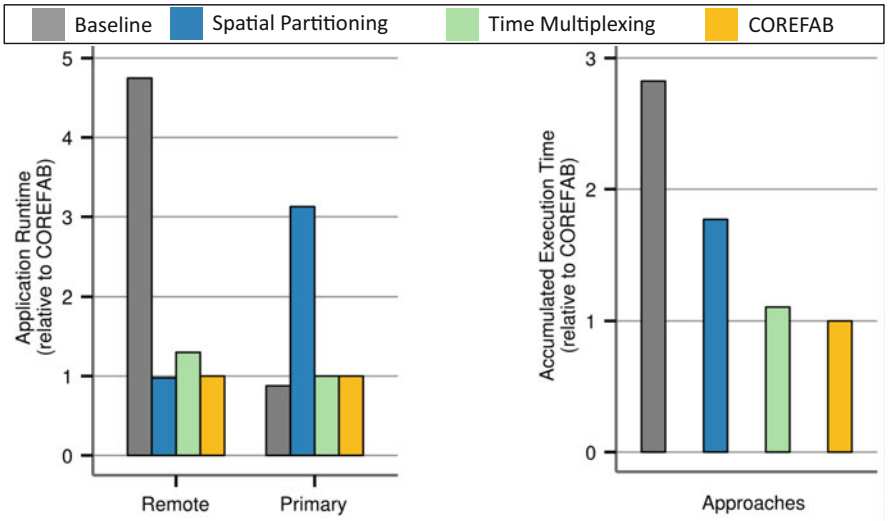

Fig. 19 Runtime results comparing state-of-the-art approaches to COREFAB. Baseline: a reconfigurable processor and three GPPs that cannot execute CIs

summary, COREFAB utilizes the reconfigurable resources more effectively than state-of-the-art approaches to optimize the performance of reconfigurable multi-core SoCs.

6 Conclusion

This chapter presented an overview of how non-functional requirements of cyber-physical systems are addressed by the runtime-reconfigurable processor platform *i*-Core. The benefits of a very tight integration of reconfigurable fabric and CPU (*i*-Core attaches an FPGA-based reconfigurable fabric directly to the CPU pipeline, see Sect. 1) were shown for several non-functional requirements. Not only were increases in performance shown by reconfiguration-aware task scheduling (Sect. 2) and providing access to *i*-Core's accelerators to general-purpose processors in the same SoC (Sect. 5), but also improvements in reliability (Sect. 3) and worst-case execution time guarantees (Sect. 4). In summary, our research in the context of *i*-Core leverages a processor with a runtime-reconfigurable instruction set to provide a holistic approach that targets the requirements of cyber-physical systems. Our future work focuses on security aspects of cyber-physical systems, where early results have shown that runtime reconfiguration is an effective countermeasure to side-channel attacks.

Acknowledgements This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89) and as part of the priority program "Dependable Embedded Systems" (SPP 1500). Parts of the research summarized in this chapter were conducted together with Dr. Artjom Grudnitsky and Dr. Hongyan Zhang who earned their PhDs under the supervision of Prof. Dr. Jörg Henkel in 2015 and 2017, respectively. We want to thank them for fruitful discussions.

References

1. Abdelfattah, M.S., Bauer, L., Braun, C., Imhof, M.E., Kochte, M.A., Zhang, H., Henkel, J., Wunderlich, H.J.: Transparent structural online test for reconfigurable systems. In: IEEE International On-Line Testing Symposium (IOLTS), pp. 37–42 (2012)
2. AbsInt: aiT Worst-Case Execution Time Analyzers. Website: http://www.absint.com/ait/ (2018). [Online; accessed 10-Aug-2018]
3. Achronix: Speedchip FPGA chiplets. Website: https://www.achronix.com/product/speedchip/ (2018). [Online; accessed 10-Aug-2018]
4. Axer, P., Ernst, R., Falk, H., Girault, A., Grund, D., Guan, N., Jonsson, B., Marwedel, P., Reineke, J., Rochange, C., Sebastian, M., Hanxleden, R.V., Wilhelm, R., Yi, W.: Building timing predictable embedded systems. ACM Trans. on Embed. Comput. Syst. **13**(4), 82:1–82:37 (2014)
5. Ballabriga, C., Cassé, H., Rochange, C., Sainrat, P.: OTAWA: An open toolbox for adaptive WCET analysis. In: IFIP International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS), pp. 35–46. Springer (2010)
6. Bauer, L., Braun, C., Imhof, M.E., Kochte, M.A., Schneider, E., Zhang, H., Henkel, J., Wunderlich, H.J.: Test strategies for reliable runtime reconfigurable architectures. IEEE Transactions on Computers (TC), Special Section on Adaptive Hardware and Systems **62**(8), 1494–1507 (2013)
7. Bauer, L., Grudnitsky, A., Shafique, M., Henkel, J.: PATS: a performance aware task scheduler for runtime reconfigurable processors. In: Field-Programmable Custom Computing Machines (FCCM), pp. 208–215 (2012)
8. Bauer, L., Henkel, J.: Run-time Adaptation for Reconfigurable Embedded Processors. Springer Science+Business Media, LLC (2011)
9. Bauer, L., Shafique, M., Henkel, J.: A computation-and communication-infrastructure for modular special instructions in a dynamically reconfigurable processor. In: Int. Conf. on Field Programmable Logic and Applications, pp. 203–208 (2008)
10. Bauer, L., Shafique, M., Henkel, J.: Cross-architectural design space exploration tool for reconfigurable processors. In: Conference on Design, Automation and Test in Europe (DATE), pp. 958–963 (2009)
11. Bauer, L., Shafique, M., Henkel, J.: Concepts, architectures, and run-time systems for efficient and adaptive reconfigurable processors. In: NASA/ESA Conference on Adaptive Hardware and Systems (AHS), pp. 80–87 (2011)
12. Benkrid, K., Liu, Y., Benkrid, A.: A highly parameterized and efficient FPGA-based skeleton for pairwise biological sequence alignment. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **17**(4), 561–570 (2009)
13. Biondi, A., Balsini, A., Pagani, M., Rossi, E., Marinoni, M., Buttazzo, G.: A framework for supporting real-time applications on dynamic reconfigurable FPGAs. In: Real-Time Syst. Symp., pp. 1–12 (2016)
14. Chen, L., Mitra, T.: Shared reconfigurable fabric for multi-core customization. In: Design Automation Conference (DAC), pp. 830–835 (2011)
15. Damschen, M., Bauer, L., Henkel, J.: Extending the WCET problem to optimize for runtime-reconfigurable processors. ACM Trans. on Archit. and Code Optim. **13**(4), 45:1–45:24 (2016)

16. Damschen, M., Bauer, L., Henkel, J.: CoRQ: Enabling runtime reconfiguration under WCET guarantees for real-time systems. IEEE Embedded Systems Letters **9**(3), 77–80 (2017)
17. Damschen, M., Bauer, L., Henkel, J.: Timing analysis of tasks on runtime reconfigurable processors. IEEE Trans. on Very Large Scale Integration Syst. **25**(1), 294–307 (2017)
18. Das, A., Nguyen, D., Zambreno, J., Memik, G., Choudhary, A.: An FPGA-based network intrusion detection architecture. IEEE Transactions on Information Forensics and Security **3**(1), 118–132 (2008)
19. De Schryver, C., Shcherbakov, I., Kienle, F., Wehn, N., Marxen, H., Kostiuk, A., Korn, R.: An energy efficient FPGA accelerator for Monte Carlo option pricing with the Heston model. In: International Conference on Reconfigurable Computing and FPGAs (ReConFig), pp. 468–474. IEEE (2011)
20. Dennl, C., Ziener, D., Teich, J.: On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library. In: IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 45–52 (2012)
21. Edwards, S.A., Lee, E.A.: The case for the precision timed (PRET) machine. In: Proc. of Design Automat. Conf., pp. 264–265. ACM (2007)
22. Galuzzi, C., Bertels, K.: The instruction-set extension problem: A survey. ACM Trans. on Reconfig. Technol. and Syst. **4**(2), 18 (2011)
23. Grudnitsky, A., Bauer, L., Henkel, J.: COREFAB: concurrent reconfigurable fabric utilization in heterogeneous multi-core systems. In: Int. Conf. on Compilers, Architecture and Synthesis for Embed. Syst. (CASES), pp. 1–10 (2014)
24. Grudnitsky, A., Bauer, L., Henkel, J.: MORP: Makespan optimization for processors with an embedded reconfigurable fabric. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), pp. 127–136 (2014)
25. Grudnitsky, A., Bauer, L., Henkel, J.: Efficient partial online synthesis of special instructions for reconfigurable processors. IEEE Transactions on Very Large Scale Integration (VLSI) Systems **25**(2), 594–607 (2017)
26. Henkel, J., Bauer, L., Becker, J., Bringmann, O., Brinkschulte, U., Chakraborty, S., Engel, M., Ernst, R., Härtig, H., Hedrich, L., Herkersdorf, A., Kapitza, R., Lohmann, D., Marwedel, P., Platzner, M., Rosenstiel, W., Schlichtmann, U., Spinczyk, O., Tahoori, M., Teich, J., Wehn, N., Wunderlich, H.J.: Design and architectures for dependable embedded systems. In: IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), pp. 365–374 (2011)
27. Henkel, J., Bauer, L., Dutt, N., Gupta, P., Nassif, S., Shafique, M., Tahoori, M., Wehn, N.: Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In: IEEE/ACM Design Automation Conference (DAC) (2013)
28. Henkel, J., Herkersdorf, A., Bauer, L., Wild, T., Hübner, M., Pujari, R.K., Grudnitsky, A., Heisswolf, J., Zaib, A., Vogel, B., et al.: Invasive manycore architectures. In: Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 193–200 (2012)
29. Hoang, T., Nguyen, V.L.: An efficient FPGA implementation of the advanced encryption standard algorithm. In: IEEE Int. Conf. on Computing Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), pp. 1–4 (2012)
30. Honegger, D., Oleynikova, H., Pollefeys, M.: Real-time and low latency embedded computer vision hardware based on a combination of FPGA and mobile CPU. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pp. 4930–4935 (2014)
31. Lam, S.K., Srikanthan, T., Clarke, C.: Selecting profitable custom instructions for area-time-efficient realization on reconfigurable architectures. IEEE Transactions on Industrial Electronics **56**(10), 3998–4005 (2009)
32. Li, Y.T.S., Malik, S.: Performance analysis of embedded software using implicit path enumeration. In: Workshop on Languages, Compilers, & Tools for Real-time Syst., pp. 88–98 (1995)
33. Lockwood, J.W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., Luo, J.: NetFPGA–an open platform for gigabit-rate network switching and routing. In: IEEE International Conference on Microelectronic Systems Education (MSE), pp. 160–161 (2007)

34. Puschner, P., Koza, C.: Calculating the maximum execution time of real-time programs. Real-Time Syst. **1**(2), 159–176 (1989)
35. Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., et al.: A reconfigurable fabric for accelerating large-scale datacenter services. ACM SIGARCH Computer Architecture News **42**(3), 13–24 (2014)
36. Rossi, E., Damschen, M., Bauer, L., Buttazzo, G., Henkel, J.: Preemption of the partial reconfiguration process to enable real-time computing with FPGAs. ACM Transactions on Reconfigurable Technology and Systems (TRETS) **11**(2), 10 (2018)
37. Sabeghi, M., Sima, V.M., Bertels, K.: Compiler assisted runtime task scheduling on a reconfigurable computer. In: Field Programmable Logic and Applications (FPL), pp. 44–50 (2009)
38. Schoeberl, M.: Time-predictable computer architecture. EURASIP Journal on Embed. Syst. pp. 2:1–2:17 (2009)
39. Tessier, R., Burleson, W.: Reconfigurable computing for digital signal processing: A survey. Journal of VLSI signal processing systems for signal, image and video technology **28**(1-2), 7–27 (2001)
40. Tessier, R., Pocek, K., DeHon, A.: Reconfigurable Computing Architectures. Proceedings of the IEEE **103**(3), 332–354 (2015)
41. Thiele, L., Wilhelm, R.: Design for timing predictability. Real-Time Syst. **28**(2-3), 157–177 (2004)
42. Watkins, M., Albonesi, D.: ReMAP: a reconfigurable heterogeneous multicore architecture. In: International Symposium on Microarchitecture (MICRO), pp. 497–508 (2010)
43. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools. ACM Trans. Embed. Comput. Syst. **7**(3), 36:1–36:53 (2008)
44. Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., Ferdinand, C.: Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. Trans. on Comput.-Aided Design of Integrated Circuits and Syst. **28**(7), 966–978 (2009)
45. Zhang, H., Bauer, L., Henkel, J.: Resource budgeting for reliability in reconfigurable architectures. In: IEEE/ACM Design Automation Conference (DAC) (2016)
46. Zhang, H., Bauer, L., Kochte, M.A., Schneider, E., Braun, C., Imhof, M.E., Wunderlich, H.J., Henkel, J.: Module diversification: Fault tolerance and aging mitigation for runtime reconfigurable architectures. In: IEEE International Test Conference (ITC), pp. 1–10 (2013)
47. Zhang, H., Bauer, L., Kochte, M.A., Schneider, E., Wunderlich, H.J., Henkel, J.: Aging resilience and fault tolerance in runtime reconfigurable architectures. IEEE Transactions on Computers (TC), Special Section on Innovation in Reconfigurable Computing Fabrics from Devices to Architectures **66**(6), 957–970 (2017)
48. Zhang, H., Kochte, M.A., Imhof, M.E., Bauer, L., Wunderlich, H.J., Henkel, J.: GUARD: Guaranteed reliability in dynamically reconfigurable systems. In: IEEE/ACM Design Automation Conference (DAC) (2014)
49. Zhang, H., Kochte, M.A., Schneider, E., Bauer, L., Wunderlich, H.J., Henkel, J.: STRAP: Stress-aware placement for aging mitigation in runtime reconfigurable architectures. In: IEEE/ACM International Conference on Computer-Aided Design (ICCAD) (2015)