WCET Guarantees for Opportunistic Runtime Reconfiguration

Marvin Damschen, Lars Bauer, Jörg Henkel

Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT), Germany
{damschen, lars.bauer, henkel}@kit.edu

***Abstract*—Time-critical systems need to be analyzable for timing guarantees. There is an increasing demand for *predictable performance* that modern processor architectures fail to provide since they focus on average-case performance only. Recent work has demonstrated that runtime reconfiguration of hardware accelerators via an FPGA is a viable way to achieve high performance for optimized worst-case execution time (WCET) guarantees. Since execution of the worst-case path is highly improbable, configuring accelerators for this path costs reconfigurable area that could better be used to accelerate more probable paths. This work presents the first approach that comprises (1) an online average-case execution time (ACET) optimization while (2) maintaining the optimized WCET guarantee utilizing reconfigurable accelerators. We achieve this by a new design-time technique which determines the runtime slack bounds that allow speculative reconfiguration of accelerators that benefit the ACET. Combined with an online slack monitoring approach that introduces negligible overheads by using a performance counter, we show a runtime reduction of up to 10.4% for a complex and real-world application on top of an already-optimized WCET guarantee.**

I. Introduction

Application domains like automotive, avionics as well as the rapidly growing domain of smart and autonomous machines (e.g., collaborative robots, automated driving, smart drones, etc.) have led to the omnipresence of time-critical systems in our everyday life [1]. For these systems, failing to meet a timing deadline can lead to severe malfunctions. Thus, as part of a *timing validation*, the maximum execution time under any circumstances, the so-called *worst-case execution time* (WCET) [2], needs to be guaranteed for every task that should run on such a system.

Time-critical systems have a rapidly increasing demand for *predictable performance* due to the vast amounts of sensor data that need to be processed, e.g., in autonomous driving. Modern processor architectures that optimize for average-case execution achieve high performance due to increasingly complex microarchitectures that employ, e.g., out-of-order scheduling pipelines, several hardware threads and multiple (shared) cache layers. However, these architectural features induce an explosion of microarchitectural states that renders WCET guarantees virtually infeasible [3].

An alternative to increasingly complex microarchitectures to achieve performance is via hardware accelerators. They provide high performance for a specific functionality and speedup a task's most compute-intensive parts, the so-called *computational kernels*. To maintain the flexibility of the system while employing hardware acceleration, embedded systems are increasingly designed to integrate field-programmable gate arrays (FPGA) that enable reconfiguration of different accelerators at runtime, e.g., reconfigurable embedded platforms like the Xilinx Zynq or Intel (formerly Altera) SoC FPGA combine general-purpose CPUs with a reconfigurable area on a single chip and enable the utilization of numerous accelerators on a constrained chip area.

Recent work has demonstrated that runtime reconfiguration of hardware accelerators is a viable way to achieve high performance that is analyzable for WCET guarantees [4–6]. The latency of a reconfigurable hardware accelerator is under direct control of the application designer and is often precisely known. Where in average-case optimizing systems the reconfigurable area is allocated to accelerators that result in the best speedup *on average*, the main constraint in time-critical systems is to statically guarantee WCET bounds. Thus, reconfigurable area is allocated to accelerators that reduce the execution time of the statically-determined worst-case path of a task. However, executing the worst-case path and finishing in WCET is usually highly improbable [2]. In fact, WCET analysis approximates the WCET of a task by an upper execution time bound and thus, the actual runtime of the task will virtually *always* be faster than the guaranteed WCET as shown in Fig. 1 (left). Ultimately this means that (1) the average-case execution time (ACET) of a task is generally considerably lower than the WCET and (2) configuring accelerators to optimize the WCET of a task can waste reconfigurable area on program paths that might never be executed in practice. It is, however, highly desirable to achieve a high utilization of accelerators and optimize average-case execution to fulfill additional non-functional constraints like, e.g., power or thermal constraints.

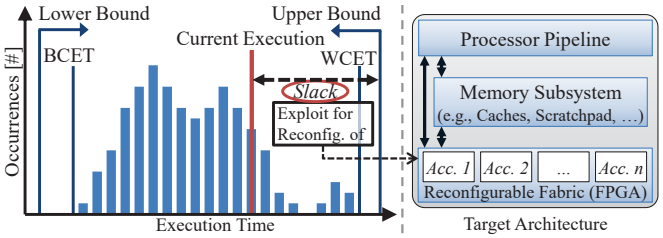

Fig. 1. The WCET of a task is bounded using static timing analysis (left). At runtime, *slack* becomes a resource that can be exploited for reconfiguration of accelerators (right).

Our novel contributions are as follows:

- first approach to optimize static WCET guarantees *and* runtime optimization of the ACET (maintaining WCET guarantees) using runtime reconfiguration of hardware accelerators
- analysis of slack bounds that enable safe reconfiguration for average-case performance under WCET guarantees
- an approach for monitoring the *slack* of a task (the amount of time it executed parts of code faster than in worst-case) using a simple performance counter

II. Related Work

While runtime reconfiguration of accelerators is an established concept in embedded systems in general [7], it gained traction in recent years for achieving performance in systems that need to fulfill hard real-time guarantees [1]. Several works in this direction are concerned with scheduling task sets that employ runtime reconfiguration [5, 8]. In [5] an overview over state-of-the-art real-time scheduling for reconfigurable systems is provided, and a scheduling framework as well as an analysis for periodic multi-priority real-time tasks is presented. The model divides tasks into software and hardware subtasks, where hardware subtasks model the execution of hardware accelerators on a reconfigurable fabric. Software subtasks request reconfiguration and execution of

978-1-7281-2350-9/19/$31.00 ©2019 IEEE

hardware subtasks and self-suspend until the hardware subtask has finished execution.

Whereas previously-mentioned scheduling approaches assume that each task comes with a fixed set of hardware accelerators that needs to be configured, the work of [8] addresses the problem of finding a set of configurations (of a processor with a reconfigurable instruction set) for periodic task graphs that enable timing constraints to be met. A periodic task graph with deadlines is scheduled and the schedule is partitioned into configurations for the reconfigurable fabric. Each configuration is assigned an instruction set to optimize the tasks' WCET. Further work on finding sets of hardware accelerators that minimize the WCET of a task are available for non-reconfigurable [9] and reconfigurable [10] processors. While all of the above-mentioned approaches utilize runtime reconfiguration to fulfill real-time constraints, none of the approaches considers online optimization of average-case execution once constraints are met. Thus, they are unable to optimize for dynamic workloads in complex applications like signal processing or computer vision applications.

When a task executes faster than the worst case, the execution time difference between the guaranteed WCET and the current execution, i.e., the *slack* (see Fig. 1), becomes a resource that can be used to optimize additional constraints, e.g., opportunistic monitoring for security or reliability issues has been presented [11]. Several works on slack exploitation target energy reduction [12–14], either using slack reclamation or procrastination scheduling approaches: Slack reclamation dynamically schedules tasks at a lower DVFS level when slack allows slower, more energy-efficient execution [13, 14]. Procrastination scheduling uses slack to delay tasks for maximized durations of processor sleep [12]. While these approaches *use* slack within a schedule for lower energy consumption in a non-speculative way (the slowdown at lower DVFS levels is a priori known), our approach *increases* slack by speculatively configuring accelerators that benefit ACET instead of WCET. Despite speculation, we maintain WCET guarantees. The resulting slack may be used for slack reclamation, however, combining both approaches is outside the scope of this paper.

Our work also bears resemblance to mixed-criticality scheduling [15], especially *HI-LO* mode switching where the system starts in the *LO* mode with tightened deadlines of high-critical tasks to enable scheduling of additional low-critical tasks. In case a high-critical task exceeds its tightened deadline, the system enters the *HI* mode that guarantees the deadlines of all high-critical tasks and abandons low-critical tasks. Switching to the *HI* mode is generally considered a fault and a switch back to *LO* is usually not considered [15]. In contrast, our work explicitly targets switching between different accelerator configurations within a single task as part of the system's normal operation and without violating deadlines.

Orthogonal to the aforementioned works, slack was used to optimize the ACET using a complex (i.e., hard to analyze but high-performance) microarchitecture that provides a simple, timing-analyzable architecture mode [16]. In case insufficient slack is detected at runtime, the architecture enters the simple mode that is the basis for WCET guarantees. However, this approach does not provide optimization of WCET guarantees.

In summary, state-of-the-art approaches either do not consider slack but focus on utilizing runtime reconfiguration of accelerators for optimizing worst-case execution only, or they do consider slack but not for a more effective average-case use of a reconfigurable area. In this work, both properties are achieved for the first time: we present an approach that optimizes the WCET of a task using accelerators *and* the average-case performance using reconfiguration within the runtime slack.

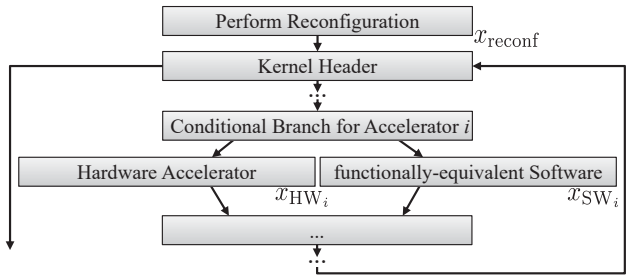

Fig. 2. CFG of a kernel that configures and utilizes accelerators in the Stalling Model. Not all utilized accelerators need to be configured (constrained area), but can be executed in a functionally-equivalent software implementation.

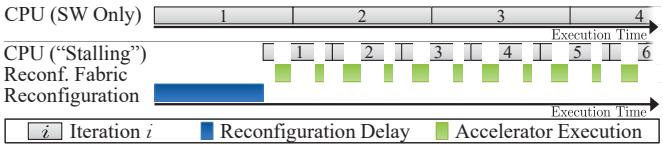

Fig. 3. Visualization of execution without utilizing accelerators (top) and with configuring accelerators before executing a kernel (bottom).

III. System Model

To model the deployment of reconfigurable hardware accelerators inside the control flow graph (CFG) of a task during WCET analysis, we base on the "stalling" model, which is commonly used in the real-time context [4, 5, 9]. The stalling model is depicted in Fig. 2 and Fig. 3. It is briefly summarized in the following. The stalling model extends the Implicit Path Enumeration Technique (IPET) [17] for WCET bound calculation, which is the program path analysis technique state-of-the-art timing analyzers rely on [2, 18]. IPET models program flow as arithmetic constraints in an ILP-formulated problem. The objective function determines the CPU cycles executed on a path in the task's CFG. To find the worst-case path (that determines the WCET), 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 a microarchitectural analysis. Similar to flow networks, the variables are constrained by modeling the control flow and capturing relative execution counts of basic blocks as ILP constraints. For a program with N basic blocks, the objective function is given as [17]:

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

Each kernel in the stalling model is preceded by a basic block that initiates reconfiguration of hardware accelerators that are used within the kernel body (x_{reconf} in Fig. 2). Reconfiguration takes a certain amount time, the so-called *reconfiguration delay*. The reconfiguration delay depends on the size of the configuration data and the bandwidth of the reconfiguration port. During reconfiguration, the task waits and only proceeds its execution after the reconfiguration delay has passed (this enables precise WCET estimates [4] and more high-level analyses like multi-priority task scheduling [5]). Afterwards, the CPU can execute the kernel and utilize hardware accelerators as shown in Fig. 3 (bottom). The functionality of a hardware accelerator (x_{HW_i}) is alternatively available as a software implementation (x_{SW_i}), e.g., the software implementation that the accelerator was derived from when using high-level synthesis (we assume the software implementation will always have a higher latency than the hardware accelerator). This way, more opportunities to generate hardware accelerators can be identified at compile time than actually accommodated by the reconfigurable area of the specific target platform (see Fig. 1 (right),

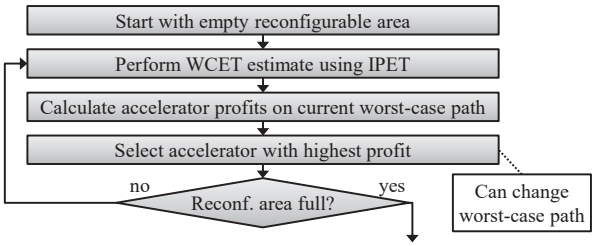

Fig. 4. WCET optimization (offline). Accelerators are selected to obtain the safe configuration. Each time an accelerator is selected, the WCET needs to be estimated again

an overview of compile-time methods is presented in [19]). Several ways to model the reconfigurable area exist [20]. This work is not limited to any specific reconfigurable area model. The problem of choosing a subset of hardware accelerators to optimize, e.g., average-case or worst-case runtime, is called *selection* [9, 10, 19]. At runtime, a conditional branch tests whether a specific accelerator was configured and is available in hardware. If this is the case, the functionality is executed by the hardware accelerator, and in software otherwise.

The following section gives a brief description of how selection can be solved for WCET- and performance-optimizing configurations based on the presented model. Afterwards, extensions to the model are presented that enable multiple reconfigurations within a kernel to switch between different configurations at runtime.

IV. Opportunistic Reconfiguration

The main idea of our work is to employ two configurations for the reconfigurable fabric: a time-wise *safe configuration* (a selection of accelerators that optimizes the WCET bound) and a *performance configuration* (a selection of accelerators that optimizes the ACET). At the start of a kernel, the reconfigurable area is configured using the *safe configuration*. During execution the task's slack towards the WCET guarantee, i.e., the amount of time it executed parts of code faster than the worst case, is continuously sampled and accumulated. Once sufficient slack is accumulated, the *performance configuration* is configured onto the reconfigurable area and the average-case execution is accelerated. The *performance configuration* results in a higher WCET bound than the *safe configuration*, and in the worst case it could experience a slower execution than what was guaranteed, i.e., the accumulated slack could be reduced. Therefore, special care needs to be taken for the case that the slack might be depleted. We might need to switch back to the *safe configuration* to avoid this case, and we need sufficient slack left not to violate the guaranteed WCET bound. With high probability, however, the execution of the *performance configuration* will be faster (it optimizes the average case) and the task will finish with a lower execution time than when executing in the *safe configuration*. Our new speculative approach is a major difference to, e.g., slack reclamation approaches mentioned in Section II.

Our approach consists of an offline preparation phase in which the safe and performance configurations are created and their information is annotated to the task under optimization, as well as the actual online optimization phase that performs reconfigurations within the WCET bound using slack.

A. Offline Preparation

The input to the offline preparation is the reconstructed control-flow graph (CFG) of the task's binary, which is obtained as the first step during WCET analysis [2].

1) Safe Configuration: Several methods to obtain a suitable safe configuration exist [9, 10]. In the following, a greedy algorithm is described that repeatedly performs WCET estimation of the CFG and accelerator selection using the stalling model as shown in Fig. 4. The first WCET estimate will result in a worst-case path that does not utilize any hardware accelerator: because the software implementation (x_{SW_i} in Fig. 2) has a longer latency than the hardware accelerator, it is the worst-case choice. After each WCET estimate the accelerator that provides the maximum profit on the current worst-case path is selected and added to the temporary safe configuration, where the profit of an accelerator a is estimated as:

$$\mathrm{profit}_{\mathrm{WCET}}(a) = \sum_{x_i \text{ calls functionality of } a} x_i \cdot (\mathrm{latency}_{\mathrm{sw}} - \mathrm{latency}_{\mathrm{hw}})$$

I.e., the total profit of selecting a is estimated to be the latency difference between its software implementation and hardware accelerator multiplied by the total amount of times a is executed in the *current* worst-case path. After annotating the selection of accelerator a' with maximum profit to the CFG such that all calls to a' utilize the hardware accelerator (instead of functionality-equivalent software), another WCET estimate needs to be performed. The worst-case path might have changed, because the execution time of the previous worst-case path was reduced. WCET estimation and accelerator selection are repeated alternately until the whole reconfigurable area of the target platform is occupied.

1) Performance Configuration: The performance configuration is obtained with a similar greedy algorithm. We utilize the same latency estimates ($\mathrm{latency}_{\mathrm{sw}}$ and $\mathrm{latency}_{\mathrm{hw}}$) that were obtained during WCET estimation. Instead of worst-case path information, however, profiling information is used to determine the profit of an accelerator, i.e., the application is run for a typical use case to determine the number of calls n_a to each accelerator a. Thus, the profit on the ACET is estimated as:

$$\mathrm{profit}_{\mathrm{ACET}}(a) = n_a \cdot (\mathrm{latency}_{\mathrm{sw}} - \mathrm{latency}_{\mathrm{hw}})$$

Again, accelerators are greedily added to the performance configuration until the whole reconfigurable area is occupied. In contrast to the worst-case path, the average-case calls to accelerators do not change when additional accelerators are selected. Therefore, only a single profiling run is required.

1) WCET Bounds: The final offline preparation step is to determine WCET bounds for program parts and configurations of the task that are used to perform decisions during online optimization. The task's guaranteed WCET bound is determined based on the safe configuration. Additionally, WCET bounds are determined for single iterations of each kernel. As shown in Fig. 2, a single iteration starts with the kernel header and ends with a branch from the end of the kernel body back to the header. The bounds of an iteration are determined as $\mathrm{WCET}^{\mathrm{safe}}_{\mathrm{iter}}$ and $\mathrm{WCET}^{\mathrm{perf}}_{\mathrm{iter}}$ for the safe and performance configuration, respectively. This information is required for online slack monitoring and to decide when to switch the configuration as explained in the following section.

B. Online Optimization

2) Slack Monitoring: To enable online slack monitoring, we utilize a simple performance counter that counts CPU cycles similar to the cycle count register of the "Performance Monitoring Unit" in ARM Cortex-R cores used in the Xilinx Zynq UltraScale+ platform. For brevity, we define four operations that control counting and accumulation of CPU cycles: `mobeg`, `moend`, `rdslck` and `wrslck`. These operations are added to the kernels that should be optimized as shown in Listing 1. At the beginning of each iteration of a kernel, the counter is initialized with the kernel's per-iteration WCET bound $\mathrm{WCET}^{\mathrm{safe}}_{\mathrm{iter}}$ using `mobeg`, which copies an unsigned integer value from a register argument into the counter. Then, during the kernel iteration, the counter is decremented in every cycle. At the end of the kernel iteration, `moend` stops the

```
1 wrslck 0                                   (reset slack accumulator)
2 for (...){                                 (kernel header)
3     mobeg WCET^safe_iter                   (set counter for current iteration)
4     rdslck slack                           (read accumulated slack)
5     conditionally reconfigure based on slack, see Fig. 5
6     ...                                    (original kernel body)
7     moend}                                 (add counter value to accumulated slack)
```

Listing 1. Operations introduced for slack monitoring

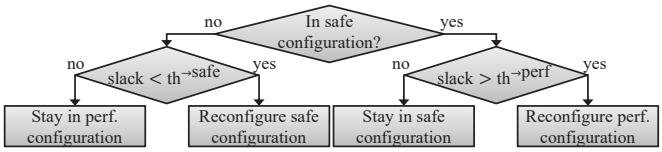

Fig. 5. Decision (software) in each kernel iteration based on current slack and thresholds ($th^{\rightarrow perf}$ and $th^{\rightarrow safe}$)

counter and adds the current value to the accumulator. The counter is guaranteed to have a value ≥ 0 when the kernel is executed in the safe configuration, because the counter was initialized with $WCET^{safe}_{iter}$. In the performance configuration, however, the counter could count down until $WCET^{safe}_{iter} - WCET^{perf}_{iter}$ ($WCET^{perf}_{iter} > WCET^{safe}_{iter}$, therefore negative). In this case, the accumulated slack is reduced. The currently accumulated slack is read using `rdslck`, which copies the accumulator value to a register argument. In case the accumulated slack is below a certain threshold while in the performance configuration, the safe configuration needs to be configured again. In which way the thresholds are obtained and enforced using the conditional reconfiguration (Line 5) is described in the following.

2) Reconfiguration within WCET Bounds: While the model described in Section III enables reconfiguration only before entering a kernel, we introduce conditional reconfiguration that decides whether to reconfigure or not in each iteration of the kernel based on the currently accumulated slack as shown in Fig. 5. As shown in Listing 1, conditional reconfiguration is executed before the kernel body in every kernel iteration (Line 5). It manages the state of the reconfigurable area and triggers a reconfiguration in case the accumulated slack reached a certain threshold. To determine the minimum threshold of accumulated slack that enables to switch from safe to performance configuration (while maintaining WCET guarantees), the worst-case execution –immediately after reconfiguration of the performance configuration was triggered– needs to be considered. First, it takes $rdelay^{perf}$ cycles to configure the performance configuration. Then, the kernel iteration takes maximum $WCET^{perf}_{iter}$ cycles, which means that the accumulated slack is reduced by $|WCET^{safe}_{iter} - WCET^{perf}_{iter}|$. Finally, when the conditional reconfiguration block is executed again and we need to switch back to the safe configuration, it takes $rdelay^{safe}$ to perform the reconfiguration. In summary, the minimum accumulated slack to be able to switch from safe to performance configuration and remain within the WCET guarantee in the worst case is:

$$th^{\rightarrow perf} := rdelay^{perf} + |WCET^{safe}_{iter} - WCET^{perf}_{iter}| + rdelay^{safe} \quad (2)$$

As mentioned before, a kernel iteration reduces the accumulated slack by $|WCET^{safe}_{iter} - WCET^{perf}_{iter}|$ in the worst case and configuring the safe configuration takes $rdelay^{safe}$. Thus, to safely switch back from performance to safe configuration, the reconfiguration needs to be triggered once the accumulated slack is lower than:

$$th^{\rightarrow safe} := |WCET^{safe}_{iter} - WCET^{perf}_{iter}| + rdelay^{safe} \quad (3)$$

For any value $> th^{\rightarrow safe}$, there is still enough accumulated slack to safely execute one iteration of the kernel (even in worst case) and switch back to the safe configuration afterwards.

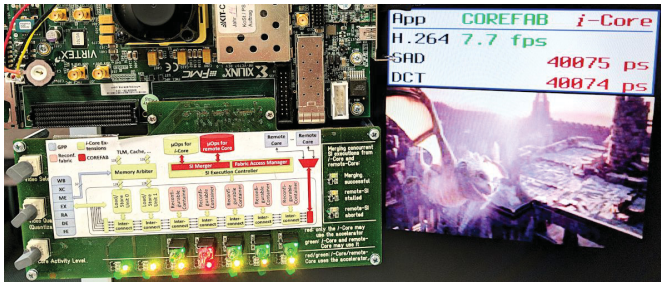

Fig. 6. Virtex-7-based prototype executing H.264

During the offline preparation phase (explained Section IV-A), the slack is annotated as constant 0 such that only the initial reconfiguration to the safe configuration is accounted for during WCET analysis (just like without applying our approach).

V. Experimental Evaluation

We evaluated this work on a runtime-reconfigurable processor [4, 21] that was extended for execution with hard real-time guarantees. The implementation is based on the Gaisler LEON3 SoC that implements a SPARC V8 in-order microarchitecture and supports several real-time operating systems[1]. Its 7-stage pipeline was extended for execution of reconfigurable accelerators: the Execute stage allows stalling the pipeline and passing operands to the accelerators. Consequently, accelerators are executed just like multi-cycle instructions that do not influence data cache analysis when determining WCETs of basic blocks. In this work, a simple performance counter (cycle counter plus accumulator) is added that implements the operations for slack monitoring explained in Section IV-B1. The offline preparation algorithms (Section IV-A) are implemented by running AbsInt aiT [22] in batch mode to obtain WCET estimates and iteratively generate constraints in aiT's AIS2 constraint language to model selected accelerators. As aiT is closed-source software, we could not directly integrate support for reconfigurable accelerators. Instead, every call to an accelerator in the binary was substituted by an ADD opcode and a constraint that sets the delay for the new ADD instruction to the delay of the specific accelerator during WCET estimation. Guaranteed reconfiguration delays are obtained using a reconfiguration controller like [6] and are annotated using additional constraints.

Our approach is suitable for any application that exhibits a different utilization of hardware accelerators for different inputs. This is generally the case for complex real-world applications such as the H.264 encoder, which is evaluated in the following. The H.264 encoder uses 9 accelerators (ca. 89 KiB bistream size each), which cover the most compute-intensive kernels shown in Table I. Each kernel reconfigures the full reconfigurable area (modeled to be of similar size like Xilinx Zynq XC7Z010). Multimedia applications are regularly subject to hard real-time constraints in the domain of computer vision (e.g., vehicle detection and tracking [23] or face recognition in digital cameras [24]). The H.264 encoder is a real-world application that contains complex control flow and compute-intensive algorithms that represent diverse computational kernels. It covers most of the properties tested in the Mälardalen[3] or TACLeBench[4] WCET Benchmarks, e.g., Discrete Cosine Transform is contained in all three and the H.264 decoder – that is part of TACLeBench– performs a subset of the computations performed in the H.264 encoder that we evaluate. We compiled

[1]https://www.gaisler.com/leon3

[2]C Lines of Code that are replaced by utilizing an accelerator (without comments or whitespace)

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

[4]http://www.tacle.eu/index.php/activities/taclebench

TABLE I
ACCELS. IN DIFFERENT PATHS OF THE H.264 APPLICATION

Accelerated func. and Descript.	MB type	Working Set	CLoC[2]
MotionEstimation Kernel			
SATD: Sum of Abs. Transf. Differences	P and I	16×16 px	123
SAD: Sum of Abs. Differences	P and I	16×16 px	24
EncodeMacroBlock Kernel			
MC_Hz: Motion Comp. Interpol. Horiz.	P	4 px	51
IPred_HDC: Intra Prediction Horiz.	I	16×16 px	35
IPred_VDC: Intra Prediction Vert.	I	16×16 px	19
DCT: Discrete Cosine Transf.	P and I	4×4 px	76
HT2x2: Hadamard Transform	P and I	2×2 px	12
HT4x4: Hadamard Transform	I	4×4 px	111
LoopFilter Kernel			
LoopFilter: In-Loop Deblock. Filter	P and I	4 px	82

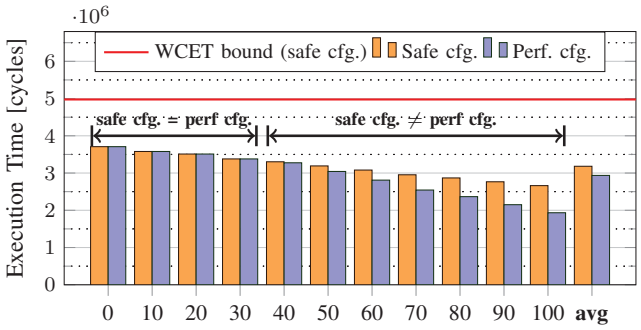

Fig. 7. Execution time of EncodeMacroBlock for safe configuration and different performance configurations obtained for different execution profiles

the application using BCC 4.4.2 (Gaisler's extended GCC 4.4.2) at O1[5] for a frame size of 396 macroblocks (i.e., CIF resolution). Note that higher resolutions would benefit our approach by reducing the relative overhead of reconfiguration delays. Measured execution times are obtained using the cycle-accurate SystemC-based simulator of the reconfigurable processor. Before performing the evaluation, we calibrated aiT and the simulator by harmonizing hardware parameters and verifying the results of test-cases, e.g., load-store sequences. Hardware parameters like reconfigurable area constraints, reconfiguration bandwidth and partial bitstream sizes of accelerators were obtained from a Xilinx Virtex-7-based hardware prototype of the complete reconfigurable processor (see Fig. 6). The CPU is modeled with 400 MHz (LEON3's frequency when implemented as an ASIC), the reconfigurable area at 100 MHz (frequency of accelerators as implemented on a Virtex-7).

A. Results

1) Offline Preparation: The H.264 encoder has two main execution paths with different accelerator requirements, depending on whether a macroblock (MB) is encoded either by referencing an MB from a previous frame (P-MB) or using the current frame only (I-MB). Table I shows which accelerators are used in the I-MB or P-MB path (or both). We will focus on the EncodeMacroBlock kernel that performs the actual encoding. It is the most complex kernel and the one that provides an opportunity to create distinct safe and performance configurations (see Table I). Which of the total 396 MBs is encoded as either I-MBs or P-MBs at runtime is input-dependent: hectic video scenes have a high ratio of I-MBs (up to 100%, e.g., video from a camera pointing sideways out of a moving car), steady scenes have a low ratio of I-MBs. Therefore, the performance configuration differs for different execution profiles (synthesized input data that is encoded as I-MBs and P-MBs randomly distributed in the desired ratio). During WCET optimization, however, the current worst-case path encodes all MBs either as P-MBs or I-MBs (the worst-case path changes while preparing the safe configuration, see Fig. 4). The final safe configuration (see Section IV-A1) selects MC_Hz, DCT, HT2x2 and HT4x4. In this configuration, the worst-case path does not encode I-MBs but only P-MBs.

Figure 7 shows execution time results for different execution profiles (% of I-MBs, resulting in different performance configurations) of the EncodeMacroBlock kernel. The performance configuration differs from the safe configuration when 40% or more of the total MBs in a frame are I-MBs. In these cases, IPred_HDC and IPred_VDC are selected instead of MC_Hz and a performance increase of up to 27.4% is achieved at $x = 100$ (7.7% on average, without reconfiguration delay). Unaccelerated execution takes between $20.1 \cdot 10^6$ ($x = 0$) and $19.4 \cdot 10^6$ ($x = 100$) cycles, i.e., the measured speedup of the safe configuration is between $7.1\times$ and $5.4\times$.

1) Online Optimization: On top of the speedup obtained by the safe configuration, the online optimization aims to reduce the average-case execution time. As shown in Fig. 7, our approach is suitable to the H.264 encoder when video frames with $\geq 40\%$ I-MBs can be expected (i.e., at least moderate movement). We proceed using the performance configuration obtained in this case (IPred_HDC, IPred_VDC, DCT, HT2x2, HT4x4) in the following for all execution profiles. In this case, $|\text{WCET}^{\text{safe}}_{\text{iter}} - \text{WCET}^{\text{perf}}_{\text{iter}}| = |12630 - 16374| = 3744$, i.e., an iteration encoding P-MBs takes 3744 cycles longer for the performance configuration compared to safe configuration in the worst case. Figure 8 shows execution time results of the EncodeMacroBlock kernel when our approach is applied for different reconfiguration bandwidths and I-MB ratios. A reconfiguration bandwidth of 400 MB/s (left) is supported by our Xilinx Virtex-7-based hardware prototype, while recent Xilinx FPGAs ("UltraScale+") support a reconfiguration bandwidth of 800 MB/s[6] (right). Switching between safe and performance configuration takes 91307 and 45658 cycles for the respective bandwidths. In both cases, our approach is beneficial for frames that contain $\geq 50\%$ of I-MBs. Even when the performance configuration results in a lower execution time at 40% I-MBs (see Fig. 7), the speedup is voided by the additional reconfiguration delay (of switching from safe to performance configuration after accumulating sufficient slack). The maximum execution time reduction is 19.8% and 23.0% for a reconfiguration bandwidth of 400 MB/s and 800 MB/s, respectively. For frames that contain $< 50\%$ of I-MBs, the execution time can be slowed down (as expected) by up to 11% ($x = 0$ for both reconf. bandwidths). When considering the execution profiles targeted by the performance configuration ($\geq 40\%$ of I-MBs), the average execution time reduction ($\frac{\text{avg}}{\geq 40}$) is 8.1% and 10.2% for a reconfiguration bandwidth of 400 MB/s and 800 MB/s, respectively. This is 97% and 98% of the theoretically possible performance improvement, additional reconfiguration delay is the main overhead, but crucial to apply this technique. Including cases not targeted by the performance configuration (the safe configuration performs better), the total average execution time reduction is 1.5% and 3.0% (95% and 96% of theoretically possible) for the respective reconfiguration bandwidth. The statically guaranteed WCET bounds were never hit in our experiments.

Finally, ca. 45% (between 38.2% and 51.7% in our measure-

[5]At higher optimization levels, GCC emitted so-called *irreducible loops* that increase the WCET estimate compared to O1.

[6]See: "Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics" (DS923)

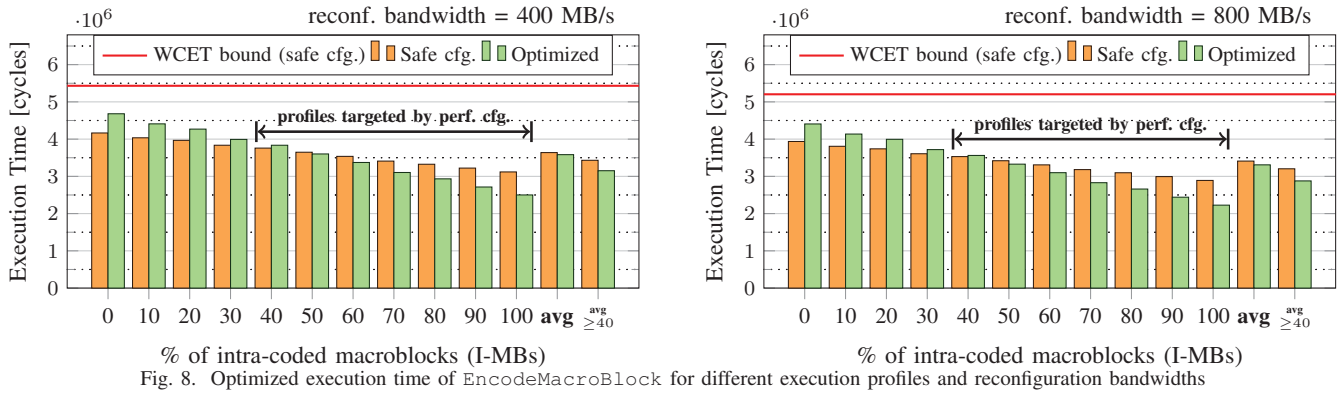

Fig. 8. Optimized execution time of `EncodeMacroBlock` for different execution profiles and reconfiguration bandwidths

ments) of the H.264 encoder's execution time are spent in the `EncodeMacroBlock` kernel, which our technique is applied to. As in other realistic applications, the remaining execution time is spent in less complex kernels (using one or two accelerators). Thus, our technique approximately reduces the total execution time on average in the targeted execution profiles ($_{\geq 40}^{\text{avg}}$) by 3.6% and 4.6%; as well as up to 8.9% and 10.4% for a reconfiguration bandwidth of 400 MB/s and 800 MB/s, respectively. On total average (including atypical execution profiles like still images), the execution time is reduced by by 0.7% and 1.4% (again, for a reconfiguration bandwidth of 400 MB/s and 800 MB/s, respectively). Note that the execution time reductions were achieved compared to the already-optimized safe configuration, by only adding slack monitoring using a single performance counter as described in Section IV-B to a proven runtime-reconfigurable processor design. The results show that our technique can effectively provide a unique feature using runtime reconfiguration: not only can an optimized worst case be guaranteed, but at the same time the reconfigurable area can be leveraged to optimize for average-case performance.

VI. Conclusion

This work is the first step towards optimized static WCET guarantees *and* runtime optimization of the ACET using runtime reconfiguration of hardware accelerators. It might very well enable the use of runtime reconfiguration for performance in time-critical systems –where nowadays only static configurations are used– as it enables utilization of slack for reconfiguration of accelerators that benefit average-case execution while maintaining optimized WCET guarantees. At negligible overheads (a performance counter was added to a runtime-reconfigurable processor), our technique allows at runtime to repurpose reconfigurable area from a WCET-optimizing configuration for more-probable program paths than the (usually improbable) worst-case path. This resulted in average-case execution time reductions on top of an already-optimized WCET guarantee: up to 8.9% and 10.4% (on average 3.6% and 4.6% for targeted profiles) at a reconfiguration bandwidth of 400 MB/s and 800 MB/s, respectively, for the whole H.264 encoder application (up to 19.8% and 23.0% for a single kernel, on top of a speed up of 5.4× to 7.1× that was achieved by the WCET-optimizing *safe configuration*).

Acknowledgment

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89).

References

[1] T. Mitra, J. Teich, and L. Thiele, "Time-Critical Systems Design: A Survey," *IEEE Design & Test*, vol. 35, no. 2, pp. 8–26, 2018.

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

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

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

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

[6] L. Pezzarossa, M. Schoeberl, and J. Sparsø, "A Controller for Dynamic Partial Reconfiguration in FPGA-Based Real-Time Systems," in *IEEE Intl. Symp. on Real-Time Dist. Comput. (ISORC)*. IEEE, 2017, pp. 92–100.

[7] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable Computing Architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–354, 2015.

[8] H. P. Huynh and T. Mitra, "Runtime reconfiguration of custom instructions for real-time embedded systems," in *Proc. of the Conf. on Design, Automation and Test in Europe*. IEEE, 2009, pp. 1536–1541.

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

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

[11] D. Lo, M. Ismail, T. Chen, and G. E. Suh, "Slack-Aware Opportunistic Monitoring for Real-Time Systems," in *Real Time and Embed. Technol. and Applications Symp.* IEEE, 2014, pp. 203–214.

[12] R. Jejurikar and R. Gupta, "Dynamic Slack Reclamation with Procrastination Scheduling in Real-time Embedded Systems," in *Proc. of Design Automat. Conf.*, ser. DAC '05. ACM, 2005, pp. 111–116.

[13] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg, "FAST: Frequency-aware Static Timing Analysis," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 1, pp. 200–224, Feb. 2006.

[14] Y. C. Lee and A. Y. Zomaya, "On effective slack reclamation in task scheduling for energy reduction." *JIPS*, vol. 5, no. 4, pp. 175–186, 2009.

[15] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, 2013.

[16] A. Anantaraman, K. Seth, K. Patil *et al.*, "Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-time Systems," *SIGARCH Comput. Archit. News*, vol. 31, no. 2, pp. 350–361, May 2003.

[17] Y.-T. S. Li, S. Malik, and A. Wolfe, "Efficient microarchitecture modeling and path analysis for real-time software," in *Proc. of Real-Time Syst. Symp.* IEEE, 1995, pp. 298–307.

[18] S. Altmeyer, B. Lisper, C. Maiza *et al.*, "WCET and mixed-criticality: What does confidence in WCET estimations depend upon?" in *OASIcs-OpenAccess Series in Informatics*, vol. 47. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

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

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

[21] M. Damschen, L. Bauer, and J. Henkel, "CoRQ: Enabling Runtime Reconfiguration Under WCET Guarantees for Real-Time Systems," *IEEE Embedded Systems Letters (ESL)*, vol. 9, no. 3, pp. 77–80, 2017. [Online]. Available: https://doi.org/10.1109/LES.2017.2714844

[22] AbsInt, "aiT Worst-Case Execution Time Analyzers," Website: http://www.absint.com/ait/, 2018, [Online; accessed 30-Dec-2018].

[23] M. Betke, E. Haritaoglu, and L. S. Davis, "Real-time multiple vehicle detection and tracking from a moving vehicle," *Machine vision and applications*, vol. 12, no. 2, pp. 69–83, 2000.

[24] M. Yang, Y. Wu, J. Crenshaw *et al.*, "Face detection for automatic exposure control in handheld camera," in *IEEE Intl. Conf. on Comput. Vision Syst. (ICVS)*. IEEE, 2006, pp. 17–17.