Just-in-time Instruction Set Extension - Feasibility and Limitations for an FPGA-based Reconfigurable ASIP Architecture

Mariusz Grad and Christian Plessl
Paderborn Center for Parallel Computing, University of Paderborn
{mariusz.grad|christian.plessl}@uni-paderborn.de

***Abstract*—In this paper, we study the feasibility of moving the instruction set customization process for reconfigurable ASIPs to runtime under the precondition that current FPGA devices and tools are used. To this end we propose a tool flow for just-in-time ASIP customization which identifies suitable custom instructions to accelerate arbitrary binary applications that execute on a virtual machine. The tool flow targets our previously introduced Woolcano reconfigurable ASIP architecture, which augments the PowerPC core in a Xilinx Virtex 4FX CPU with runtime reconfigurable instructions. We evaluate the tool flow with a comprehensive set of applications from the SPEC2006, SPEC2000, MiBench, and SciMark2 benchmark suites and compare the speedups that can be achieved with the overhead of the complete ASIP specialization process. We show that an average speedup of $5\times$ can be achieved for benchmarks from the embedded computing domain. The overhead of custom instruction identification and hardware generation for these benchmarks is less than 50 minutes and will be compensated if the applications execute for more than 2 hours. Finally, we evaluate caching strategies to reduce the time until this break even point is reached.**

I. 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 number of reconfigurable ASIP architectures have been proposed in academic research [2]–[6] which implement custom instruction in reconfigurable logic. There exist also a number of commercially available CPU architectures that allow for customizing the instruction set, e. g., 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, the idea of adapting the instruction set during runtime has been hardly explored.

A number of obstacles make the exploitation of just-in-time ISE challenging: 1) there are only very few silicon implementations of reconfigurable ASIP architectures commercially available; 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. In our previous research, we have addressed obstacles 1) and 2). In [6] we have introduced the Woolcano reconfigurable instruction set architecture. Woolcano is based on a Xilinx Virtex 4FX FPGA and allows for augmenting the PowerPC core in the device with user-defined instructions that can be changed at runtime using partial reconfiguration. In [8] we have presented a circuit library and data path generator that can generate custom instructions for this architecture. In our recent work [9] we have presented new heuristics for reducing the runtime of methods for identifying and selecting custom instructions for just-in-time ISE.

The goal of this work is to gain insights into the controversial question whether just-in-time processor customization is a worthwhile idea under the assumption that existing commercially available FPGA devices and tools are used. Specifically we investigate how the long runtimes of FPGA implementation tools, mentioned as obstacle 3) above, limit the applicability of the approach. 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 total required execution time until a net speedup is achieved stays within practical bounds. In this paper we study this question for a set of benchmark applications from embedded and scientific computing targeting our Woolcano architecture.

II. Related Work

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

From the hardware perspective, we do not target static but reconfigurable ASIP architectures, such as our Woolcano architecture [6] or comparable architectures, like CHIMAERA [4], PRISC [3] or PRISM [10]. These architectures

Published in: *Proc. Reconfigurable Architectures Workshop (RAW)*
 DOI: 10.1109/IPDPS.2011.153

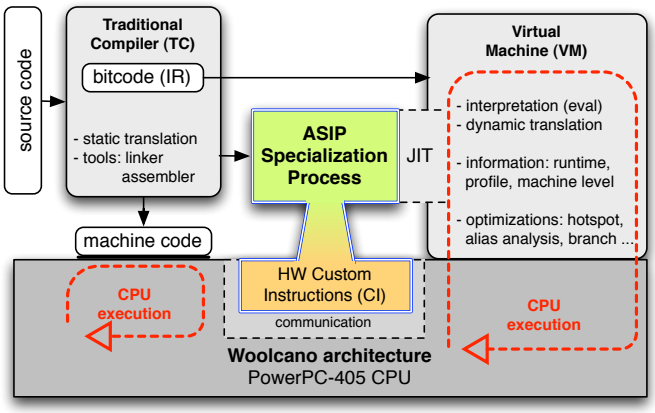

Figure 1: Overview of the tool flow

provide programmable functional units that can be dynamically reconfigured during the runtime in order to implement arbitrary custom instructions. We have shown in previous work [6], [9] that it is possible to accelerate applications from the SPEC2006 benchmark suite up to $44\times$.

Research in the areas of ISE algorithms for ASIP architectures is extensive, a recent survey can be found in [11]. 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 the case for just-in-time ISE. In this work, we leverage our preliminary work [9] in which we have studied new heuristics for effectively pruning the search space for ISEs. We have 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 [12]. Binary translation has also been used for cases where the source and target ISA are identical with the objective to create a binary with a higher degree of optimization [13], [14].

Our work is conceptually similar to these approaches as we also do not translate between different instruction set, but optimize 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 [15]. 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. Our 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 data-path, while WARP uses a bus-attached FPGA co-processor that is more loosely coupled with the CPU. Hence, we can offload operations at the instruction-level where WARP needs offload whole loops to the accelerators in order to cope with

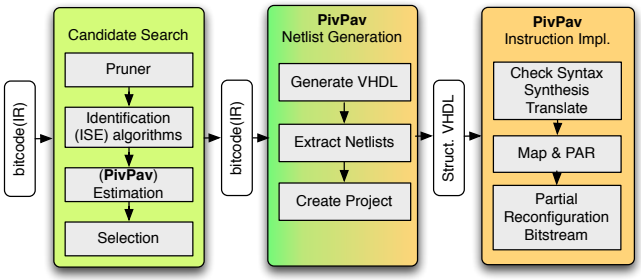

Figure 2: ASIP specialization process

longer communication delays. Further, WARP operates at the machine code level and reconstructs the program's higher-level structure with decompilation, while we rely on higher-level information that is present in the virtual machine. Finally, WARP assumes a custom system-on-chip, while we target a commercially available standard FPGAs.

In [16] Beck and Carro present work on binary translation of Java programs for a custom reconfigurable ASIP architecture with coarse-grained reconfigurable data-path units. They show that for a set of small benchmarks an average speedup of $4.6\times$ and power reduction of $10.9\times$ can be achieved. The identification and synthesis of new instructions occurs at runtime, however the paper does not specify what methods are used for instruction identification and what overheads arise from instruction synthesis.

III. Tool Flow

In Figure 1 we illustrate the difference between a conventional static ISE tool flow and our proposed just-in-time ISE tool flow. In both cases, the application source code is compiled into an intermediate representation. This intermediate representation is either translated to machine code for a specific CPU architecture at compile time (static compilation) or at runtime using a virtual machine (dynamic compilation). For dynamic compilation, the intermediate representation is usually denoted as bytecode or bitcode. ISE has been applied so far almost exclusively in static compilation. That is, the ISE algorithms operate on the intermediate representation, select the parts of the code to be translated to custom instructions, and generate hardware descriptions for the custom instructions and software binaries that use these instructions. The synthesis of the custom instructions occurs offline before the application is executed.

In this work we study the feasibility of moving this ASIP specialization process to runtime for systems that use a virtual machine with just-in-time compilation of bitcode. Hence, this process is performed concurrently with the execution of the application. As soon as it is completed and configurations for the application-specific instructions have been generated, the adaptation phase occurs where ASIP architecture is reconfigured and the application binary is modified such that the newly available custom instructions are used.

The details of the ASIP specialization process (ASIP-SP) 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 JIT ASIP-SP. To this end, in [9] we have studied three state-of-the art ISE algorithms and proposed a set of efficient heuristics for *pruning* the search space for ISE to the basic blocks for which the best performance improvements can be expected. We have shown that the runtime of the ISE algorithms can be reduced by *two orders of magnitude* by sacrificing 1/4 of the speedup. In this paper, we uses the *@50pS3L* pruning filter and the MAXMISO linear complexity ISE algorithm, refer to [9] for a detailed explanation. The MAXMISO algorithm identifies a set of custom instruction candidates and afterwards the *selection* process selects only the best of them with the help of the performance *estimation* data. The *estimation* data are computed by our PivPav tool [8] and they represent the performance difference for every candidate when executed in software or in hardware. This is possible since PivPav has a database with a wide collection of the pre-synthesized hardware IP cores together with more than 90 different metrics, see [8] for details. The next two phases in the ASIP-SP cover the generation of hardware from a candidate and are also implemented with the help of the PivPav tool.

The second phase, *Netlist Generation*, generates VHDL from the candidate's bitcode and prepares an FPGA CAD project for synthesizing the candidate. The *Generate VHDL* task is performed with PivPav's data path generator. This generator iterates over the candidate's data path and translates every instruction to 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 speedup the *synthesis* and the *translation* processes during the FPGA CAD tool flow, that is, PivPav is used as a netlist cache. Finally, PivPav creates an FPGA CAD project for Xilinx ISE, sets up the parameters of the FPGA, and adds the VHDL and the netlist files.

In the third phase, *Instruction Implementation*, the previously prepared project is processed with the Xilinx FPGA CAD tool flow, resulting 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.

IV. Experimental evaluation & discussion

In Table I, we present the experimental data obtained with the developed tool flow. The scientific applications (shown in the upper part of the table) are part of the SPEC2006 and SPEC2000 and the embedded applications (shown in the lower part) are part of the SciMark2 and MiBench benchmark suites. All data were obtained from the original, unmodified sequential applications.

A. Source code, characterization and compilation

The second column of Table I denotes the number of source files processed by the compiler (llvm-gcc) to generate the bitcode. The third column shows the number of *lines of code (LOC)*. The embedded applications have $24\times$ less *LOC* than the scientific applications. The forth column shows the time needed by the LLVM compiler to generate the bitcode. These values cover also the runtime of the standard *(-O3)* optimizations. On average the embedded applications were compiled and optimized $28\times$ faster than the scientific ones, which is proportional to their size differences in terms of LOC. The following two columns represent the number of basic blocks and the number of instructions, respectively. The ratio between the average *LOC* and the average *ins* is equal almost to two for both types of applications. This indicates that for every line of source code about two bitcode instructions were generated, which means that the generated bitcode does not include many complex instructions. The *VM* column 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, we have used our own data sets. 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 *Native* 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 *Native* 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 to 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.

B. Maximum performance of the ASIP-SP

The *ASIP ratio* column describes the upper limit of performance improvement that can be achieved with the Woolcano reconfigurable ASIP architecture. These values show the hypothetical best-case in which all candidates found by the MAXMISO ISE algorithm 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 us to prune the set of candidates that are evaluated and implemented to a tractable subset. To this end, in [9] we have studied suitable pruning

App	Source Code		Compilation to Bitcode			Execution runtimes			ASIP	Code coverage			Kernel Size	
	files	LOC	real	blk	ins	VM	Native	Ratio	Ratio	live	dead	const	size	freq
	#	#	[s]	#	#	[s]	[s]	x	x	[%]	[%]	[%]	[%]	[%]
164.gzip	20	8605	3.89	1006	6925	23.71	18.47	1.28	1.17	38.86	44.66	16.48	4.52	91.05
179.art	1	1270	1.06	376	2164	69.92	74.70	0.94	1.46	42.05	28.47	29.48	5.04	91.63
183.equake	1	1513	1.71	257	2670	7.97	6.79	1.17	2.08	75.39	8.91	15.69	15.32	94.8
188.ammp	31	13483	10.10	4244	26647	23.18	17.24	1.34	3.44	19.22	70.89	9.89	3.43	95.79
429.mcf	25	2685	0.97	284	1917	23.94	24.06	1.00	1.08	75.9	13.09	11.01	20.34	94.18
433.milc	89	15042	10.88	1538	14260	20.95	16.43	1.28	1.26	61.67	34.72	3.61	10.83	93.47
444.namd	32	5315	22.77	5147	47534	39.94	34.31	1.16	1.61	31.71	62.81	5.48	7.33	93.59
458.sjeng	23	13847	8.49	3373	20531	180.41	155.66	1.16	1.13	48.49	49.44	2.07	46.22	100.0
470.lbm	6	1155	1.36	104	1988	5.68	5.36	1.06	2.61	55.23	24.9	19.87	29.38	93.12
473.astar	19	5829	3.68	757	6010	66.00	67.68	0.98	1.21	78.79	5.31	15.91	8.3	94.11
AVG-S	25	6874	6.49	1709	13064	46.17	42.07	1.14	**1.71**	52.73	34.32	12.95	15.07	94.17
adpcm	6	448	0.29	43	305	29.22	28.35	1.03	1.21	85.41	1.29	13.3	39.92	91.78
fft	3	187	0.26	47	304	18.47	18.49	1.00	2.94	60.61	24.58	14.81	45.58	97.56
sor	3	74	0.13	19	129	15.83	15.85	1.00	6.93	63.64	9.09	27.27	10.0	99.99
whetstone	1	442	0.25	44	284	28.66	28.50	1.01	17.78	34.74	26.32	38.95	9.54	93.27
AVG-E	3	288	0.23	38	255	23.04	22.80	1.01	**7.21**	61.10	15.32	23.58	26.26	95.65
RATIO	7.6	24	28	45	51	2	1.85	1.13	0.24	0.86	2.24	0.55	0.57	0.98

Table I: Experimental data obtained for the scientific and embedded applications. AVG-S represents the averages for scientific applications and AVG-E for the embedded applications. Ratio = AVG-S / AVG-E

methods. Hence, the speedup quoted in the *ASIP ratio* column should be treated only as an upper bound on the achievable performance.

For scientific applications the average maximum performance improvement over a *Native* execution is $1.5\times$, embedded applications can be accelerated by a factor of $7.13\times$. The considerable difference of $4.21\times$ suggests that the MAXMISO ISE algorithm could find better candidates in the smaller embedded applications or that the embedded applications have more pronounced kernels. Still, the reconfigurable ASIP architecture is considerably faster than the underlying CPU alone for both benchmark domains, hence 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.

C. Code Coverage & Kernel Size

The *Code Coverage* columns show the relative percentages of the size of *live*, *dead* and *constant* code. We have determined these values by executing each application for different input data sets and recording the execution frequency of each basic block *(blk)*. After execution, we compare the change in execution frequency per block between the different runs. If the frequency is equal to 0 the code is marked as *dead*. If the frequency is different from 0 but did not change for different inputs the code is marked as *constant* and if the frequency has changed, the block is marked as *live*. The frequency information for each *blk* is used to compute the speedup of the application.

The last two columns present data on the size of the kernel of the application. This data is also derived from the frequency data. We define the kernel of an application as the code that is responsible for more than 90% of the execution time. For determining the kernel size we sort the basic blocks by their total execution time. Then we select as many basic blocks as required (in the order of execution time) until the threshold of 90% is reached. The size of the kernel is measured as the total number of instructions contained in these basic blocks. For scientific applications, 15% of the code causes 94% of the total application execution time. For embedded applications the average relative kernel size is almost twice as high (26%), which is one of the causes why higher speedups can be achieved for these applications. The average size of the kernel is 1960 instructions for scientific and 67 instructions for embedded applications. These numbers also support conventional wisdom that applications spend about 90% of the time in about 10% of the code, which is also known as the Pareto Principle.

V. Runtime costs of the ASIP-SP process

As elaborated in Section III the achievable application speedup, which we have discussed in the previous section, and the execution time of the tool flow determine for how long the application needs to be executed until the hardware generation overhead is amortized, that is, a net speedup is achieved. In this section, we will analyze the runtime of the three different phases of ASIP specialization that have been introduced in Figure 2.

A. Candidate Search

As described in Section III, 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 [9] to reduce the search space for instruction candidates. The number of selected candidates, after pruning, is indicated in the *can* column of Table II.

The third column of Table II 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

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

Table II: The runtime overheads for the ASIP-SP process.

@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 *5th* and *6th* column Table I. That is, the pruning mechanism reduced the size of the bitcode that needs to be analyzed in the identification task by a factor of $36.49\times$ and $4.9\times$ 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.

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 10th column in Table I which assumes that *all* candidates are moved to hardware, the average speedup drops by 45% from $1.71\times$–$1.20\times$ for scientific applications and by 44% from $7.21\times$–$4.98\times$ for the embedded ones. What is also interesting is that the ratio of 0.24 between the scientific and embedded applications is the same for both cases, i. e., with enabled and disabled pruning mechanisms. Comparing the *fft* with the *470.lbm* applications illustrates the main difference between embedded and scientific applications. Both applications have a similar speedup of $2.40\times$ vs. $2.53\times$, respectively, but differ significantly in the number of candidates that need to be translated to hardware to achieve this speedups (14 vs. 179 candidates). This correlates with the previously described observation that scientific applications have a significantly larger kernel size.

B. 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 data path generator which produces the *structural VHDL* code. The data path generator traverses the data path 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* tasks retrieves the netlists files for each hardware component used in the candidates VHDL description from the PivPav database. This step allows to reduce 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 consuming task of the netlist generation phase. The average total runtime for these three tasks is presented in the *C2V* column of Table III and amounts to $3.22\,s$. As the standard deviation is only 0.10 this time can be considered as constant.

C. 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 III. 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

	C2V	Syn	Xst	Tra	Bitgen	Sum
	[s]	[s]	[s]	[s]	[s]	[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	

Table III: Constant overheads involved in the ASIP-SP process. *C2V* corresponds to the *Netlist Generation* phase in Figure 1. Syn, Xst, Tra, and Bitgen are the FPGA CAD tool flow processes and correspond to the syntax check, synthesis, translate, partial reconfiguration processes, respectively, which can be found in the third phase in Figure 1.

re-synthesize 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 on average for $8.99\,s$.

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 $1.4\times$ for small candidates to $2.5\times$ 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 III, 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 II. The column *const* represents the runtime of constant processes shown in Table III. The column *map* stands for the mapping process, the column *par* for the *place and route*, and 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. On average it takes less than 50 minutes (49:53m) to generate all candidates for the embedded applications but more than 4:30 hours (270:28m) 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 (678:13m) and 17 hours (1021:22m), respectively and is caused primarily by the high constant time overheads (*const*).

This observation emphasizes the importance of the pruning algorithms, in particular 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 146:34m to 24:28m that is by a factor of $5.99\times$, which is exactly the difference in the number of candidates *(can)* between the scientific and the embedded applications.

D. 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 process are compensated.

A simplistic way of computing the break even time would be to divide the total runtime overhead (*sum* in Table II) by the time saved during one execution of the application, which can be computed using the *VM execution time* and the *ASIP ratio* (speedup) (see Table I). 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 IV-C). The resulting break even times are presented in the last column of Table II.

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 *(>4:30h)* and modest performance gains of $1.2\times$. As described above, the long runtimes are caused 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 LLVM 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 1960 LLVM 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 performances gains with modest runtime overheads. For an average embedded application a $5\times$ 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 LLVM instructions in the selected candidates. Scientific applications have on average 7.31 instructions per candidate, 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 code blocks *(blk)* in which they are identified are also small. The average basic block has only 7.64 (6.71) LLVM instructions for a scientific (embedded) application (see Table I).

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 II). 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 specialization process when using code that has been compiled from imperative languages.

	Faster FPGA CAD tool flow[%]			
Cache	0	30	60	90
hit [%]	[h:m:s]	[h:m:s]	[h:m:s]	[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

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

VI. 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.

A. 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 instructions. 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.

B. 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 smaller device would definitively 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 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 [17].

C. Extrapolation

In Table IV 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 II. One can note also that these values do not scale linearly because we consider the frequency information for basic blocks.

For this evaluation, we varied the assumed cache hit rate between 0%–90%. That is, for simulating a cache with 20% hit rate, we have populated the cache with 20% of the required bitstreams for a particular application, whereas the selection which bitstreams are stored in the cache is random. Whenever there is a *hit* in the cache for a given candidate then the whole runtime associated with the generation of the candidate is subtracted from the total runtime, see *sum* column in Table II. 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 a half ($1.94\times$) from *1:59:55h* to *1:01:42h*. This means that the whole runtime of the ASIP-SP could be compensated in a bit more than a one hour and for the rest of the time the adapted architecture would provide a performance gain by an average factor of $5\times$. These assumptions are modest values since the *Cache hit* rate depends only on the size of the cache and our Dell T3500 workstation (see [9, Section V.B]) could be easily replaced by a faster one.

VII. Conclusion and Future Work

In this work we have studied the feasibility and limitations of just-in-time ISE process for an FPGA-based reconfigurable ASIP architecture. We have presented a detailed study comparing the achievable performance and the properties of custom instruction candidates for embedded computing and SPEC benchmarks. The study has shown that for embedded applications an average speedup of $5\times$ 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 data path synthesis tools.

References

[1] R. Leupers and P. Ienne, Eds., *Customizable Embedded Processors: Design Technologies and Applications*, ser. Systems on Silicon. Morgan Kaufmann Publishers, Aug. 2006.

[2] M. J. Wirthlin and B. L. Hutchings, "A dynamic instruction set computer," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, Apr. 1995, pp. 99–107.

[3] R. Razdan and M. D. Smith, "A high-performance microarchitecture with hardware-programmable functional units," in *Proc. Int. Symp. on Microarchitecture (MICRO)*. New York, NY, USA: ACM, 1994, pp. 172–180.

[4] Z. A. Ye, A. Moshovos, S. Hauck, and P. Banerjee, "CHIMAERA: A high-performance architecture with a tightly-coupled reconfigurable functional unit," in *Proc. Int. Symp. on Computer Architecture (ISCA)*, 2000, pp. 225–235.

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

[6] M. Grad and C. Plessl, "Woolcano: An architecture and tool flow for dynamic instruction set extension on Xilinx Virtex-4 FX," in *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. CSREA Press, Jul. 2009.

[7] J. M. Arnold, "S5: the architecture and development flow of a software configurable processor," in *Int. Conf. on Field Programmable Technology (ICFPT)*, Dec. 2005, pp. 121–128.

[8] M. Grad and C. Plessl, "An open source circuit library with benchmarking facilities," in *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. CSREA Press, Jul. 2010.

[9] ——, "Pruning the design space for Just-In-Time processor customization," in *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE Computer Society, Dec. 2010.

[10] M. Wazlowski, L. Agarwal, T. Lee, A. Smith, E. Lam, P. M. Athanas, H. F. Silverman, and S. Ghosh, "PRISM-II compiler and architecture," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, Apr. 1993, pp. 9–16.

[11] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," in *Proc. Int. Conf. on Architecture of Computing Systems (ARCS)*, ser. LNCS, no. 4943. Springer, 2008, pp. 209–220.

[12] 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.

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

[14] K. Ebciolu and E. Altman, "Daisy: Dynamic compilation for 100 percent architectural compatibility," in *Proc. Int. Symp. on Computer Architecture (ISCA)*. New York: ACM, 1997, pp. 26–37.

[15] F. Vahid, G. Stitt, and R. Lysecky, "Warp processing: Dynamic translation of binaries to fpga circuits," *Computer*, vol. 41, pp. 40–46, July 2008.

[16] A. C. S. Beck and L. Carro, "Dynamic reconfiguration with binary translation: breaking the ILP barrier with software compatibility," in *Proc. Design Automation Conference (DAC)*. New York, NY, USA: ACM, 2005, pp. 732–737.

[17] 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*, ser. CC'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 178–192.