Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2012, Article ID 418315, 21 pages
doi:10.1155/2012/418315

Research Article

On the Feasibility and Limitations of Just-in-Time Instruction Set Extension for FPGA-Based Reconfigurable Processors

Mariusz Grad and Christian Plessl

Paderborn Center for Parallel Computing, University of Paderborn, 33098 Paderborn, Germany

Correspondence should be addressed to Mariusz Grad, mariusz.grad@uni-paderborn.de

Received 13 May 2011; Revised 19 August 2011; Accepted 16 September 2011

Academic Editor: Viktor K. Prasanna

Reconfigurable instruction set processors provide the possibility of tailor the instruction set of a CPU to a particular application. While this customization process could be performed during runtime in order to adapt the CPU to the currently executed workload, this use case has been hardly investigated. In this paper, we study the feasibility of moving the customization process to runtime and evaluate the relation of the expected speedups and the associated overheads. To this end, we present a tool flow that is tailored to the requirements of this just-in-time ASIP specialization scenario. We evaluate our methods by targeting our previously introduced Woolcano reconfigurable ASIP architecture for a set of applications from the SPEC2006, SPEC2000, MiBench, and SciMark2 benchmark suites. Our results show that just-in-time ASIP specialization is promising for embedded computing applications, where average speedups of 5x can be achieved by spending 50 minutes for custom instruction identification and hardware generation. These overheads will be compensated if the applications execute for more than 2 hours. For the scientific computing benchmarks, the achievable speedup is only 1.2x, which requires significant execution times in the order of days to amortize the overheads.

1. Introduction

Instruction set extension (ISE) is a frequently used approach for tailoring a CPU architecture to a particular application or domain [1]. The result of this customization process is an application-specific instruction set processor (ASIP) that augments a base CPU with custom instructions to increase the performance and energy efficiency.

Once designed, the ASIP's instruction set is typically fixed and turned into a hardwired silicon implementation. Alternatively, a reconfigurable ASIP architecture can implement the custom instructions in reconfigurable logic. Such reconfigurable ASIPs have been proposed in academic research [2–6], and there exist a few commercially available CPU architectures that allow for customizing the instruction set, for example, the Xilinx Virtex 4/5FX FPGAs or the Stretch S5 processor [7]. But although the adaptation of the instruction set during runtime is technically feasible and provides a promising technology to build adaptive computer systems which optimize themselves according to the needs of the actually executed workload [8], the idea of adapting the instruction set during runtime has been hardly explored.

A number of obstacles make the exploitation of just-in-time (JIT) ISE challenging: (1) there are only very few commercially available silicon implementations of reconfigurable ASIP architectures, (2) methods for automatically identifying custom instructions are algorithmically expensive and require profiling data that may not be available until runtime, and (3) synthesis and place-and-route tool flows for reconfigurable logic are known to be notoriously slow. While it is evident that even long runtimes of design tools will be amortized over time provided that an application-level speedup is achieved, it is so far an open question whether the total required execution time until a net speedup is achieved stays within practical bounds. The goal of this work is to gain insights into the question whether just-in-time processor customization is feasible and worthwhile under the assumption that we rely on commercially available FPGA devices and tools.

In our previous work we have presented initial results in each of these three areas. We have introduced the Woolcano reconfigurable instruction set architecture in Grad and Plessl [6] (obstacle 1). Woolcano is based on a Xilinx Virtex 4FX FPGA and augments the PowerPC core in the device with user-defined custom instructions (UDCI) that can be changed at runtime using partial reconfiguration. In Grad and Plessl [9] we presented a circuit library and data path generator that can generate custom instructions for this architecture. In recent work [10] we have presented new heuristics for reducing the runtime of methods for identifying and selecting custom instructions for JIT ISE (obstacle 2). Further, we have presented a first evaluation [11] of how the long runtimes of FPGA implementation tools, mentioned as (obstacle 3) above, limit the applicability of the approach.

This paper makes the following specific contributions over our previous work.

(i) In contrast to our previous work which treated the individual subproblems in JIT ISE in isolation, this paper presents them in a comprehensive way and covers the architecture, the design methods, and the corresponding tool flow along with a more detailed evaluation.

(ii) We provide an extended discussion and formal description of our candidate identification, estimation, selection, and pruning methods for a just-in-time context. Further we describe the algorithms which were developed for candidate estimation and selection in detail.

(iii) Finally, we present an extended experimental evaluation of the candidate identification and estimation methods. In particular, we focus on the candidate identification process and evaluate the suitability of three state-of-the-art ISE algorithms for our purposes by comparing their runtime, number of identified instruction candidates, and the impact of constraining the search space.

2. Related Work

This work is built on research in three areas: reconfigurable ASIP architectures, ISE algorithms, and just-in-time compilation, which have mostly been studied in separation in related works. Just-in-time ISE inherently needs a close integration of these topics; hence a main contribution of this work is the integration of these approaches into a consistent methodology and tool flow.

From the hardware perspective, this work does not target the static but reconfigurable ASIP architectures such as our Woolcano architecture [6] or comparable architectures like CHIMAERA [4], PRISC [3], or PRISM [12]. These architectures provide programmable functional units that can be dynamically reconfigured during the runtime in order to implement arbitrary custom instructions.

Research in the areas of ISE algorithms for ASIP architectures is extensive; a recent survey can be found in Galuzzi and Bertels [13]. However, the leading state-of-the-art algorithms for this purpose have an exponential algorithmic complexity which is prohibitive when targeting large applications and when the runtime of the customization process is a concern as it is in the case for JIT ISE. This work leverages our preliminary work [10] in which new heuristics were studied for effective ISEs search space pruning. It was shown that these methods can reduce the runtime of ISE algorithms by two orders of magnitude.

The goal of this work is to translate software binaries on the fly into optimized binaries that use application-specific custom instructions. Binary translation is used, for example, to translate between different instruction sets in an efficient way and has been used, for example, in Digital's FX!32 product for translating X86 code to the Alpha ISA [14]. Binary translation has also been used for cases where the source and target ISAs are identical with the objective to create a binary with a higher degree of optimization [15, 16].

This work is conceptually similar to these approaches as it also does not translate between different instruction sets, but optimizes binaries to use specific user-defined instructions in a reconfigurable ASIP. This kind of binary translation has hardly been studied so far. One comparable research effort is the WARP project [17]. The WARP processor is a custom system on chip comprising a simple reconfigurable array, an ARM7 processor core, and additional cores for application profiling and place and route. This work differs from WARP in several ways. The main difference is that we target a reconfigurable ASIP with programmable processing units in the CPU's datapath, while WARP uses a bus-attached FPGA coprocessor that is more loosely coupled with the CPU. Hence, this work allows to offload operations at the instruction level where WARP needs to offload whole loops to the accelerators in order to cope with longer communication delays. Further, WARP operates at the machine-code level and reconstructs the program's higher-level structure with decompilation, while this work relies on higher-level information that is present in the virtual machine. Finally, WARP assumes a custom system on chip, while this work targets commercially available standard FPGAs.

Beck and Carro [18] present work on binary translation of Java programs for a custom reconfigurable ASIP architecture with coarse-grained reconfigurable datapath units. They show that for a set of small benchmarks an average speedup of 4.6x and power reduction of 10.9x can be achieved. The identification and synthesis of new instructions occur at runtime; however, the paper does not specify what methods are used for instruction identification and what overheads arise from instruction synthesis.

3. General ASIP Specialization Tool Flow and Just-in-Time Runtime System

Figure 1 illustrates the difference between a conventional static ASIP specialization process (ASIP-SP) and a runtime system with a just-in-time ASIP-SP support. The ASIP-SP is responsible for (a) generating *bistreams* for configuring the underlying reconfigurable ASIP hardware architecture with instruction extensions and (b) for modifying the source code

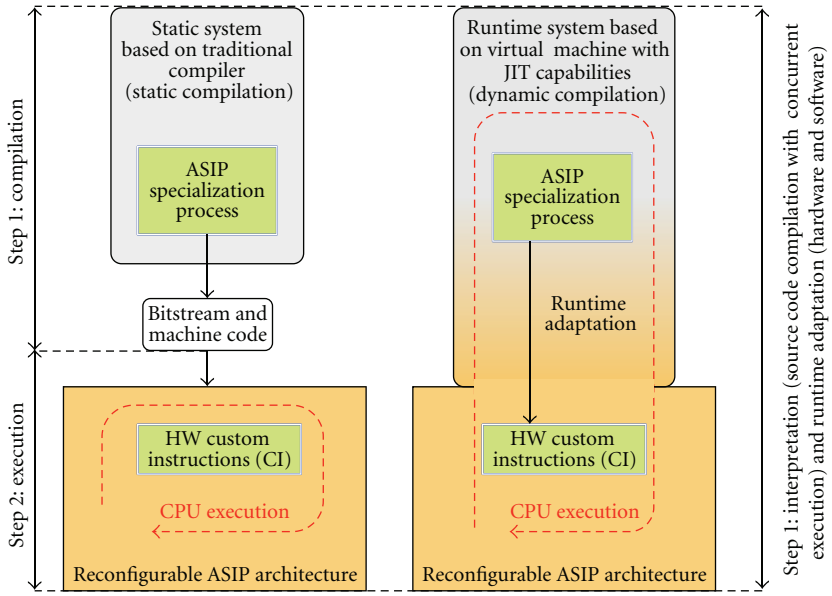

Figure 1: Overview of ASIP specialization process for conventional static and runtime systems.

to actually utilize the newly created instructions. So far, ASIP specialization has been applied almost exclusively in static systems, where steps (a) and (b) occur off line before the application is executed.

This work studies the feasibility of moving the ASIP specialization to runtime by integrating it into a virtual machine with just-in-time compilation capabilities. In such a system the ASIP specialization is performed concurrently with the execution of the application. As soon as (a) and (b) are available, the runtime adaptation phase occurs where ASIP architecture is reconfigured and the application binary is modified such that the newly available custom instructions are utilized. The main advantages of executing ASIP specialization as part of the runtime system are the following.

(i) The system can optimize its operation by reconfiguring the instruction set and by changing the code at runtime, which is fundamentally more powerful than static ASIP specialization.

(ii) The system can collect execution time, profiling, and machine level information in order to identify the code sections that are actually performance limiting at runtime; these sections are ideal candidates to be accelerated with custom instructions.

(iii) The virtual machine has the capability to execute various dynamic optimizations like hotspot detection, alias analysis, or branch prediction to further optimize the performance.

4. Our Tool Flow Implementation

For the purpose of evaluating the potential of just-in-time ASIP specialization, we have developed a prototypical tool flow that is presented in Figure 2. Our tool flow executes ASIP specialization as part of a runtime system as introduced in the previous section. However, since Xilinx's proprietary FPGA design tools can be executed only on X86 CPUs, our current version of the tool flow runs the ASIP-SP on a host computer and not on the Woolcano ASIP architecture itself; see Section 10 for details considering the experimental setup.

The details of our implemented tool flow and the Woolcano hardware architecture are presented in Figure 2. The process comprises three main phases: *Candidate Search*, *Netlist Generation*, and *Instruction Implementation*.

During the first phase, Candidate Search, suitable candidates for custom instructions are identified in the application's bitcode with the help of ISE algorithms which search the data flow graphs for suitable instruction patterns.

The ISE algorithms are computationally intensive with runtimes ranging from seconds to days, which is a major concern for the just-in-time ASIP specialization. To avoid such scenarios, the candidate identification process is preceded by *basic block pruning* heuristics which prune the search space for candidate identification algorithms to the basic blocks from which the best performance improvements can be expected. It was shown by Grad and Plessl [10] that the runtime of the ISE algorithms can be reduced by *two orders of magnitude* by sacrificing 1/4 of the speedup. The ISE algorithm identifies a set of custom instruction candidates. Afterwards the *selection* process using the performance estimation data singles out only the best one.

The *estimation* data are computed by the PivPav tool [9], and they represent the performance difference for every candidate when executed either in software or in hardware. This is possible since PivPav has a database with a wide collection of the presynthesized hardware IP cores together with more than 90 different metrics; see Grad and Plessl [9] for details. The next two phases in the process cover the generation of hardware from a software candidate and are also implemented with the help of the PivPav tool.

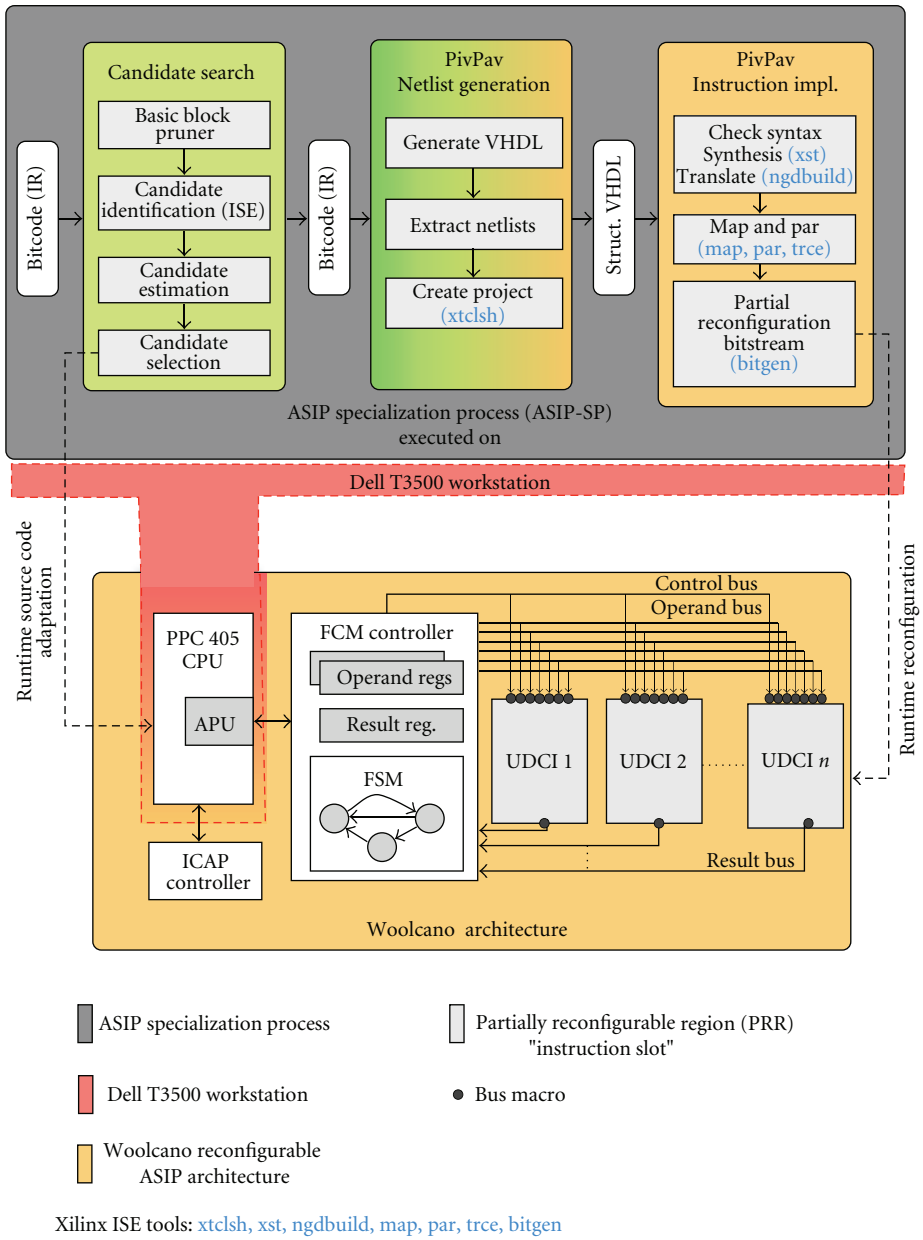

Figure 2: Overview of the developed tool flow and the targeted Woolcano hardware architecture. During experimental evaluation, instead of a PPC405 CPU, the ASIP specialization process was executed on a Dell T3500 workstation; see Section 10 for details.

The second phase, *Netlist Generation*, generates a VHDL code from the candidate's bitcode and prepares an FPGA CAD project for synthesizing the candidate. The *Generate VHDL* task is performed with PivPav's datapath generator. This generator iterates over the candidate's datapath and translates every instruction into a matching hardware IP core, wires these cores, and generates structural VHDL code for the custom instruction. Next, PivPav *extracts the netlist* for the IP cores from its circuit database. This is performed for every IP core instantiated during the VHDL generation and is used to speed up the *synthesis* and the *translation* processes during the FPGA CAD tool flow; that is, PivPav is used as a netlist cache. Finally, an FPGA CAD project for Xilinx ISE is created, the parameters of the FPGA are set up, and the VHDL and the netlist files are added.

In the third phase, *Instruction Implementation*, the previously prepared project is processed with the Xilinx FPGA CAD tool flow. This results in an FPGA configuration bitstream for the given custom instruction candidate. This bitstream can be loaded to the Woolcano architecture using partial reconfiguration. These steps are also handled by PivPav.

5. Woolcano Hardware Architecture

The bottom part of Figure 2 shows the Woolcano dynamically reconfigurable ASIP architecture. The main components

of the architecture are the PowerPC core, the internal configuration access port (ICAP) controller, the fabric co-processor module (FCM) controller, and the partial reconfiguration regions for implementing UDCI which we denote also as *instruction slots*. The FCM controller implements the interface between the CPU core and the UDCI. It forwards the inputs to the instruction slots via the operand bus and, after the custom instruction has finished computing, it transfers the output back to the CPU via the result bus. The control bus is used for sending control information, for example, activation or abort signals, to the UDCI.

Bus macros are placed at the interface between the instruction slot and the (a) operand, (b) control, and (c) result busses for enabling *dynamic partial reconfiguration* of the instruction slots. The instruction slots can be reconfigured via ICAP or the external configuration port of the FPGA.

The FCM controller was implemented as a *finite-state machine* (FSM) and is responsible for connection to the *auxiliary processor unit* (APU) interface of the *PowerPC 405* core. Its main function is to implement the APU protocol for transferring data and for dispatching instructions. During the UDCI execution the CPU is blocked and it is waiting for the UDCI results. The architectural constraints of the APU allow only for two input and one output operands to the UDCI. This restriction limits the amount of data a UDCI instruction can operate on, which in turn limits the achievable speedup. To circumvent this limitation, the FCM core implements internal operand registers for supplying the UDCI with additional operands.

The number of instruction slots as well as their input and output operands are compile-time configurable architecture parameters denoted as $C_{\max}$, $\text{in}_{\max}$, $\text{out}_{\max}$, respectively. Since all inputs and outputs to the instruction slots must be fed through Xilinx bus macros, the size and geometric placement options of the bus macros limit the number of input operands and results.

6. Candidate Identification

The candidate identification process identifies subgraphs in the *intermediate representation* (IR) code, which are suitable for fusing into a new UDCI which can be implemented for the Woolcano architecture. Suitable candidates are rich in *instruction-level parallelism* (ILP) while satisfying the architectural constraints of the target architecture.

6.1. Formal Problem Definition. Formally, we can define the candidate identification process as follows. Given a data flow graph (DFG) $G = (V, E)$, the architectural constraints $\text{in}_{\max}$, $\text{out}_{\max}$ and a set of infeasible instructions F, find all candidates (subgraphs) $C = (V', E') \subseteq G$ which satisfy the following conditions:

$$C_{\text{in}} \leq \text{in}_{\max}, \tag{1}$$

$$C_{\text{out}} \leq \text{out}_{\max}, \tag{2}$$

$$V' \cap F = \emptyset, \tag{3}$$

$$\forall t \in C : \text{convex}(t). \tag{4}$$

Here,

(i) the DFG is a *direct acyclic graph* (DAG) $G(V, E)$, with a set of *nodes* or *vertices* V that represent *IR operations* (instructions), *constants* and *values*, and an edge set E represented as binary relation on V which represents the data dependencies. The edge set E consists of ordered pairs of vertices where an edge $e_{ij} = (v_i, v_j)$ exists in E if the result of operation or the value defined by v_i is read by v_j.

(ii) $C = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

(iii) C_{in} is the set of *input nodes* of C, where a node $v_i \notin C$ with $(v_i, v_j) \in E$ for some node $v_j \in C$ is called an input node.

(iv) C_{out} is the set of *output nodes* of C, where a node $v_0 \in C$ with $(v_0, v_k) \in E$ for some node $v_k \notin C$ is called an output node.

(v) $\text{in}_{\max}$, $\text{out}_{\max}$ are constants that specify the input/output operand constraints for UDCIs which apply to the instruction slot implementation of the Woolcano architecture.

(vi) $F \subseteq V$ is a subset of illegal graph nodes (IR instructions) which are not allowed to be included in an UDCI. This set includes for instance all memory access instructions like *loads* and *stores* (since a UDCI cannot access memory) and other instructions which are not suitable for a hardware implementation, for example, for performance reasons.

(vii) Convex means that there does not exist a path between two vertices in C which contains a node that does not belong to C. In other words candidate C is convex if $v_i \in C, i = 0, \ldots, k$ for all paths $\langle v_0, \ldots, v_k \rangle$ with $v_0 \in C$ and $v_k \in C$ where path is defined as follows. A *path* from a vertex u to a vertex v in a graph C is a sequence $\langle v_0, v_1, \ldots, v_k \rangle$ of vertices such that $v_0 = u$, $v_k = v$ and $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots, k$. Convexity ensures that C can be executed atomically in the hardware. This property is required to make sure that an instruction can be executed as an atomic operation without exchanging intermediate results with the CPU.

Translating these formal definitions to practice means that the identification of UDCI candidates occurs at the level of *basic blocks* (BBs). BBs are suitable units for this purpose since they have a DAG structure as required by the ISE algorithms. In addition, it is feasible to enforce the convexity condition (cf. condition (4)) on them. When selecting IR code for a UDCI implementation, illegal instructions, such as control flow or memory operations, must be excluded (cf. condition (3)). Finally, the number of inputs and outputs of the candidate has to respect the architectural constraints (cf. conditions (1) and (2)). These architectural constraints are variable and are defined by the FCM controller and by the interface to the partially reconfigurable slots into which the UDCIs are loaded.

Table 1: Candidate identification ISE algorithms comparison.

Algorithm	# of inputs	# of outputs	Worst-case complexity	Overlapping candidates
MaxMiso	Invariant (∞)	Invariant (1)	$O(n)$	No
SingleCut	Variant	Variant	$O(\text{exp})$	Yes
Union	Variant	Variant	$O(\text{exp})$	No

6.2. Supported Instruction Set Identification Algorithms. For the identification process, we implemented three state-of-the-art ISE algorithms that are considered as the most suitable for runtime processor specialization: the *SingleCut* (SC) [19], the *Union* (UN) [20], and the *MaxMiso* (MM) [21] algorithm. The most relevant properties of these algorithms are presented in Table 1.

All identification algorithms do not try to find candidates in the entire application at once. Instead, they focus on the individual basic blocks of the application. Hence, in order to prune the search space, the ISE algorithm is executed selectively only for the most promising BBs (cf. Section 9). Each ISE algorithm analyzes the basic block to identify all feasible candidates. Further, some algorithms allow to constrain the number of inputs and outputs of candidates to match the architectural constraints of the targeted ASIP architecture. Finally, all algorithms fulfill the condition (3).

The advantage of the MM algorithm is its linear complexity $O(n)$. However, it finds only candidates which have a single output and does not allow to constrain number of inputs. Therefore, some of the generated candidates need to be discarded at additional costs later in the selection phase because they validate conditions and thus they cannot be implemented. In contrast, the SC and UN algorithms already allow for restricting the desired number of inputs and outputs for candidates during the identification phase. Finally, the SC algorithm may produce overlapping candidates. Hence, when using this algorithm an additional phase is needed in the selection process to eliminate the overlapping candidates.

7. Candidate Estimation

Since the number of UDCIs that can be implemented concurrently in the Woolcano architecture is limited, only the best candidates are selected for an implementation. The corresponding selection process which is described in the following section is based on the estimated performance of every candidate when executing in software on the CPU or in hardware as a UDCI. Based on these performance estimates the subsequent selection process can decide whether it is affordable and beneficial to implement a candidate as a hardware UDCI instruction.

7.1. Software Estimation. The Woolcano architecture consists of a PowerPC 405 CPU hard core which is used for software execution. In contrast to modern general-purpose CPUs, the PowerPC CPU has a relatively simple design. It has a scalar, in-order, five-stage pipeline, and most instructions have a latency of a single cycle. This simple design makes the task of software estimation relatively easy since the execution of the instructions in the candidate is sequential. Hence, the presented bellow estimation method is not a novel idea, and it is based on research published in Gong et al. [22].

We estimate the performance of the execution with the expression shown in (5) which corresponds to the sum of latencies of all instructions found in the candidate multiplied by the CPU clock period. This estimation technique has an algorithmic complexity of $O(n)$ where n is the number of PowerPC instructions found in a candidate

$$T_{\text{sw}} = T_{\text{cpu}} \cdot L_{\text{sum}}\ [\text{ns}],$$
$$L_{\text{sum}} = \sum_{i=0}^{n} L_i\ [\text{ns}]. \tag{5}$$

Here,

(i) T_{cpu} is the clock period of used PowerPC CPU.

(ii) L_{sum} is the sum of latencies of all instructions found in a candidate, where n is the number of instructions and L_i is the latency of the ith instruction found in a candidate (the instruction latencies have been determined from the PowerPC manual [23]).

The use of the T_{cpu} in (5) ensures that the differences in clock periods between the PowerPC CPU and the UDCI hardware implementation are taken into account.

The presented method yields correct results only when the candidate's IR code is translated one to one into the matching PowerPC instructions. In the case of a mismatch between these two, the estimation results are inaccurate. The mismatch can happen due to a few reasons, that is, folding a few IR instructions into a single target instruction, or because of differences in the register sets. The PowerPC architecture has a fixed amount of registers (32 general-purpose registers) whereas the IR code uses an unlimited number of *virtual registers*. For larger code, which requires more registers than available, the backend of the compiler produces additional instructions which will move data from registers into temporary variables kept on the stack. These additional instructions are not covered in the software estimation process. For such cases it has been shown that the estimation inaccuracy can be as high as 29% [24].

7.2. Hardware Estimation. Since each instruction candidate can be translated to a wide variety of functionally equivalent datapaths, the task of hardware estimation is much more complicated than the task of software estimation. In the following, we choose to illustrate approach used in this work by means of an actual example. Listing 1 shows an excerpt from a raytracing algorithm. The corresponding IR code of one of the identified candidates is presented in Listing 2. This candidate is constructed from adders and multipliers and corresponds to the scalar product which is shown in Listing 1 on the 5th line.

When translated to hardware, the DFG **structure** of the candidate is preserved; that is, instead of complex high-level synthesis [25], a more restricted and thus simpler datapath

```
(1)  bool hitSphere (const ray &r, const sphere& s..)
(2)  {
(3)  ...
(4)  vecteur dist = s.pos - r.start;
(5)  float B = rdx * dx + rdy * dy + rdz * dz;
(6)  float D = B * B - dist * dist + s.size * s.size;
(7)  if (D < 0.0f) return false;
(8)  ...
(9)  }
```

Listing 1: Part of the raytracing source code.

```
(1) ...
(2) %0 = mul float  %rdx, %dx
(3) %1 = mul float  %rdy, %dy
(4) %2 = add float  %0,   %1
(5) %3 = mul float  %rdz, %dz
(6) %4 = add float  %2,   %3
(7) ...
```

Listing 2: IR of candidate found in 5th line of previous listing.

synthesis (DPS) process [1] is required which does not generate complex finite state machines in order to schedule and synchronize computational tasks. The first step in the hardware estimation for DPS involves translating each IR instruction (node) into a corresponding *hardware operator*, where operators may exist as purely *combinational* operators or as *sequential* operators. Sequential operators produce a valid result after one or—in the case of pipelined operators—several clock cycles. Also, functionally equivalent operators can have a large variety of different properties such as hardware area, speed, latency, and power consumption. Therefore, the hardware estimation tasks have to deal with three different types of datapath structures: combinational, sequential, and hybrid datapaths, where a mix of sequential and combinational operators exists. Examples for such datapaths for the discussed candidate are shown in Figure 3. The hardware estimation process used in this work for estimating the delay of a UDCI supports all of these scenarios and is formally defined as

$$C_{\text{hw}} = T_{\text{udci}} \cdot R_D \cdot P_{\max}\,[\text{ns}], \tag{6}$$

$$T_{\text{udci}} = \max\{L\}\,[\text{ns}], \tag{7}$$

$$P_{\max} = \max\{P\}\,[\#]. \tag{8}$$

Here,

(i) T_{udci} corresponds to the minimal allowable *clock period*, which is visualized as the tallest green box in Figure 3. For combinational datapaths, scenario (a), it is equivalent to the latency of critical path, whereas for the sequential datapaths, scenario (b), to the maximal latency of all operators (`add[1]` in this example). For hybrid datapaths, scenario (c), it corresponds to the highest latency of all sequential operators and combinational paths; in this case, to the sum of combinational `mult[0]` and sequential `add[1]` operator latencies.

(ii) R_D is an experimentally determined routing delay parameter which is used to decrease the T_{udci}. The routing delays are equivalent to the communication latencies between connected operators caused by the routing switch boxes and wires. The precise value of R_D is unknown until the physical placement of the operators is performed in the FPGA; however, experiments showed that R_D often corresponds to about half of all circuit latencies.

(iii) $P_{\max}$ is the maximum number of pipeline stages. It can be interpreted as the maximum number of all green rectangles covering a given DFG. For scenarios (a), (b), (c), and (d), $P_{\max}$ equals to 1, 3, 2, and 24, respectively.

(iv) L is a set of latencies generated in Algorithm 1, and its maximum for all operators defines the minimal allowable clock period; see (7). The graphical interpretation of L, as presented in Figure 3, corresponds to a list of the height of all green boxes. Thus, combinational datapaths, scenario (a), have the highest latencies, whereas the smallest latency can be found in highly pipelined sequential datapaths, scenario (d). The latency of each operator is obtained from the PivPav circuit library with the `Latency()` function found in the algorithm in the 4th line.

(v) P is a set of all pipeline stages generated by Algorithm 1. P is used in (8) to select the maximum number of stages in a given datapath. The number of pipeline stages for operator is retrieved with `Pipeline()` function and it is presented in square brackets in the operator name; thus, the `mult [10]`- reflects the 10 stages multiplier.

Algorithm 1 is used to compute the values of the L and P sets, which are associated with the height and the number of green boxes, respectively. In the first line of the algorithm, initialization statements are found. In the second line,

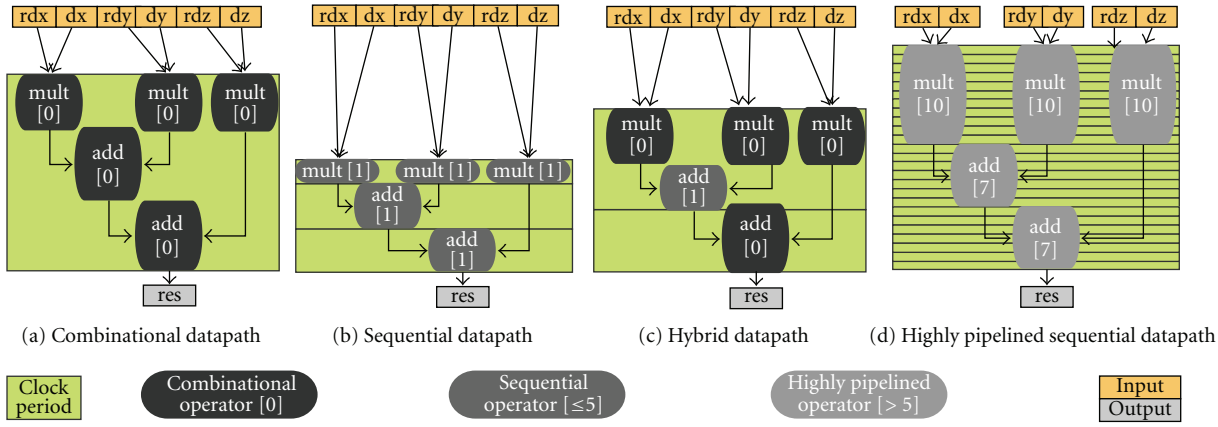

Figure 3: Different types of a DFG presented for a UDCI candidate shown in Listing 2.

```
(1) L ← P ← φ
(2) for p in critical_paths do
(3)     for n ∈ nodes(p) do
(4)         L_p ← L_p + Latency(n)
(5)         P_p ← P_p + Pipeline(n)
(6)         if Pipeline(n) ≠ 0 then
(7)             L ← L ∪ L_p and L_p ← 0
(8)         end if
(9)     end for
(10)    if Pipeline(n) = 0then
(11)        P_p ← P_p + 1
(12)    end if
(13)    L ← L ∪ L_p and L_p ← 0
(14)    P ← P ∪ P_p and P_p ← 0
(15) end for
```

Algorithm 1: Hardware estimation.

the algorithm iterates and generates results for every *critical path*, which indicates that a path leads from the *input* to the *output* node. In next three lines, for each node in a given path, the latency and the number of pipeline stages are accumulated in L_p and P_p temporal variables, respectively. If the given node is sequential, the new green box is created and the current latency value L_p is moved to the L set (lines (6)–(8)). Lines (10)–(15) are executed when the algorithm reaches the last node in the given critical path. Thus, lines (10)–(12) add an additional pipeline stage if the last node is a combinational one, whereas lines (13)-(14) move the values of temporal variables to the resulting sets. Since the candidate's template does not overlap, each node of the candidate is visited only once, and therefore this estimation technique is of $O(n)$ complexity.

In order to illustrate the estimation process in detail, we show how C_{hw} is estimated for (b) sequential datapath and (d) highly pipelined sequential datapath representation of the candidate. The metrics of the used operators obtained from the PivPav circuit library are described below and are presented in Table 2. The upper and lower parts of the table show the metrics of the operators used in scenarios (b) and (d), respectively. The estimation results found in Table 3 indicate that the sequential datapath is able to produce the first result almost twice as fast as the highly pipelined datapath (1.82x). However, processing many data (d) is able to fill the pipeline and work with 24 data at once, generating the results, accordingly to the formula $T_{\text{udci}} * R_D$, every 9.5 ns, whereas (b) generates a result only every 41.78 ns.

The T_{udci} and $P_{\max}$ factors in the equations depend on the characteristics of every used hardware operator; see *Latency()* and *Pipeline()* functions in Algorithm 1. Thus, the key to accurate hardware estimation is the quality of the characterization database that provides performance, latency, and area metrics for each operator. For some selected operators, for example, for floating point operators obtained from *IP Core Libraries*, data sheets that characterize each operator with the required performance metrics are available, for example, [26]. For most other operators—in particular those created on demand by HDL synthesis tools, such as integer, logic, or bit manipulation operators—no data sheets exist. Moreover, the characterization data in data sheets are not exhaustive and do not cover all FPGA families, models, and speed grades, which is problematic, since even within one device family the performance of operators can vary significantly. For example, the data sheet for Xilinx Coregen quotes the maximum speed of a floating-point multiplier as 192 MHz or 423 MHz, respectively, for two FPGA models from the Xilinx Virtex-6 family (Table 23 versus Table 25) [26]. This huge range makes it impractical to estimate accurate performance metrics for devices that are not even tabulated in the data sheets.

To meet the characterization data requirements for the hardware operators, we have developed the open-source PivPav tool [9], which we leverage also in this work to obtain accurate characterization metrics. PivPav generates a library of hardware operators, where each operator is specifically optimized for the targeted FPGA family. For each operator, the performance and many other metrics are automatically extracted from the implementation. These metrics are made available to the estimation process and other processes in the ASIP specialization tool flow via the PivPav API, which

Table 2: Excerpt from metrics requested by the candidate estimation process from PivPav circuit library for the XC4VFX100-FF1152-10 FPGA device.

HW Oper.	P_p #	L_p ns	Max FRQ after PAR [MHz]	FF #	LUT #	Slice #	BUF #	DSP #
mul	1	24.81	40.3	66	76	46	103	0
add	1	32.15	31.1	66	377	250	103	5
mul	10	7.31	136.8	66	134	150	103	4
add	7	7.19	139.0	66	556	326	103	4

Table 3: Results of hardware estimation process for DFGs presented in Figures 3(b) and 3(d) implemented with two different sets of operators found in Table 2.

		Sequential datapath (Figure 3(b))	Highly pipelined sequential datapath (Figure 3(d))
L	[ns]	24.81 and 32.15	7.31 and 7.19
T_{udci}	[ns]	32.15	7.31
P	#	3 and 2	24 and 17
$P_{\max}$	#	3	24
R_D	#	1.30	1.30
C_{hw}	[ns]	125.39	228.07

is illustrated in Figure 2. Since all the data are generated beforehand with the benchmarking facilities of the PivPav, there is no need to run the FPGA CAD tool flow in the estimation process.

It is worth noticing that the presented estimation method is not in itself a novel idea. There are many timing analysis approaches which perform equivalent steps to the one presented above [27]. Therefore, this subsection does not contribute to the hardware timing analysis field. The novelty in this estimation approach relies on the precise characterization data that were generated with PivPav tool. These data together with presented methods allow to precisely estimate the hardware performances for UDCI instructions.

8. Candidate Selection

Once the set of candidates has been determined and estimation data are available, the selection process makes the final decision about which candidates should be implemented in the hardware.

First, all the candidates that violate at least one of the constraints presented below are rejected:

$$C_{|\text{in}|} \leq \text{in}_{\max}, \tag{9}$$

$$C_{|\text{out}|} \leq \text{out}_{\max}, \tag{10}$$

$$\frac{C_{\text{sw}}}{C_{\text{hw}}} \leq \text{threshold}. \tag{11}$$

Here,

(i) $C_{|\text{in}|}$ and $C_{|\text{out}|}$ are equivalent to the constraints described in conditions (1) and (2), respectively. They correspond to architectural constraints for the number of input and output operands to the UDCI. They are applied to the ISE algorithms that are not able to perform this step themselves, such as the MM algorithm.

(ii) C_{sw} and C_{hw} correspond to the software and hardware estimations, respectively. If threshold = 1.0, then there are no performance gains; when threshold >1.0 there is a performance gain since it takes more cycles to execute the candidate in software than in hardware. Finally, if threshold <1.0, the hardware implementation has a lower performance than the software.

After applying these conditions, the search space of the selection process is significantly reduced, since candidates that are either infeasible or would provide only low speedups are discarded. As a result, the runtime of the subsequent steps in the tool flow is considerably lower.

8.1. Candidate Selection. In general, selecting the optimal set of UDCI instructions under different architectural constraints is a demanding task. Related work has studied different selection approaches, such as greedy heuristics, simulated annealing, ILP, or evolutionary algorithms; see, for example, Meeuws et al. [28] or Pozzi et al. [19]. For the purpose of the ASIP specialization, we use the greedy candidate selection algorithm that is presented in Algorithm 2 and which has a computational complexity of $O(|C_{\text{input}}|)$. When using the SC algorithm which may produce overlapping candidates for ISE identification our algorithm rejects any candidates that overlap with any candidate that has been selected so far. The aim of this process is to select up to $C_{\max}$ candidates from the set of C_{input} candidates generated by the identification process that offers the greatest advantage in terms of some metric M; in this case the *application performance*:

$$C_{\text{res}} = \max\left(\forall C_i \in C_{\text{input}} : \sum M(C_i)\right). \tag{12}$$

Here,

(i) C_{res} is the resulting set of best candidates, $|C_{\text{res}}|$ is a size of this set, and $C_{\max}$ is the architectural constraint representing the number of supported UDCI instructions.

(ii) M is a metric function defined as

$$M(C_i) = \frac{C_{\text{sw}}(C_i)}{C_{\text{hw}}(C_i)}. \tag{13}$$

while $|\mathcal{C}_{\text{res}}| \leq \mathcal{C}_{\max}$ **do**
  $c_i \leftarrow \max\{M(\mathcal{C}_i) \mid \mathcal{C}_i \in \mathcal{C}_{\text{input}}\}$
  if c_i does not overlap with $\mathcal{C}_{\text{res}}$ **then**
    $\mathcal{C}_{\text{res}} \leftarrow \mathcal{C}_{\text{res}} \cup c_i$
  end if
  $\mathcal{C}_{\text{input}} \leftarrow \mathcal{C}_{\text{input}} \cap c_i$.
end while

Algorithm 2: Best candidate selection.

8.2. Selection Metrics. The metric function is used as a policy in the greedy selection process and is responsible for selecting only the best candidates. While in (13), the *application performance* policy is used, and nothing prevents basing the decision preference on a different metric. It is worth mentioning that the PivPav tool could be used to provide a wealth of other metrics since the circuit library stores more than 90 different properties about every hardware operator. These properties could be used for instance to develop resource usage or power consumption policies. Consequently, they can be used to estimate the size of the final *bitstream* and partial reconfiguration time, the runtimes of netlist generation and instruction implementation, or many other metrics. Finally, all these policies could be merged together into a sophisticated ASIP specialization framework which would

(i) maximize the performances,

(ii) minimize the power consumption, and

(iii) constrain the resource area to the sizes of UDCI slot.

Such a combined metric can be defined as an *integer linear programming* model. While this method would allow a more precise selection of candidates based on more parameters, its algorithmic complexity is higher than $O(|C_{|\text{input}|}|)$, resulting in runtimes that are much longer, often by orders of magnitude. Since it is important to keep the runtimes of the ASIP-SP as low as possible, the tradeoff between the gains and the costs of the metric function is an interesting research topic in itself.

9. Pruning the Search Space

Pruning is the first process executed in the ASIP-SP outlined in Figure 2. Pruning uses a set of algorithms which act as *filters* to shrink the search space for the subsequent processes by rejecting or by passing certain BBs.

This decision is based on the data obtained from *program analysis* which provides information about loops, the sizes of BBs, and the contained instruction types. In addition, the ASIP-SP makes it possible to discard dead code by running the filters only for the code which was executed at least once.

The objective of the pruning process is described by the following term:

$$\max\left(\frac{\text{metric function}}{\text{runtime of candidate identification}}\right). \quad (14)$$

The pruning aims to maximize the ratio between a *metric function* to the time spent in the candidate identification process. The *metric function* is defined in (13) and is equivalent to the *application performance* gain. The denominator of the equation takes into the account only the *runtime of the candidate identification* since in comparison to this runtime the runtime of the *candidate estimation* and *selection* processes is insignificant.

For this work, we are using this *@50pS3L* filter heuristic, which has been shown in our previous work [10] to provide the best results for just-in-time systems. This filter has three sequential pruning stages which can be decoded from its name. The first filter stage (indicated by @) filters out *dead code*; that is, it discards all BBs that have not been executed at least once. This information is available during the ASIP-SP runtime without access to profiling data. The second filter stage (indicated by *50p*) selects BBs based on their size. Here, only BBs that have a size of at least 50% of the size of the largest BB in the application are selected. Preferring large over small BBs simplifies the task of identifying large candidates which are likely to provide more speedups. Finally, the last filter stage (indicated by *3L*) selects only BBs which are part of a loop and selects the 3 largest of these BBs. The rationale of this filter is that the BBs contained in a loop have a higher chance of being executed, and promoting candidates which are more frequently executed is one of the methods to increase the overall application speedup.

10. Experimental Setup

While this work targets the reconfigurable ASIPs, like our Woolcano architecture presented in Figure 2, due to practical limitations it is currently not feasible to execute the complete ASIP-SP on an embedded reconfigurable ASIP architecture. The specialization process heavily uses the LLVM compiler framework and the Xilinx ISE tools which require high-performance CPUs and desktop operating systems. These resources are not available in currently existing ASIP architectures. Hence, we used Linux and a Dell T3500 workstation (dual core Intel Xeon W3503 2.40 GHz, 4 M L3, 4.8 GT/s, 12 GB DDR3/1333 MHz RAM) as a host computer in place of the PowerPC 405 CPU of the Woolcano architecture to execute the ASIP-SP; see Figure 2.

The lack of the possibility to run the complete tool flow on the ASIP has a number of consequences for the experimental evaluation. Instead of running the ASIP-SP as a single process, we are forced to spilt this process into two steps. In the first step, the host computer is used to generate the partial bitstreams by executing the tasks corresponding to the upper half of Figure 2. In the second step, we switch to the Woolcano architecture where we use the generated bitstreams to reconfigure the UDCI slots and to measure the performance improvements.

It is also worth noticing that this two-step process has an impact on several reported measurements. First, all performance measurements reported in Table 5 and 6, in columns *Max ASIP-SP speedups* and *ASIP ratio*, are performed for Woolcano's PowerPC405 CPU and not for the host CPU.

Table 4: Characteristics of scientific and embedded applications. AVG-S represents the averages for scientific applications and AVG-E for the embedded applications. Ratio = AVG-S/AVG-E.

App	Sources		Compilation to IR				IR in BBs			Code coverage			Kernel size		
	files	loc	real	fun	blk	ins	max	avg	udci	live	dead	const	size	ins	freq
	#	#	[s]	#	#	#	#	#	[%]	[%]	[%]	[%]	[%]	#	[%]
164.gzip	20	8605	3.89	33	1006	6925	59	6.88	29.68	38.86	44.66	16.48	5.78	400	90.34
179.art	1	1270	1.06	21	376	2164	43	5.76	21.53	42.05	28.47	29.48	9.84	213	92.45
183.equake	1	1513	1.71	15	257	2670	132	10.39	23.0	75.39	8.91	15.69	15.32	409	92.9
188.ammp	31	13483	10.10	98	4244	26647	382	6.28	25.74	19.22	70.89	9.89	3.38	901	95.81
429.mcf	25	2685	0.97	18	284	1917	77	6.75	13.09	75.9	13.09	11.01	25.77	494	98.46
433.milc	89	15042	10.88	87	1538	14260	363	9.27	32.59	61.67	34.72	3.61	10.83	1545	93.99
444.namd	32	5315	22.77	84	5147	47534	291	9.24	37.82	31.71	62.81	5.48	7.33	3486	93.64
458.sjeng	23	13847	8.49	86	3373	20531	69	6.09	21.1	48.49	49.44	2.07	44.6	9157	100.0
470.lbm	6	1155	1.36	16	104	1988	405	19.12	57.55	55.23	24.9	19.87	32.75	651	97.57
473.astar	19	5829	3.68	45	757	6010	70	7.94	27.45	78.79	5.31	15.91	6.39	384	91.3
AVG_S	24.70	6874	6.49	50	1709	13065	189.1	8.77	28.95	52.73	34.32	12.95	16.20	1764	94.65
adpcm	6	448	0.29	6	43	233	39	7.21	33.48	60.66	29.18	10.16	41.97	128	91.79
fft	3	187	0.26	10	47	297	41	6.53	42.09	58.88	30.26	10.86	44.08	134	95.98
sor	3	74	0.13	4	19	99	22	7.06	34.34	46.51	50.39	3.1	24.81	32	97.52
whetstone	1	442	0.25	12	44	285	32	6.58	34.04	32.75	36.27	30.99	10.21	29	93.27
AVG_E	3.25	288	0.23	8	38.3	228.5	33.5	6.85	35.99	49.70	36.52	13.78	30.27	80.75	94.64
RATIO	7.60	23.89	28.22	6.29	44.67	57.18	5.64	1.28	0.80	1.06	0.94	0.94	0.54	21.85	1.00

Table 5: Specialization process executed for whole applications when targeting the Woolcano architecture without capacity constraints. The performance of the custom instructions has been determined with the PivPav tool. ISE algorithms: MM: MaxMiso, SC: SingleCut, UN: Union. SC and UN search is constrained to 4 inputs and 1 input.

App	Executionruntimes			ISE algorithm runtime			Candidates found			Max ASIP-SP Speedup		
	VM	Nat	Ratio	MM	SC	UN	MM	SC	UN	MM	SC	UN
	[s]	[s]	x	[ms]	[ms]	[ms]	#	#	#	x	x	x
164.gzip	23.71	18.47	1.28	40.6	549.0	11170.0	1621	44177	43682	1.172	1.213	1.213
179.art	69.92	74.70	0.94	12.3	55.1	3350.0	371	3534	3513	1.526	21.414	21.414
183.equake	7.97	6.79	1.17	13.5	457.9	4351.0	672	9690	9690	2.147	25.972	25.972
188.ammp	23.18	17.24	1.34	145.7	15840	—	7547	122441	—	3.449	20.826	—
429.mcf	23.94	24.06	1.00	11.1	68.7	200.5	571	3571	3562	1.112	1.112	1.112
433.milc	20.95	16.43	1.28	78.1	5065	—	3573	59450	—	1.301	21.546	—
444.namd	39.94	34.31	1.16	227.5	35854	—	11490	125970	—	1.609	24.846	—
458.sjeng	180.41	155.66	1.16	123.7	6244.1	235195.7	5540	83173	83035	1.118	1.137	1.137
470.lbm	5.68	5.36	1.06	8.6	2777.1	—	490	18216	—	2.554	**44.622**	—
473.astar	66.00	67.68	0.98	33.4	914.8	303796653	1408	37025	32368	1.159	1.19	1.19
AVG_S	46.17	42.07	1.14	69.45	6783	30405092	3328	50724	17585	1.71	16.39	5.20
adpcm	29.22	28.35	1.03	1.7	15.0	3869.4	83	819	819	1.243	1.309	1.293
fft	18.47	18.49	1.00	1.6	9.7	33.1	87	553	552	3.1	14.413	14.413
sor	15.83	15.85	1.00	0.7	4.3	14.6	35	384	375	14.418	14.422	14.418
whetstone	28.66	28.50	1.01	1.6	9.5	64.0	69	435	435	**18.012**	18.012	**18.012**
AVG_E	23.04	22.80	1.01	1.40	9.62	995.27	68.50	547.75	545.25	9.19	12.04	12.03
RATIO	2.00	1.85	1.13	49.61	705.05	30549.59	48.59	92.61	32.25	0.19	1.36	0.43

Table 6: The runtime overheads for the ASIP-SP.

App	Candidate Search: *@50pS3L*					ASIP	Runtime overheads				Break-even
	real	pruner	blk	ins	can	ratio	const	map	par	sum	break-even time
	[ms]	effic	#	#	#	x	[m:s]	[m:s]	[m:s]	[m:s]	[d:h:m:s]
164.gzip	1.44	71.79	2	100	19	1.00	56:22	13:02	18:28	87:52	206:22:15:50
179.art	1.05	23.37	3	79	9	1.01	26:42	8:58	13:20	49:00	1:12:18:13
183.equake	2.25	8.33	2	244	11	1.00	32:38	7:56	16:12	56:46	259:02:28:33
188.ammp	3.27	52.29	1	382	92	1.41	272:58	102:12	142:49	517:59	0:14:56:39
429.mcf	1.05	28.2	1	77	5	1.00	14: 50	4:06	7:48	26:44	213:20:05:55
433.milc	6.6	26.71	2	673	9	1.00	26:42	6:44	15:08	48:34	568:06:08:05
444.namd	7.68	57.43	3	776	129	1.03	382:45	117:24	178:04	678:13	6:16:00:48
458.sjeng	1.8	184.11	3	121	8	1.00	23:44	6:56	12:58	43:38	2403:01:35:57
470.lbm	10.62	2.43	3	961	179	2.53	531:07	181:51	308:24	1021:22	1:03:29:48
473.astar	2.25	38.2	3	184	33	1.00	97:54	29:46	46:59	174:39	5149:02:19:14
AVG_S	3.80	49.29	2.30	358	49	**1.20**	146:34	47:53	76:01	270:28	881:00:33:54
adpcm	0.84	5.59	2	61	8	1.08	23:44	6:00	10:34	40:18	0:04:34:10
fft	0.78	3.78	2	75	14	2.40	41:32	11:44	20:56	74:12	0:01:53:07
sor	0.24	2.21	1	22	2	1.00	5:56	4:48	10:12	20:56	0:00:24:19
whetstone	0.54	7.7	2	49	9	15.43	26:42	11:34	25:52	64:08	0:01:08:04
AVG_E	0.60	4.82	1.75	52	8	**4.98**	24:28	8:31	16:53	**49:53**	**0:01: 59:55**
RATIO	6.33	10.23	1.31	6.95	5.99	0.24	5.99	5.62	4.50	5.42	10580

Further, in order to compute the *break-even* time reported in Table 6 we used the *runtime overheads* values from the same table which were measured on the host computer. Therefore, this value is computed as if Woolcano's PowerPC CPU had the processing power of the host machine. Finally, while the ASIP specialization tool flow is capable of performing UDCI reconfiguration during runtime, in practice, we had to switch from the first to the second step manually.

The hardware limitations of Woolcano, in particular the number of UDCI slots, in practice do not allow us to measure the performance improvements on a real system for all applications. To this end, for these applications we estimate the speedups with the help of the techniques presented in Section 7.

11. Applications for Experimental Evaluation

Table 4 shows the characteristics of used applications divided into two groups. The upper part of the table shows data for applications obtained from the SPEC2006 and SPEC2000 benchmark suites which represent scientific computing domain whereas the lower part represents applications from the embedded computing domain obtained from the SciMark2 and MiBench.

While the used benchmark suites count 98 different applications altogether, we could not run our evaluation on all of them due to cross-compilation errors. Hence, from the set of available applications, we selected the ones which are the most representative and allow us to get comprehensive insights into the JIT ASIP specialization methodology. While we have used slightly different application sets in our previous publications which evaluated specific parts of the tool flow, we have chosen an adapted common set of applications for this work in order to get a consistent end-to-end evaluation of the whole tool flow.

11.1. Source Code Characterization and Compilation. The second and third columns of Table 4 contain the number of *source files* and *lines of code* and tell that scientific applications have on average 23.89x more code than the embedded applications. This difference influences the *compilation time* shown in the fourth column which for scientific applications is 28.22x longer on average, but still the average compilation time is only 6.5 s. The next three columns express the characteristics of the *bitcode* reflecting the total number of *functions*, *basic blocks*, and *intermediate instructions*, respectively. For the scientific applications the ratio between *ins* (13065) and the *LOC* (6874) is 1.9. This means that an average single high-level code line is expressed with almost two IR instructions and less than one (0.8) for embedded applications. Since scientific applications have 24 times more LOCs than embedded applications, this results in a 57x difference in the IR instructions.

The ISE algorithms operate on the BBs, and thus the *IR in BBs* column indicates the characteristics of these BBs in more detail. The *max* column indicates the BB with the highest number of IR instructions and *avg* is the average number of IR instructions in all BBs. These values in combination with the data presented in Figures 4, 5, and 6 allow to understand the *runtime* of the ISE and the *number of candidates*.

For embedded applications the largest BBs cover on average more than 14.7% of the application whereas the largest basic block for scientific applications covers only 1.4% of the total application. The difference between average-size BBs for embedded and scientific applications is 1.28x and results in a small average number of IR instructions of less than

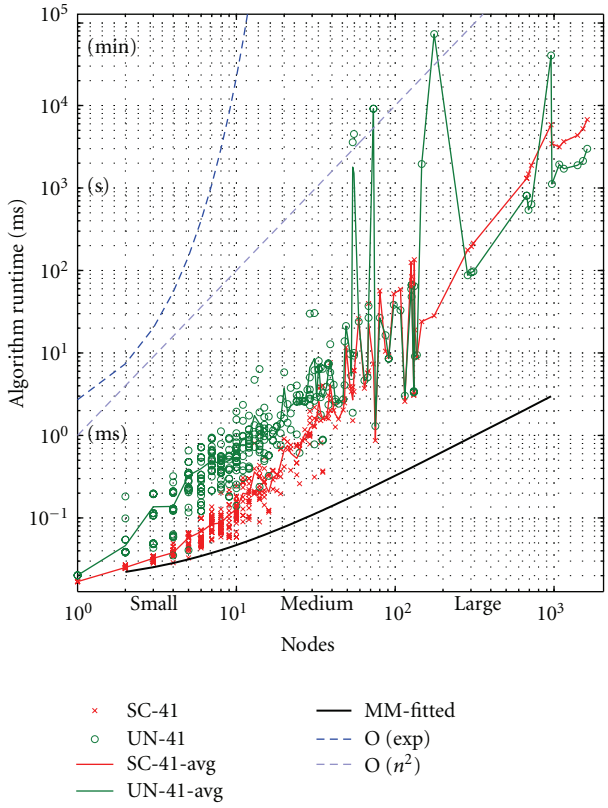

Figure 4: Runtimes of the ISE identification algorithm for different basic block sizes (nodes). The SC: SingleCut, UN: Union, MM: MaxMiso algorithms. The label "41" means that the SC and UN search has been constrained to 4 inputs and 1 output.

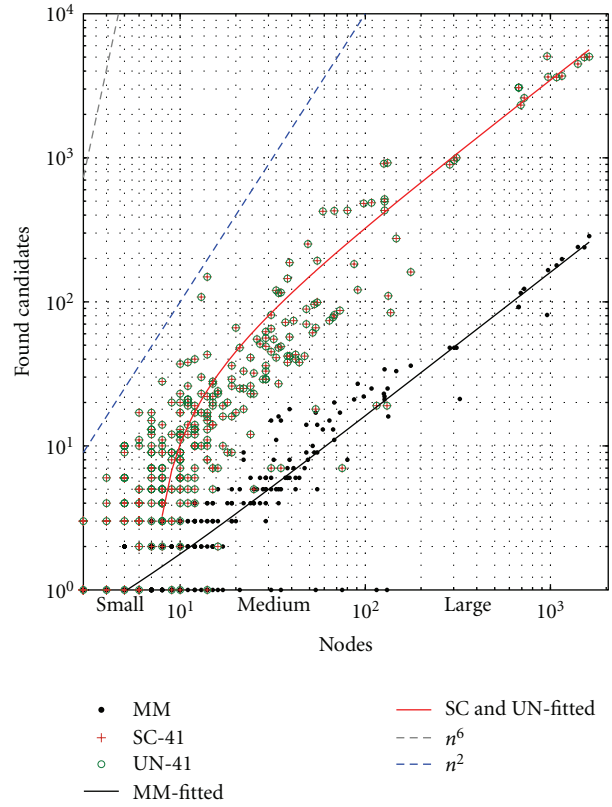

Figure 5: Number of found candidates by ISE algorithms for a large spectrum of BB nodes, where SC and UN are constrained to 4 inputs and 1 output.

10 for both cases. The small size of BBs in our applications needs to be attributed to the actual benchmark code, the compiler, and the properties of the intermediate representation. Our experiments have shown that the size of the BBs does not change significantly for different compiler optimizations, transformations, or with the size of application (LOC).

11.2. Feasible UDCI Instructions. The *udci* column lists the percentage of all IR instructions which are feasible for a hardware implementation. Feasible instructions include the arithmetic, logic, and cast instructions for all data types and make up to 1/3 of all instructions of the application. Considering the small average size of BBs this means that the size of an average-found candidate is only between 2 and 3 IR instructions. This fact emphasizes the need for a proper BB and candidate selection and stresses even more the importance of the proper pruning algorithms in order to avoid spending time with analyzing candidates that will likely not result in any speedup.

11.3. Code Coverage. The *Code Coverage* columns show the percentages of the size of *live*, *dead*, and *constant* code. These values were determined by executing each application for different input data sets and by recording the execution frequency of each BB. For the SPEC benchmark suite applications, the standard *test*, *train*, and *ref* data sets were used, whereas for the embedded ones, due to the unavailability of standard data sets, each application was tested with at least three different custom-prepared input data sets. After execution, the change in execution frequency per block between the different runs was compared. If the frequency was equal to 0, the code was marked as *dead*. If the frequency was different from 0 but did not change for different inputs, the code was marked as *constant*, and if the frequency has changed, the block was marked as *live*. This frequency information was used to compute the *break-even points* in the following section. In addition, the *live* frequency information indicates that roughly only 50% of the application has a dynamic behavior in which the ISE algorithms are interested in searching for candidates.

11.4. Kernel Size. The last three columns contain data on the size of the kernel of the application. These data are derived from the frequency data. The kernel of an application is defined as the code that is responsible for more than 90% of the execution time. The size of the kernel is measured as the total number of IR instructions contained in the basic blocks which represent the kernel. For scientific applications, 16.20% of the code affects 94.65% of the total application execution time, and it corresponds to more than 1.7 k IR

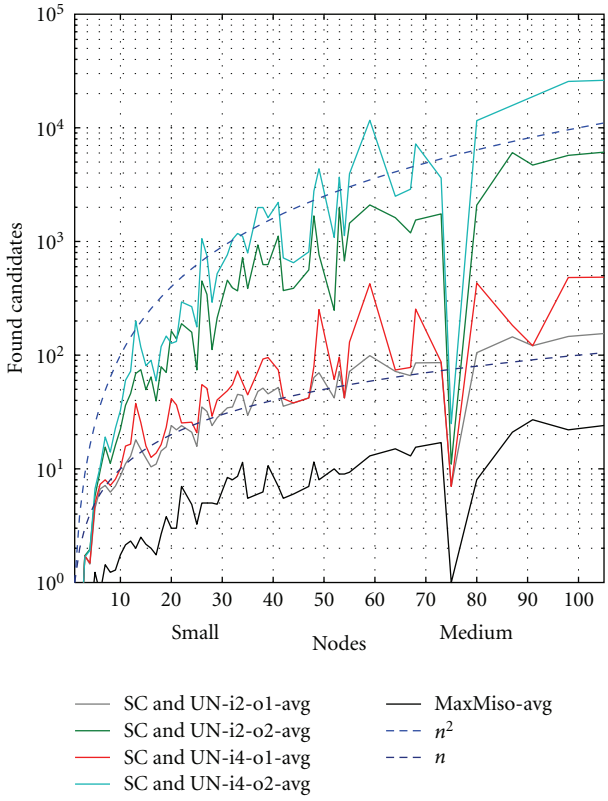

Figure 6: Number of found candidates by ISE algorithms for a medium spectrum of BB nodes, where SC and UN have variable constraints.

instructions. For embedded applications, the average relative kernel size is 30.27% and is expressed only with 80 IR instructions. These numbers indicate that it is relatively easier to increase the performances of the embedded applications than the scientific ones, since they require 22x smaller UDCI instructions.

11.5. Execution Runtimes. The *VM* column in Table 5 represents the application runtime when executed on the LLVM virtual machine. The runtime of the application depends heavily on the input data which, in the case of the scientific applications, were obtained from the *train* datasets of the SPEC benchmark suite. Due to the unavailability of standard data sets for the embedded applications, custom-made data sets were used. For both application classes, the input data allowed to exercise the most computationally intensive parts of the application for a few or several tens of seconds. The *Nat* column shows the *real* runtime of the application when statically compiled, that is, without the overhead caused by the runtime translation. The *Ratio* column shows the proportion of *Nat* and *VM* and represents the overhead involved with the interpretation during the runtime. For the small embedded applications, the overhead of the *VM* is insignificant (1%). For the large scientific applications, the average overhead caused by the VM equals on average 14%. However, it is important to notice that for some applications like *179.art* or *473.astar*, the *VM* was significantly faster than the statically compiled code by 6% and 2%, respectively. This means that the *VM* optimized the code in a way which allowed to overcome the overhead involved in the optimization as well as the dynamic just-in-time compilation.

12. Experimental Evaluation of the Candidate Identification

In this section, we evaluate and compare the ISE algorithms which have been used for candidate identification as presented in Figure 2. Our evaluation covers the runtime characteristics of the algorithms as well as the number of identified candidates for different architectural constraints, the maximum gain in application performance, and the runtime of our benchmark algorithms when statically compiled to native code and when executing in a virtual machine. The discussion is based on data presented in Table 5 which was obtained for the benchmark applications introduced in the previous section.

12.1. ISE Algorithm Runtimes and Comparison. The average ISE algorithm runtimes are presented in the 5th to 7th columns of Table 5. As stated previously, the MM algorithm has a linear complexity and therefore is the fastest, resulting in a 0.22 s runtime for `444.namd`, which is the largest application. Due to its larger algorithmic complexity, the runtime of the UN algorithm should generally exceed the runtime of SC but this is not the case for applications which include specific types of BBs. For example, it took 3837 ms to process such a specific BB consisting of 55 nodes in the `adpcm` application, which is 99.15% of the overall runtime of the UN algorithm. On contrast, the same BB was analyzed by the SC algorithm in a mere 4.7 ms.

Since both SC and UN have exponential complexity, their runtimes are a few orders of magnitude higher than for MM. In average, MM is 96.94x faster than the SC algorithm.

The identification times for BBs of similar sizes also vary significantly for the same algorithm since the number of candidates that need to be actually considered in a BB depends not only on the total size of the BB but also on the structure of the represented DFG, the number and location of infeasible instructions in the DFG, the architectural constraints, and other factors. For example, it took the SC algorithm 1707 ms to analyze a BB of `433.milc` with 102 instructions, while the analysis of a slightly larger BB in `470.lbm` with 120 instructions took only 76 ms, which is a 22.5x difference in runtime. This example illustrates that it is impossible to accurately estimate the runtime of the exponential ISE algorithms SC and UN in advance when basing the estimation solely on the size of the BB.

It is worth pointing out that for keeping the search space and thus the algorithm runtimes manageable, we had to apply rather tight architectural constraints for the custom instructions (4 inputs, 1 output) in our comparison of ISE algorithm in Table 5. When loosening these constraints, the execution times for the SC and UN algorithms rapidly grow from seconds to many hours.

The overall runtime characterization of the instruction identification algorithms is summarized in Figure 4 which plots the runtime of the different algorithms for varying BB sizes. Each data point represents an average which was computed by running the ISE algorithm 1000 times on each BB. The multitude of data points for a fixed BB size illustrates that the runtime of the same algorithm can vary over several orders of magnitude for BBs which have an equal size but differ in their structure as pointed out above. This effect is particularly strong for larger BBs where many variants of DAG structures exist.

For visualizing the overall behavior of the algorithms we have also added the average runtime for the SC and the UN algorithms for each BB size. It can be observed that the variability for the UN algorithm is larger than for the SC algorithm. Another interesting observation is that although the SC and UN algorithms have a worst-case algorithmic complexity of $O(\exp)$, on average their runtime is only polynomial $O(n^2)$, which can be seen by comparing the blue dotted lines with the red and green lines, respectively.

We are able to fit the runtime of the MM algorithm with a linear polynomial model which is represented with a black line which has an almost ideal characteristic (goodness of the fit: $R^2 = 0.9995$). This means that the runtime of the MM algorithm always depends linearly on the BB size $O(n)$. Unfortunately, the behavior of the other algorithms is not sufficiently regular to perform a meaningful curve fitting with similar quality.

In general, we can say that the MM is the fastest algorithm and outperforms the SC and UN algorithms easily in terms of runtime for small, medium, and large basic blocks. For small BBs of up to 10 instructions, the runtime difference is in the range of up to an order of magnitude; for medium inputs (10^2 instructions), up to two orders of magnitude and for the largest BBs (10^3 and more instructions), a difference of more than three orders of magnitude can be observed. While the runtime of the MM stays on the millisecond time scale even for the largest inputs, the SC and UN algorithms work on a scale of seconds or minutes.

It is important to note that runtime of the exponential algorithms tend to literally explode when these algorithms are constrained less tightly than 4 input 1 output (41), in particular when allowing a larger number of outputs. For instance, when applying an 8-input 2-output (82) constraint, common runtimes are in the order of 10^8 ms, which is three orders of magnitude higher than for the 41 constraint.

In terms of runtime, SC is approximately one order of magnitude faster than UN. However, the runtime for both algorithms grows similarly for increasing BB sizes with the exception of significant outliers for the UN algorithm, for peaks with a runtime difference of three orders of magnitude which can be observed for large BBs. A similar behavior was also found for architectural constraints other than 41.

The results presented here have been obtained using a special *benchmarking mode* of our tool flow where the instruction candidates are identified but not copied to a separate data structure for further processing. Additionally, the time needed to reject overlapping candidates for SC algorithm as well as the time needed for the MM algorithm to validate condition (1) presented in Section 6 was not included. As a result, the runtimes of the candidate identification algorithms will be slightly longer in practice when they are applied as part of the complete tool flow.

12.2. Candidates Found by the ISE Algorithms. The number of candidates found by the ISE identification algorithms is presented in columns 8, 7, and 9 of Table 5. In addition, an overview of all identified candidates as a function of the BB size is shown in Figure 5, while Figure 6 presents a close up of the same data for medium-sized BBs. As illustrated by the *red* line in Figure 5, the SC and the UN algorithms generate an equal number of candidates, given that the same architectural constraints are used and that any overlapping UDCI candidates generated by the SC algorithm are removed.

In general, the total number of subgraphs that can be created from an arbitrary graph G is exponential $\exp(n)$ in the number of nodes of G. For ASIP specialization scenarios, that is, when architectural constraints are applied cf. conditions (1), (2), and (4) [29] has shown that the number of subgraphs is bounded by $n^{(C_{\text{in}}+C_{\text{out}}+1)}$. Thus, for the 4-input/1-output constraints applied in this study, the search space is equal to n^6, which is represented by the gray dotted line found in the upper-left corner of Figure 5. When applying the final constraint condition (3), the search space is significantly reduced from n^6 by at least a power of 4 to n^2, which is presented with the blue dotted line above all results.

The *black* line represents a linear fitting for the MM algorithm, whereas the *red* line shows a second-order polynomial curve fitting for the SC and UN algorithms. The goodness of these fits represented with R^2 parameter is equal to 0.9663 for MM and to 0.9564 for the SC and UN algorithms. Therefore, it is safe to assume that the number of candidates for the 4-input/1-output constraint is limited by a second-order polynomial.

Figure 6 shows that the longer runtimes of the SC and the UN algorithm also result in the identification of more candidates. The difference for small and medium BBs is up to one order of magnitude and increases for even larger BBs.

The data points (number of candidates) were obtained by running the ISE algorithms on the applications presented in Section 11. Thus, each data point is associated with a single BB, and closely located data points tell that there are many BBs of similar sizes.

It can be observed in Figure 6 that there are less data points with large BBs than medium or small BBs. This is a consequence of the distribution of basic block sizes; that is, most BBs found in these applications have sizes of up to 100 instructions. For such BBs, the number of feasible candidates reaches more than 10 when using the MM algorithm and more than 100 when using the SC and UN algorithms. Given that the average application has at least a few dozens (38 for embedded) or hundreds (1709 for scientific) of BBs this results in thousands of feasible candidates that are suitable for hardware implementation. In our experiments, we considered 3328 MM (50724 SC) candidates for the scientific and 68.5 MM (547.75 SC) candidates for the embedded applications. These high numbers are more than

enough since an average ISA consists of around 80 core instructions for X86 platform and around 100 for PowerPC. If one assumes 10% modification to the ISA, it results in a task of selecting less than 10 UDCI instructions from a set of thousands of feasible candidates.

The number of found candidates depends strongly on the architectural constraints that are applied. Tighter constraints, that is, allowing a smaller number of inputs and outputs, lead to a smaller number of candidates for the SC and UN algorithm. This behavior can be seen in Figure 6 where the average number of candidates is plotted as a function of the BB size for various constraints. There are two groups of constraints: the 21,41 and 22,42, between which a rising gap of one order of magnitude is established. Applying the MM algorithm to BBs with a size of 100 instructions leads to more than 10 feasible candidates whereas applying SC or UN leads to more than 100 candidates for the first set of constraints and even two orders of magnitude more candidates (10^4) for the second set of constraints. This validates the second-order polynomial characteristic n^2 of the number of candidates for the SC or UN algorithms.

The similar behavior of the lines representing the average number of identified candidates is caused by the *less or equal* ($\leq$) relationships found in conditions (1) and (2). That is, the less constrained algorithms (like 41) include all candidates of more constrained ones like 21. The area between the *red* and *gray* line corresponds exactly to the number of additional candidates found in less constrained algorithms. Also, the graphs illustrate that the number of candidates depends much stronger on the number of allowable outputs than on the number of allowable inputs.

For BBs with sizes of approximately 75 instructions, we see an interesting decay from which all ISE algorithms suffer. This decay is found only in a concrete benchmark and is the result of a high concentration of illegal instructions in basic blocks of those sizes, for which only a few feasible candidates were found.

Finally, it can be seen that the MM algorithm has a linear characteristic $a \cdot n$ where $a \leq 1$. The SC and UN algorithms also show a linear characteristic with $a \geq 1$ for the case of the 21 and 41 constraints, whereas for the 22 and 42 constraints, the characteristic changes by a power (n^2).

12.3. Achievable Performance Gain. The *Max ASIP-SP Speedup* columns presented in Table 5 describe the upper limit of performance improvement that can be achieved with the Woolcano reconfigurable ASIP architecture and the presented ASIP-SP. These values show the hypothetical best-case potential in which all candidates found by three different ISE algorithms are implemented as custom instructions. In reality, the overheads caused by implementing all possible instructions and the limited hardware resources of the reconfigurable ASIP require pruning of the set of candidates that are evaluated and implemented to a tractable subset. Therefore, the speedup quoted in these columns should be treated only as an upper boundary on the achievable performance.

The ISE algorithms have a lot to offer, reaching a speedup of up to 44.62x for SC algorithm and 18.01x for MM and UN algorithms. The average speedups achieved with MM, SC, and UN are 1.71, 16.39, and 5.20, respectively for the scientific applications and 9.19, 12.04, and 12.03 for the embedded applications. For all applications, the average speedups achieved with MM, SC, and UN are 3.85, 15.15, and 10.02, with the value of median 1.57, 16.22, and 7.85, respectively. These results clearly indicate that the SC algorithm is superior for static systems where identification runtimes are not a major concern.

For a JIT specialization process, one needs to balance the achievable speedup with the identification time. Comparing the ratios of average speedup to identification runtime for embedded applications results in the following ratios: 6.56 (MM), 1.25 (SC), and 0.01 (UN). These figures suggest that the MM algorithm is the most suitable for such systems. In addition, the considerable difference of 0.19 between average speedups for different application sets suggests that the MM algorithm could find better candidates in the smaller applications with more pronounced kernels and that these applications will benefit most from JIT-based systems.

It is important to remember that these results were obtained for the first time for the FPGA-based Woolcano architecture and not as presented in related work for a fixed CMOS ASIP architecture. This distinction is significant since the same hardware custom instructions will achieve significantly higher speedups when implemented in CMOS technology, often by more than one order of magnitude. But at the same time, a fixed architecture will sacrifice the flexibility and runtime customization capabilities of the Woolcano architecture.

13. Runtime Overhead of the ASIP Specialization Process

As elaborated in the previous section, the reconfigurable ASIP architecture is considerably faster than the underlying CPU alone for both benchmark domains. Thus, the overheads of just-in-time software compilation, optimization, and custom instruction generation can be amortized provided that the application will be executed long enough. In this section, we analyze the achievable performance gains by ASIP specialization, presented in Figure 2, and the runtime costs of the three different phases of that process. These figures are used to compute for how long the application needs to be executed until the hardware generation overheads are amortized, that is, when a net speedup is achieved.

13.1. Candidate Search. As described in Section 4, the *Candidate Search* phase is responsible for finding and selecting only the best custom instruction candidates from the software. As this task is frequently very time consuming, we are using our pruning mechanisms introduced in Section 9 to reduce the search space for instruction candidates. The number of selected candidates, after pruning, is indicated in the *can* column of Table 6.

The third column of Table 6 represents the *pruning efficiency* ratio which is defined as the quotient of two terms. The first term is the ratio of the average maximum ASIP

speedup to the runtime of the identification algorithm when no pruning is used. The second term is the same ratio when using the *@50pS3L* pruning mechanism. The pruning efficiency can be used as a metric to describe the relative gain in the speedup-to-identification-time ratio with and without pruning.

The *blk* and *ins* columns represent the number of basic blocks and instructions which have been passed to the identification process. These numbers are significantly lower than the total number of blocks and instructions presented in the *6th* and *7th* columns of Table 4. That is, the pruning mechanism reduced the size of the bitcode that needs to be analyzed in the identification task by a factor of 36.49x and 4.4x for scientific and embedded applications, respectively.

The overall runtime of the data pruning, identification, estimation, and selection is aggregated in the *real* column. The total candidate search time is in the order of milliseconds and thus insignificant in comparison to the overheads involved in the hardware generation.

13.2. Performance Improvements. The column *ASIP ratio* represents the speedup of the augmented hardware architecture when all candidates selected by Candidate Search are offloaded from the software to custom instructions. In contrast to the maximum performance shown in the 11th column in Table 5 which assumes that *all* candidates are moved to hardware, the average speedup drops by 30% from 1.71x–1.20x for scientific applications and by 46% from 9.19x–4.98x for the embedded ones. Comparing the *fft* with the *470.lbm* applications illustrates the main difference between embedded and scientific applications. Both applications have a similar speedup of 2.40x versus 2.53x, respectively, but differ significantly in the number of candidates that need to be translated to hardware to achieve these speedups (14 versus 179 candidates). This correlates with the previously described observation that scientific applications have a significantly larger kernel size.

13.3. Netlist Generation. The tasks discussed in this section are represented by the second phase in Figure 2. The task *Generate VHDL* is performed with the PivPav datapath generator which produces the *structural VHDL* code. The datapath generator traverses the datapath graph of the candidate and matches every node with a VHDL component. This is a constant time operation requiring 0.2 s per candidate. The *extract netlist* task retrieves the netlist files for each hardware component used in the candidate's VHDL description from the PivPav database. This step allows reduction of the FPGA CAD tool flow runtimes, since the synthesis process needs to build only a final netlist for the top module. The next step is to *create the FPGA CAD project* which is performed by PivPav with the help of the *TCL* scripting language. After the project is created, it is configured with the FPGA parameters and the generated VHDL code as well as the extracted netlist files are added. On average this process took 2.5 s per candidate, making this the most time-consuming task of the netlist generation phase. The average total runtime for these three tasks is presented in the *C2V* column of Table 7 and amounts to 3.22 s. As the standard deviation is only 0.10, this time can be considered as constant.

Table 7: Constant overheads involved in the ASIP-SP. *C2V* corresponds to the *Netlist Generation* phase in Figure 2. Syn, Xst, Tra, and Bitgen are the FPGA CAD tool flow processes and correspond to the syntax check, synthesis, translate, and partial reconfiguration bitstream generation processes, respectively, which can be found in the third phase in Figure 2.

	C2V [s]	Syn [s]	Xst [s]	Tra [s]	Bitgen [s]	Sum [s]
Average	3.22	4.22	10.60	8.99	151.00	178.03
Stdev	0.10	0.10	0.23	1.22	2.43	

13.4. Instruction Implementation. Once the project is created it can be used to generate the partial reconfiguration bitstream representing the FPGA implementation of the custom instruction. This step is performed with the FPGA CAD tool flow which includes several steps. First, the VHDL source code is checked for any *syntax* errors. The runtime of this task is presented in the second column of Table 7. On average it takes 4.22 s to perform this task for every candidate. Since the *stdev* is very low (0.10) we can assume that this is a constant time too.

Once the source code is checked successfully the *synthesis* process is launched. Since all the netlists for all hardware components are retrieved from a database there is no need to resynthesize them. The synthesis process thus has to generate a netlist just for the top-level module which on average took 10.60 s. The runtime of this task does not vary a lot since the VHDL source code for all candidates has a very similar structure and changes only with the number of hardware components. After this step all netlists and constraint files are consolidated into a single database with the *translate* task, which runs for 8.99 s on average.

In the next step, the most computationally intensive parts of the tool flow are executed. These are the *mapping* and the *place and route* tasks which are not constant time processes as the previous tasks, but their duration depends on the number of hardware components and the type of operation they perform. For instance, the implementation of the shift operator is trivial in contrast to a division. The spectrum of runtimes for the *mapping* process ranges from 40 s for small candidates up to 456 s for large and complex ones, whereas the *place and route* task takes 56 s–728 s. There is no strict correlation between the duration of these processes; the ratio of *place and route* and *mapping* runtimes vary from 1.4x for small candidates to 2.5x for large candidates. The last step in the hardware custom instruction generation process is the bitstream generation. Our measurements show that this is again a constant time process depending only on the characteristics of the chosen FPGA. Surprisingly, the runtime of this task is substantial. On average, 151 s per candidate are spent to generate the partial reconfiguration bitstream. This runtime is constant and does not depend on the characteristics of a candidate. In many cases, the bitstream creation consumed more time than all other tasks of the instruction

synthesis process combined (including synthesis and place-and-route). The runtime is mainly caused by using the early access partial reconfiguration Xilinx 12.2 FPGA CAD tools (EAPR). In comparison, creating a full-system bitstream that includes not only the custom instruction candidate but also the whole rest of the FPGA design takes just 41 s on average when using the regular (non-EAPR) Xilinx FPGA CAD tools.

In Table 7, we summarize the runtime of the processes which cause constant overheads that are independent of the candidate characteristics. These are the *Candidate to VHDL translation (C2V)*, *Syntax Check (Syn)*, *Synthesis (Xst)*, *Translation (Tra)*, and *Partial Reconfiguration Bitstream Generation (Bitgen)*. The total runtime for these processes is 178.03 s and is inevitable when implementing even the most simple custom instruction. The *Bitgen* process accounts for 85% of the total runtime.

The overall runtime involved in the FPGA CAD Tool Flow execution is presented in the column *Runtime Overheads* in Table 6. The column *const* represents the runtime of constant processes shown in Table 7. The column *map* stands for the mapping process, the column *par* for the *place and route*, and the values in the column *sum* adds all three columns together. These columns aggregate the total runtime involved in the generation of all candidates for a given application. The candidate's partial reconfiguration times were not included in these runtimes since they consume just a fraction of a second [6]. On average it takes less than 50 minutes (49 : 53 min) to generate all candidates for the embedded applications but more than 4 : 30 hours (270 : 28 min) for the scientific applications. One can see that this large difference is closely related to the number of candidates and that *sum* column grows proportionally with the number of candidates. This behavior can be observed for example for the *444.namd* and the *470.lbm* applications, which consist of 179 and 129 candidates, respectively. The total runtime overhead for them is more than 11 hours (678 : 13 min) and 17 hours (1021 : 22 min), respectively and is caused primarily by the high constant time overheads (*const*).

This observation emphasizes the importance of the pruning algorithms, particularly for the large scientific applications. We can observe the difference for the embedded applications where a smaller number of candidates exists. On average, the *const* time drops for the scientific applications from 146 : 34 min to 24 : 28 min, that is, by a factor of 5.99x, which is exactly the difference in the number of candidates *(can)* between the scientific and the embedded applications.

13.5. Break-Even Times. In this section, we analyze the *break-even time* for each application; that is, the minimal time each application needs to execute before the overheads caused by the ASIP-SP is compensated.

A simplistic way of computing the break-even time would be to divide the total runtime overhead (*sum* in Table 6) by the time saved during one execution of the application, which can be computed using the *VM execution time* and the *Max ASIP Speedup* (speedup) (see Table 5). This computation assumes a scenario, where the size of the input data is fixed and the application is executed several times.

We have followed a more sophisticated approach of computing the break-even time, which assumes that more input data is processed instead of multiple execution of the same application. Hence, the additional runtime is spent only in the parts of the code which are *live* while code parts that are *const* or *dead* are not affected. To this end, we use the information about the execution frequency of basic blocks and the variability of this execution frequency for different benchmark sizes which we have collected during profiling; see Section 11.3. The resulting break-even times are presented in the last column of Table 6.

It is evident that there exists a major difference in the break-even times for the embedded and the scientific applications. While the break-even time of the embedded applications is in the order of minutes to a few hours, the scientific applications need to be executed for days to amortize the overhead caused by custom instruction implementation (always under the assumption that *all* candidates are implemented in hardware). The reason for these excessive times is the combination of rather long ASIP-SP runtimes *(>4 : 30 h)* and modest performance gains of 1.2x. As described above, the long runtimes are caused by implementing many candidates. One might expect that this large number of custom instructions should cover a sizable amount of the code and that significant speedups should be obtained, but evidently this is not the case. The reason for this is that the custom instructions are rather small, covering only 6.9 IR instructions on average. Although there are many custom instructions generated, they cover only a small part of the whole computationally intensive kernels of the scientific application, which has a size of 1764 IR instructions on average. Adding more instructions will not solve this issue since every candidate adds an additional FPGA CAD tool flow overhead.

In contrast, the break-even point for embedded applications is reached more easily. On average, the break-even time is five orders of magnitude lower for these applications. In contrast to the scientific applications, the custom instructions for embedded application can cover a significant part of the computationally intensive kernel. This results in reasonable performance gains with modest runtime overheads. For an average-embedded application, a 5x speedup can be achieved, resulting in a runtime overhead of less than 50 minutes and a break-even time of less than 2 hours.

The difference between scientific and embedded applications is not caused by a significant difference in the number of IR instructions in the selected candidates. Scientific applications have on average 7.31 instructions per candidate, while embedded applications have on average 6.5 instructions per candidate.

Since we cannot decrease the size of the computational kernel, we should strive for finding larger candidates in order to cover a larger fraction of the kernel. Unfortunately, this turns out to be difficult because the reason that the candidates are small is that the BBs *(blk)* in which they are identified are also small. The average basic block has only 7.64 (5.94) IR instructions for a scientific (embedded) application (see Table 5).

The pruning mechanism we are using is directing the search for custom instruction to the largest basic blocks;

hence, the average basic block that passes the pruning stage has 155.65 instructions for a scientific and 29.71 for embedded application (see Table 6). However, even these larger blocks include a sizable number of the hardware-infeasible instructions, such as accesses to global variables or memory, which cannot be included in a hardware custom instruction. As a result, there are only 7.31 instructions per candidate in a scientific application which causes high break-even times for them.

This observation illustrates that there are practical limitations for the ASIP-SP when using code that has been compiled from imperative languages.

14. Reduction of Runtime Overheads

In this section, we propose two approaches for reducing the total runtime overheads and in turn also the break-even times: partial reconfiguration bitstream caching and acceleration of the CAD tool flow.

14.1. Partial Reconfiguration Bitstream Caching. As in many areas of computer science, caching can be applied also in the context of our work. Much like virtual machines cache the binary code that was generated on the fly for further use, we can cache the generated partial bitstreams for each custom instruction. To this end, each candidate needs to have a unique identifier that is used as a key for reading and writing the cache. We can, for example, compute a signature of the LLVM bitcode that describes the candidate for this purpose. The cached bitstreams can be stored for example in an on-disk database.

14.2. Acceleration of the CAD Tool Flow. A complementary method for reducing the runtime overheads is to accelerate the FPGA CAD tool flow. There are several options to achieve this goal. One possibility is to use a faster computer that provides faster CPUs and faster and larger memory or to run the FPGA tool concurrently. Alternatively, it may be possible to use a smaller FPGA device, since the *constant* processes of the tool flow depend strongly on the capacity of the FPGA device. We have used a rather large *Virtex-4 FX100* device, therefore switching to a smaller device would definitely reduce the runtime of the tool flow. Another option would be to use a memory file system for storing the files created by the tool flow. As the FPGA CAD tool flow is known to be I/O intensive, this should speed up the tool flow. Finally, we could change our architecture to a more coarse-grained architecture with simplified computing elements and limited or fixed routing. It has been shown that it is possible to develop customized tools for such architectures which work significantly faster [30].

14.3. Extrapolation. In Table 8 we calculate the average *breaking-even time* for the embedded applications when applying these ideas. When the cache is disabled and we do not assume any performance gain from the tool flow, the first value is equal to the *AVG_E* row and the last column in Table 6. One can note also that these values do not scale linearly because we consider the frequency information for basic blocks.

Table 8: The average *breaking-even time* for the embedded applications using a *partial reconfiguration bitstream* cache and a faster FPGA CAD tool flow.

	Faster FPGA CAD tool flow [%]			
Cache hit [%]	0 [h:m:s]	30 [h:m:s]	60 [h:m:s]	90 [h:m:s]
0	01:59:55	01:24:48	00:48:27	00:12:07
10	01:47:44	01:15:25	00:43:06	00:10:46
20	01:32:59	01:05:05	00:37:11	00:09:18
30	**01:28:09**	**01:01:42**	**00:35:15**	**00:08:49**
40	01:13:08	00:51:11	00:29:15	00:07:19
50	01:01:00	00:42:42	00:24:24	00:06:06
60	00:48:50	00:34:10	00:19:32	00:04:53
70	00:35:12	00:24:38	00:14:05	00:03:31
80	00:29:19	00:20:31	00:11:43	00:02:56
90	00:14:07	00:09:53	00:05:39	00:01:24

For this evaluation, we varied the assumed cache hit rate to be between 0%–90%. That is, for simulating a cache with a 20% hit rate, we have populated the cache with 20% of the required bitstreams for a particular application, whereas the selection whose bitstreams are stored in the cache is random. Whenever there is a *hit* in the cache for a given candidate, the whole runtime associated with the generation of the candidate is subtracted from the total runtime; see *sum* column in Table 6. The values in the *Faster FPGA CAD tool flow* columns are decreasing linearly with the assumed speedup.

If we assume that the FPGA CAD tool flow can be accelerated by 30% and that we have 30% cache hits, the average break-even time drops almost by half (1.94x), from *1:59:55 h* to *1:01:42 h*. This means that the whole runtime of the ASIP-SP could be compensated in a bit more than one hour and for the rest of the time the adapted architecture would provide a performance gain by an average factor of 5x. These assumptions are modest values since the *cache hit* rate depends only on the size of the cache, and our Dell T3500 workstation could be easily replaced by a faster one.

15. Conclusion and Future Work

In this work, we have studied the just-in-time ASIP-SP for an FPGA-based reconfigurable architecture. The most significant parts of this process, including the candidate identification, estimation, selection, and pruning mechanisms were not only described with precise formalisms but were also experimentally evaluated. In particular, we discussed and compared characteristics of three state-of-the-art instruction set extension algorithms in order to study the candidate identification mechanism in detail. This study included not only algorithm runtimes, number of found UDCI candidates, their properties, and impact of algorithm constraints on the search space, but more importantly the achievable maximum

performance gains for various embedded computing and scientific benchmark applications.

The study of the ASIP process was performed both separately for every element but more importantly also for the entire ASIP process, where the feasibility and limitations were investigated. The study has shown that for embedded applications an average speedup of 5x can be achieved with a runtime overhead of less than 50 minutes. This overhead can be compensated if the application executes for two hours or for one hour when assuming a 30% cache hit rate and a faster FPGA CAD tool flow. Our study further showed that the larger and more complex software kernels of scientific applications, represented by the SPEC benchmarks, do not map well to custom hardware instructions targeting the Woolcano architecture and lead to excessive times until the break-even point is reached. The reason for this limitation can be found in the properties of the intermediate code generated by LLVM when compiling C code, in particular, rather small basic block sizes with an insufficient amount of instruction level parallelism. Similar results are expected for other imperative languages. Simultaneously, this work has explored the potential of our Woolcano reconfigurable architecture, the ISE algorithms, and pruning mechanism for them as well as the PivPav estimation and datapath synthesis tools.

References

[1] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications*, Morgan Kaufmann, San Francisco, Calif, USA, 2006.

[2] M. J. Wirthlin and B. L. Hutchings, "Dynamic instruction set computer," in *Proceedings of the 3rd IEEE Symposium on FPGAs for Custom Computing Machines, (FCCM '95)*, pp. 99–107, IEEE Computer Society, April 1995.

[3] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proceedings of the 27th International Symposium on Microarchitecture, (MICRO '94)*, pp. 172–180, ACM, New York, NY, USA, November 1994.

[4] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "Chimaera: a high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proceedings of the 27th Annual International Symposium on Computer Architecture, (ISCA '00)*, pp. 225–235, ACM, June 2000.

[5] P. M. Athanas and H. F. Silverman, "Processor reconfiguration through instruction-set metamorphosis," *Computer*, vol. 26, no. 3, pp. 11–18, 1993.

[6] M. Grad and C. Plessl, "Woolcano: an architecture and tool flow for dynamic instruction set extension on Xilinx Virtex-4 FX," in *Proceedings of the 9th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA '09)*, pp. 319–322, CSREA Press, Monte Carlo Resort, Nev, USA, July 2009.

[7] J. M. Arnold, "S5: the architecture and development flow of a software configurable processor," in *Proceedings of the International Conference on Field Programmable Technology, (ICFPT '05)*, pp. 121–128, IEEE Computer Society, Kent Ridge Guild House, Singapore, December 2005.

[8] S. Borkar, "Design challenges of technology scaling," *IEEE Micro*, vol. 19, no. 4, pp. 23–29, 1999.

[9] M. Grad and C. Plessl, "An open source circuit library with benchmarking facilities," in *Proceedings of the 10th International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA, '10)*, T. P. Plaks, D. Andrews, R. F. DeMara et al., Eds., pp. 144–150, CSREA Press, Las Vegas, Nev, USA, July 2010.

[10] M. Grad and C. Plessl, "Pruning the design space for just-in-time processor customization," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '10)*, pp. 67–72, IEEE Computer Society, Cancun, Mexico, December 2010.

[11] M. Grad and C. Plessl, "Just-in-time instruction set extension—feasibility and limitations for an FPGA-based reconfigurable ASIP architecture," in *Proceedings of the 18th Reconfigurable Architectures Workshop, (RAW '11)*, pp. 278–285, IEEE Computer Society, May 2011.

[12] M. Wazlowski, L. Agarwal, T. Lee et al., "PRISM-II compiler and architecture," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines, (FCCM '93)*, pp. 9–16, IEEE Computer Society, April 1993.

[13] C. Galuzzi and K. Bertels, "The instruction-set extension problem: a survey," in *Proceedings of the International Conference on Architecture of Computing Systems, (ARCS '08)*, Lecture Notes in Computer Science, no. 4943, pp. 209–220, Springer/Kluwer Academic, Dresden, Germany, February 2008.

[14] R. J. Hookway and M. A. Herdeg, "DIGITAL FX!32: combining emulation and binary translation," *Digital Technical Journal*, vol. 9, no. 1, pp. 3–12, 1997.

[15] V. Bala, E. Duesterwald, and S. Banerjia, "Transparent dynamic optimization," Tech. Rep. number HPL-1999-78, HP Laboratories Cambridge, 1999.

[16] K. Ebcioglu and E. R. Altman, "DAISY: dynamic compilation for 100% architectural compatibility," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 26–37, New York, NY, USA, June 1997.

[17] F. Vahid, G. Stitt, and R. Lysecky, "Warp processing: dynamic translation of binaries to FPGA circuits," *Computer*, vol. 41, no. 7, pp. 40–46, 2008.

[18] A. C. S. Beck and L. Carro, "Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility," in *Proceedings of the 42nd Design Automation Conference, (DAC '05)*, pp. 732–737, New York, NY, USA, June 2005.

[19] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1209–1229, 2006.

[20] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, (CASES '04)*, pp. 69–78, Washington, DC, USA, September 2004.

[21] C. A. William, W. Fornaciari, L. Pozzi, and M. Sami, "A DAG-based design approach for reconfigurable VLIW processors," in *Proceedings of the Design, Automation and Test in Europe Conference, (DATE '99)*, pp. 778–779, ACM, Munich, Germany, January 1999.

[22] J. Gong, D. D. Gajski, and S. Narayan, "Software estimation from executable specifications," *Journal of Computer Software Engineering*, vol. 2, pp. 239–258, 1994.

[23] *The PowerPC 405TM Core*, IBM, 1998.

[24] A. Ray, T. Srikanthan, and W. Jigang, "Practical techniques for performance estimation of processors," in *Proceedings of the International Workshop on System-on-Chip for Real-Time Applications, (IWSOC '05)*, pp. 308–311, IEEE Computer Society, Washington, DC, USA, 2005.

[25] B. So, P. C. Diniz, and M. W. Hall, "Using estimates from behavioral synthesis tools in compiler-directed design space exploration," in *Proceedings of the 40th Design Automation Conference*, pp. 514–519, New York, NY, USA, June 2003.

[26] *Floating-Point Operator v5.0*, Xilinx.

[27] N. Maheshwari and S. S. Sapatnekar, *Timing Analysis and Optimization of Sequential Circuits*, Springer/Kluwer Academic Publishers, Norwell, Mass, USA, 1999.

[28] R. Meeuws, Y. Yankova, K. Bertels, G. Gaydadjiev, and S. Vassiliadis, "A quantitative prediction model for hardware/software partitioning," in *Proceedings of the International Conference on Field Programmable Logic and Applications, (FPL '07)*, pp. 735–739, Amsterdam, The Netherlands, August 2007.

[29] P. Bonzini and L. Pozzi, "Polynomial-time subgraph enumeration for automated instruction set extension," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pp. 1331–1336, Nice, France, April 2007.

[30] E. Bergeron, M. Feeley, and J. P. David, "Hardware JIT compilation for off-the-shelf dynamically reconfigurable FPGAs," in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction (CC/ETAPS'08)*, pp. 178–192, Springer-Verlag, Berlin, Heidelberg, 2008.