Timing Analysis of Tasks on Runtime Reconfigurable Processors

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

***Abstract*— Real-time embedded systems need to be analyzable for timing guarantees. Despite significant scientific advances, however, timing analysis lags years behind current microarchitectures with out-of-order scheduling pipelines, several hardware threads, and multiple (shared) cache layers. To satisfy the increasing performance demands, analyzable performance features are required. We propose a novel timing analysis approach to introduce runtime reconfigurable instruction set processors as one way to escape the scarcity of analyzable performance while preserving the flexibility of the system. We introduce extensions to the state-of-the-art Integer linear programming (ILP)-based program path analysis for computing precise worst case time bounds in the presence of the widely used technique to continue processor execution during reconfiguration by emulating not yet reconfigured custom instructions (CIs) in software. We identify and safely bound a timing anomaly of runtime reconfiguration, where executing faster than worst case time during reconfiguration extends the execution time of the whole program. Stalling the processor during reconfiguration (easier to analyze but not state-of-the-art for reconfigurable processors) is not required in our approach. Finally, we show the precision of our analysis on a complex multimedia application with multiple reconfigurable CIs for several hardware parameters and give advice on how to deal with reconfiguration delay under timing guarantees.**

***Index Terms*— Custom instructions (CIs), real time, reconfigurable processors, timing analysis, timing anomaly, worst case execution time (WCET).**

I. Introduction

IN CONTRAST to general-purpose computing systems, embedded systems must meet nonfunctional requirements. Several application domains, e.g., automotive, avionics, or robotics, require tasks to execute under timing constraints. Failing to finish a task before a given deadline can lead to catastrophes; therefore timing validation is performed to guarantee the timing constraints [1]. As part of the timing validation, a schedulability analysis is performed to guarantee that a given task set can be scheduled at runtime under any circumstances. To perform a schedulability analysis, the worst case execution time (WCET) of every task from the task set needs to be known [1].

Manuscript received December 31, 2015; revised April 8, 2016; accepted May 11, 2016. This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

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

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2016.2572304

Statically determining the WCET of a task is a complex problem. Due to the halting problem, it is in general not possible to determine the precise WCET of a task or its worst case input [2]. Furthermore, modern processor design focuses on reducing the average execution time, which complicates the WCET analysis. Features, such as pipelining, caches, and branch prediction, introduce a microarchitectural state, i.e., the latency of an instruction is dependent on the execution history. Situations where a locally faster execution, e.g., a cache hit instead of cache miss, leads to an increase of the global execution time are the so-called timing anomalies [3]. They enforce exhaustive exploration of every possible microarchitectural state in timing analysis as assuming a local worst case, e.g., cache miss, when incomplete information about the state is available, can result in an unsafe WCET bound. This leads to a state explosion which may make the analysis infeasible in certain cases due to memory and computation time constraints [4]. Thus, abstract interpretation [5] is applied to statically obtain a guaranteed upper bound of the WCET of a task, assuming uninterrupted execution. 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.

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 [6]. Therefore, the performance features amenable for timing analysis are requested [7]–[9] to face the increasing performance demands of real-time systems. Improving the WCET of a task is highly desirable as it may enable an embedded system to meet previously infeasible timing constraints or make room for power optimizations.

To escape the scarcity of timing-analyzable performance features, we introduce the timing analysis of tasks on runtime reconfigurable processor designs, in which the core instruction set architecture (cISA) of the processor core is extended by custom instructions (CIs) which initiate the execution of kernels on the reconfigurable fabric. Fig. 1 shows a system with a reconfigurable instruction set processor (generalized from [10]–[12]). It consists of an in-order reduced instruction set computing CPU with a reconfigurable fabric, scratchpad memory (SPM), and separate data and instruction caches (D$ and I$). With commercial platforms such as the Xilinx Zynq system on chip (SoC) that couples an ARM Cortex A9 multicore with a Xilinx reconfigurable fabric on a single chip, the reconfigurable processors have become off-the-shelf devices. In contrast to these, we specifically choose

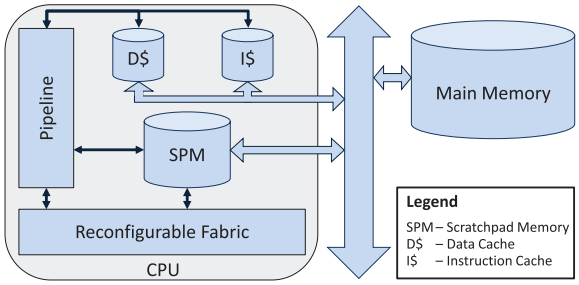

Fig. 1. SoC with a reconfigurable processor.

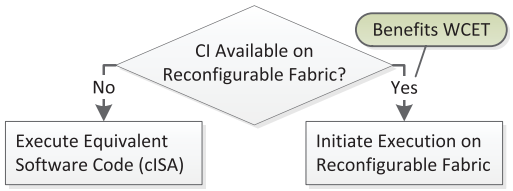

Fig. 2. Software emulation entails testing whether a specific reconfigurable CI is currently configured (available).

an in-order pipeline to be able to obtain precise execution time bounds using the state-of-the-art static timing analysis. Predictable performance is achieved with our novel models for analyzing tasks utilizing reconfigurable CIs. CIs are specified by (multicycle) datapaths, which are implemented as configurations on the reconfigurable fabric. A configured CI takes a certain area share of the fabric. The fabric can accommodate several CIs at once constrained by its total area (see [13] for an overview of area models). The time required for reconfiguring a CI at runtime depends on the size of the configuration and is called reconfiguration delay; a CI ready for execution is referred to as available. A CI that is not yet readily configured or was replaced by another CI is unavailable. In contrast to cISA instructions, a CI can be unavailable when it is due for execution. Two common approaches exist to deal with this problem in reconfigurable processors that optimize the average case on the best effort basis: stalling [14], [15], i.e., halting execution until the pending reconfiguration finishes, and software emulation [16], [17], i.e., branching to CI-equivalent software, which can be executed on the cISA (see Fig. 2).

The main contribution of this paper is a timing analysis for environments, in which faster paths (e.g., containing hardware-accelerated CIs) through a kernel body become successively available during the execution of the kernel (e.g., software emulation is utilized for unavailable CIs and reconfigurations are performed in parallel). We present a reconfigurable processor design amenable to this timing analysis and show how the resulting information can be utilized to obtain precise WCET bounds of tasks.

Our novel contributions are as follows:

1) timing analysis of tasks on a reconfigurable instruction set and comparison of measures to deal with reconfiguration delays: stalling the core pipeline or running equivalent software code until configurations finish (software emulation);
2) identification and analysis of a timing anomaly of runtime reconfiguration, i.e., a situation where executing iterations of a kernel faster than worst case time during reconfiguration can extend the execution time of the whole program;
3) identification of key requirements for a reconfigurable instruction set processor with timing guarantees and the design of a reconfiguration unit to fulfill them.

We argue that a reconfigurable instruction set gives application designers more control over the microarchitecture than a fixed instruction set, which benefits timing analysis. Our evaluation results show that with precise analysis of proven reconfigurable processor design, runtime instruction set reconfiguration can be an enabling feature to provide timing-analyzable performance.

II. Related Work

Significant amount of work on reconfigurable instruction set processors has been performed. The demonstrated benefits are code size reduction, lower power consumption, and increased average-case performance [18]. Current research in reconfigurable instruction set processors is moving toward heterogeneous reconfigurable multicore architectures [19]–[21]. However, worst case timing analysis of parallel tasks is still new ground even on general-purpose multicore architectures [6], [9].

A. WCET-Optimizing Instruction Set Architectures

Little work on instruction set adaptation has been performed with respect to WCET. In [22], an instruction set selection for WCET optimization on an Application-specific instruction-set processor is performed. This approach performs WCET estimation using timing schema [23]. Timing schema uses a syntax tree representation of the program under analysis. Inner nodes represent the control flow, and leaf nodes are the basic blocks of the program. The WCET is estimated in a bottom–up fashion with simple recursive rules. The proposed instruction set selection targets instruction set extension for applications known at design time. Reconfiguration is not considered in this approach.

Microprogrammed coarse grained reconfigurable processor (MCGREP) [24] is a two-stage pipelined, microprogrammed processor design without caches. Every instruction has a constant delay, independent of the execution history. The two arithmetic logic unit of this processor design can be reconfigured to perform an application specific instruction in every cycle using a microcode to improve instruction throughput. MCGREP's design allows a straightforward timing analysis without the requirement of complex models. However, its evaluation assumes a single-cycle load delay and compares against microprocessors at 40 MHz with their cache switched OFF. Requiring the absence of caches for timing predictability is too restrictive, as least recently used caches are well understood in timing analysis and allow a predictable processor design with memory hierarchy [1]. In addition, the two-stage pipeline may limit the frequency of the processor. The scalability of this design in performance-demanding scenarios remains questionable.

In [25], an integration of the real-time processor CarCore [26] and the MOLEN reconfigurable custom

computing unit [11] is presented. The integration focuses on guaranteeing hard real-time constraints for the memory accesses of processor core and reconfigurable hardware. Timing analysis of binaries utilizing reconfiguration is not addressed. As a consequence, measured execution times are evaluated, while the effects on WCET bounds remain uninvestigated.

B. Runtime Reconfiguration in Hard Real-Time Systems

If supported, runtime reconfiguration is the responsibility of the task scheduler in the state-of-the-art hard-real-time systems [27]–[31], but reconfiguration within a currently executing task is not considered. Reference [27] tackles the problem of scheduling access to the reconfiguration access port for fixed-priority task sets with hard deadlines. ReconOS [30] is an operating system that provides a unified programming model for threads running in software and threads mapped to reconfigurable hardware. Previous versions were based on the eCos[1] real-time kernel. Reference [31] presents mapping and scheduling of task graphs to a uniprocessor system with reconfigurable units, which minimizes the utilization of fabric area. Reconfigurations are either performed before, or as special tasks within the schedule. However, during the execution of a task, its assigned reconfigurable unit is not reconfigured.

In [28], a per-task instruction set selection is performed using reconfiguration to meet timing constraints. A periodic task graph with deadlines is scheduled and the schedule is partitioned into configurations. Each configuration is assigned an instruction set to optimize the tasks' WCET. Their approach assumes that the schedule's WCET reduction directly corresponds to the per-task cycle reduction due to a CI that is chosen for a subset of the tasks. But this is only the case when there is no conditional execution of tasks. When having alternative execution paths, adding a CI into the current WCET path may result in another path becoming the WCET path. In this case, the gain of the CI on the overall WCET is the delta between the old and new WCET paths, which could be as small as 1 cycle, independent of the average performance improvement due to the CI. So the overall WCET gain can be significantly less than the gain in the old WCET path. A similar effect was already reported in [32] when assigning a variable to SPM for WCET reduction. Due to this effect, it is not possible to apply the intertask techniques in [28] on intratask level.

In sum, the state-of-the-art techniques either do not consider runtime reconfiguration [22], introduce reconfigurable architectures without investigating effects on WCET bounds [25], or runtime reconfiguration is the responsibility of the real-time scheduler, where reconfiguration for an already running task is prohibited [27]–[31].

III. Motivational Example

As discussed in Section II, the state-of-the-art techniques only perform reconfigurations when switching from one task to another [27]–[31]. Within a single task however, the benefits of having a reconfigurable fabric is not exploited. To show

[1]http://ecos.sourceware.org/

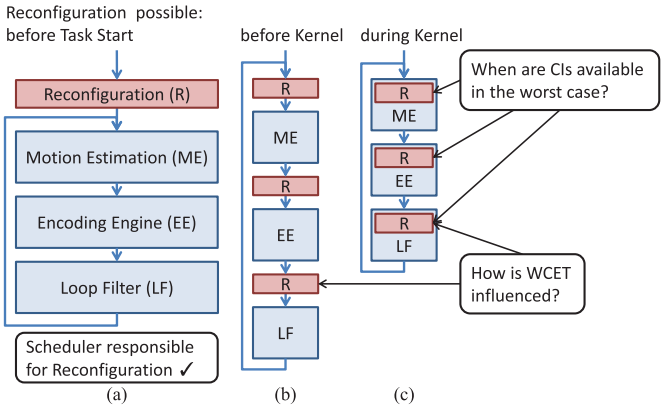

Fig. 3. Sequence of kernels, e.g., in the H.264 encoder, is well suited for runtime reconfiguration, but raise new issues in timing analysis. (a) once for the whole task, (b) per kernel, stalling the CPU or (c) per kernel, allowing the CPU to proceed in parallel.

the opportunities that are missed by such limitations, we use an H.264 video encoder as a motivational example in Fig. 3. For each input frame, the encoder goes through a sequence of kernels with different requirements of CIs to be configured onto the fabric. When configuring CIs for the whole task before it executes, the reconfigurable area has to be divided among the kernels [Fig. 3(a)]. Performing reconfiguration before each kernel allows every kernel to use the whole fabric area at the cost of reconfiguration delay [Fig. 3(b)], i.e., the start of the kernel is delayed (stalling) until its reconfiguration is completed. As the total reconfiguration delay per task increases, the question whether the idle time of the CPU during reconfiguration (stalling) can be used more effectively is posed. Instead of waiting until the reconfiguration of all CIs for a kernel finishes, the kernel can be started immediately using software emulation (see Fig. 2). As soon as reconfiguration for a CI finishes, it is utilized within the kernel and the reconfiguration delay was effectively used to make progress [Fig. 3(c)]. Software emulation leads to considerable runtime benefits at the cost of implementing equivalent software code for a CI, e.g., the runtime of loop filter, the shortest running kernel in the H.264 video encoder, is reduced by 20% using software emulation compared with stalling on a system running the reconfigurable fabric at 100 MHz, the core pipeline at 800 MHz, and a reconfiguration bandwidth of 100 MB/s (see Section VII for the detailed discussion).

Software emulation provides measurable runtime benefits that we make accessible for lower WCET bounds. A pessimistic approach would assume an infinite reconfiguration delay and thus would assume that every CI is executed in cISA for WCET estimation. While this would produce a safe time bound and enable speedups using CIs over cISA at runtime, the guaranteed worst case runtime would be worse than not using reconfiguration at all. Therefore, precise modeling of reconfigurable CIs is required for timing analysis.

In Section IV, we introduce timing analysis fundamentals that target nonreconfigurable processors or the cISA of a reconfigurable processor, and form the basis for our novel extensions for the analysis of reconfigurable CIs presented in Section V.

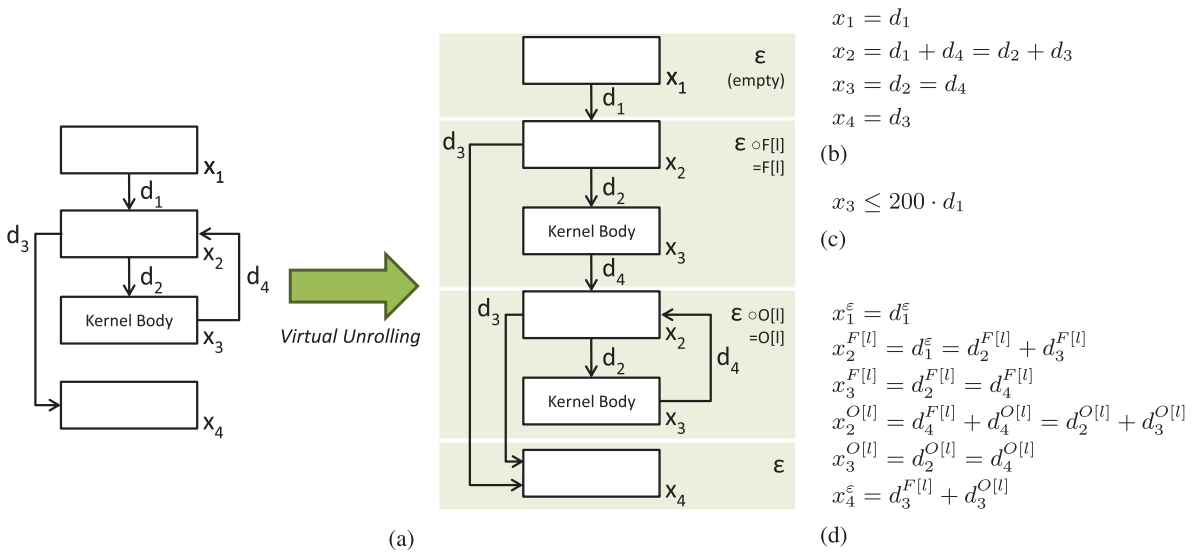

Fig. 4. IPET constraint generation for single contexts, and multiple contexts after virtual unrolling. (a) Virtually unrolling a kernel. (b) Basic program structural constraints (single context). (c) Program functionality constraint (single context) for an upper bound of 200 kernel iterations. (d) Program structural constraints after virtual unroll (multiple contexts).

IV. Timing Analysis Fundamentals

We employ static timing analysis that produces guaranteed WCET bounds instead of measurement-based approaches that produce bounds on the observed execution time [33].

The bounds obtained by static timing analysis allow safe schedulability analysis of hard real-time systems. Static timing analyzers generally perform three major analyzes consisting of several passes [1]:

1) control-flow reconstruction and static analyses for control and data flow;
2) microarchitectural analysis, computing the execution time bounds for basic blocks;
3) global bound analysis, computing execution time bounds for the whole task using the ILP-based implicit path enumeration technique (IPET) [34].

For computing guaranteed time bounds for the tasks on a reconfigurable instruction set processor, we introduce an additional reconfiguration analysis pass that generates IPET constraints after the microarchitectural analysis. When using software emulation and performing reconfigurations in parallel, the timing analyzer needs to know when a CI becomes available in the worst case. The aim of our reconfiguration analysis is to provide information to the global bound analysis about when it is safe to assume a CI to be available, i.e., when to use the path that uses the hardware CI instead of equivalent software to effectively reduce the guaranteed WCET bound. In Section IV-A, we introduce the basics of IPET-based path analysis.

A. Path Analysis

As any ILP-formulated problem, global bound analysis using IPET consists of two parts: the objective function and its constraints. For WCET analysis, the objective function determines the CPU cycles executed on a path in the task's control-flow graph (CFG). To find the WCET path, it needs to be maximized. Variables in the objective function represent the execution count of a single basic block (x_i) in the CFG and are weighted with the execution cycles of that basic block (c_i), which is determined in the microarchitectural analysis. For a program with N basic blocks, the objective function is given as [34]

$$\max_{x \in \mathbb{N}_0^N} \sum_{i=1}^{N} c_i x_i.$$

The constraints restrict the variables by modeling the control flow and capturing relative execution counts of basic blocks. The more infeasible paths can be excluded with constraints, the tighter the WCET bound will get. Program structural constraints can be derived automatically from the CFG, and program functionality constraints need to be user-specified or provided by further analysis passes.

1) Basic Constraints: An overview of how to generate IPET constraints is given in [34]. Besides the variables x_i representing the execution counts of basic blocks, variables d_i for the execution counts of edges in the CFG are used, e.g., consider the loop kernel in Fig. 4(a) (left). The loop header (represented by x_2) can be entered from outside using the edge represented by d_1 or from the previous iteration using d_4. The same basic block can be left when the loop condition becomes false and the kernel is exited using d_3 or it can proceed to another iteration when the loop condition is true using d_2. Therefore, $x_2 = d_1 + d_4 = d_2 + d_3$ [see Fig. 4(b)]. An upper bound of 200 for the number of kernel iterations can be given by the program functionality constraint $x_3 \leq 200 \cdot d_1$ [see Fig. 4(c)].

2) Execution Context: The simple constraints of Section IV-A1 only consider a single execution context, i.e., only one abstract microarchitectural state is considered at the beginning of each basic block. For a specific basic

block, this context needs to include any additional delay that may occur on any path to this basic block. In a loop, e.g., the first iteration will encounter much more cache misses than the following iterations, but the cache misses of the first iteration need to be considered for all the following iterations. The resulting WCET of the task will be very pessimistic. Therefore, more fine granular analyses considering different paths to reach a basic block separately have been developed [35], [36].

Following the definition in [35], a context is a sequence (denoted by $*$ in the formula) of first (C) and recursive (R) calls of functions as well as first (F) and following/other (O) iterations of loops. The set $\mathcal{T}$ of all contexts for a program $\mathcal{P}$ is [35]

$$\mathcal{T} := \{C[c], R[c], F[l], O[l] : c \in \text{calls}(\mathcal{P}), l \in \text{loops}(\mathcal{P})\}^*$$

with calls($\mathcal{P}$) and loops($\mathcal{P}$) being the sets of all calls and all loops in $\mathcal{P}$, respectively.

3) Analyzing Basic Blocks in Multiple Contexts: Contexts allow separate timing analysis of basic blocks for the different paths to reach them. Analysis starts with the empty context ε. When a function is called or a loop body is executed, the context changes. It can be considered as a stack: when a function is called or a loop is entered, this information is appended using the '$\circ$' operator. Upon exit, the information is removed. Recursive calls and following loop iterations replace (first) calls or iterations on top of the stack, respectively. For example, a basic block inside a loop $l \in \text{loops}(\mathcal{P})$, which is reached by calling $c \in \text{calls}(\mathcal{P})$, entering l and executing it repeatedly would be in the context $C[c]O[l]$. A more detailed explanation can be found in [35] and the theoretical background in [37].

In the CFG, the context influence of a loop is represented by virtually unrolling the loop once, as shown in Fig. 4(a). Using this representation, multicontext IPET constraints can be generated, as shown in Fig. 4(d). In general, each basic constraint in Section IV-A1 is generated for each distinguished context of the designated basic block, e.g., x_3 in Fig. 4 can be entered using d_2 and exited using d_3 in context $F[l]$ as well as $O[l]$. In addition, the constraints capture context changes performed using the '$\circ$' operator, e.g., when entering, reentering, or exiting a loop.

V. Timing Analysis Extensions for Runtime Reconfigurable Processors

The main contributions of this paper are the path analysis extensions to obtain precise WCET bounds of tasks on runtime reconfigurable processors and are presented in Section V-B. Section V-A introduces properties, which we assume the microarchitecture to provide and which we exploit during timing analysis.

A. Microarchitectural Analysis

The input to microarchitectural analysis (see Section IV) is the reconstructed CFG from the binary under analysis. Its output is a WCET bound per basic block and context (see Section IV-A2) incorporating cache misses/hits, memory access latencies, pipeline stalls, and other delays which can occur in the system. To determine the WCET of a basic block including CIs, the CI latencies need to be known, and possible influences on the rest of the microarchitecture—especially on core pipeline and caches—need to be accessible to the respective analysis passes.

The delay for initiating and performing the reconfiguration of a CI needs to be analyzable statically. In addition, reconfigurations in parallel to execution may not spoil any timing guarantees of the microarchitecture, e.g., the delay for the CPU to perform bus accesses. We assume to be able to initiate a sequence of CI reconfigurations using the core pipeline (e.g., by accessing a device on the bus) and then either use stalling or software emulation while reconfigurations take place. The requested CIs for an upcoming kernel and the order of configurations need to be obtainable by analyzing the binary. This information is used to generate constraints for path analysis, as described in Section V-B.

By way of example, an implementation of a reconfiguration unit, which provides these properties is presented in Section VI.

B. Path Analysis Constraints for Software Emulation

Using software emulation, CI functionality can be executed on two separate paths: the CI itself or CI-equivalent software, depending on whether the CI is available or not (see Fig. 2). In the following, we always initiate a sequence of reconfigurations of CIs immediately before entering a kernel. Kernel execution and reconfigurations are performed in parallel. With the delay for configuring each CI known (in cycles of the core pipeline), we can determine the total delay for a CI to become available in the sequence. Once the CI is available, the program path using the CI will be taken for all of its invocations in the remaining kernel iterations. The main challenge is to precisely analyze at which point during kernel execution these path changes—by CIs becoming available successively—will happen: How far will the kernel have progressed in the worst case and what exactly is the worst case?

1) Assumptions: For analyzing tasks that use reconfigurable CIs for worst case timing with manageable complexity, we apply the following assumptions.

1) We assume that the software emulation of a CI always has a longer delay than the CI itself. In case the CI took longer than the software emulation, it would not make sense to use a CI anyway. In addition, the CI is a single instruction, potentially saving numerous instruction cache accesses and pipeline hazards. When there is no constraint forcing the analyzer to take the CI path, this assumption results in always accounting for the software emulation of a CI when it is invoked.
2) Running CIs in software emulation are never moved to hardware and we conservatively assume the availability of a CI not to change during a kernel iteration, even when reconfiguration finishes early in the iteration and the CI is the last executed instruction. This assumption eases analysis and is safe, as the WCET of a single kernel iteration is reduced when a CI becomes available, but may introduce pessimism.

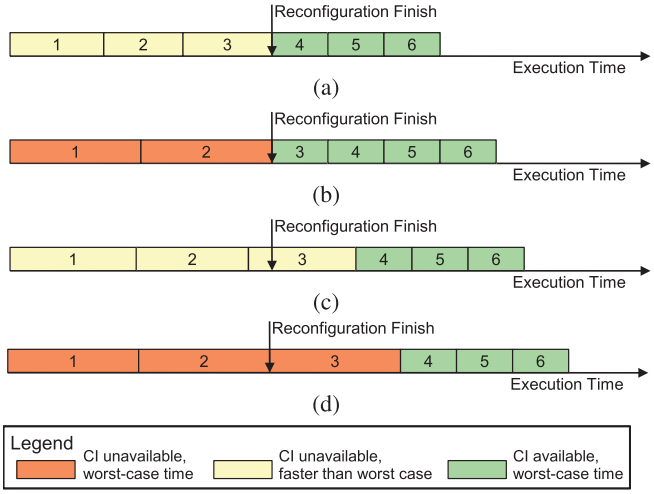

Fig. 5. Different cases for execution times of kernel iterations before the CI becomes available. (a) Faster than worst case, assuming no iterations overlap RF. (b) Worst case, assuming no iterations overlap RF. (c) Faster than WCET before RF. Iteration overlapping RF leads to extended execution time (timing anomaly). (d) Applied case for safe WCET bounds.

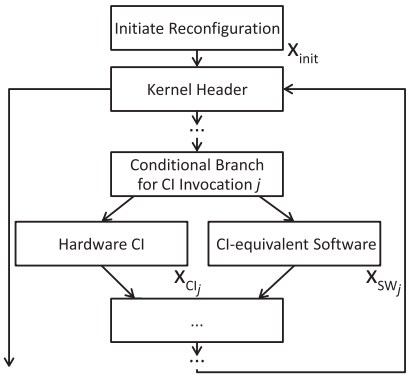

Fig. 6. CFG of a kernel invoking a CI with software emulation.

2) Worst Case CI Availability: In the following, we want to bound the worst case number of kernel iterations for every CI, in which the CI is unavailable. During these iterations, the CI invocations need to be executed using software emulation. To obtain a safe bound, we need to determine under which circumstances the execution time is maximized when a CI becomes available after its constant reconfiguration delay. For this, consider the timelines in Fig. 5 for a kernel with six iterations in total. Iterations without the CI available are yellow for executing faster than WCET and orange for executing them in WCET. After the marked configuration delay for the specific CI [reconfiguration finish (RF)], every following kernel iteration (green) makes use of the CI path for all of its invocations (possibly multiple per iteration). Clearly, these iterations need to run in worst case time to maximize the execution time after the configuration delay. The remaining questions to maximize the execution time are as follows.

1) Given an upper bound of kernel iterations, what share needs to be run after the reconfiguration delay?
2) What is a safe time bound for iterations before finishing the reconfiguration?

First, let us consider question 1) and assume no iteration of the kernel can overlap the point at which the configuration finishes. Let WCET_1 be the WCET of one kernel iteration with the CI available, r the reconfiguration delay for the CI, and m the remaining iterations after executing with software emulation during r. Under these assumptions, the execution time becomes

$$\text{Execution time} = r \ + m \cdot \mathrm{WCET}_1.$$

As m is the only variable in this equation and every constant is positive, the equation is maximized when m is maximized. Under the assumption that no iteration of the kernel can overlap the RF, this is achieved by executing every iteration during r in worst case time, as shown in Fig. 5(b). Counterintuitively, the execution time is increased when we have fewer iterations with the CI unavailable. To finally answer (1) and (2), we need to consider iterations overlapping the RF. As the last iteration of the kernel with the CI unavailable can now reach into the part of the timeline in which the CI is already available, this iteration can prolong the execution time. This can lead to a case, where executing iterations faster than worst case time during reconfiguration extends the execution time of the whole kernel. We call this the timing anomaly of runtime reconfiguration. An example is shown in Fig. 5(c). For bounding this timing anomaly, we need to apply the case shown in Fig. 5(d) in timing analysis. Iterations are executed in worst case time, as this results in minimal progress during the reconfiguration delay. An additional iteration starting after the RF is assumed to execute with the CI unavailable in worst case time (the maximum prolongation) for bounding a potentially overlapping iteration. To generate safe constraints, this case is always applied to bound the timing anomaly at the cost of overestimation.

For a scenario with multiple CIs in a kernel, we can analogously argue for the worst case of CI_i to become available after CI_{i-1} when considering the RF of CI_{i-1} as point zero on the timeline and accounting for the additional iteration (potential timing anomaly) of CI_{i-1}. In Section V-B3, we will formally express these considerations for multiple CIs.

3) Basic Constraints: Suppose that without loss of generality, the CIs for the following kernel are configured in the sequence $\mathrm{CI}_1, \ldots, \mathrm{CI}_n$. We denote r_i as the delay to configure CI_i. With the existing timing analysis and the properties of Section V-A, we can determine WCET_i, the WCET of one kernel iteration with $\mathrm{CI}_1, \ldots, \mathrm{CI}_{i-1}$ available (and the others unavailable) by setting the CI availability in simple constraints. Consider the conditional branch to CI-equivalent software (see Fig. 2) in the case a CI should be invoked, but is unavailable. As shown in Fig. 6, for every CI invocation j in the binary, the CI and its software emulation reside on separate paths in the CFG, which are immediately joined after the CI functionality is executed. Let x_{SW_j} be the variable representing the first basic block of the software emulation path of invocation j, a constraint of $x_{\mathrm{SW}_j} = 0$ forces the path analysis to exclude this path and account for the hardware CI path using x_{CI_j} only. For determining WCET_i, we generate a constraint for every invocation of $\mathrm{CI}_1, \ldots, \mathrm{CI}_{i-1}$

to exclude software emulation. WCET_0 is the special case of not generating any CI-specific constraints and effectively always executing the software emulation for every CI to be configured.

Suppose we know u_k, the number of iterations in which CI_k is unavailable (and therefore CI_s, $s > k$ unavailable), but all previous CIs (if any) CI_t, $t < k$ are available. The total number of iterations CI_i is unavailable is $\sum_{k=1}^{i} u_k$. Let $u_0 = \mathrm{WCET}_0 = 0$, then we can define the remaining reconfiguration time needed for CI_i after CI_{i-1} became available as

$$s_i := \sum_{k=1}^{i} r_k - \sum_{k=0}^{i-1} u_k \cdot \mathrm{WCET}_k. \tag{1}$$

In other words, this is the time to configure $\mathrm{CI}_1, \ldots, \mathrm{CI}_i$ minus the time already spent in the first $\sum_{k=0}^{i-1} u_k$ iterations. Formally, we can define u_i recursively as follows:

$$u_i := \begin{cases} \lceil s_i / \mathrm{WCET}_i \rceil + 1, & \text{if } s_i > 0 \\ 1, & \text{if } s_i + \mathrm{WCET}_{i-1} > 0 \wedge s_i \leq 0 \\ 0, & \text{else.} \end{cases} \tag{2}$$

If s_i becomes ≤ 0 (the second and third cases of u_i), the time $\sum_{k=1}^{i} r_k$ until CI_i becomes available is already covered by the time spent in the iterations in which $\mathrm{CI}_1, \ldots, \mathrm{CI}_{i-1}$ are unavailable. As discussed in Section V-B2, an additional iteration for iterations overlapping the point in time the reconfiguration finishes can become necessary, however. This is the case, if CI_i became available in the additional iteration of CI_{i-1} [the second case, e.g., iteration 3 in Fig. 5(d)]. If CI_i became available in the previous iteration [e.g., iteration 2 in Fig. 5(d)], the additional iteration of CI_{i-1} already covers the additional iteration of CI_i, such that CI_{i-1} and CI_i become available in the same iteration. It would be safe but pessimistic to not differentiate between the second and third cases and always add an additional iteration. For a single CI_1, the equation becomes $u_1 = \lceil r_1 / \mathrm{WCET}_1 \rceil + 1$.

Assuming no invocations of CI_i are contained in a nested loop inside the kernel, we can directly generate constraints to restrict the number of kernel iterations, in which CI_i is unavailable in hardware and the software emulation needs to be executed. The restriction of the absence of CI invocations in nested loops will be removed in Section V-B5. For a single invocation j of CI_i inside the kernel, let $x_{\mathrm{SW}_{i,j}}$ be the variable representing the number of executions of the first basic block in CI_i's software emulation. Let x_{init} be the number of executions of the basic block which initiates reconfiguration before entering the kernel, as shown in Fig. 6. To include the previous reconfiguration analysis into global bound computation, the following constraint is generated:

$$x_{\mathrm{SW}_{i,j}} \leq \sum_{k=1}^{i} u_k \cdot x_{\mathrm{init}}. \tag{3}$$

This constraint ensures that $x_{\mathrm{SW}_{i,j}}$ is accounted for at most as often as CI_i was determined to be unavailable. Even though we can determine the exact number of kernel iterations that start with CI_i unavailable, the constraint is generated as an inequality, because it can happen that the worst case path in the kernel does not include invocation j of CI_i. Generating a constraint $x_{\mathrm{SW}_{i,j}} = \sum_{k=1}^{i} u_k \cdot x_{\mathrm{init}}$ with $u_i > 0$ would force the timing analyzer to include invocation j of CI_i in its path analysis, possibly hindering it from finding the real worst case path.

4) Conditional Execution: As discussed previously, the constraints generated by (3) correctly allow the timing analyzer to find worst case paths not containing every CI invocation. However, for the upper bounds on how often the software emulation of a CI invocation needs to be executed, because the hardware is not yet available, the maximum possible number of iterations making use of this invocation is considered. Even when the specific CI invocation is not part of the worst case path. This may be very pessimistic, e.g., consider a CI which is unavailable for ten iterations of a kernel in total. Consider the CI is executed only five times during the ten iterations of its unavailability, then the CI needs to be invoked five times using software emulation during the whole execution of the kernel. Our conservative constraints, however, force the analyzer to assume ten invocations of the CI need to be executed in software emulation. As only five iterations exist in which the CI is invoked and unavailable, five iterations of the kernel after the unavailability of the CI are considered to emulate the CI in software, even though the hardware is already available. This pessimism can be removed by performing an extended analysis of WCET_i and determining for every invocation j of an unavailable $\mathrm{CI}_i, \ldots, \mathrm{CI}_n$ whether this invocation is part of the worst case path, i.e., $x_{\mathrm{SW}_{i,j}} > 0$ (in the analysis of WCET_i). When generating the constraint for the number of software emulations of invocation j of CI_i, u_k is only added if this invocation is part of WCET_k. We define

$$\mathrm{inp}(i, j, k) := \begin{cases} 1, & \text{invocation } j \text{ of } \mathrm{CI}_i \text{ is part of the} \\ & \text{worst-case path of } \mathrm{WCET}_k \\ 0, & \text{else} \end{cases} \tag{4}$$

$\mathrm{inp}(i, j, k)$ is obtained from the solved ILP which was used to determine WCET_k by simply testing whether $x_{\mathrm{SW}_{i,j}} > 0$. From this analysis, we obtain for every invocation of a CI the exact number of times the software emulation is executed when entering the kernel in the worst case path. We obtain the following equality:

$$x_{\mathrm{SW}_{i,j}} = \sum_{k=1}^{i} \mathrm{inp}(i, j, k) \cdot u_k \cdot x_{\mathrm{init}}. \tag{5}$$

This constraint is generated for every invocation of all CIs instead of the constraint in (3).

5) Loop Nests: When considering an invocation j of CI_i which is contained in a nested loop inside the kernel, the constraints generated by (3) would result in a too low upper bound for the number of times the software emulation is executed, because j is assumed to be executed at most once per iteration. In a nested loop, however, j can be executed multiple times per iteration, maximal as often as the basic block its conditional branch for software emulation is contained in (see Fig. 6). For brevity, let $\mathrm{nf}(\mathrm{CI}_i, j)$ be the statically known product of upper loop bounds for every level of loop nest to reach the

basic block which contains invocation j of CI_i from the kernel iteration top level (nesting factor) and $\mathrm{nf}(\mathrm{CI}_i, j) = 1$ if it is not contained in a loop nest.

We obtain the following constraint:

$$x_{\mathrm{SW}_{i,j}} \leq \mathrm{nf}(\mathrm{CI}_i, j) \cdot \sum_{k=1}^{i} u_k \cdot x_{\mathrm{init}}. \tag{6}$$

When including the analysis of conditional execution, we obtain the equality

$$x_{\mathrm{SW}_{i,j}} = \mathrm{nf}(\mathrm{CI}_i, j) \cdot \sum_{k=1}^{i} \mathrm{inp}(i, j, k) \cdot u_k \cdot x_{\mathrm{init}}. \tag{7}$$

Using (7), constraints for all invocations of CI_i are generated. After generating constraints for all invocations of $\mathrm{CI}_1, \ldots, \mathrm{CI}_n$, we can perform the global bound analysis of a task including reconfigurable CIs with a single context.

6) Multiple Contexts: For extending the constraints of Sections V-B3-V-B5 for multicontext timing analysis, we need to generate the constraint of (7) for every context the kernel can appear in. Let $t(x_{\mathrm{init}}) \subseteq \mathcal{T}$ be the subset of execution contexts x_{init} can appear in (see Section IV-A2 for the definition of contexts). Again, we want to express the reconfiguration delay of a CI as iterations of the kernel. However, in a multicontext analysis, a loop l can be in its first $F[l]$ and other $O[l]$ iterations. Therefore, the basic constraint in (3) becomes

$$x_{\mathrm{SW}_{i,j}}^{\vartheta \circ F[l]} + x_{\mathrm{SW}_{i,j}}^{\vartheta \circ O[l]} \leq \sum_{k=1}^{i} u_k^{\vartheta} \cdot x_{\mathrm{init}}^{\vartheta} \quad \forall \vartheta \in t(x_{\mathrm{init}}). \tag{8}$$

Note that the number of iterations in which CI_i is unavailable u_i^{ϑ} is now also context-dependent as denoted by the superscript $\vartheta \in t(x_{\mathrm{init}})$, because the iterations of the kernel now have different delays depending on the context the kernel is entered in. Therefore, we need to redefine u_i^{ϑ} in the following. The reconfiguration of CI_1 starts in the first iteration of the kernel, therefore in context $\vartheta \circ F[l]$. The worst case delay for one iteration of the kernel is now context-dependent and especially dependent on whether we are in the first or other iterations of the kernel. We denote the first iteration, in which all CIs to be reconfigured are unavailable, as $\mathrm{WCET}_1^{\vartheta \circ F[l]}$. Following iterations in parallel to reconfiguration are all in context $\vartheta \circ O[l]$. $\mathrm{WCET}_1^{\vartheta \circ O[l]}$ denotes the worst case time bound of an iteration in this context in parallel to configuring CI_1. $\mathrm{WCET}_i^{\vartheta \circ F[l]}$ and $\mathrm{WCET}_i^{\vartheta \circ O[l]}$, the WCET of one first or other kernel iteration with $\mathrm{CI}_1, \ldots, \mathrm{CI}_{i-1}$ available and $\mathrm{CI}_i, \ldots, \mathrm{CI}_n$ unavailable when the kernel is entered in context ϑ, can be determined analogously to the single-context analysis explained in Section V-B3.

Now, let us consider u_1^{ϑ}, the number of iterations in which CI_1 is unavailable. To account for the first iteration of the kernel, we simply subtract its delay from the reconfiguration delay of CI_1 and increase u_1^{ϑ} by 1. Together with the additional iteration discussed in Section V-B2, we have at least two iterations with CI_1 unavailable. The formula directly resembles the single-context definition in (2) when inserting the values for CI_1, u_1^{ϑ} becomes

$$u_1^{\vartheta} := \begin{cases} \left\lceil \left(r_1 - \mathrm{WCET}_1^{\vartheta \circ F[l]}\right) / \mathrm{WCET}_1^{\vartheta \circ O[l]} \right\rceil + 2, \\ \qquad\qquad \text{if } r_1 - \mathrm{WCET}_1^{\vartheta \circ F[l]} > 0 \\ 2, \qquad\qquad \text{else.} \end{cases} \tag{9}$$

As in the single-context analysis, we need to determine the remaining reconfiguration delay after the time spent in the first $\sum_{k=0}^{i-1} u_k^{\vartheta}$ iterations when determining u_i^{ϑ}. However, we now need to consider a different context for the first iteration than for the others. Therefore, the context-sensitive extension of s_i is defined as follows:

$$s_i^{\vartheta} := \sum_{k=1}^{i} r_k - \Bigg(\mathrm{WCET}_1^{\vartheta \circ F[l]} + \left(u_1^{\vartheta} - 1\right) \cdot \mathrm{WCET}_k^{\vartheta \circ O[l]} + \sum_{k=2}^{i-1} u_k^{\vartheta} \cdot \mathrm{WCET}_k^{\vartheta \circ O[l]} \Bigg). \tag{10}$$

Finally, u_i^{ϑ} for $i > 1$ becomes

$$u_i^{\vartheta} := \begin{cases} \left\lceil s_i^{\vartheta} / \mathrm{WCET}_i^{\vartheta \circ O[l]} \right\rceil + 1, & \text{if } s_i^{\vartheta} > 0 \\ 1, & \text{if } s_i^{\vartheta} + \mathrm{WCET}_{i-1}^{\vartheta \circ O[l]} > 0 \wedge s_i^{\vartheta} \leq 0 \\ 0, & \text{else.} \end{cases} \tag{11}$$

As in the single-context case, u_i^{ϑ} for $i > 1$ can become 0 when CI_{i-1} and CI_i become available in the same iteration.

For including the analysis of conditional CI execution and loop nests as explained in Sections V-B4 and V-B5, $\mathrm{inp}(i, j, k)$ also needs to be extended for contexts as it is dependent on the WCET of one kernel iteration. $\mathrm{inp}(i, j, k)$ determines whether invocation j of CI_i is part of the worst case path defining WCET_k. We define the context-aware $\mathrm{inp}^{\vartheta}(i, j, k)$ as

$$\mathrm{inp}^{\vartheta}(i, j, k) := \begin{cases} 1, & \text{invocation } j \text{ of } \mathrm{CI}_i \text{ is part of the} \\ & \text{worst case path of } \mathrm{WCET}_k^{\vartheta \circ F[l]} \\ & \text{or } \mathrm{WCET}_k^{\vartheta \circ O[l]} \\ 0, & \text{else.} \end{cases} \tag{12}$$

Including the analysis of conditional CI execution and loop nests analogously to the single-context constraint in (7), we obtain

$$x_{\mathrm{SW}_{i,j}}^{\vartheta \circ F[l]} + x_{\mathrm{SW}_{i,j}}^{\vartheta \circ O[l]} = \mathrm{nf}(\mathrm{CI}_i, j) \cdot \sum_{k=1}^{i} \mathrm{inp}^{\vartheta}(i, j, k) \cdot u_k^{\vartheta} \cdot x_{\mathrm{init}}^{\vartheta}. \tag{13}$$

Using this equation, we can generate constraints for all invocations of $\mathrm{CI}_1, \ldots, \mathrm{CI}_n$ and perform the global bound analysis of a task including reconfigurable CIs with multiple contexts.

C. Stalling Versus Software Emulation

As modeling software emulation with parallel reconfiguration needs to make pessimistic assumptions to achieve a safe worst case bound and an additional analysis needs to be

performed to generate path analysis constraints, it needs to be determined in which cases the approach is competitive or superior to stalling under timing guarantees. For clarity, we will only consider a single execution context in the following, but multiple contexts in the evaluation.

When stalling, execution halts for the duration of reconfiguring all CIs required in the upcoming kernel. Afterward, every kernel iteration is executed in its worst case execution bound with all CIs available, let us denote this worst case bound as WCET_{n+1}. Therefore, when stalling, the WCET for a kernel with an upper bound of I iterations (while neglecting the time taken to exit the kernel) is

$$\sum_{k=1}^{n} r_k + I \cdot \mathrm{WCET}_{n+1}. \tag{14}$$

From the considerations for generating path analysis constraints for the software emulation approach in Section V-B, we can directly bound the execution time for a kernel. For brevity, we assume the basic constraints of Section V-B3. The upper bound of kernel iterations in which reconfiguration takes place is $\sum_{k=1}^{n} u_k$. Furthermore, during u_k iterations, a kernel iteration has an upper bound of WCET_k. The remaining $I - \sum_{k=1}^{n} u_k$ iterations have an upper bound of WCET_{n+1} each. Therefore, we obtain the following execution time bound for all kernel iterations:

$$\sum_{k=1}^{n} u_k \cdot \mathrm{WCET}_k + \left(I - \sum_{k=1}^{n} u_k \right) \cdot \mathrm{WCET}_{n+1}. \tag{15}$$

For the software emulation approach resulting in a lower worst case time bound, we need to satisfy the inequality of (15) $<$ (14). When simplifying this inequality, we obtain

$$\sum_{k=1}^{n} u_k \cdot (\mathrm{WCET}_k - \mathrm{WCET}_{n+1}) < \sum_{k=1}^{n} r_k. \tag{16}$$

It can be noted that inequality (16) is independent of the total iterations of the kernel, whether software emulation is beneficial over stalling can be decided by analyzing the first $\sum_{k=1}^{n} u_k$ iterations. For a specific iteration included in u_k, $\mathrm{WCET}_k - \mathrm{WCET}_{n+1}$ denotes the additional time the software emulation takes because some CIs are unavailable. The software emulation approach results in a lower time bound for a kernel if and only if the total additional time remains lower than the total reconfiguration time. The practical implications of this analysis are investigated in the evaluation.

VI. Runtime Reconfigurable Processor Infrastructure for Timing Guarantees

For evaluating this paper, we modified a reconfigurable processor that was so far used for optimizing the average case for execution with hard real-time guarantees. It is based on the Gaisler LEON3 SoC (see [38] for an overview). The LEON3 processor has the SPARC V8 in-order microarchitecture and supports several real-time operating systems.[2] Its seven-stage pipeline was extended into a reconfigurable core: the execute stage allows stalling the pipeline and passing

[2]http://www.gaisler.com/index.php/products/operating-systems

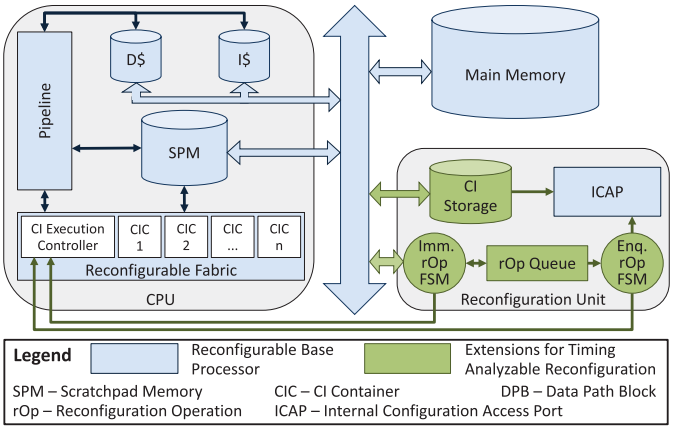

Fig. 7. Overview of SoC used for evaluation.

operands to the fabric. A CI is executed by the CI execution controller in a protocol similar to other multicycle instructions like division and directly accesses register operands or the non-cacheable SPM only (see Fig. 7). This way, in microarchitectural analysis, a CI is just another multicycle instruction, which does not influence data cache analysis. The reconfigurable fabric is partitioned into containers of identical size [13]. A CI can be configured into any set of containers (possibly multiple) and potentially replaces a currently configured CI [17].

For this paper, we designed a reconfiguration unit (see Fig. 7) that performs timing-analyzable reconfiguration using the integrated configuration access port (ICAP) and is accessible by the core pipeline as a memory-mapped on-chip bus device. For achieving predictability, we statically select which CIs to reconfigure at which program points and in what order. The resulting reconfiguration sequences are fed as reconfiguration operations (rOps) to the reconfiguration unit by a sequence of stores to its address. The rOps are executed by two simple finite state machines, one for immediate rOps and the other one for enqueued rOps. The first rOp of every sequence is clear, which immediately flushes the rOp queue and aborts a potentially running reconfiguration to ensure there are no pending rOps from the previous kernel. Afterward, a sequence of configure CI rOps follows with memory address operands. They are enqueued and the addressed configurations are fed to the ICAP sequentially. Feeding configurations over the on-chip bus would lead to bus contention and result in a high variability of memory access delay for the pipeline. Therefore, the reconfiguration unit contains an SRAM memory specifically for configurations (CI storage). The CI storage is filled with all configurations required by the task over the bus at task load. At runtime, the reconfiguration unit solely configures CIs from its local CI storage, enabling statically analyzable reconfiguration delays. By default, execution proceeds after storing the rOps to the reconfiguration unit. Optionally, the last rOp of the reconfiguration sequence can be the stall immediate rOp to halt execution until reconfiguration finishes. The complete reconfiguration unit, including on-chip bus interfaces, utilizes 560 lookup table, 347 latches, and SRAM memory to store 552 kB of bitstreams (used in our

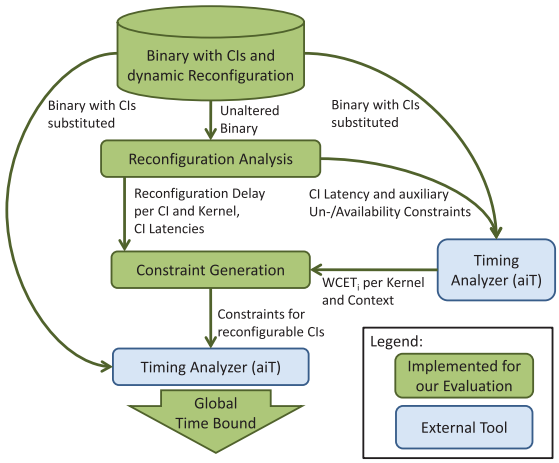

Fig. 8. Evaluation tool flow.

```
instruction 0x40001238 additionally takes:
    ((36*def("cISA_freq_mul"))-1) cycles;
```

(a)

```
flow sum: point(0x40001244) == (2*67)
    point(0x400011a0);
```

(b)

Fig. 9. Generated constraints in aiT's format (AIS2). (a) Example for setting the CI latency. (b) Example for a generated constraint for CI availability.

evaluation) on a Xilinx Virtex 5 field-programmable gate array (FPGA).

VII. Experimental Evaluation

A. Implementation and Setup

The static timing analysis flow as described in Section IV was split in several steps, as shown in Fig. 8.

1) Reconfiguration analysis is performed on the compiled binary, giving absolute reconfiguration delays per CI.
2) AbsInt aiT [39] is used to determine $\text{WCET}_i^{\vartheta}$ (see Section V-B) for all kernels. As aiT is closed-source software, we could not directly integrate support for CIs. Therefore, every CI opcode in the binary was substituted by an ADD opcode and a constraint in aiT's AIS2 language to set the delay for the new ADD instruction to the delay of the specific CI [see Fig. 9(a)]. The aiT outputs an XML report, which we parsed to determine every u_i^{ϑ} for every kernel and generate the constraints described in Section V-B in AIS2 [see Fig. 9(b)].
3) Our generated constraints were used to calculate the global WCET bound using aiT.

The analysis is performed offline and runs on a workstation. It does not induce any runtime overheads. We evaluated our analysis with an H.264 encoder application which uses nine CIs covering the most compute-intensive kernels.

1) *Motion Estimation Kernel:*
   a) `SATD:` Sum of absolute (Hadamard-)transformed differences on 16×16 pixels (px).
   b) `SAD:` Sum of absolute differences on 16×16 px.

TABLE I
Parameters Investigated

Parameter [Unit]	Symbol	Values
CPU frequency [MHz]	f_{CPU}	100, 200, 400, 800, 1600
Fabric frequency [MHz]	f_{fabric}	100
Configuration Port Frequency [MHz]	f_{ICAP}	25, 50, 100

2) *Encode Macroblock Kernel:*
   a) `MC_Hz:` Motion compensated interpolation horizontal on 4 px.
   b) `IPred_HDC:` Intraprediction horizontal on 16×16 px.
   c) `IPred_VDC:` Intraprediction vertical on 16×16 px.
   d) `Discrete Cosine Transform (DCT):` DCT on 4×4 px.
   e) `HT2x2:` Hadamard transform on 2×2 px.
   f) `HT4x4:` Hadamard transform on 4×4 px.
3) *Loop Filter Kernel:*
   a) `LoopFilter:` In-loop deblocking edge filter on 4 px.

Every kernel configuration requires the whole CI containers. Therefore, before entering a kernel, reconfigurations for all containers are initiated to meet the kernel's CI requirements. It contains complex control flow with numerous decisions and nested loops. Most of the properties tested in the Mälardalen WCET Benchmarks[3] are covered, e.g., DCT is contained in both. For evaluating the overestimation of the static analysis, we executed the same binary obtained from BCC 4.4.2 (Gaisler's extended GCC 4.4.2) at O1 in our SystemC-based cycle-accurate simulator which models the reconfigurable system shown in Fig. 7. Before performing the evaluation, we calibrated aiT and our simulator by harmonizing hardware parameters and verifying the results of test cases, e.g., load-store sequences.

B. Results

In the following, we analyze the influences of stalling and software emulation on WCET bounds of a single kernel. Afterward, we analyze overestimation and WCET reduction when using reconfigurable CIs on the whole H.264 encoder.

1) Software Emulation Versus Stalling on a Single Kernel: 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 (Quarter Common Interchange Format (QCIF) resolution). The loop filter is the kernel of lowest complexity in our H.264 encoder, and it contains a single CI and allows detailed analysis of worst case CI availability. The guaranteed time bounds are compared with the results obtained by executing the same binary in our simulator. Table I gives an overview of the parameters investigated. f_{fabric} stays constant at 100 MHz and we choose multiples of it for f_{CPU} which resemble realistic setups (rounded to the next power of two), e.g., the LEON3 processor which we extended for reconfigurable CIs is advertised as running at 400 MHz when implemented as an application-specified integrated circuit, its

[3] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

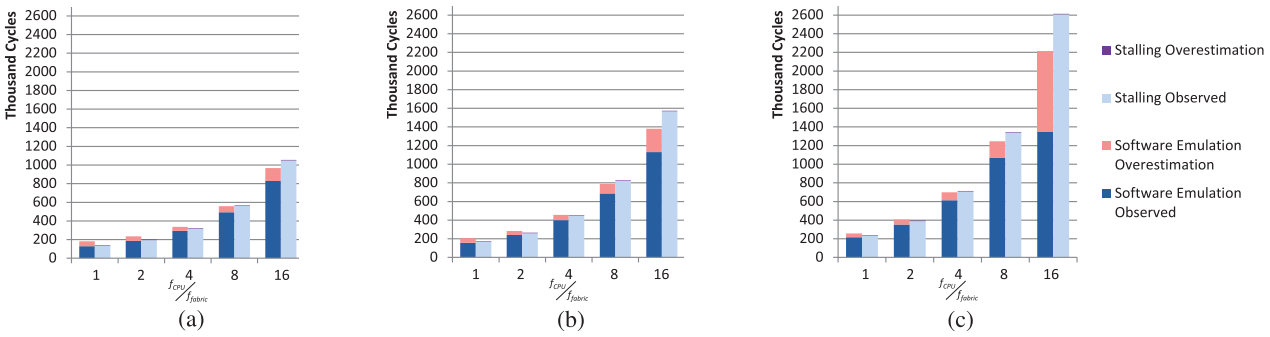

Fig. 10. Observed runtimes and guaranteed WCET bounds for loop filter. (a) $f_{\text{ICAP}} = f_{\text{fabric}}$. (b) $f_{\text{ICAP}} = (1/2)\, f_{\text{fabric}}$. (c) $f_{\text{ICAP}} = (1/4)\, f_{\text{fabric}}$.

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 are measured in cycles of the CPU pipeline; therefore, they are determined by the relation of f_{CPU}, f_{fabric}, and f_{ICAP}. Fig. 10(a) shows the results for $f_{\text{CPU}}/f_{\text{fabric}} \in \{2^0 = 1, \ldots, 2^4 = 16\}$ and $f_{\text{ICAP}} = f_{\text{fabric}} = 100$ MHz. This corresponds to running the internal configuration access port (ICAP) at its maximum frequency, and results in a reconfiguration bandwidth of 400 MB/s when configuring 32 bit of data every cycle. This reconfiguration bandwidth is possible when using a dedicated on-chip CI storage (see Section VI), we utilize the block random-access memory resources of Xilinx FPGAs in our prototype. When increasing the minimum evaluated CPU pipeline frequency of 100 MHz by a factor of c for a fixed ICAP frequency, the reconfiguration delay, measured in CPU cycles, increases by the factor c as well. In addition, the runtime benefit of hardware CIs compared with software emulation decreases. According to our prediction in Section V-C, software emulation results in a lower time bound than stalling for $f_{\text{CPU}}/f_{\text{fabric}} \in \{8, 16\}$, but not for $f_{\text{CPU}}/f_{\text{fabric}} \in \{1, 2, 4\}$. The time bounds obtained for the kernel shown in Fig. 10(a) reflect these predictions. Software emulation results in 30.28%, 16.39%, and 4.48% higher time bounds than stalling for $f_{\text{CPU}}/f_{\text{fabric}} \in \{1, 2, 4\}$, respectively. For $f_{\text{CPU}}/f_{\text{fabric}} \in \{8, 16\}$, software emulation results in 1.38% and 8.25% lower time bounds. When considering the observed worst case runtime, however, software emulation always takes less time than stalling, i.e., 2.84%, 4.57%, 7.28%, 11.98%, and 20.85% less for $f_{\text{CPU}}/f_{\text{fabric}} \in \{1, 2, 4, 8, 16\}$, respectively. The reason for this discrepancy is that for analyzing software emulation for the worst case time bound, pessimistic assumptions about CI availability need to be made as detailed in Section V-B2. In contrast, we know exactly at which point in a kernel a CI is available when stalling: directly from the beginning, after stalling for a statically known amount of cycles. Therefore, overestimation for software emulation ranges from maximal 40.17% with $f_{\text{CPU}} = f_{\text{fabric}}$ to 13.25% with $f_{\text{CPU}}/f_{\text{fabric}} = 8$, while the overestimation for stalling is never above 4.54%, again maximal with $f_{\text{CPU}} = f_{\text{fabric}}$.

Fig. 10(b) and (c) shows the results for $f_{\text{ICAP}} = 50\text{ MHz} = (1/2) \cdot f_{\text{fabric}}$ and $f_{\text{ICAP}} = 25\text{ MHz} = (1/4) \cdot f_{\text{fabric}}$, i.e., a reconfiguration bandwidth of 200 and 100 MB/s, respectively. This corresponds, e.g., to systems making use of cheaper but slower flash memory for the CI Storage instead of SRAM-based memory and therefore requiring a lower f_{ICAP}. The overall trend is that overestimation is lower with slower memories. In software emulation, the pessimism of assuming an additional iteration of CI unavailability (see Section V-B2), has less influence on the guaranteed time bound as there are more iterations with the CI unavailable in total. At $f_{\text{CPU}}/f_{\text{fabric}} = 16$ and $f_{\text{ICAP}} = (1/4) \cdot f_{\text{fabric}}$ ($f_{\text{CPU}} = 64 \cdot f_{\text{ICAP}}$), overestimation rises again and reaches its overall maximum of 63.73%. As shown in Table II, timing analysis guarantees that a maximum of 19 iterations (99 total iterations minus $81 = u_1$ plus 1, as discussed in Section V-B2) of the kernel need to be executed after reconfiguration delay at this point. In the observed runtime however, all iterations are finished in software during reconfiguration.

When stalling, overestimation also decreases slightly with slower memory as the reconfiguration delay takes a bigger share on the overall execution time and does not introduce overestimation. As more iterations in software emulation can be executed during reconfiguration and overestimation decreases, the resulting time bound is only 1.07% higher than

TABLE II
CI UNAVAILABILITY IN WCET BOUNDS FOR LOOP FILTER

$f_{\text{CPU}}/f_{\text{fabric}}$	1	2	4	8	16
u_1 at $f_{\text{ICAP}} = f_{\text{fabric}}$	3	4	6	11	21
u_1 at $f_{\text{ICAP}} = \frac{1}{2} \cdot f_{\text{fabric}}$	4	6	11	21	41
u_1 at $f_{\text{ICAP}} = \frac{1}{4} \cdot f_{\text{fabric}}$	6	11	21	41	81

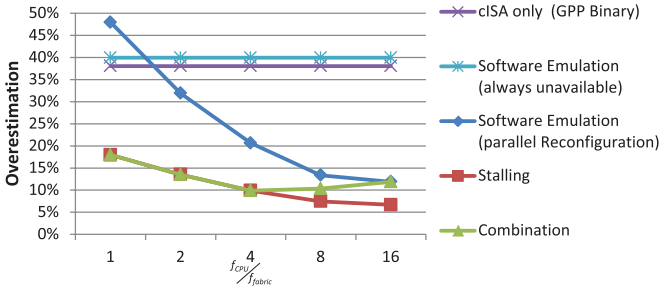

Fig. 11. H.264 overall overestimation without CI invocations (cISA only) and different alternatives of invoking CIs. Software emulation (always unavailable) introduces CI invocations, but never executes them in hardware. Combination chooses either software emulation or stalling per kernel to optimize the timing bound (see Section V-C).

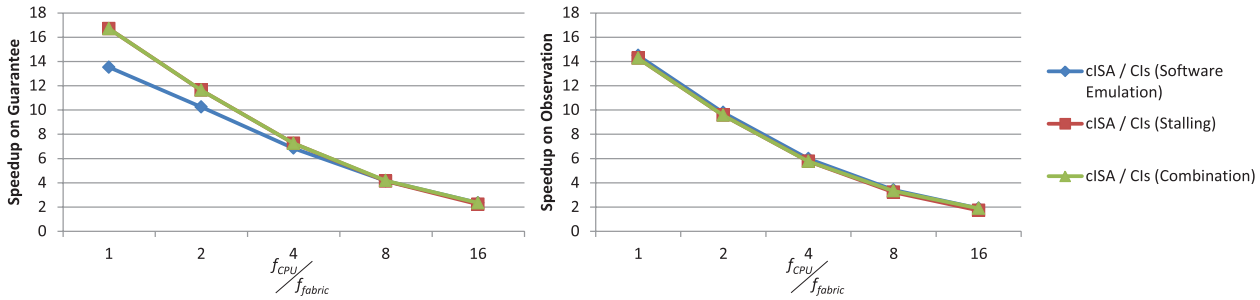

Fig. 12. H.264 overall speedup on the guaranteed timing bound (left) and the observed runtime (right).

stalling at $f_{\text{CPU}}/f_{\text{fabric}} = 4$, $f_{\text{ICAP}} = (1/2) \cdot f_{\text{fabric}}$, and already 2.02% lower at $f_{\text{ICAP}} = (1/4) \cdot f_{\text{fabric}}$. The observed execution time for software emulation is 10.33% and 13.11% lower than for stalling, respectively.

In sum, software emulation benefits from slow reconfiguration bandwidths or high CPU frequencies. In our results, to reduce the guaranteed time bound over stalling, the CPU needs to run at least at eight times the ICAP frequency due to pessimism. In the observed runtime, software emulation is always beneficial over stalling.

2) Overestimation: In this section, we analyze the influence of CIs on overestimation of WCET bounds for the H.264 encoder encoding 20 frames in QCIF resolution. Higher resolutions would increase the number of iterations per kernel and therefore diminish the effects of the reconfiguration delays on the total execution time. All results were taken with $f_{\text{ICAP}} = f_{\text{fabric}}$ and several values for $f_{\text{CPU}}/f_{\text{fabric}}$.

Fig. 11 shows the percentage of overestimation for a general-purpose version of our H.264 encoder without any CIs (cISA execution only), and several alternatives of using reconfigurable CIs. As the runtime in the general-purpose case is unaffected by the fabric frequency, the amount of overestimation is constant at 38.04%. Introducing additional control flow by inserting the conditions for software emulation without actually configuring CIs—denoted by software emulation (always unavailable)—increases the overestimation slightly to 39.93%.

As mentioned in Section VII-B1, the pessimism for bounding the timing anomaly when using software emulation is highest when the reconfiguration of CIs takes only few iterations of a specific kernel. Overestimation reaches its maximum for software emulation when $f_{\text{CPU}} = f_{\text{fabric}}$ with 47.95% and its minimum at $f_{\text{CPU}}/f_{\text{fabric}} = 16$ (maximum iterations during reconfiguration) with 11.89%. In sum, in our results, the overestimation is less for software emulation than for the general-purpose CPU when the pipeline frequency is twice the fabric frequency or higher. When using stalling, overestimation reaches its maximum of 17.97% when $f_{\text{CPU}} = f_{\text{fabric}}$ and its minimum of 6.66% when $f_{\text{CPU}}/f_{\text{fabric}} = 16$. It is generally lower than when using software emulation (see also Section V-B2) or cISA instructions only.

We can observe that increasing $f_{\text{CPU}}/f_{\text{fabric}}$ results in lower overestimation for both approaches for dealing with reconfiguration delay. This has two reasons.

1) Increasing $f_{\text{CPU}}/f_{\text{fabric}}$ results in more kernel iterations which can be executed during the reconfiguration delay of a CI using software emulation. This means that the pessimism of assuming an additional iteration in software to bound the timing anomaly (see Section V-B2) has a lesser share on the total iterations and therefore a lesser effect on the timing bound.
2) Increasing $f_{\text{CPU}}/f_{\text{fabric}}$ also increases the share of execution time (measured in CPU cycles) spent on the fabric. The execution time on the fabric does not introduce overestimation and therefore the total overestimation decreases.

Software emulation is affected by (1) and (2), while stalling is only affected by (2). Therefore, increasing $f_{\text{CPU}}/f_{\text{fabric}}$ has a stronger effect on the overestimation of software emulation than that of stalling.

Using the analysis of Section V-C, we use the combination of software emulation and stalling to choose the more beneficial approach per kernel. As a result, we apply software emulation at $f_{\text{CPU}}/f_{\text{fabric}} = 8$ for two out of three kernels and stalling for the other one. $f_{\text{CPU}}/f_{\text{fabric}} = 16$ is equivalent to software emulation only. It turns out that while overestimation is higher than using stalling in these cases, the resulting time bound is lower. This is achieved by our models guaranteeing that the reconfiguration delay can be hidden effectively. All other cases are equivalent to stalling only.

3) Speedup: Fig. 12 shows the speedup obtained by using runtime reconfiguration compared with execution on the cISA only. The left graph shows the speedup obtained in the guaranteed runtime, and the right graph shows the speedup of the observed runtime. As in Section VII-B2, all results in this section are obtained with $f_{\text{ICAP}} = f_{\text{fabric}}$. The speedup in the guaranteed runtime is higher than in the observed runtime for stalling and the combination of stalling and software emulation from Section VII-B2. The reason for this effect is that in addition to the actual speedup introduced by CIs, the overestimation is reduced as discussed in Section VII-B2. Therefore, the speedup in the predicted runtime is on average 24.43% higher (minimum 17.01% at $f_{\text{CPU}}/f_{\text{fabric}} = 1$ and maximum 29.42% at $f_{\text{CPU}}/f_{\text{fabric}} = 16$) than for the observed runtime when stalling. Software emulation results in a 11.5% higher speedup in the predicted runtime than in the observed runtime on average (minimum −6.70% at $f_{\text{CPU}}/f_{\text{fabric}} = 1$ and maximum 23.37% at $f_{\text{CPU}}/f_{\text{fabric}} = 16$).

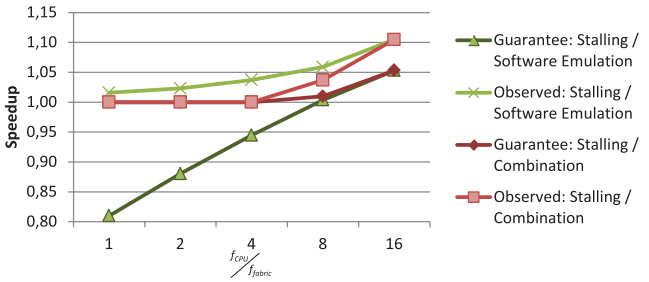

Fig. 13. Speedup of software emulation and combination over stalling only in H.264.

Fig. 13 takes stalling as a baseline and compares the guaranteed and observed results of software emulation and the combination of both approaches to it. Software emulation is always beneficial to the observed runtime with an average reduction of 4.8%, a minimum of 1.58% with $f_{\text{CPU}}/f_{\text{fabric}} = 1$, and a maximum of 10.48% with $f_{\text{CPU}}/f_{\text{fabric}} = 16$. For the guaranteed WCET bound, however, software emulation is beneficial only when the CPU pipeline runs faster than the fabric at a factor of $f_{\text{CPU}}/f_{\text{fabric}} = 8$ or more, because of overestimation (see Section VII-B2). Using software emulation for suitable kernels and stalling for others, the combination does not increase the runtime over stalling in any case. In cases where software emulation is beneficial to some kernels, the combination achieves the same or better guaranteed runtime reduction than using software emulation only.

VIII. Conclusion

We have presented a novel timing analysis approach for tasks on runtime reconfigurable processors, and it supports static analysis of runtime reconfiguration of multiple CIs and multiple execution contexts. In our evaluation, we have shown the precision of the analysis and the benefit of using CIs on WCET reduction and reduced overestimation. We compared the effects on safe estimated WCET bounds of executing CI-equivalent software (software emulation) with halting execution (stalling) during the reconfiguration delay. In the observed worst case, software emulation was always beneficial over stalling. However, in the estimated time bound, software emulation was superior only when the CPU pipeline frequency was higher than the fabric frequency by a factor of eight or more as stalling can be analyzed more precisely. An analysis to choose either stalling or software emulation per kernel was introduced and evaluated to combine the benefits of both approaches. In sum, we have shown that runtime instruction set reconfiguration can be an enabling feature to provide timing-analyzable performance.

References

[1] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 7, pp. 966–978, Jul. 2009.

[2] P. Puschner and C. Koza, "Calculating the maximum, execution time of real-time programs," *Real-Time Syst.*, vol. 1, no. 2, pp. 159–176, 1989.

[3] J. Reineke *et al.*, "A definition and classification of timing anomalies," in *Proc. WCET*, vol. 4. 2006.

[4] R. Wilhelm *et al.*, "Static timing analysis for hard real-time systems," in *Verification, Model Checking, and Abstract Interpretation*. Berlin, Germany: Springer, 2010, pp. 3–22.

[5] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. ACM Symp. Principles Program. Lang.*, 1977, pp. 238–252.

[6] M. Schoeberl, "Time-predictable computer architecture," *EURASIP J. Embedded Syst.*, vol. 2009, Jan. 2009, Art. no. 2.

[7] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Syst.*, vol. 28, nos. 2–3, pp. 157–177, 2004.

[8] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *Proc. ACM Design Autom. Conf.*, 2007, pp. 264–265.

[9] P. Axer *et al.*, "Building timing predictable embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 4, 2014, Art. no. 82.

[10] S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao, "The Chimaera reconfigurable functional unit," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 12, no. 2, pp. 206–217, Feb. 2004.

[11] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, and E. M. Panainte, "The MOLEN polymorphic processor," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1363–1375, Nov. 2004.

[12] L. Bauer, M. Shafique, S. Kramer, and J. Henkel, "RISPP: Rotating instruction set processing platform," in *Proc. ACM 44th Design Autom. Conf.*, 2007, pp. 791–796.

[13] C. Steiger, H. Walder, and M. Platzner, "Operating systems for reconfigurable embedded platforms: Online scheduling of real-time tasks," *IEEE Trans. Comput.*, vol. 53, no. 11, pp. 1393–1407, Nov. 2004.

[14] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS processor with a reconfigurable coprocessor," in *Proc. IEEE Symp. Field-Program. Custom Comput. Mach.*, Apr. 1997, pp. 12–21.

[15] R. D. Wittig and P. Chow, "OneChip: An FPGA processor with reconfigurable logic," in *Proc. IEEE Int. Symp. FPGAs Custom Comput. Mach.*, Apr. 1996, pp. 126–135.

[16] M. Dales, "Managing a reconfigurable processor in a general purpose workstation environment," in *Proc. IEEE Comput. Soc. Conf. Design, Autom. Test Eur.*, Mar. 2003, pp. 980–985.

[17] L. Bauer, M. Shafique, and J. Henkel, "A computation- and communication- infrastructure for modular special instructions in a dynamically reconfigurable processor," in *Proc. IEEE Int. Conf. Field Program. Logic Appl.*, Sep. 2008, pp. 203–208.

[18] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 2, 2011, Art. no. 18.

[19] J. Henkel *et al.*, "Invasive manycore architectures," in *Proc. 17th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, 2012, pp. 193–200.

[20] J. Cong, M. A. Ghodrat, M. Gill, B. Grigorian, H. Huang, and G. Reinman, "Composable accelerator-rich microprocessor enhanced for adaptivity and longevity," in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)*, Piscataway, NJ, USA, 2013, pp. 305–310.

[21] A. Grudnitsky, L. Bauer, and J. Henkel, "COREFAB: Concurrent reconfigurable fabric utilization in heterogeneous multi-core systems," in *Proc. ACM Int. Conf. Compil., Archit. Synthesis Embedded Syst.*, 2014, Art. no. 5.

[22] P. Yu and T. Mitra, "Satisfying real-time constraints with custom instructions," in *Proc. IEEE Int. Conf. Hardw./Softw. Codesign Syst. Synthesis*, Sep. 2005, pp. 166–171.

[23] C. Park and A. C. Shaw, "Experiments with a program timing tool based on source-level timing schema," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 1990, pp. 72–81.

[24] J. Whitham and N. Audsley, "MCGREP—A predictable architecture for embedded real-time systems," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 2006, pp. 13–24.

[25] S. Uhrig, S. Maier, G. Kuzmanov, and T. Ungerer, "Coupling of a reconfigurable architecture and a multithreaded processor core with integrated real-time scheduling," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Symp.*, Apr. 2006, pp. 1–4.

[26] S. Uhrig, S. Maier, and T. Ungerer, "Toward a processor core for real-time capable autonomic systems," in *Proc. IEEE Int. Symp. Signal Process. Inf. Technol.*, Dec. 2005, pp. 19–22.

[27] F. Dittmann and S. Frank, "Hard real-time reconfiguration port scheduling," in *Proc. IEEE Conf. Design, Autom. Test Eur.*, Apr. 2007, pp. 1–6.

[28] H. P. Huynh and T. Mitra, "Runtime reconfiguration of custom instructions for real-time embedded systems," in *Proc. IEEE Conf. Design, Autom. Test Eur.*, Apr. 2009, pp. 1536–1541.

[29] A. Luppold, B. Menhorn, H. Falk, and F. Slomka, "A new concept for system-level design of runtime reconfigurable real-time systems," *ACM SIGBED Rev.*, vol. 10, no. 4, pp. 57–60, 2013.
[30] A. Agne *et al.*, "ReconOS: An operating system approach for reconfigurable computing," *IEEE Micro*, vol. 34, no. 1, pp. 60–71, Jan./Feb. 2014.
[31] J. A. Clemente, J. Resano, and D. Mozos, "An approach to manage reconfigurations and reduce area cost in hard real-time reconfigurable systems," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 4, Mar. 2014, Art. no. 90.
[32] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET centric data allocation to scratchpad memory," in *Proc. 26th IEEE Int. Real-Time Syst. Symp. (RTSS)*, Washington, DC, USA, Dec. 2005, pp. 223–232.
[33] R. Wilhelm *et al.*, "The worst-case execution-time problem—Overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, May 2008, Art. no. 36.
[34] Y.-T. S. Li, S. Malik, and A. Wolfe, "Efficient microarchitecture modeling and path analysis for real-time software," in *Proc. IEEE Real-Time Syst. Symp.*, Dec. 1995, pp. 298–307.
[35] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise WCET prediction by separated cache and path analyses," *Real-Time Syst.*, vol. 18, nos. 2–3, pp. 157–179, 2000.
[36] C. Rochange and P. Sainrat, "A context-parameterized model for static analysis of execution times," in *Transactions on High-Performance Embedded Architectures and Compilers II*. Berlin, Germany: Springer, 2009, pp. 222–241.
[37] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand, "Analysis of loops," in *Compiler Construction*. Berlin, Germany: Springer, 1998, pp. 80–94.
[38] J. Henkel, L. Bauer, M. Hübner, and A. Grudnitsky, "i-Core: A run-time adaptive processor for embedded multi-core systems," in *Proc. Int. Conf. Eng. Reconfigurable Syst. Algorithms*, 2011, pp. 1–8.
[39] *aiT Worst-Case Execution Time Analyzers*, accessed on Sep. 13, 2015. [Online]. Available: http://www.absint.com/ait/

Marvin Damschen (M'15) received the B.Sc. and M.Sc. degrees in computer science with a minor in mathematics from the University of Paderborn, Paderborn, Germany, in 2012 and 2014, respectively. He is currently pursuing the Ph.D. degree with the Chair for Embedded Systems, Karlsruhe Institute of Technology, Karlsruhe, Germany, under the supervision of Prof. J. Henkel.

His current research interests include timing analysis and architectures for real-time embedded systems with special focus on runtime reconfigurable architectures.

Lars Bauer (M'10) received the M.Sc. and Ph.D. degrees in computer science from the Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, in 2004 and 2009, respectively.

He is currently a Research Group Leader and Lecturer with the Chair for Embedded Systems, KIT. His current research interests include architectures and management for adaptive multi-/many-core systems.

Dr. Bauer received two dissertation awards (EDAA and FZI), two best paper awards (AHS'11 and Design Automation and Test in Europe'08), and several nominations.

Jörg Henkel (M'95–SM'01–F'15) received the M.S. and Ph.D. (summa cum laude) degrees from the Technical University of Braunschweig, Braunschweig, Germany.

He is currently with the Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, where he directs the Chair for Embedded Systems (CES). He was with NEC Laboratories, Princeton, NJ, USA.He holds ten U.S. patents. His current research interests include design and architectures for embedded systems with focus on low power and reliability. He is an Initiator and a Spokesperson of the national priority program on Dependable Embedded Systems of the German Science Foundation and the Site Co-Ordinator (Karlsruhe site) of the three-university collaborative research center on invasive computing.

Prof. Henkel has received the 2008 DATE best paper award, the 2009 IEEE/ACM William J. Mc Calla International Conference On Computer Aided Design (ICCAD) best paper award, and the International Conference on Hardware/Software Codesign and System Synthesis 2011, 2014, and 2015 best paper awards. He was the General Chair of major CAD events including ICCAD and ESWeek. He is the Chairman of the IEEE Computer Society, Germany Section, and was the Editor-in-Chief of the ACM Transactions on Embedded Computing Systems for two terms. He is currently the Editor-in-Chief of IEEE Design&Test.