Extending the WCET Problem to Optimize for Runtime-Reconfigurable Processors

MARVIN DAMSCHEN, LARS BAUER, and JÖRG HENKEL, Karlsruhe Institute of Technology (KIT), Germany

The correctness of a real-time system does not depend on the correctness of its calculations alone but also on the non-functional requirement of adhering to deadlines. Guaranteeing these deadlines by static timing analysis, however, is practically infeasible for current microarchitectures with out-of-order scheduling pipelines, several hardware threads, and multiple (shared) cache layers. Novel timing-analyzable features are required to sustain the strongly increasing demand for processing power in real-time systems.

Recent advances in timing analysis have shown that runtime-reconfigurable instruction set processors are one way to escape the scarcity of analyzable processing power while preserving the flexibility of the system. When moving calculations from software to hardware by means of reconfigurable custom instructions (CIs)—additional to a considerable speedup—the overestimation of a task's worst-case execution time (WCET) can be reduced. CIs typically implement functionality that corresponds to several hundred instructions on the central processing unit (CPU) pipeline. While analyzing instructions for worst-case latency may introduce pessimism, the latency of CIs—executed on the reconfigurable fabric—is precisely known.

In this work, we introduce the problem of selecting reconfigurable CIs to optimize the WCET of an application. We model this problem as an extension to state-of-the-art integer linear programming (ILP)-based program path analysis. This way, we enable optimization based on accurate WCET estimates with integration of information about global program flow, for example, infeasible paths. We present an optimal solution with effective techniques to prune the search space and a greedy heuristic that performs a maximum number of steps linear in the number of partitions of reconfigurable area available. Finally, we show the effectiveness of optimizing the WCET on a reconfigurable processor by evaluating a complex multimedia application with multiple reconfigurable CIs for several hardware parameters.

CCS Concepts: • **Computer systems organization** → **Real-time system specification**; **Real-time system architecture**; Embedded software; • **Hardware** → *Hardware accelerators*; *Application specific instruction set processors*

Additional Key Words and Phrases: WCET, real-time, timing analysis, predictability, custom instructions, instruction set extension, reconfigurable processors

ACM Reference Format:
Marvin Damschen, Lars Bauer, and Jörg Henkel. 2016. Extending the WCET problem to optimize for runtime-reconfigurable processors. ACM Trans. Archit. Code Optim. 13, 4, Article 45 (December 2016), 24 pages.
DOI: http://dx.doi.org/10.1145/3014059

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89).
Authors' addresses: Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT), Haid-und-Neu-Str. 7, 76131 Karlsruhe, Germany; emails: {marvin.damschen, lars.bauer, henkel}@kit.edu.

DOI: http://dx.doi.org/10.1145/3014059

1. INTRODUCTION

Real-time embedded systems are ubiquitous in our everyday life, for example, in safety-critical domains such as automotive, avionics, or robotics. The correctness of a real-time system does not only depend on the correctness of its calculations but also on the non-functional requirement of adhering to deadlines where, under circumstances safety-critical, output signals are produced. Failing to meet a deadline may lead to severe malfunctions, and therefore they need to be guaranteed in a process called *timing validation* [Wilhelm et al. 2009]. 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 [Wilhelm et al. 2009].

Determining an accurate upper WCET bound of a task is a complex problem, because performance-enhancing features of modern processors like pipelining, caches and branch prediction introduce a microarchitectural state. This microarchitectural state results in a dependency of the latency of instructions on the execution history. Assuming worst-case behavior of a microarchitectural component, for example, a cache miss, in situations where the microarchitectural state cannot be determined statically does not necessarily result in a safe WCET bound for the whole task. The state of one component may influence other microarchitectural components, for example, whether a cache access is a hit or miss can influence whether a branch condition is calculated in time or potentially mispredicted. Such an effect is called *timing anomaly* [Reineke et al. 2006], and it enforces exhaustive exploration of every possible microarchitectural state when determining the worst-case bound for executing a sequence of instructions.

Modern high-performance processors like the IBM POWER8 [Sinharoy et al. 2015] feature microarchitectures with out-of-order scheduling pipelines, several hardware threads and multiple (shared) cache layers. These average-case performance enhancing features cause an explosion of possible microarchitectural states that render timing analysis practically infeasible [Axer et al. 2014]. However, the demand for processing power in real-time systems is strongly increasing, for example, automated driving requires vast amounts of sensor data to be processed under timing constraints. Therefore, high-performance microarchitectures amenable for WCET analysis are requested [Thiele and Wilhelm 2004; Edwards and Lee 2007; Axer et al. 2014].

Recent advances in timing analysis of tasks on reconfigurable processors have proven instruction set extensions by reconfigurable *custom instructions* (CIs) to be an effective means to achieve predictable performance [Damschen et al. 2016]. CIs initiate execution of hardware accelerators configured on a reconfigurable fabric that is tightly coupled to a processor core (see Tessier et al. [2015] for an overview of reconfigurable architectures). An application binary in such an architecture provides directives to a *Reconfiguration Unit* to configure the CIs' accelerators onto the reconfigurable fabric. Reconfigurations are performed for the requirements of an upcoming *kernel* (also known as hot spot), that is, a compute-intensive part of the application, for example, a loop nest. In Damschen et al. [2016], it was shown that—additional to a considerable speedup—the overestimation of a task's WCET can be reduced by moving calculations from software code to hardware CIs. CIs typically implement functionality that corresponds to several hundred instructions when executed on the central processing unit (CPU) pipeline, possibly including conditional branches and other control flow. While analyzing instructions for worst-case latency may introduce pessimism due to, for example, pipeline hazards or instruction cache misses, the latency of the hardware accelerators—executed on the reconfigurable fabric—is precisely known.

In this work, we propose an approach of *selecting WCET-optimizing sets of CIs* for computational kernels that seamlessly integrates into state-of-the-art timing analysis.

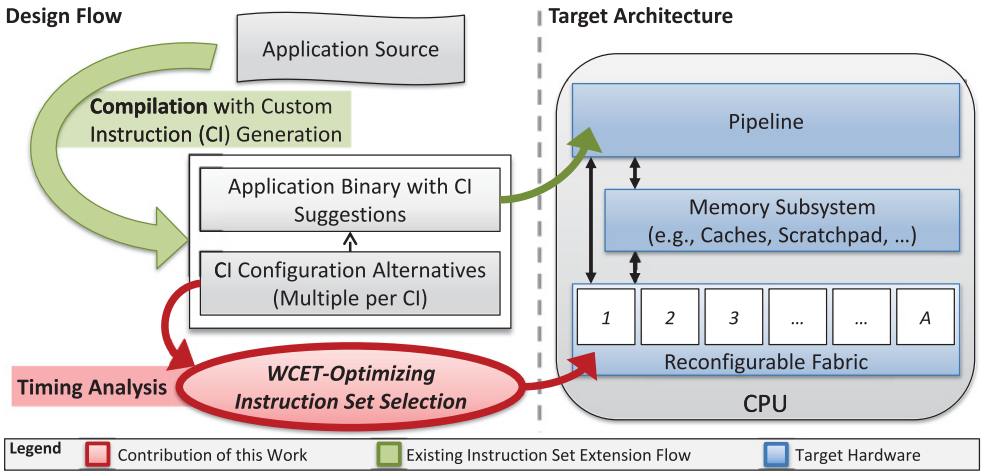


Fig. 1. Toolflow performing *WCET-Optimizing Instruction Set Selection* integrated with timing analysis. As input to our approach, we take application binary with suggestions where to place custom instructions as well as different implementation alternatives per custom instruction, differing in resource requirements and latency.

While we do not target the reduction of overestimation of a task's WCET bound or resolving the problem of timing anomalies in this work, we present an effective approach to statically select sets of reconfigurable CIs to optimize a task's WCET bound and advance research on timing-analyzable high-performance architectures. One main problem in selecting WCET-optimizing CIs is *the instability of the worst-case path*, that is, when reducing the latency of the worst-case path by inserting a CI, a whole different path can become the new worst-case path. Therefore, WCET bound estimation is an integral part of WCET-optimizing CI selection. Figure 1 shows our envisioned toolflow. CI selection, also referred to as *instruction set selection*, is the second of the two main steps in the so-called *instruction set extension problem* [Galuzzi and Bertels 2011]. The first step is the *CI generation* that is performed when compiling the application source code. In this step, kernels are identified in the application and partitioned into segments of code to execute in software and segments to execute in hardware. For the segments to execute in hardware, several alternatives that differ in resource demands as well as latencies are generated and then synthesized into configurations for the reconfigurable fabric. CIs provide an assembly-level interface to execute the hardware segments. Which CIs are implemented in hardware instead of the original software code and how much resources to allocate per CI is determined by the CI selection according to an optimization goal, for example, average-case performance. Several approaches to CI generation exist that can provide CIs and implementation alternatives as input to CI selection [Galuzzi and Bertels 2011]. Differing from existing CI selection approaches targeting average-case performance, our novel WCET-optimizing selection requires the application binary, as it is the only way to be able to obtain precise WCET bound estimates [Wilhelm et al. 2008]. To obtain a finished binary with generated CIs while keeping the flexibility to execute the original software, we introduce *CI super blocks*, which we will detail in Section 3. Essentially, we move the selection step of the instruction set extension problem from the compiler to the timing analyzer, that is, *post-compilation*, by introducing a conditional jump that either jumps to the hardware CI, if configured, or the original software code otherwise. This results in an effective technique that considerably reduces the guaranteed WCET bound compared to the original task that does not use CIs.

The novel contributions of our work are as follows:

—Modeling the WCET-optimizing instruction set selection problem with support for global program flow information and reconfiguration delay by extending state-of-the-art models used in timing analyzers like AbsInt aiT [AbsInt 2016] or OTAWA [Ballabriga et al. 2010].
—An optimal solution that effectively reduces the search space by mapping selection candidates to *weak compositions of an integer*, that is, the algorithm recursively generates all distributions of reconfigurable fabric area to CIs while adhering to area constraints. Recursion subtrees corresponding to distributions of area that cannot be utilized in CI implementations are pruned early. In our evaluation, we show that less than 1% of all possible 570,240 selections need to be evaluated when optimizing the `EncodeMacroBlock` kernel as part of the H.264 encoder with our optimal search algorithm.
—A heuristic solution that performs a maximum number of WCET estimates linear in the partitions of area available for configuring CIs on the reconfigurable fabric. It reduces the runtime of optimization down to 11.18% of the optimal search algorithm in the before-mentioned `EncodeMacroBlock` kernel, the most-complex kernel evaluated. Its results produced maximum 2.52% lower speedups on the WCET than optimal in our evaluations.

We show that previous work targeting optimization of the worst-case path, for example, instruction cache locking or scratchpad memory allocation of program code, share similarities with the WCET-optimizing instruction set selection problem but cannot be adapted to obtain optimal solutions. For introducing runtime instruction set reconfiguration as an enabling feature to provide timing-analyzable performance, novel models and solutions are required.

2. RELATED WORK AND MOTIVATION

WCET-optimizing instruction set selection bears resemblance to other static optimizations targeting the worst-case path like instruction cache locking or scratchpad memory allocation of program code. In this section, we point out the differences of these problems to WCET-optimizing instruction set selection. Additionally, we discuss state-of-the-art solutions to instruction set selection specifically and explain their shortcomings.

Caches are used to effectively reduce the average memory access latency of a CPU. It is very difficult to predict whether a memory access can be served by the cache (cache hit) or needs to be served by main memory (cache miss). WCET analysis always needs to consider a cache miss when it cannot guarantee a cache hit. This typically leads to overestimation of the WCET bound. *Cache locking* is a software-controlled mechanism to load code segments into the cache and prevent them from being evicted. Several works utilize instruction cache locking to reduce overestimation resulting from cache analysis and thus lowering the WCET bound [Falk et al. 2007; Liu et al. 2009; Plazar et al. 2012]. Similarly, the instruction cache can be replaced by allocating program code directly to predictable scratchpad memory [Falk and Kleinsorge 2009]. Even though these techniques are complementary to instruction set selection, the question arises whether the same algorithms can be applied. Similarly to instruction set selection, the instruction cache locking and program code allocation problem entail WCET estimation to determine the worst-case path and using this information to select code segments that can be most profitably sped up for lowering the WCET bound. However, both need to choose between two alternatives for a code segment only: utilizing the fast memory (i.e., locking it in the cache or allocating it in scratchpad memory) or main memory. Instruction set selection has several alternatives to choose from: the original software or different CIs implementing the same functionality with different degrees

Timing Schema Rules (excerpt):

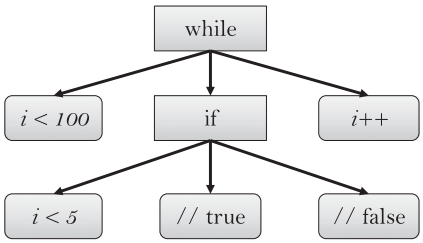
— T(`if` (Exp) T `else` F) = T(Exp) + max (T(T), T(F))

— T(`while` (Exp) Body = T(Exp) + $n \cdot$ (T(Exp) + T(Body))

Program Code:

```
int i = 0;
while (i < 100) {
  if (i < 5)
    ..; // t_T = 8
  else
    ..; // t_F = 4
  i++; }
```

Syntax Tree:



Bottom-up Calculation (with T(Exp) = 1)**:**

(1) t_{if} = T(`if` $(i < 5)$ T `else` F) = T$(i < 5)$ + max $(t_{\text{T}}, t_{\text{F}}) = 1 + \max(8, 4) = 9$

⇒ `true` case explicitly determined as worst-case path.

(2) T(`while` $(i < 100)$ Body) = T$(i < 100) + 100 \cdot$ (T$(i < 100) + t_{\text{if}} +$ T$(i++)$)

$= 1 + 100 \cdot (1 + 9 + 1) = \underline{1101}$

⇒ Decision: optimize `true` case, e.g., using cache locking or a custom instruction.

Actual WCET:

T$(i < 100) + 101 \cdot$ T$(i < 5) + 5 \cdot t_{\text{T}} + 95 \cdot t_{\text{F}} + 100 = \underline{622}$

⇒ In contrast to the result obtained by Timing Schema, the `false` case dominates the WCET. Optimizations relying on timing schema would therefore allocate resources on the wrong path.

Fig. 2. Simple example that shows how WCET optimization approaches that rely on timing schema perform suboptimal decisions.

of parallelism and therefore different delays and resource requirements. Even with extensions for evaluating multiple alternatives to choose from (e.g., the different CI implementations), existing algorithms for cache locking would remain unsuitable for our problem. Falk et al. [2007] and Liu et al. [2009] model the problem similarly using *Execution Flow Graphs* and *Execution Flow Trees*, respectively. However, the execution flow is modeled on the level of function calls. As we target kernels, we aim to model the function-internal control flow.

Plazar et al. [2012] as well as Falk and Kleinsorge [2009] model function-internal control flow similarly to the instruction set selection presented by Yu et al. [2005], which in turn is an integer linear programming (ILP) formulation of a WCET estimation technique called *timing schema* [Park and Shaw 1990]. Timing schema is a tree-based WCET estimation technique (see Ermedahl et al. [2005] for an overview of estimation techniques). In current timing analyzers, it was succeeded by the more powerful Implicit Path Enumeration Technique (IPET) [Li et al. 1995], which we will discuss in Section 3.1. Timing schema is still commonly used in state-of-the-art WCET optimization approaches however, because it is computationally cheap and it enables WCET optimization to be modeled as a single ILP (as opposed to the combinatorial problem that we present in Section 4). In timing schema, the estimation is calculated by building a representation that generally corresponds to the abstract syntax tree of the program and traversing it bottom-up by simple recursive rules. Infeasible path information cannot efficiently be applied, because the recursive rules are local to program statements [Ermedahl et al. 2005]. This can lead to imprecise WCET estimates as shown in the simple example in Figure 2: The rules are unable to capture the global information that the `true`case of the `if` statement can appear at maximum 5 times in the worst-case

path. In this example, timing schema produces an estimate based on a program path that executes the `true`case 100 times, and therefore this case seems to be the most profitable candidate to be optimized. However, this path never appears in an actual execution of the program. State-of-the-art timing analyzers can correctly determine that the `false`case dominates the WCET in the example in Figure 2 using value analysis and generating constraints for IPET. Therefore, when utilizing a computationally cheap, but imprecise, WCET estimation technique like timing schema during WCET optimization, the allocated resources may not even be utilized in the final WCET bound that is obtained using a timing analyzer. Additionally, state-of-the-art timing analyzers support powerful annotation languages to provide global path information [Kirner et al. 2011] (the impact on WCET optimization is evaluated in Section 7.3). Thus, we propose to extend state-of-the-art timing analysis using IPET to support WCET optimization, as opposed to treating WCET optimization and timing analysis as two separate processes.

Yu and Mitra [2005] perform WCET-optimizing instruction set selection for *instruction set extensible processors*. These processors contain custom functional units that can be configured to implement frequently used instruction patterns for speedups by exploiting instruction level parallelism and operator chaining [Yu and Mitra 2004]. According to their processor model, the presented heuristic assumes a uniform cost per selected pattern (i.e., occupation of one custom functional unit). The WCET-optimizing instruction set is selected per task, that is, during task execution the instruction set is fixed. Therefore, the cost of configuring a selected pattern is not taken into account in their approach. In our work, we target dynamic reconfiguration of custom instructions with varying area demands (1 up to A units of the reconfigurable fabric area). For evaluating the profit of an instruction on reducing the WCET estimate, we need to factor in its required area demands as well as its reconfiguration delay. The impact of reconfiguration delay on WCET optimization is evaluated in Section 7.2.

3. SYSTEM MODEL

Our optimization is applied to the reconstructed control-flow graph (CFG) of an application in binary form, as it is the only way to obtain obtain safe and precise WCET estimates [Wilhelm et al. 2008]. Besides the binary, we require additional compile-time information: potential CIs and their possible configurations to choose from (see Galuzzi and Bertels [2011] for an overview). The *granularity* of a CI, that is, the amount of software it replaces, depends on the specific target architecture. In our evaluation, CIs replace 12 to 123 lines of C code (see Section 7.1). For configuring the CIs in hardware, we assume reconfigurable fabric area to be allocatable in up to A discrete units. This corresponds to the common area model of dividing the fabric area into A equally sized partitions like in the 1D or 2D partitioned area models in Steiger et al. [2003]. Reconfigurations are performed before beginning execution of a kernel, for example, as shown in Figure 4(a). Let $\mathcal{CJ}$ be the set of all CIs. We assume a specific configuration j of a CI $k \in \mathcal{CJ}$ in hardware to have a constant delay $t_{k,j}$ (cycles spent in the pipeline's execution stage), to require area on the reconfigurable fabric $a_{k,j} \in [1, A]$, and to take a constant reconfiguration delay $r_{k,j}$ for configuring it on the fabric. For a constant reconfiguration delay, a constant bandwidth for transferring configuration data to the reconfigurable fabric's configuration memory needs to be guaranteed. We assume the CPU to be stalled during reconfiguration in this work, and therefore the system bus could be utilized for reconfiguration at a guaranteed bandwidth. For more details on achieving a constant reconfiguration delay, see, for example, the Reconfiguration Unit in Damschen et al. [2016].

Additional to hardware configurations, a CI can be implemented using its original software code $j = 0$. The software implementation does not have a constant delay $t_{k,0}$, because it is subject to, for example, cache and pipeline analysis in the specific

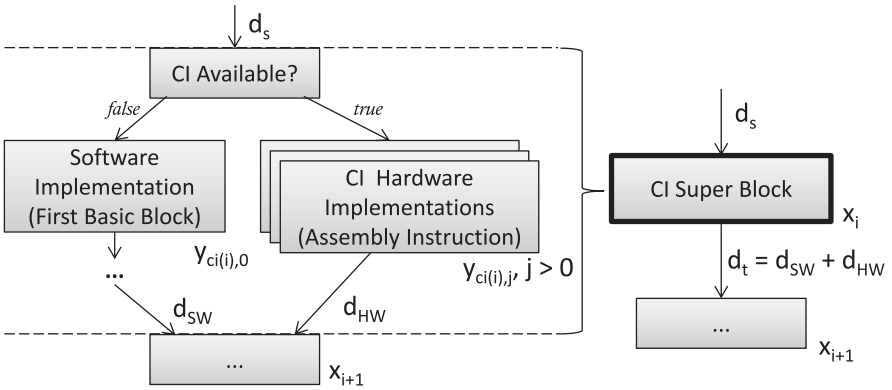


Fig. 3. CI super block as part of a CFG.

context that it is executed in. It does not require fabric area nor reconfiguration delay (i.e., $a_{k,0} = r_{k,0} = 0$). For providing the flexibility to execute the original software for generated CIs, we introduce *CI super blocks*. As shown in Figure 3, CI super blocks begin with a conditional branch before every CI (the actual instruction in the binary) which jumps to the functionally equivalent software code when the CI is not implemented in hardware. If a configuration for the CI is available on the reconfigurable fabric, then the CI is executed instead of jumping to the software. The CI super block ends by joining paths of hardware CI and software. Multiple CI super blocks in the binary can execute the same CI k. Let $\mathcal{B}$ be the set of all blocks, that is, basic blocks (not contained in super blocks) as well as super blocks. The function ci(i) determines which CI k is executed by a super block $i \in \mathcal{B}$, that is,

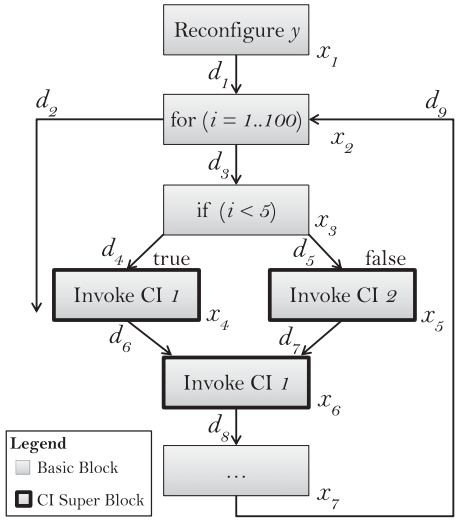
$$\text{ci}: \mathcal{B} \to \mathcal{CI} \cup \{0\}, i \mapsto k, \text{ with } \text{ci}(i) = 0 \notin \mathcal{CI} \text{ if } i \text{ is a basic block (not a super block).} \quad (1)$$

The context-dependent delay for executing implementation j of CI super block i is denoted as $e_{i,j}$ for hardware as well as software implementations. While CI execution on the reconfigurable fabric itself is context independent ($t_{\text{ci}(i),j}$ is constant, for $j > 0$), invoking the CI from the CPU pipeline can add additional cycles, for example, because of pipeline hazards or instruction fetch miss of the CI. Therefore, $e_{i,j} \geq t_{\text{ci}(i),j}$ for $j > 0$. Consider the example of Figure 4(a): It provides input to the WCET-optimizing instruction set selection. In this example, two CIs were generated, one with $m_1 = 2$ and the other with $m_2 = 3$ different hardware implementations. From *microarchitectural analysis* [Wilhelm et al. 2008], we obtain the worst-case bound per block that considers, for example, cache, pipeline, or branch prediction effects (see Figure 4(b)). This way, $e_{4,0}$ and $e_{6,0}$ can be unequal, even when they execute the same CI ($\text{ci}(\{4, 6\}) = 1$) in the same implementation ($j = 0$). $e_{i,j}$ is the main parameter that we use to calculate WCET estimates based on a specific selection of implementations in Section 4. Equation (1) is used to concisely formulate Equations (5)–(9) on which our WCET estimation is based. Effectively, we obtain a CFG that is parameterizable by a chosen selection using CI super blocks. In the following, we will introduce the WCET bound estimation technique we utilize and show how we can extend it to our problem formulation to evaluate and direct our optimization.

3.1. Calculating WCET Bounds Using IPET

We utilize IPET [Li et al. 1995] for WCET bound calculation during optimization, as it is the program path analysis technique state-of-the-art timing analyzers rely on Wilhelm et al. [2008] and Altmeyer et al. [2015]. IPET models program flow as arithmetic

(a) Input to WCET-Optimizing Timing Analysis:



Reconfigurable area:
$A = 5$
Generated CIs:
$\mathcal{CI} = \{1, 2\}, |\mathcal{CI}| = 2$
Hardware implementations per CI:
$m_1 = 2, m_2 = 3$
Area demands ($a_{k,0}$: software):
$a_1 = (0, 3, 3), a_2 = (0, 2, 4, 5)$
Reconfiguration delays:
$r_1 = (0, 10, 10), r_2 = (0, 7, 12, 16)$
CI latencies on reconf. fabric (undefined for software: $t_{k,0} = \bot$):
$t_1 = (\bot, 10, 12), t_2 = (\bot, 15, 11, 9)$

(b) Obtained from Microarchitectural Analysis:

Worst-case basic block delays:
$c_1, c_2, c_3, c_7 \in \mathbb{N}$
Invoked CIs:
$\mathbf{ci}(\{1, 2, 3\}) = 0, \mathbf{ci}(\{4, 6\}) = 1, \mathbf{ci}(\{5\}) = 2$
Worst-case CI Super Block delays (in order of invoked CI):
$e_4 = (50, 10, 12), e_6 = (48, 10, 12), e_5 = (60, 18, 14, 12)$
($e_{k,j} \geq t_{k,j}$ for $j > 0$, because execution history-dependent, see Section 3)

(c) Generated Constraints:

Program Structure (by IPET):

$$1 = x_1 = d_1 \text{ (\textit{kernel entry constraint})}$$
$$x_2 = d_1 + d_9 = d_2 + d_3$$
$$x_3 = d_3 = d_4 + d_5$$
$$x_4 = d_4 = d_6$$
$$x_5 = d_5 = d_7$$
$$x_6 = d_6 + d_7 = d_8$$
$$x_7 = d_8 = d_9$$

Global Information:

$$x_3 \leq 100 \cdot d_1 \text{ (\textit{upper loop bound})}$$
$$x_4 \leq 5 \cdot d_1 \text{ (\textit{true case max. 5 times})}$$

CIs and Reconfigurable Fabric:

$$\sum_{j=0}^{2} y_{1,j} = 1, \sum_{j=0}^{3} y_{2,j} = 1$$
(exactly one configuration per CI)
$$\sum_{k=1}^{2} \sum_{j=0}^{m_k} a_{k,j} y_{k,j} \leq 5 \text{ \textit{(area constraint)}}$$

(d) Generated Combinatorial Objective Function:

$$\min_{y \in \{0,1\}^{2 \times 4}} \left(\max_{x \in \mathbb{N}_0^6} \left(c_1 x_1 + c_2 x_2 + c_3 x_3 + c_7 x_7 \right.\right.$$
$$\left.\left. + \sum_{j=0}^{2} e_{4,j} y_{1,j} x_4 + \sum_{j=0}^{3} e_{5,j} y_{2,j} x_5 + \sum_{j=0}^{2} e_{6,j} y_{1,j} x_6 \right) + \sum_{j=0}^{2} y_{1,j} r_{1,j} + \sum_{j=0}^{3} y_{2,j} r_{2,j} \right)$$

Fig. 4. Simple example of how an instance of the problem formulated in Sections 3 and 4 is generated.

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 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 Li et al. [1995]:

$$\max_{x \in \mathbb{N}_0^N} \sum_{i=1}^{N} c_i x_i. \tag{2}$$

Constraints restrict the variables by modeling the control flow and capturing relative execution counts of basic blocks. The more infeasible paths can be excluded by constraints, the tighter the WCET bound will be.

An overview of how to generate IPET constraints is given in Li et al. [1995]. Besides the variables x_i representing the execution counts of basic blocks, variables d_i for every edge in the CFG are used. For example, consider Figure 4. IPET generates the program structure constraints in Figure 4(c) as follows. The loop header (represented by x_2) can be entered from outside using the edge represented by d_1 or from a previous iteration using d_9. The same basic block can be exited when the loop condition is false and the kernel is exited using d_2, or it can proceed to another iteration when the loop condition is true using d_3. Therefore, $x_2 = d_1 + d_9 = d_2 + d_3$. Global information like the upper bound of 100 loop iterations can be given by the constraint $x_3 \leq 100 \cdot d_1$.

IPET is a standard technique in state-of-the-art timing analyzers as it can capture global control flow information without the need to explicitly enumerate program paths. Several extensions, for example, for complex flows and hardware timing effects depending on the history of executed instructions, have been published [Ballabriga et al. 2010; Ermedahl et al. 2005; Wilhelm et al. 2008].

4. PROBLEM FORMULATION

In order to obtain precise WCET estimates that utilize global program flow information during instruction set selection, we integrate the system model from Section 3 and global bound calculation using IPET. Selecting an instruction set to optimize the WCET bound essentially means we aim to minimize the WCET over all possible selections, that is, we aim to *minimize* the *maximum* execution time. In the following, we extend the ILP formulation of IPET (see Section 3.1) for capturing the implementation alternatives of a CI $k \in \mathcal{CI}$. We introduce new variables $y_{k,j} \in \{0, 1\}$ for every implementation j with $y_{k,j} = 1$ if CI k is implemented using alternative j and $y_{k,j} = 0$ otherwise. For example, $y_{k,0} = 1$ would mean we do not implement CI_k in hardware but utilize its original software instead (see Section 3 and Figure 3). We introduce the constraint

$$\sum_{j=0}^{m_k} y_{k,j} = 1 \quad \forall k \in \mathcal{CI}. \tag{3}$$

To ensure that exactly one implementation is chosen—potentially in software ($j = 0$) or hardware ($j > 0$)—with m_k being the number of hardware configurations of CI k. To only allow solutions that do fit onto the reconfigurable fabric, we introduce the following area constraint:

$$\sum_{k=1}^{|\mathcal{CI}|} \sum_{j=0}^{m_k} a_{k,j} y_{k,j} \leq A \in \mathbb{N}_0, \tag{4}$$

that is, the sum of area on the reconfigurable fabric $a_{k,j}$ required to implement all CIs k using the selected implementation j (for which $y_{k,j} = 1$) needs to be lower than or equal

to the total fabric area A. Any $y \in \{0,1\}^{|\mathcal{CJ}|\times M}$, with $M = \max_{k\in\mathcal{CJ}} m_k + 1$, satisfying Equation (3) and Equation (4) is a feasible instruction set selection. As shown in Figure 4(c), the obtained constraints are used to extend constraints generated by IPET.

In the following, we will develop the objective function for optimizing the WCET in the presence of CI super blocks. Our system model from Section 3 enables us to capture every implementation alternative as a single super block in the CFG (see Figure 3). The total cycle contribution of CI k's super block i to the WCET bound is given as follows:

$$\sum_{j=0}^{m_k} e_{i,j} y_{k,j} x_i. \tag{5}$$

For example, when choosing the software implementation, the cycle contribution becomes $e_{i,0}x_i$, which directly resembles the contribution of a basic block in IPET's objective function (see Equation 2). The WCET for a given selection y without accounting for reconfiguration delay can be determined as follows:

$$\mathrm{WCET}'(y) := \max_{x\in\mathbb{N}_0^{|\mathcal{B}|}} \left(\sum_{\substack{i=1 \\ \mathrm{ci}(i)\notin\mathcal{CJ}}}^{|\mathcal{B}|} c_i x_i + \sum_{\substack{i=1 \\ \mathrm{ci}(i)\in\mathcal{CJ}}}^{|\mathcal{B}|} \sum_{j=0}^{m_{\mathrm{ci}(i)}} e_{i,j} y_{\mathrm{ci}(i),j} x_i \right). \tag{6}$$

Additionally, we need to account for the reconfiguration delay of our selection. Neglecting it could result in suboptimal selections in which the time spent configuring the selected CIs outweighs the time saved by performing hardware-accelerated calculations (more details in Section 7.2). Every CI super block utilized in a kernel is configured exactly once before entering the kernel (with zero reconfiguration delay for software implementation). Therefore, we obtain the WCET including reconfiguration delay as:

$$\mathrm{WCET}_r(y) := \mathrm{WCET}'(y) + \sum_{k=1}^{|\mathcal{CJ}|} \sum_{j=0}^{m_k} y_{k,j} r_{k,j}. \tag{7}$$

For every selection y, we obtain an ILP instance determining the resulting WCET for y. For example, when selecting the software implementation for every CI, we obtain the following objective function that resembles an objective function of an IPET problem instance without any CIs (see Equation (2)):

$$\max_{x\in\mathbb{N}_0^{|\mathcal{B}|}} \left(\sum_{\substack{i=1 \\ \mathrm{ci}(i)\notin\mathcal{CJ}}}^{|\mathcal{B}|} c_i x_i + \sum_{\substack{i=1 \\ \mathrm{ci}(i)\in\mathcal{CJ}}}^{|\mathcal{B}|} e_{i,0} x_i \right). \tag{8}$$

Note that for every choice of y, only $\mathrm{WCET}_r(y)$ changes while the constraints remain static once they were generated.

Putting it all together, the WCET-optimizing instruction set selection problem becomes a combinatorial problem with the following objective function:

$$\min_{y\in\{0,1\}^{|\mathcal{CJ}|\times M}} \left(\max_{x\in\mathbb{N}_0^{|\mathcal{B}|}} \left(\sum_{\substack{i=1 \\ \mathrm{ci}(i)\notin\mathcal{CJ}}}^{|\mathcal{B}|} c_i x_i + \sum_{\substack{i=1 \\ \mathrm{ci}(i)\in\mathcal{CJ}}}^{|\mathcal{B}|} \sum_{j=0}^{m_{\mathrm{ci}(i)}} e_{i,j} y_{\mathrm{ci}(i),j} x_i \right) + \sum_{k=1}^{|\mathcal{CJ}|} \sum_{j=0}^{m_k} y_{k,j} r_{k,j} \right). \tag{9}$$

The objective function for our example in Figure 4 is shown in Figure 4(d). As we have finite choices for $y \in \{0,1\}^{|\mathcal{CJ}|\times M}$ ($|\mathcal{CJ}|$ and M are finite), we could transform Equation (9)

into a single ILP by resolving the min of Equation (6) into one constraint per choice of y. However, this would result in up to $2^{|\mathcal{CJ}|\cdot M}$ constraints of high complexity, which becomes practically infeasible even for small values. Also note that we do not need to evaluate the ILPs for the IPET instance of the whole application, but only per kernel. Therefore, the ILPs are considerably less complex (fewer variables and constraints) than the ILP for determining the WCET of the whole application. In the following section, we will show how the search space can be pruned and feasible y are generated efficiently.

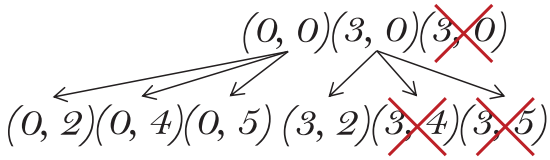
5. OPTIMAL SOLUTION

In theory, we can have up to $2^{|\mathcal{CJ}|\cdot M}$ possible selections y that we need to evaluate. In practice, however, the search space is considerably smaller for the following reasons:

—The number of possible hardware configurations m_k per CI k varies a lot, for example, in our evaluation we had a minimum of 1 to a maximum of $78 = M$ implementations for CIs (including software implementation) within one kernel (more details Section 7). From these $\sum_{k=1}^{|\mathcal{CJ}|}(m_k + 1) \leq |\mathcal{CJ}| \cdot M$ different CI implementations in total, again, in practice, only a small subset is relevant. For the CI with 78 different implementations, many implementations had different degrees of parallelism and latencies but required the same amount of area and reconfiguration delay when synthesized to the reconfigurable fabric. When considering only the minimum-latency implementation per required fabric area, our algorithm was able to prune the number of implementations to 10 relevant ones. Therefore, in practice the relevant number of implementations per CI k is much smaller than $m_k + 1$.

—Additionally, the possible selections can be pruned considerably when applying the area constraint early (see Equation (4)). Let us consider the inner sum of Equation (4); it gives us the allocation of area per CI for one selection as a tuple $a = (a_1, a_2, \ldots, a_{|\mathcal{CJ}|})$. To prune the search space, we want to find the number of unique tuples fulfilling Equation (4). Having a total area of A on the reconfigurable fabric means the number of selections utilizing the *whole* fabric is equal to the number of possibilities to distribute the area such that $\sum_{k=1}^{|\mathcal{CJ}|} a_k = A$ (allowing $a_k = 0$ for the software implementation), that is, the number of selections utilizing the whole fabric is the number of so-called *weak compositions* of the integer A into exactly $|\mathcal{CJ}|$ parts that is $\binom{A+|\mathcal{CJ}|-1}{|\mathcal{CJ}|-1}$ [Heubach and Mansour 2004]. The number of all unique tuples fulfilling Equation (4), that is, $\sum_{k=1}^{|\mathcal{CJ}|} a_k \leq A$, is exactly $\sum_{s=0}^{A}\binom{s+|\mathcal{CJ}|-1}{|\mathcal{CJ}|-1} < 2^{A+|\mathcal{CJ}|}$.

Combining both observations leads to an additional opportunity for pruning, which our optimal search algorithm shown in Algorithm 1 exploits. We recursively generate the *weak compositions* of A into exactly $|\mathcal{CJ}|$ parts as tuples $a = (a_1, a_2, \ldots, a_{|\mathcal{CJ}|})$. In the initial call optSearch(A, 1, y) the algorithm enumerates the possible values of $a_1 \in \{0, \ldots, A\}$ (Line 3). For every value of a_1, it tries to find the best implementation (minimum latency) of CI 1 requiring exactly a_1 area (Line 4). The recursive calls optSearch($A - a_1$, 2, y) take place only if an implementation for a chosen a_1 is found. Otherwise, the whole recursion subtree for the value of a_1 is pruned. Every leaf of the recursion tree ($k = |\mathcal{CJ}| + 1$) defines a unique selection y fulfilling Equation (3) as well as Equation (4) and is evaluated by solving the ILP of Equation (9) (Line 12). Figure 5 visualizes how pruning is applied and how generated tuples correspond to selection candidates for the input provided by the example in Figure 4. The effectiveness of our approach of pruning the search space by recursively generating weak compositions of A is evaluated in Section 7.4.1. While it shows effective in practice, the number of candidates to be evaluated can still grow exponentially in A and $|\mathcal{CJ}|$. Therefore, we also present a heuristic solution in the following section.



Generate possibilities for $a = (a_1, 0)$ only that correspond to implementations of CI 1.

Based on a given $(a_1, 0)$, generate possibilities for $a = (a_1, a_2)$ (implementations of CI 2).

The subtree $(3,0) = (a_{1,2}, 0)$ is pruned ($\times$), because the respective implementation $j = 2$ of CI 1 requires the same area as $j = 1$, but does not provide a latency benefit, i.e., $a_{1,1} = a_{1,2} = 3 \wedge t_{1,1} < t_{1,2} \wedge r_{1,1} \leq r_{1,2}$. $(3,4)$ and $(3,5)$ are pruned, because $a_{1,j} + a_{2,j'} > A = 5$ (area constraint).
Finally, from theoretically $|\{0,1\}^{2\times 3}| = 64$ possible inputs $y \in \{0,1\}^{2\times 3}$ to the objective function, only

$$6 = \left| \left\{ \underbrace{\begin{pmatrix} 1&0&0&0 \\ 1&0&0&0 \end{pmatrix}}_{\hat{=}a=(0,0)}, \underbrace{\begin{pmatrix} 0&1&0&0 \\ 1&0&0&0 \end{pmatrix}}_{\hat{=}a=(3,0)}, \underbrace{\begin{pmatrix} 1&0&0&0 \\ 0&1&0&0 \end{pmatrix}}_{\hat{=}a=(0,2)}, \underbrace{\begin{pmatrix} 1&0&0&0 \\ 0&0&1&0 \end{pmatrix}}_{\hat{=}a=(0,4)}, \underbrace{\begin{pmatrix} 1&0&0&0 \\ 0&0&0&1 \end{pmatrix}}_{\hat{=}a=(0,5)}, \underbrace{\begin{pmatrix} 0&1&0&0 \\ 0&1&0&0 \end{pmatrix}}_{\hat{=}a=(3,2)} \right\} \right|$$

possible selections y need to be evaluated using timing analysis to find the optimal solution.

Fig. 5. Visualization of how pruning is applied and how generated tuples correspond to selection candidates for the input provided by the example in Figure 4. For clarity, tuples that were pruned because a chosen a_k did not correspond to a possible implementation of CI k are omitted.

ALGORITHM 1: Recursive Search for Optimal Selection

```
1:  y^best ← (0, ..., 0), WCET^best ← ∞        ▷ For brevity, global variables to obtain result
2:  function optSearch(A, k, y)                 ▷ Remaining area A, CI k, (partial) selection y
3:      if k < |CI| + 1 then                    ▷ Called to select CI k
4:          for a_k ← 0, A do                   ▷ For all possible possible values of a_k
5:              y'_k ← getMinLatencyImpl(k, a_k)
6:                  ▷ Minimum latency implementation for CI k allocating exactly a_k area
7:              if y'_k ≠ 0 then                ▷ Implementation allocating exactly a_k area exists
8:                  y' ← (y_1, ..., y_{k-1}, y'_k, 0, ..., 0)^T
9:                  optSearch(A − a_k, k + 1, y')
10:             end if
11:         end for
12:     else                                    ▷ All CIs selected, evaluate
13:         if WCET_r(y) <WCET^best then        ▷ Calculate WCET bound for y, see Eq. (7)
14:             y^best ← y, WCET^best ← WCET_r(y)   ▷ Save global best evaluated so far
15:         end if
16:     end if
17: end function
```

6. HEURISTIC SOLUTION

We introduce a greedy heuristic that performs a number of WCET estimates linear in the number of partitions into which the reconfigurable fabric area was divided, that is, maximal A estimates (for $A > 0$). It is shown in Algorithm 2. The heuristic starts with implementing all CIs in software, that is, not allocating any area of the reconfigurable fabric for CIs. For every CI, it assigns a profit that calculates the WCET reduction on the *current* worst-case path when choosing an alternative implementation. Let $j(y_k)$ be

ALGORITHM 2: Greedy Heuristic for WCET-Optimizing Instruction Set Selection

```
1: repeat
2:     if ∑_{k=1}^{|𝒞𝒥|} a_{k,j(y_k)} = A then return
3:     end if
4:     x ← result x which determines WCET′(y), see Equation (6)
5:     y′ ← y
6:     y ← UPGRSELECTION(y′, x)
7: until y′ = y                                   ▷ Unable to upgrade any CI
8:
9: function UPGRSELECTION(y, x)
10:    profit^best ← 0, y^next ← y
11:    freePartitions ← A − ∑_{k=1}^{|𝒞𝒥|} a_{k,j(y_k)}
12:    for k ← 1, |𝒞𝒥| do
13:        profit ← -1, s ← 0
14:        while profit < 0 ∧ s < freePartitions do
15:            y′_k ← GETMINLATENCYIMPL(k, a_{k,j(y_k)} + s)   ▷ Try to find upgrade for CI k
16:            if y′_k ≠ 0 then                   ▷ If upgrade with exactly s additional area found
17:                y′ ← (y_1, ..., y_{k−1}, y′_k, y_{k+1}, ..., y_{m_k})
18:                profit ← profit(y′_k, y_k, x)   ▷ Defined in Eq. (10), profit > 0 ⇒ y′_k = y⁺_k
19:            end if
20:            s ← s + 1
21:        end while
22:        if profit > profit^best then
23:            y^next ← y′, profit^best ← profit          ▷ Save best y⁺_k found so far
24:        end if
25:    end for
26:    return y^next
27: end function
```

the implementation selected for CI_k in y. We define the profit of selecting y'_k over y_k for a CI k as follows:

$$\mathrm{profit}(y'_k, y_k, x) := \underbrace{\sum_{\substack{i=1 \\ \mathrm{ci}(i)=k}}^{|\mathcal{B}|} (e_{i,j(y_k)} - e_{i,j(y'_k)})x_i}_{\text{latency reduction on current worst-case path}} - \underbrace{(r_{k,j(y'_k)} - r_{k,j(y_k)})}_{\text{additional reconfiguration cost}}, \qquad (10)$$

where x is obtained by solving Equation (6) and keeping the values of the variables x_i, that is, x determines $\mathrm{WCET}'(y)$. Note that the profit can become negative if the latency reduction on the current worst-case path is smaller than the additional reconfiguration delay for the additional area. The heuristic calculates the profit for selecting the *next best implementation* y_k^+ instead of y_k for every CI k (Line 18). The *implementation* y_k^+ *for a CI* k is the implementation that can be chosen with minimum increase in the amount of area over y_k resulting in a positive profit. There might be several implementations according to this definition with the same required area. In this case, y_k^+ is the implementation with minimum latency $t_{k,j}$ (and minimum j). If no such implementation y_k^+ exists (i.e., no implementation with positive profit was found), then the CI is not considered for selecting a different implementation. Among the CIs for which y_k^+ exists, the algorithm greedily chooses the one with the maximum profit and upgrades y to select y_k^+ for the chosen CI k (Line 23). This process is repeated such that in every iteration CI k with maximum $\mathrm{profit}(y_k^+, y_k, x)$ is upgraded. The algorithm terminates when no y_k^+ for

Table I. Kernels and Custom Instructions (CI) in the H.264 Application

CI Name and Short Description	Working Set	#Confs.	Min. Part.	Max. Part.	CLoC[1]
MotionEstimation Kernel					
SATD: Sum of Abs. Transf. Differences	16×16 px	77	1	9	123
SAD: Sum of Abs. Differences	16×16 px	3	1	4	24
EncodeMacroBlock Kernel					
MC_Hz: Motion Compens. Interpol. Horiz.	4 px	29	1	6	51
IPred_HDC: Intra Prediction Horiz.	16×16 px	3	1	1	35
IPred_VDC: Intra Prediction Vert.	16×16 px	5	1	4	19
DCT: Discrete Cosine Transf.	4×4 px	21	1	5	76
HT2x2: Hadamard Transform	2×2 px	1	1	1	12
HT4x4: Hadamard Transform	4×4 px	17	1	4	111
LoopFilter Kernel					
LoopFilter: In-Loop Deblock. Filter	4 px	6	1	4	82

any k exists anymore or insufficient area is left to be allocated for selecting y_k^+ (Line 7). In every iteration, we either select a CI for increasing its allocated area by a minimum of one or the algorithm terminates. For every iteration but the last we perform one WCET estimate. Therefore, we perform a maximum of A WCET estimates.

7. EXPERIMENTAL EVALUATION

7.1. Evaluation Setup

We evaluated this work on a reconfigurable processor presented in Henkel et al. [2011, 2012] that we extended for execution with hard real-time guarantees. The implementation is based on the Gaisler LEON3 SoC, with a 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 us to stall the pipeline and pass operands to the fabric. A CI is executed by a so-called CI Execution Controller. Its protocol is similar to other multi-cycle instructions like division and directly accesses register operands or non-cacheable Scratchpad Memory (see Bauer et al. [2008] for more details). This way, in microarchitectural analysis, when determining WCETs of basic blocks, a CI is just another multi-cycle instruction that does not influence data cache analysis (if applied). The reconfigurable fabric is divided into A equally sized partitions, complying to common models of allocating reconfigurable fabric area as assumed in our system model (see Section 3). A *Reconfiguration Unit* with private memory to store configurations provides predictable reconfiguration of CIs. Initiating a specific configuration is done by a single store of the CPU using the memory-mapped interface of the Reconfiguration Unit (see Damschen et al. [2016] for more details).

Our timing analysis of tasks on reconfigurable processors has been detailed in previous works and was evaluated using the commercial timing analyzer AbsInt aiT [AbsInt 2016] (see Damschen et al. [2016]). In this work, we extend WCET analysis as an integral part of WCET optimization. Therefore, we implemented our optimal search and heuristic selection algorithms as *processors* within the open-source WCET estimation framework OTAWA [Ballabriga et al. 2010]. We extended the existing analysis support for the LEON3 CPU in OTAWA to support CI opcodes, CI super blocks with configuration-dependent latency and reconfiguration delay. We evaluated our analysis with an H.264 encoder application that uses nine CIs covering the most compute-intensive kernels shown in Table I. Multimedia applications in general are

[1]C Lines of Code that are replaced by utilizing a hardware CI (without comments or whitespace).
[2]http://www.gaisler.com/index.php/products/operating-systems.

regularly subject to hard real-time constraints in the domain of computer vision. Notable examples are advanced driver assistance systems, for example, vehicle detection and tracking [Betke et al. 2000], but also consumer electronics, for example, face recognition in digital cameras [Yang et al. 2006]. The H.264 encoder contains complex control flow with numerous decisions and nested loops. Most of the properties tested in the Mälardalen[3] or TACLeBench[4] WCET Benchmarks are covered, for example, Discrete Cosine Transform is contained in all three. The H.264 decoder—that is part of TACLeBench—performs a subset of the computations performed in the H.264 encoder that we evaluate. Especially the `EncodeMacroBlock` kernel stresses our selection heuristic (more details in Section 7.4.2), as it contains separate compute-intensive paths that share some CIs. The kernel iterates over macroblocks (MBs). Which path is executed within a kernel iteration depends on the type of MB, either *I-MB* or *P-MB*, determined by the `MotionEstimation` kernel, that is, it is input dependent. the I-MB and P-MB paths also contain separate CIs leading to *instability of the worst-case path*, that is, adding more partitions to the current worst-case path can result in the other path becoming the worst case. We compiled the application using BCC 4.4.2 (Gaisler's extended GCC 4.4.2) at O1 and performed our selection on the encoder for a frame size of 99 MBs (QCIF resolution). At higher optimization levels, GCC emitted *irreducible loops*, that is, complex loop structures that cannot be extracted as well-defined loop routines by the timing analyzer. Therefore, O1 provided the lowest WCET bound for the baseline executing all CI super blocks in software. The selection is performed offline and runs on a workstation with an AMD FX-6300 CPU and 12GB of RAM. The result is used to generate a single configuration for every kernel that includes CI super blocks. The configurations are supplied to the optimized application on the target system by loading them into the private memory of the Reconfiguration Unit before executing the application. Before entering a kernel that includes CI super blocks, its specific configuration is triggered. The pipeline stalls for the reconfiguration delay and continues with entering the kernel once reconfiguration finishes.

The parameters evaluated were different numbers of partitions A (300 slices each on a Xilinx Virtex 7), reconfiguration bandwidths as well as relations of CPU frequency and fabric frequency $f_{\text{CPU}}/f_{\text{fabric}}$. f_{fabric} stays constant at 100MHz, and we choose multiples of it for f_{CPU} that resemble realistic setups. For example, running the CPU at $f_{\text{CPU}} = 400\text{MHz}$, which the LEON3 CPU is advertised as running at as an application-specific integrated circuit (ASIC) implementation, would correspond to the parameter $f_{\text{CPU}}/f_{\text{fabric}} = 4$. The successor of the LEON3, LEON4 is advertised running at 1500MHz, corresponding to $f_{\text{CPU}}/f_{\text{fabric}} = 15$. The commercially available Xilinx Zynq-7000 SoC couples an ARM Cortex A9 at 866 MHz with a Xilinx 7-Series reconfigurable fabric, corresponding to $f_{\text{CPU}}/f_{\text{fabric}} \approx 9$. Note that while the WCET in seconds (WCET cycles/f_{CPU}) is anticipated to get lower (better) with higher f_{CPU}, the WCET cycles are increasing (at a constant f_{fabric}), because hardware CIs perform less computations on the reconfigurable fabric within one CPU cycle.

In Sections 7.2 and 7.3, we focus on the effects of considering reconfiguration delay and infeasible path information on the selection result, respectively. In Section 7.4, we analyze the effectiveness of pruning during optimal search and compare runtime as well as quality of selection results of our optimal search and heuristic algorithms. Note that all discussed results are upper bounds of the actual WCET. In general, it is not possible to obtain the actual WCET [Wilhelm et al. 2008].

[3]http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.
[4]http://www.tacle.eu/index.php/activities/taclebench.

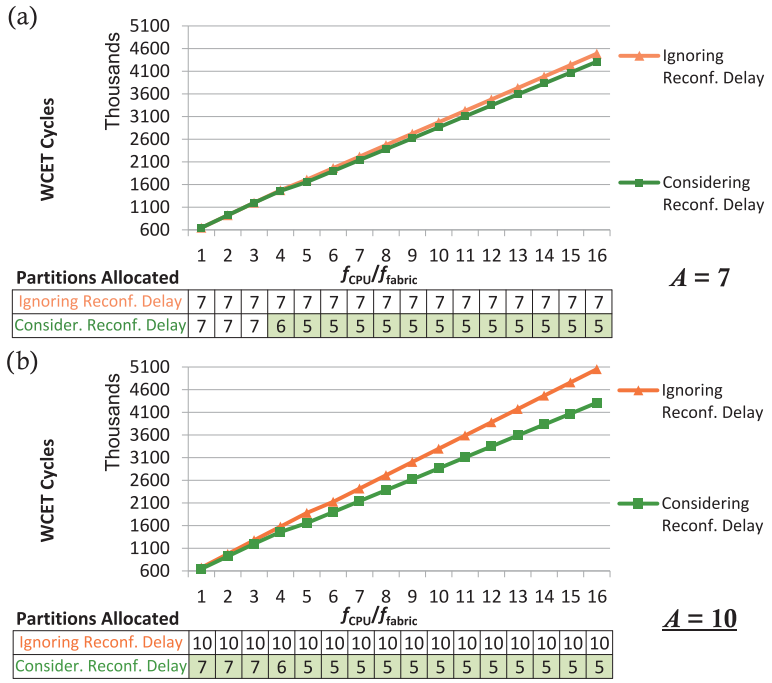


Fig. 6. Optimal results for the `EncodeMacroBlock` kernel of the H.264 encoder and different values of $f_{\text{CPU}}/f_{\text{fabric}}$, A as well as a reconfiguration bandwidth of 200MB/s. Comparing results considering reconfiguration delay during selection and results not considering it.

7.2. Impact of Reconfiguration Delay on WCET-Optimizing Selection

In this section, we evaluate the impact of reconfiguration delay on WCET-optimizing CI selection. Figure 6 shows results obtained by applying our optimal search algorithm (see Section 5) to the `EncodeMacroBlock` kernel of the H.264 encoder for $f_{\text{CPU}}/f_{\text{fabric}} \in [1:16]$ and reconfiguration bandwidth of 200MB/s (half of the theoretical maximum in current Xilinx FPGAs). We compare the results obtained by considering the reconfiguration delay during selection, as in Equation (9), with results obtained by ignoring it (i.e., $r_{k,j} = 0\ \forall k \in \mathcal{CI}, j \in m_k$). The final WCET bound always includes the reconfiguration delay required to configure the selection result.

Figure 6(a) shows the results for $A = 7$, that is, the algorithm can allocate up to 7 partitions for the selection to optimize the WCET of this kernel. For $f_{\text{CPU}}/f_{\text{fabric}} \in [1:3]$, the selections and the resulting WCET bound are equal. For higher frequencies of the CPU, the WCET bound obtained by ignoring the reconfiguration delay during selection is higher than the WCET bound obtained by considering the reconfiguration delay with a maximum of 4.08% increase at $f_{\text{CPU}}/f_{\text{fabric}} = 16$. More importantly, the lower WCET bounds are obtained with *fewer partitions*. It is not beneficial to use all 7 partitions with $f_{\text{CPU}}/f_{\text{fabric}} \in [4:16]$, because the CIs having the biggest effect on reducing the WCET bound are implemented in hardware already. Increasing the number of allocated partitions for these CIs yields diminishing returns in their latency reduction. In total, this leads to an increase of the WCET bound, because the additional reconfiguration delay outweighs the latency reduction of the WCET path. This effect becomes even more apparent, with $A = 10$ as shown in Figure 6(b), keeping all other parameters as in Figure 6(a). In this case, ignoring the reconfiguration delay already yields a higher WCET bound by 4.02% at $f_{\text{CPU}}/f_{\text{fabric}} = 1$ and up to 17.14% at $f_{\text{CPU}}/f_{\text{fabric}} = 16$ over

considering the reconfiguration delay. Furthermore, at $f_{\text{CPU}}/f_{\text{fabric}} = 16$ only half the partitions are required when considering the reconfiguration delay (5 partitions, as compared to 10 when ignoring it). The effect of obtaining lower WCET bounds with fewer partitions when considering the reconfiguration delay during selection compared to not considering it becomes more severe with higher reconfiguration delay (measured in CPU cycles) per allocated partition. The reconfiguration delay per partition increases when $f_{\text{CPU}}/f_{\text{fabric}}$ is increased (e.g., when using a higher-frequency CPU) or the reconfiguration bandwidth is lowered (e.g., when using cheaper memory). Additionally, raising A further would again lead to worse selections when not considering the reconfiguration delay, as more partitions would be allocated for only little CI latency improvement.

In sum, not considering the reconfiguration delay during WCET-optimizing CI selection can lead not only to suboptimal results but also to *higher* WCET bounds allocating *more* partitions than required in the optimal results (considering reconfiguration delay). Existing approaches for selecting CIs to optimize the WCET, target application-specific instruction set processors (ASIPs) instead of reconfigurable processors, and therefore do not consider reconfiguration delay (see Section 2). While runtime reconfiguration provides the flexibility to utilize the whole fabric area per kernel and was proven to provide substantial WCET reductions [Damschen et al. 2016], the reconfiguration delay needs to be considered during selection to avoid suboptimal results. Previous approaches targeting ASIPs can therefore not be applied to reconfigurable processors as the results obtained by our approach show.

7.3. Impact of Infeasible Path Information on WCET-Optimizing Selection

As motivated in Figure 2, previous WCET-optimizing selection and allocation approaches relying on timing schema cannot utilize information about the global program flow. Therefore, the global flow information provided by annotation languages in state-of-the-art timing analyzers (see Kirner et al. [2011] for an overview), which is crucial to precise WCET bounds, cannot be utilized during optimization, and therefore decisions are made on imprecise WCET estimates. In the evaluations of our approach, the CFG was annotated with infeasible path information using the XML-based FFX language [Bonenfant et al. 2012] supported by OTAWA. During WCET bound estimation, these annotations are translated into IPET constraints (see Section 3.1). Similarly to all other IPET constraints used in our optimization approach, the constraints need to be generated once for the whole optimization process and can be reused for all WCET bound estimations.

Figure 7 shows results with and without infeasible path information obtained by applying our optimal search algorithm (see Section 5) to the `EncodeMacroBlock` kernel of the H.264 encoder for several parameters. For evaluating the effects of infeasible path information, we annotate the path enconding I-MBs (see Section 7.1) as infeasible, which becomes the worst-case path only at some point when adding partitions to CIs (the exact point depends on $f_{\text{CPU}}/f_{\text{fabric}}$ and the reconfiguration bandwidth). For most selections and especially when not allocating any partitions (original software instead of hardware CIs only), the P-MB path is the worst-case path. Still, we can show that annotating the I-MB path as infeasible has a considerable effect on the resulting WCET bound. A reconfiguration bandwidth of 400MB/s—the theoretic maximum in current Xilinx FPGAs—is used in Figure 7(a) for allocating $A = 5$ partitions. At $f_{\text{CPU}}/f_{\text{fabric}} = 1$, the difference between optimized WCET bound with and without infeasible path information is maximal with 12.71% more WCET cycles when not utilizing the infeasible path information. The additional WCET cycles are a result of allocating partitions to CIs that lie on the path marked as infeasible. Our approach enables us to utilize this information during optimization and therefore does not allocate any partitions to the infeasible path when this information is provided. For $f_{\text{CPU}}/f_{\text{fabric}} \in [2:4]$, the difference

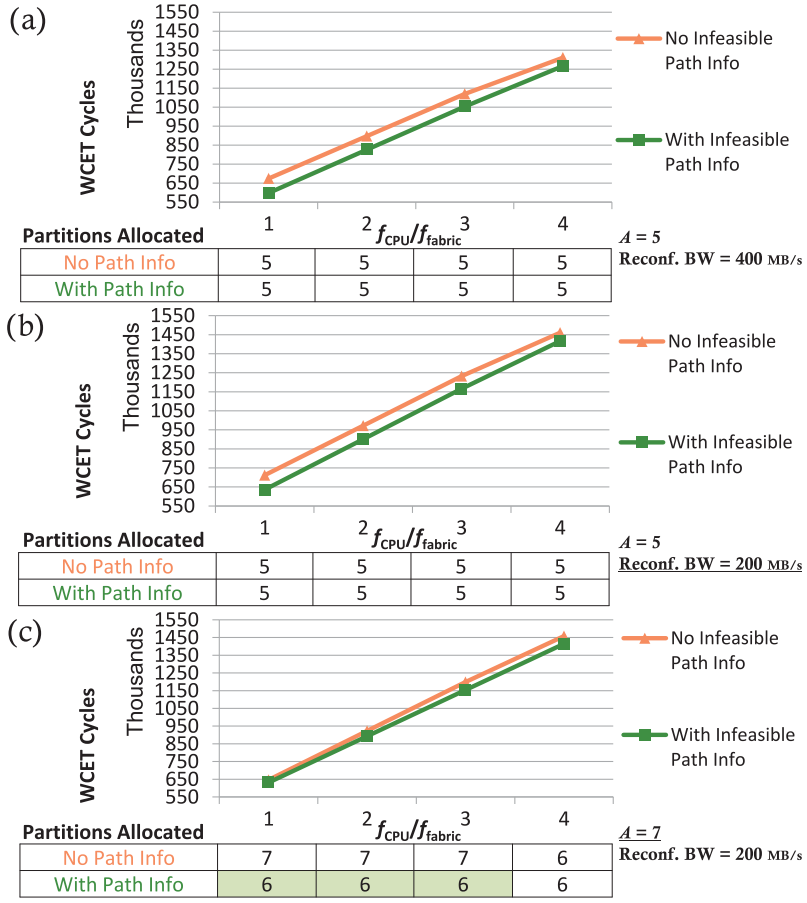


Fig. 7. Optimal results for the `EncodeMacroBlock` kernel of the H.264 encoder and different values of $f_{\text{CPU}}/f_{\text{fabric}}$, A as well as reconfiguration bandwidth. Comparing results utilizing Infeasible Path Information (I-MB Path marked infeasible) and not utilizing it.

decreases down to 3.45% at $f_{\text{CPU}}/f_{\text{fabric}} = 4$, because the increased speed of the CPU relative to the fabric compensates the partition wasted on the infeasible path when not utilizing the global flow information. One might suspect that this effect is only possible at this high reconfiguration bandwidth, because the reconfiguration delay of adding an additional partition to the worst-case path while utilizing infeasible path information might be too high to reduce the WCET bound otherwise (see Section 7.2). However, with half the reconfiguration bandwidth (200 MB/s), the results remain valid with 11.95% additional cycles at $f_{\text{CPU}}/f_{\text{fabric}} = 1$ and 3.08% at $f_{\text{CPU}}/f_{\text{fabric}} = 4$ when comparing the selection not utilizing WCET path information with the selection that does. For higher values of $f_{\text{CPU}}/f_{\text{fabric}}$ than 4, the instability of the worst-case path leads to the infeasible path not appearing for $A = 5$. Keeping the reconfiguration bandwidth at 200MB/s and increasing A leads to an additional effect shown in Figure 7(c). The difference between the WCET bounds obtained with infeasible path information and without is generally lower, with maximal 3.90% additional WCET cycles at $f_{\text{CPU}}/f_{\text{fabric}} = 3$. The reason is that with infeasible path information the optimal choice only allocates 6 of $7 = A$ available partitions, the reconfiguration delay of adding an additional partition to the worst-case path is now too high to effectively reduce the WCET bound. Adding an

Table II. Evaluation Results `LoopFilter` Kernel, $|\mathcal{CJ}| = 1$

Measures \ A	**0**	**1**	**2**	**3**	**4**
Total Possible Selections	7	7	7	7	7
Weak Comp. of A into $\lvert\mathcal{CJ}\rvert$	1	2	3	4	5
Opt. Estimates	1	2	3	4	5
Heur. Estimates	1	1	2	3	3
Opt. Runtime [ms]	122.0	123.2	119.2	120.0	124.4
Heur. Runtime [ms]	118.0	112.4	122.4	122.4	119.2
WCET unoptimized [cycles]	4,467,172				
Opt. Speedup	1	1	19.8516	19.8516	19.8516
Heur. Speedup	1	1	19.8516	19.8516	19.8516

additional partition to the infeasible path when not utilizing this information therefore adds reconfiguration delay to the WCET bound without providing actual benefit. Therefore, similarly to the impact of reconfiguration delay evaluated in Section 7.2, utilizing infeasible path information during WCET-optimizing CI selection provides better results and can even provide *better results with fewer partitions*. Previous WCET-optimizing approaches for CI selection and memory allocation relied on timing schema and therefore were unable to utilize global flow information (see Section 2). However, utilizing this information is crucial to obtain good selection results.

While our approach enables utilizing global flow information and considering the reconfiguration delay during optimization, it adds complexity by utilizing IPET over simpler techniques like timing schema to obtain WCET estimates. In the following, we will evaluate the quality of the results of our heuristic compared to optimal search as well as runtimes to demonstrate the practicality of our approach.

7.4. Runtimes, Pruning, and Quality of Heuristic Selection

Sections 7.2 and 7.3 demonstrated the importance of considering reconfiguration delay as well as global program flow information during WCET optimization. In contrast to previous approaches, our approach enables us to consider both types of information. However, this requires evaluating several IPET instances and therefore raises the question whether the runtimes of the optimization remain within acceptable bounds.

Table II to V show evaluation results for all major kernels of the H.264 encoder application. We fixed $f_{\text{CPU}}/f_{\text{fabric}}$ at 4 and a reconfiguration bandwidth of 400MB/s, as it reflects the realistic setup of running the CPU at 400MHz (which the LEON3 processor is advertised as running at when implemented as an ASIC) and the reconfigurable fabric at 100MHz, as well as running the configuration port at its maximum speed. The scalability and effectiveness of pruning during the optimal search as well as the heuristic is evaluated by running the algorithms for $A \in [0:21]$. Giving optimal search the freedom to allocate up to $A = 21$ partitions results in the maximum number of candidates for the most complex kernel `EncodeMacroBlock` (see Table V). The maximum number of candidates is reached for lower A for the `MotionEstimation` ($A = 11$) and `LoopFilter` ($A = 4$) kernels, and the additional measurements for these kernels are therefore omitted. Table II to Table V are in increasing order of kernel complexity (number of instructions and number of CI super blocks). The first line of the tables is the total number of possible selections calculated as $\prod_{k \in \mathcal{CJ}}(m_k + 1)$, that is, the number of all combinations of configurations, plus the original software implementation, per CI without any restrictions. Weak compositions of A were explained in Section 5 as a technique we apply for pruning selection candidates during optimal search. The number of weak compositions of A into exactly $|\mathcal{CJ}|$ parts are calculated as $\sum_{s=0}^{A} \binom{s+|\mathcal{CJ}|-1}{|\mathcal{CJ}|-1}$

[Heubach and Mansour 2004]. *Opt. Estimates* and *Heur. Estimates* are the number of WCET estimates calculated using Equation (7) during optimization using optimal search and the heuristic, respectively. The last lines of the tables are the runtimes of the optimizations and the speedups obtained on the WCET estimate of the kernel, comparing the selection result to the software-only implementation.

7.4.1. Effectiveness of Pruning and Scalability of Optimal and Heuristic Selection. Table II shows the results obtained for the evaluated kernel of least complexity, `LoopFilter`. The kernel includes one CI super block only, with seven implementation alternatives and therefore seven possible selections in total. For only one CI, the optimal search performs as many WCET estimate calculations as the number of weak compositions of A into exactly $|\mathcal{CI}|$ parts (denoted as *number of compositions* for the remainder of this text) until $A = 4$. For higher A, the number of estimates remains constant, as no possible implementation for the CI requiring more than four partitions exists. The heuristic never performs more than three estimates while the optimal search performs a maximum of five; however, the complexity of the kernel is too low to show any measurable runtime effect.

The kernel of next higher complexity is `MotionEstimation`, and its evaluation results are shown in Table III. It includes two CI super blocks having 4 and 78 different implementations, for a total of 312 possible selections. This is enough to demonstrate the effectiveness of pruning by finding selections that correspond to weak compositions of A (see Section 5), as even at $A = 11$ the 78 possible compositions are only a quarter of the total number of possible selections. The additional pruning of the search space in our recursive search further reduces the search space to 30 candidates, which is 9.62% of the total selection candidates and 38.46% of the number of compositions. The heuristic further reduces the number of WCET estimations performed to 11, 36.67% of the estimations performed by the optimal search, the runtime benefit is barely measurable, however.

`EncodeMacroBlock` is the most complex kernel in our evaluation. It contains six CI super blocks, resulting in a total of 570,240 possible selections. The results are shown in Table IV and Table V. Again, finding selections that correspond to weak compositions of A prunes the search space effectively, but for $|\mathcal{CI}| = 6$ the number of possible compositions of A already grows rapidly, reaching 51.91% of the total number of estimates at $A = 21$. However, the number of estimates calculated during optimal search stays much lower with a maximum of 4,200 at $A = 21$, which is 0.74% of the total number of possible selections. This is possible by pruning recursive subtrees early in our optimal search algorithm. Still, the runtime[5] of the optimal search algorithm does not scale well with increasing A. At $A = 0$ it takes 4.5s, doubling already at $A = 6$ with 9.0s, again doubling at $A = 9$ to 18.88s. For higher values of A, the runtime growth stagnates, reaching its maximum of 41.43s at $A = 21$. Especially because there may be numerous kernels within the application under optimization, these runtime values may hinder design space exploration. To reduce the optimization runtime at the potential cost of quality of the result (discussed in the following section), the heuristic can be applied. It performs a maximum of 10 estimate calculations, leading to a runtime of maximal 4.63s; 4.25s of this runtime are spent in CFG reconstruction and microarchitectural analysis, which are preparation steps to WCET bound estimation before any optimization can take place. Therefore, *solving ILPs for estimate calculations during optimization only takes a fraction of the total runtime* of the heuristic. As the dominating part of the runtime—the microarchitectural analysis—only needs to be performed once, an additional estimate required roughly under 40ms. This value is often dominated by the noise

[5]All runtime results were computed as the average of 10 median values from 12 measurements.

Table III. Evaluation Results `MotionEstimation` Kernel, $|\mathcal{CJ}| = 2$

Measures \ A	0	1	2	3	4	5	6	7	8	9	10	11
Total Possible Selections	312	312	312	312	312	312	312	312	312	312	312	312
Weak Comp. of A into $\|\mathcal{CJ}\|$	1	3	6	10	15	21	28	36	45	55	66	78
Opt. Estimates	1	2	3	4	5	6	7	8	9	10	29	30
Heur. Estimates	1	1	2	3	4	5	6	7	8	9	10	11
Opt. Runtime [ms]	4360.0	4350.8	4318.4	4329.2	4360.4	4331.2	4375.2	4391.2	4398.8	4456.8	4373.2	4317.2
Heur. Runtime [ms]	4291.6	4303.2	4332.0	4317.2	4337.6	4309.6	4342.4	4361.6	4323.6	4405.6	4344.0	4328.8
WCET unoptimized [cycles]	112,173,893											
Opt. Speedup	1	4.82	5.48	9.04	21.01	24.67	25.65	26.70	26.70	26.97	26.97	27.24
Heur. Speedup	1	4.82	5.48	9.04	21.01	24.67	25.65	26.70	26.70	26.97	26.97	27.24

Table IV. Evaluation Results `EncodeMacroBlock` Kernel, $|\mathcal{CJ}| = 6$

Measures \ A	0	1	2	3	4	5	6	7	8	9	10
Total Possible Selections	570240	570240	570240	570240	570240	570240	570240	570240	570240	570240	570240
Weak Comp. of A into $\|\mathcal{CJ}\|$	1	7	28	84	210	462	924	1716	3003	5005	8008
Opt. Estimates	1	2	3	4	5	299	517	816	1192	1629	2100
Heur. Estimates	1	1	2	3	4	5	6	7	8	9	10
Opt. Runtime [ms]	4568.4	4635.6	4757.6	5154.4	5923.2	7194.4	9568.8	12331.6	15721.2	19550.4	23131.2
Heur. Runtime [ms]	4540.8	4546.0	4556.4	4547.2	4561.6	4580.4	4605.2	4647.6	4747.2	4731.2	4784.8
WCET unoptimized [cycles]	13,348,251										
Opt. Speedup	1	2.43	3.35	4.66	9.84	10.19	10.44	10.55	10.62	10.69	10.69
Heur. Speedup	1	2.43	3.35	4.66	9.84	9.94	10.29	10.55	10.62	10.69	10.69

Table V. Evaluation Results `EncodeMacroBlock` Kernel, $|\mathcal{CJ}| = 6$

Measures \ A	11	12	13	14	15	16	17	18	19	20	21
Total Possible Selections	570240	570240	570240	570240	570240	570240	570240	570240	570240	570240	570240
Weak Comp. of A into $\|\mathcal{CJ}\|$	12376	18564	27132	38760	54264	74613	100947	134596	177100	230230	296010
Opt. Estimates	2571	3008	3384	3683	3901	4045	4130	4174	4193	4199	4200
Heur. Estimates	10	10	10	10	10	10	10	10	10	10	10
Opt. Runtime [ms]	27230.4	31072.8	34457.6	37087.2	38973.6	40324.4	41037.6	41725.2	41686.8	41751.6	41695.2
Heur. Runtime [ms]	4761.6	4740.0	4753.2	4802.0	4764.4	4792.8	4780	4592.0	4608.0	4608.4	4597.6
WCET unoptimized [cycles]	13,348,251										
Opt. Speedup	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69
Heur. Speedup	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69	10.69

between measurements. Thus, extending IPET for precise WCET estimates during WCET optimization can be suitable for design space exploration, as our results show.

In sum, the pruning techniques for the optimal search algorithm have proved to be very effective but can still lead to runtimes unsuitable for design space exploration. In these cases the heuristic can reduce the runtime down to 11.18% of the optimal search algorithm. However, the heuristic can lead to suboptimal results in certain cases, which we will evaluate in the following section.

7.4.2. Quality of Heuristic Selection. For the kernel of least complexity, `LoopFilter`, and medium complexity, `MotionEstimation`, our heuristic as well as optimal search always find the same solution for all values of A as shown in Table II and Table III, respectively. Therefore, we focus on the `EncodeMacroBlock` kernel and the evaluation results shown in Table IV, which exhibit heuristic selections that differ from optimal search (highlighted with yellow background). More specifically, the heuristic finds selections that produce 2.52% and 1.46% lower speedups at $A = 5$ and 6 than the optimal solution, respectively (while finding the optimal solution in all other cases). The reason for this is the calculation of the profit function in Equation (10) that tries to estimate the effect of a CI implementation on a previously calculated WCET bound. The problem is that the profit is calculated for the *current* worst-case path. Due to the instability of the worst-case path, adding a CI implementation y'_j that benefits the WCET bound can have a smaller effect on the total bound than on the current worst-case path. However, this is not sufficient for the heuristic to make a suboptimal choice. We additionally need a CI implementation that was assigned a lower profit than y'_j but actually has a higher effect on the total WCET bound than y'_j. This can happen when a CI configuration can appear in the current worst-case path, as well in the next longest path in the program. This is the case for the `dct4x4` CI within the `EncodeMacroBlock` kernel. For example, for $A = 5$ the heuristic chooses the suboptimal selection as follows: After allocating four partitions for the P-MB path, the I-MB path becomes the worst-case path. The heuristic calculates a profit of 257,496 cycles for implementing `ipredhdc` in hardware, allocating the last partition. However, the P-MB path is only 13,659 cycles longer at this point. Therefore, implementing `dct4x4` with a profit of 46,032 cycles in hardware and effectively reducing the WCET bound by 32,373 cycles, because it appears in the P-MB as well as the I-MB path, would have been the better choice. For $A < 5$, the I-MB path never becomes the worst-case path. For $A > 6$, the heuristic has enough partitions available to fully compensate for the suboptimal decision. Therefore, the heuristic finds the optimal results in these cases.

8. CONCLUSION

In this work, we presented how to extend state-of-the-art timing analysis using IPET to perform WCET optimization on runtime-reconfigurable processors. We formulated the WCET-optimizing instruction set selection problem, that is, selecting the WCET-optimal set of reconfigurable custom instruction implementations. Techniques for generating and pruning potential instruction set selections were discussed and realized in an optimal search algorithm. We demonstrated the effectiveness of pruning in our optimal search algorithm, which only needed to evaluate less than 1% of all possible 570,240 selections when optimizing the `EncodeMacroBlock` kernel as part of the H.264 encoder. However, as the optimal search algorithm was still not scaling well for large problem instances, we introduced a heuristic that performs maximally as many evaluations as there are partitions to allocate on the reconfigurable fabric. The heuristic was an order of magnitude faster than the optimal search for the previously mentioned kernel. Additionally, an analysis of suboptimal solutions (up to 2.52% lower speedup) obtained from the heuristic in our evaluation showed that they are a

result of competing worst-case paths during optimization that share CIs. Our problem formulation and algorithms were implemented based on the timing analyzer OTAWA, showing the seamless integration into a state-of-the-art timing analysis tool.

We showed the consequences of utilizing timing schema in WCET optimization, a WCET estimation technique that does not support global program flow information but is still commonly used in state-of-the-art WCET optimization approaches. Our novel problem formulation of WCET optimization enables considering global program flow information such as reconfiguration delay during optimization and the importance of considering these information was demonstrated. Not considering global information can lead to higher WCET bounds that require more resources than the optimal solution.

In sum, we provide novel WCET optimization approaches and introduce runtime instruction set reconfiguration as an enabling feature for timing-predictable performance. To the best of our knowledge, our model is the first formulation of a WCET optimization problem with support for global program flow information and we can envision applications to problems other than instruction set extension.

REFERENCES

AbsInt. 2016. aiT Worst-Case Execution Time Analyzers. Retrieved May 16, 2016 from http://www.absint.com/ait/.

Sebastian Altmeyer, Björn Lisper, Claire Maiza, Jan Reineke, and Christine Rochange. 2015. 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.

Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. 2014. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.* 13, 4 (2014), 82.

Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. 2010. OTAWA: An open toolbox for adaptive WCET analysis. In *SEUS*. Springer, 35–46.

Lars Bauer, Muhammad Shafique, and Jörg Henkel. 2008. A computation-and communication-infrastructure for modular special instructions in a dynamically reconfigurable processor. In *Proc. Int. Conf. on Field Programmable Logic and Applications*. IEEE, 203–208.

Margrit Betke, Esin Haritaoglu, and Larry S. Davis. 2000. Real-time multiple vehicle detection and tracking from a moving vehicle. *Mach. Vis. Appl.* 12, 2 (2000), 69–83.

Armelle Bonenfant, Hugues Cassé, Marianne De Michiel, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. 2012. FFX: A portable WCET annotation language. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*. ACM, 91–100.

Marvin Damschen, Lars Bauer, and Jörg Henkel. 2016. Timing analysis of tasks on runtime reconfigurable processors. *IEEE Trans. VLSI Syst.* (2016). DOI:http://dx.doi.org/10.1109/TVLSI.2016.2572304 online first.

Stephen A. Edwards and Edward A. Lee. 2007. The case for the precision timed (PRET) machine. In *Proc. of Design Automat. Conf.* ACM, 264–265.

Andreas Ermedahl, Friedhelm Stappert, and Jakob Engblom. 2005. Clustered worst-case execution-time calculation. *IEEE Trans. Comput.* 54, 9 (2005), 1104–1122.

Heiko Falk and Jan C. Kleinsorge. 2009. Optimal static WCET-aware scratchpad allocation of program code. In *Proc. of Design Automat. Conf.* ACM, 732–737.

Heiko Falk, Sascha Plazar, and Henrik Theiling. 2007. Compile-time decided instruction cache locking using worst-case execution paths. In *Proc. of Int. Conf. on Hardware/Software Codesign and Syst. Synthesis*. ACM, 143–148.

Carlo Galuzzi and Koen Bertels. 2011. The instruction-set extension problem: A survey. *ACM Trans. Reconfig. Technol. Syst.* 4, 2 (2011), 18.

Jörg Henkel, Lars Bauer, Michael Hübner, and Artjom Grudnitsky. 2011. i-Core: A run-time adaptive processor for embedded multi-core systems. In *Proc. Int. Conf. on Engineering of Reconfig. Syst. and Algorithms*.

Jörg Henkel, Andreas Herkersdorf, Lars Bauer, Thomas Wild, Michael Hübner, Ravi Kumar Pujari, Artjom Grudnitsky, Jan Heisswolf, Aurang Zaib, Benjamin Vogel, and others. 2012. Invasive manycore architectures. In *Proc. 17th Asia and South Pacific Design Automation Conf. (ASP-DAC)*. 193–200.

Silvia Heubach and Toufik Mansour. 2004. Compositions of n with parts in a set. *Congress Numer.* 168 (2004), 127.

Raimund Kirner, Jens Knoop, Adrian Prantl, Markus Schordan, and Albrecht Kadlec. 2011. Beyond loop bounds: Comparing annotation languages for worst-case execution time analysis. *Softw. Syst. Model.* 10, 3 (2011), 411–437.

Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. 1995. Efficient microarchitecture modeling and path analysis for real-time software. In *Proc. of Real-Time Syst. Symp.* IEEE, 298–307.

Tiantian Liu, Minming Li, and Chun Jason Xue. 2009. Minimizing WCET for real-time embedded systems via static instruction cache locking. In *Real-Time and Embed. Technol. and Applications Symp.* IEEE, 35–44.

Chang Yun Park and Alan C. Shaw. 1990. Experiments with a program timing tool based on source-level timing schema. In *Proc. of Real-time Syst. Symp.* IEEE, 72–81.

Sascha Plazar, Jan C. Kleinsorge, Peter Marwedel, and Heiko Falk. 2012. WCET-aware static locking of instruction caches. In *Proc. 10th International Symposium on Code Generation and Optimization*. ACM, 44–52.

Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. 2006. A definition and classification of timing anomalies. *WCET* 4 (2006).

Balaram Sinharoy, J. A. Van Norstrand, Richard J. Eickemeyer, Hung Q. Le, Jens Leenstra, Dung Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, José E. Moreira, and others. 2015. IBM POWER8 processor core microarchitecture. *IBM J. Res. Dev.* 59, 1 (2015), 2–1.

Christoph Steiger, Herbert Walder, Marco Platzner, and Lothar Thiele. 2003. Online scheduling and placement of real-time tasks to partially reconfigurable devices. In *Proc. of Real-Time Syst. Symp.* IEEE, 224–225.

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

Lothar Thiele and Reinhard Wilhelm. 2004. Design for timing predictability. *Real-Time Syst.* 28, 2–3 (2004), 157–177.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.* 7, 3, Article 36 (May 2008), 53 pages. DOI:http://dx.doi.org/10.1145/1347375.1347389

Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. 2009. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 28, 7 (2009), 966–978.

Ming Yang, Ying Wu, James Crenshaw, Bruce Augustine, and Russell Mareachen. 2006. Face detection for automatic exposure control in handheld camera. In *Proc. 4th IEEE Int. Conf. on Computer Vision Systems (ICVS'06)*. IEEE, 17–17.

Pan Yu and Tulika Mitra. 2004. Scalable custom instructions identification for instruction-set extensible processors. In *Proc. of Int. Conf. on Compilers, Architecture and Synthesis for Embed. Syst.* ACM, 69–78.

Pan Yu and Tulika Mitra. 2005. Satisfying real-time constraints with custom instructions. In *Proc. Int. Conf. on Hardware/Software Codesign and System Synthesis*. IEEE, 166–171.

Received May 2016; revised September 2016; accepted October 2016