Pruning the Design Space for Just-In-Time Processor Customization

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 instruction set specialization for reconfigurable ASIPs at runtime. Applying known instruction set extension algorithms for static ASIPs in this just-in-time CPU specialization context is generally possible. 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. Hence, we propose effective ways of pruning the design space which can reduce the runtime of instruction set extension algorithms by two orders of magnitude. We evaluate the proposed methods and our tool flow targeting the Woolcano reconfigurable ASIP architecture with a comprehensive set of applications from the SPEC2006, SciMark2, and MiBench benchmark suites. For some applications we show a 44-fold speedup over a fixed CPU architecture. Finally, we elaborate why linear complexity instruction set extension algorithms are most suitable for just-in-time ASIP specialization.**

I. Introduction

The last decade has witnessed an emergence of Application Specific Instruction Set Processors (ASIPs). ASIPs allow for increasing the performance and energy efficiency by specializing the instruction set of a processor according to the demands of a given application. For achieving the best overall performance the specialization process needs to identify the application parts that are most suitable for acceleration by considering which parts can translated to custom instructions, what degree of acceleration can be achieved and how frequently these parts are executed. Further, architectural constraints of the ASIP architecture need to be satisfied, such as the number of read and write ports of the register file. The complexity of these decisions demands for automating the identification of suitable custom instructions with automated instruction set extension (ISE) algorithms. Unfortunately, the algorithmic complexity of most ISE methods is exponential in the number of instructions of an application which frequently results in runtimes of several hours for complex applications.

In our work we are not targeting static but reconfigurable ASIP architectures, such as our Woolcano architecture [1] or comparable architectures like CHIMAERA or PRISC. These architectures provide programmable functional units that can be dynamically reconfigured during runtime to implement arbitrary custom instructions. Our objective is study the feasibility of just-in-time processor customization, that is to automatically adapt the processor's instruction set during runtime depending on the executed workload. For this scenario, current ISE methods are not suitable due to excessive runtimes.

Our paper makes the following contributions: We introduce and compare different approaches for significantly reducing the runtime for three state-of-the-art ISE algorithms to make them accessible to just-in-time processor customization while maintaining a high quality of the solution (application-level speedup) at the same time. For this purpose we propose new methods for effectively reducing the design space for ISE by exploiting information gathered by static code analysis, profiling and performance estimation. To validate the effectiveness of the design space reduction and its impact on the overall application performance, we evaluate our methods using a realistic model of our FPGA-based Woolcano [1] architecture for a large set of complete applications from the SPEC2006, MiBench, and SciMark2 benchmark suites. These applications cover the whole range from embedded computing to general-purpose computing, illustrating that our methods are widely applicable.

II. Related work

Although instruction set extension (ISE) has been extensively studied, see [2] for a recent survey, the issue of reducing the design space for ISE has not received any attention so far, with the exception of the *guide functions* [2]. Unfortunately these are not well suited for the online systems, which is not the case for our study.

One reason for this may be that the runtime of the ISE algorithms for static non-reconfigurable ASIPs is of less importance than for just-in-time processor specialization which is novel research area in itself. For this work, we have selected three state-of-the-art ISE methods which we consider as the most suitable for runtime processor specialization: the *Single-Cut* algorithm [3] by Atasu et al., the *Union* algorithm [4] by Yu and Mitra, and the *MaxMiso* [5] algorithm by Alippi et al. A common shortcoming of these works is their focus on accelerating a single or few basic blocks instead of complete applications and the lack of a discussion how to integrate the methods into an end-to-end tool flow. While ad-hoc heuristics have been proposed to select the basic blocks subject to custom instruction identification, e. g., all hot loops with an execution time of more than 1% of the total runtime [6] or just the most frequently executed basic block [4], no systematic study of approaches for pruning the design space has been conducted so far. However, when applying ISE in a just-in-time scenario and targeting large real-world applications with thousands of basic blocks finding effective approaches for reducing the design

Published in: *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*
 DOI: 10.1109/ReConFig.2010.19

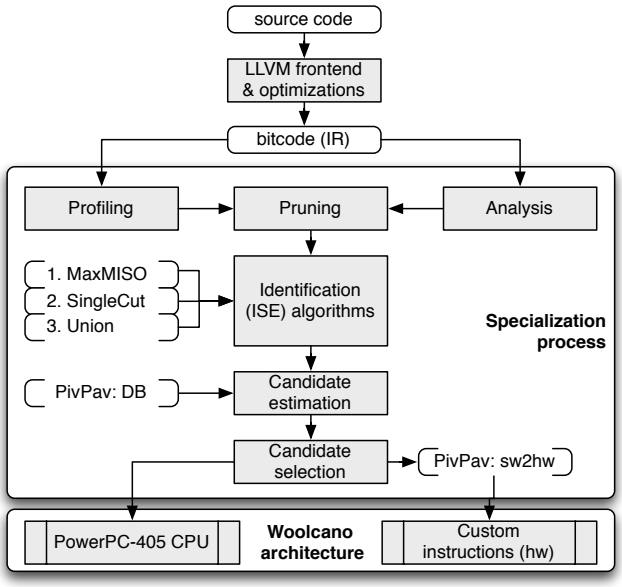

Figure 1: Overview of the tool flow

space and in turn the runtime of ISE methods is crucial to prevent excessive runtimes.

III. Background & Developed Tool Flow

Figure 1 presents an overview of the tool flow that we have developed to compare different approaches for pruning the design space for ISE. The tool flow is based on the LLVM [7] compiler framework. As the first step, the application source code is compiled using the LLVM frontend to an intermediate representation (LLVM bitcode). The intermediate code is used as the input to the specialization process that has been implemented for the purpose of this paper. In the specialization process ISE algorithms analyze the applications to identify all feasible candidates for custom hardware instructions. Then, each candidate is passed to the estimation process which computes the runtime when translating the candidate to a custom instruction (hardware) or executing it on the CPU (software). Further, estimates for the area and energy consumption are computed. Finally, based on this data the selection process chooses the best custom instruction candidates.

A. Candidate Identification

For the identification process we have implemented three state-of-the-art ISE algorithms in the tool flow: MaxMiso (MM) [5], SingleCut (SC) [3], and Union (UN) [4]. We compare the most relevant properties of this algorithms for the purpose of this work in Table I. All of these identification algorithms do not try to find custom instruction candidates in the whole application at once. Instead, they focus on the individual basic blocks (BBs) of the application. Hence we can run the ISE algorithm selectively only for the most promising BBs in order to prune the design space (cf. Sec. IV). Each ISE algorithm analyzes the basic block to identify all feasible custom instructions, which means that the custom instructions are only allowed to use a subset of operations that can be

Algorithm	# of inputs	# of outputs	algorithm complexity	overlapping candidates
MaxMiso	fixed (unlimited)	fixed (1)	O(n)	no
SingleCut	variable	variable	O(exp)	yes
Union	variable	variable	O(exp)	yes

Table I: ISE algorithms comparison

effectively translated to hardware. Further, some algorithms allow to further constrain the number of inputs and outputs of custom instructions to match the architectural constraints of the targeted ASIP architecture.

The algorithmic complexity of the SC and UN algorithms is exponential $O(exp)$ in the size of a BB. The advantage of the MM algorithm is its linear $O(n)$ complexity. However, it finds only candidates which have a single output and does not allow to constrain number of inputs. Hence some custom instruction candidates that are generated need to be discarded later at additional cost in the selection phase because they cannot be implemented. By contrast the SC and UN algorithms allow for restricting the desired number of inputs and outputs for candidates already during the identification phase. Finally, only the MM algorithm produces non-overlapping candidates. The SC and UN algorithms need an additional phase in the selection process to eliminate them.

B. Candidate Estimation

After the set of custom instruction candidates has been identified, the best candidates are selected. The selection process is based on two factors. First, the candidates must satisfy the architectural constraints of the execution architecture (in this work the Woolcano architecture [1]). Second, the performance of every candidate needs to be estimated. Hence this phase is critically important for determining the final performance.

The first step of the estimation is to translate each operation in the candidate to a corresponding hardware implementation. Every operator is characterized by a number of metrics, such as the delay, hardware area and energy consumption which is obtained from the PivPav tool [8], which we have developed specifically for this purpose. Using the characterization data, the critical path for each candidate is determined which is use to calculate the runtime of the candidate in hardware. The runtime in software is estimated by adding the instruction latencies for all operations of the custom instruction candidate. Based on these performance estimates the subsequent selection process can decide if it is affordable and beneficial to implement a candidate as a hardware custom instruction.

The key to accurate hardware estimation is the quality of the characterization database that provides performance and area for each operator. Related work [3], [4], [9] has used characterization data for fixed 0.18 or $0.13\mu m$ CMOS architectures which has been obtained by synthesizing the operators using the Synopsys Design Compiler targeting standard-cells libraries. Since this work targets the FPGA-based Woolcano reconfigurable ASIP architecture we could not use the same approach. Instead we have created the PivPav tool that not only provides a characterization database for operators but can also

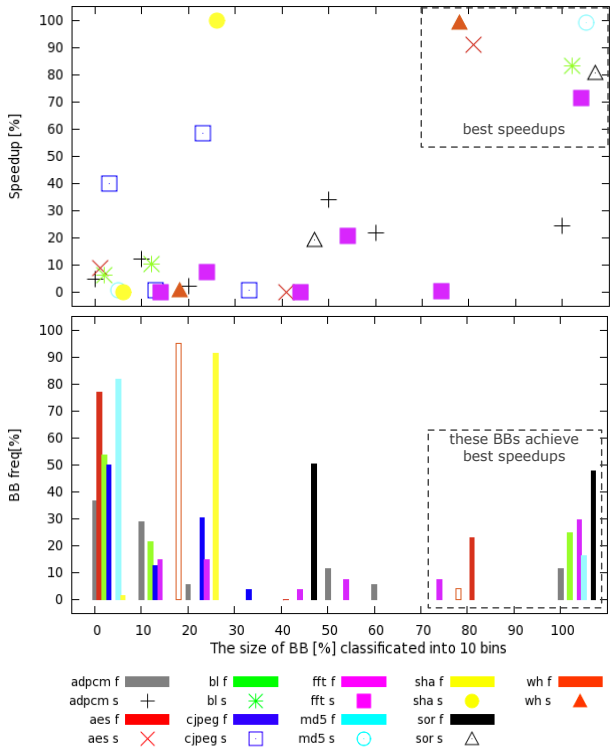

Figure 2: Relative relationship between BB sizes to application performance gain and to BB frequency for MaxMISO ISE algorithm. Absolute values are presented in Table III. s = speedup, f = frequency.

perform design space exploration and can generate the actual FPGA implementation for the custom instructions.

C. Candidate Selection

Once the set of candidates has been determined and characterized the selection process makes the final decision which candidates should be implemented in hardware. Selecting the optimal custom instructions under different architectural constrains is a demanding task. Related work has studied different selection approaches, such as greedy heuristics, simulated annealing, ILP, evolutionary algorithms, see for example [10] or [9]. For the purpose of our study, we have developed a locally optimal greedy heuristic which, in the case of overlapping candidates, tries to merge as many candidates in a single BB as possible.

IV. Pruning the Design Space

As outlined in Figure 1 the specialization process consists of a pruning process which is of major importance to this paper. The pruning process bases its decisions on data obtained from profiling and program analysis. Profiling determines execution frequency histograms for all BBs. Bitcode analysis provides information about the size of the BBs and the contained instruction types. Figure 2 illustrates the mutual dependencies of BB size, execution frequency and achievable speedup for 9 benchmarks. For each application all BBs that can be accelerated with custom instructions are sorted into 10 bins depending on their absolute sizes (measured in instructions) in a histogram-like manner. The first bin contains all BBs that have a size of 0–10% of maximal basic block size for this application, the second bin collects all BBs that have a size of 10–20% of the maximum BB size, etc. For each application and bin the total execution frequencies for all BBs assigned to each bin are shown in the lower figure. The upper figure shows the overall application-level speedup when translating the candidates in these basic blocks to custom instructions. The figure shows that the largest application speedups result from BBs which are located in the upper right corner of the Figure. The objective of our proposed pruning methodologies is to identify these highly profitable BBs and to ignore the insignificant rest of BBs.

A. Pruning Objective

As visualized in Figure 2, the best speedups (70–100%) are achieved for basic blocks in the 80–100% size bins, whereas the most frequently used BBs (up to 90%) are located in 0–30% size bins. This observation suggests that it is more significant to identify candidates in frequently used large BBs than in the most frequently used but smaller BBs. In large BBs, ISE algorithms are able to find more and larger candidates which overall result in better speedups, than accelerating more frequently executed but smaller BBs. In other words, execution frequency matters but among the most frequently executed BBs the largest BBs are most important for achieving high overall speedups.

The `aes` application is an ideal example of such a behavior. There are two red bars representing the frequencies of BBs for `aes`, the first is located at 0%, the second at 80%, both with distinctively high frequencies. However, the major speedup (91%) is achieved when accelerating the BBs represented by the second bar which are larger but less frequently executed than the first one. Another interesting application is `sor`. There are two black bars at 40% and 100%, respectively, with almost the same frequency. The sum of the bars is equal to 100% frequency, which means that we are investigating all executed BBs in the application. Once again the second bar which represents bigger BBs has a major speedup of more than 80%.

B. Pruning Algorithms

In this work we have studied a whole set of different pruning algorithms which are presented in Table II. All algorithms but `n(F/S)` try to leverage the fact stated above that the highest speedups are achieved for large and frequently executed BBs. The easiest way to restrict the set of BBs to this subset is to constrain the `x` (sizes) and `y` (frequencies) axis which corresponds to specifying the `n` and `m` parameter in the `mFnS` and `mSnF` algorithms. The remaining algorithms with the exceptions described below try to mimic the same goal, but without using the frequency data. This is particularly important for a JIT processor customization scenario in which the execution frequencies for BBs are not known. They can be collected during the execution but will remain incomplete or inaccurate until the application terminates. Hence during the

Name	Algorithm	Description
nS	n-size	select n largest BB
nF	n-freq	select n most frequently used ones
nL	n-loop	select n largest loop BBs
@nL	n-loop(@)	eliminate not used BBs and do nL
mLn*	n-(m-loop(uniq F))	do nL for every function and then mL
mSnL	n-loop(m-size)	do mS and them nL
@mSnL	n-loop(m-size(@))	eliminate not used BBs and do mSnL
mFnS	n-size(m-freq)	do mF and then nS
mSnF	n-freq(m-size)	do mS and then nF
tF	threshold = max(F)	select BBs with this threshold
tS	threshold = max(S)	select BBs with this threshold
n(F/S)	n-(freq / size)	select n BBs with largest ratio

Table II: Description of pruning algorithms. The n and m letters represent constraining numbers, the p suffix, if added to these, represents percentage of the range. The @ tells that algorithm used JIT to indicated BBs which were executed at least once, S = Size, F = Frequency and L indicates Loops.

runtime only partial information is available which makes the goal of JIT processor specialization challenging.

In order to find the frequently used BBs without profiling, we have developed a program analysis pass which examines the control flow structures in the application and indicates which BBs are contained in loops. This technique is used in the L algorithms. The chances of executing such BBs are higher than the others and therefore it is one indication that these BBs are frequently executed. A loop can consist of many BBs and the nL algorithm selects only the largest BBs. Whenever possible our analysis pass tries to skip the `preheader` and `latch` parts of a loop and focuses only on the computational BBs in the loop bodies. This technique can also leverage applications which use the common software design patterns to encapsulate every significant part of the code in a separate function. For such applications, the nLm* algorithm will try to find m loop-BBs in every function and will finally select the n largest ones from this set.

The limitation of the nL algorithm is that it can consider loop-BBs which are not used at all during execution. For instance, such loop-BBs exists in the `decrypt` and `encrypt` functions of the `aes` application. During the execution of the application only one function is used, which can easily mislead the nL algorithm. To avoid such a case, the @nL algorithm was developed. It considers only BBs in functions that have been executed at least once which is an information that can be made available in JIT systems. The difference between the tF to the nF algorithm, where $n = 1$ is that several BBs may have an identical execution frequency. For such cases the first algorithm will consider all of them whereas the second will only consider the first BB. The same applies to the tS and the nS algorithms.

Finally, the algorithm n(F/S) was implement as a reference to test our hypothesis that the design space should be constrained to the largest and most frequently executed BBs. This algorithm favors the frequent but small BBs and hence should result in lower speedups if our hypothesis is correct.

V. Experimental Evaluation & Discussion

Table III shows the experimental data obtained by executing our tool flow for each application. The number of instructions and BBs found in each application are shown in the second and third column, respectively. The upper part of the table shows data for applications representing the embedded computing domain. The lower part of the table presents data from SPEC2006 applications representing the general-purpose and scientific computing domain which in average have 8 (in median 30) times more instructions.

A. ISE Algorithm Runtimes

The fourth column shows the average runtime for each ISE algorithm executing on a Dell T3500 Workstation (dual core Intel Xeon W3503 2.40GHz, 4M L3, 4.8GT/s, 12GB DDR3 / 1333MHz RAM). As stated before, the MM algorithm has linear complexity and therefore is the fastest resulting in 0.9s runtime for `445.gombk` which is the largest application. The runtime of UN should generally outperform the runtime of SC but this is not the case for applications which include specific types of BBs. For example, it took 3837.0ms to process such a specific BB consisting of 55 nodes in the `adpcm` application which is 99.15% of overall runtime of UN algorithm. In contrast, the same BB was analyzed by SC algorithm just in 4.7ms.

Since SC and UN have both exponential complexity their runtimes are a few orders of magnitudes higher than for MM. In average, the runtime of MM is 180 times faster than the SC algorithm. In addition, due to the exponential complexity of algorithms and constraining limitations (e. g., the maximum number of inputs and outputs, infeasible operations, etc.), the identification times vary significantly even for the same algorithm and applications of similar size. For example, the runtimes for executing SC for `401.bzip2` and `433.milc` differ by 32%. Most of the 2858.6ms for analyzing `401.bzip2` was spent to analyze a single BB with 1221 nodes whereas for `433.milc` 1707ms were spent for analyzing one BB of just 119 nodes. This observation illustrates that it is not possible to accurately estimate the runtime of the exponential ISE algorithms in advance based on the size of a BB only.

We like to point out that for keeping the design space and thus the algorithm runtimes manageable we have used rather tight architectural constraints for the custom instructions (4 inputs, 1 output) in our comparison of ISE algorithm in Table III. When loosening these constraints, the execution times for the SC and UN algorithms rapidly grow from seconds to many hours.

B. Application Speedups

The last columns in Table III indicate the overall application-level speedup when using the three different ISE algorithms. These results show the potential of a reconfigurable ASIP architecture for a broad range of applications. The developed tool flow has much to offer in this respect, reaching a speedup of up to $44.6\times$ for the SC algorithm and $18.01\times$ for the MM and UN. The average speedups

App	# Instr	# BB	ISE algorithm runtime [ms]			Saved Clock Cycles [10^6]			Speedup		
ISE alg.			MM	SC	UN	MM	SC	UN	MM	SC	UN
adpcm	305	43	1.7	15.0	3869.4	20945.6	26718.0	25753.3	1.24	1.31	1.29
aes	8972	200	40.6	31624.9	62776.6	0.2	0.1	0.1	1.64	1.32	1.32
blowfish	1746	59	8.0	2495.7	1147.3	0.7	0.7	0.7	1.43	1.43	1.43
cjpeg	19468	2483	115.3	1522.8	6191.6	9.7	16.1	16.1	2.09	7.51	7.51
fft	304	47	1.6	9.7	33.1	6.1	8.4	8.4	3.10	14.41	14.41
md5	918	29	3.8	4400.8	20353.4	0.3	0.3	0.3	2.09	1.76	1.76
sha	471	40	2.6	37.8	1314.3	4.4	4.0	3.8	1.41	1.36	1.33
sor	129	19	0.7	4.3	14.6	5.3	5.3	5.3	14.42	14.42	14.42
whetstone	284	44	1.6	9.5	64.0	231.9	231.9	231.9	**18.01**	18.01	**18.01**
401.bzip2	14102	1604	72.3	3839.3	1979.3	0.4	28912.9	28912.9	1.18	1.13	1.13
429.mcf	1917	284	11.1	68.7	200.5	826.5	826.8	826.8	1.11	1.11	1.11
433.milc	14260	1538	78.1	5065.6	-	232062.4	957390.9	-	1.30	21.55	-
444.namd	47534	5147	227.5	35853.6	-	13426391.2	34036379.8	-	1.61	24.85	-
445.gobmk	135500	27469	906.3	5797.5	93029.4	2.1	2.1	2.1	1.11	1.11	1.11
456.hmmer	29337	5518	178.3	76436.2	105699875.0	54745.8	64889.6	64889.5	2.46	3.38	3.38
458.sjeng	20531	3373	123.7	6244.1	235195.7	2651.0	3023.4	3023.4	1.12	1.14	1.14
462.libquantum	5327	1047	32.5	140.2	1167.1	5619.8	5864.6	5864.6	1.27	1.28	1.28
470.lbm	1988	104	8.6	2777.1	0.0	214657.9	344885.1	0.0	2.55	**44.62**	1.00
473.astar	6010	757	33.4	914.8	303796653.0	5792.6	6767.1	6767.0	1.16	1.19	1.19

Table III: 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 & UN search is constrained to 4 inputs and 1 input.

achieved with MM, SC, and UN are $3.17\times$, $8.57\times$, $3.94\times$ or in median $1.43\times$, $1.43\times$, $1.29\times$, 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, we need to balance the achievable speedup with the identification time. Comparing the ratios of average speedup to identification runtime results in the following ratios: 1.92 (MM), 0.36 (SC), and 0.10 (UN). These figures suggest that the MM algorithm is the most suitable for such 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, often by more than an order of magnitude. But at the same time a fixed architecture will sacrifice the flexibility and runtime customization capabilities of the Woolcano architecture.

C. Comparison of Pruning Algorithms

The results of the pruning algorithms are summarized in Table IV. They are grouped by embedded and SPEC2006 applications, since their sizes differ significantly. Hence the chances for finding more pronounced computation kernels is higher for the embedded benchmarks than for SPEC2006 applications. These values represent the ratio of the average speedup to the required identification times for the whole set of applications. Higher ratios means that a given algorithm achieves a better speedup to identification time ratio which is preferable for systems with just-in-time CPU specialization that need balance both metrics.

As presented in Sec. V-B, the advantage of MM (36.69, 7.50) over SC (20.76, 1.48) and UN (6.95, 2.20) is clear, especially for the smaller embedded applications, where an average speedup of $3.71\times$ is achieved in 1.82ms in total. This is a $8.86\times$ improvement over the case where `no-pruning` is used and $4.16\times$ better as the `3(F/S)` algorithm. For the latter case the average speedup is $5.05\times$ and $2.06\times$, respectively, with an identification time of 175.75ms and 19.74ms.

The identification runtime was shortened for `@50pS3L` over `no-pruning` almost by *two orders of magnitude* ($96.73\times$) which caused the relative speedup to decrease only to 73.47%. This result clearly indicates the importance of the pruning algorithms in any specialization process. The correctness of our assumptions about the location of the best speedups in applications (`@50pS3L` vs `3(F/S)`) is supported by the fact that the identification time is shortened by $10.86\times$ while the average speedup is higher by $1.8\times$.

The `mFnS` and `mSnF` algorithms, as assumed in IV, can easily locate the most profitable BBs in applications. The results for SC and UN are better when constraining the sizes first and the frequencies afterwards, that is `2S1F` instead of `2F1S`. In addition, the ratio of speedup to identification for these algorithms is maximal when targeting only a single BB with parameters $\texttt{m} = 1$ and $\texttt{n} = 2$ due to their exponential complexity. This is not the case for the linear complexity (MM) algorithm, which achieves the best results for cases with more than one BB, that is $\texttt{n} > 1$. The capability to process BBs faster is the reason why MM outperforms SC and UN when using the speedup over identification time metric.

Generally, the best ratios are obtained when applying the ISE algorithms only to a small number of BBs. This is problematic for the loop (`L`) algorithms which focuses on loop-BBs because the amount of loop-BBs in an application is usually very high. Without an execution trace it is difficult to accurately identify the most frequently executed loop-BBs. In

Algorithm	embedded applications			SPEC2006 applications		
	MM	SC	UN	MM	SC	UN
1L	7.61	2.84	0.41	0.22	0.21	0.16
2L	9.59	1.81	0.66	0.18	0.29	0.10
3L	15.68	1.17	1.03	0.08	0.28	0.05
4L	12.70	1.00	1.01	0.07	0.27	0.05
@1L	9.06	4.31	0.53	0.82	0.25	0.53
@2L	14.99	4.17	1.07	0.82	0.45	0.53
@3L	22.90	2.11	1.51	0.32	0.41	0.20
@4L	19.95	2.11	1.51	0.25	0.37	0.14
50pS1L	9.45	1.75	0.65	0.03	0.30	0.08
50pS2L	13.87	1.81	1.00	0.08	-	-
50pS3L	28.58	1.74	1.87	0.08	-	-
@50pS1L	8.92	0.67	0.61	0.00	0.24	0.45
@50pS2L	17.70	0.39	1.24	0.00	0.54	0.49
@50pS3L	**36.69**	0.72	2.34	0.01	0.62	0.19
1L2*	7.13	2.55	0.37	0.09	0.19	0.11
3L2*	10.09	1.00	0.66	0.03	0.18	0.03
1L3*	7.13	2.55	0.37	0.09	0.19	0.11
2F1S	14.99	19.80	2.04	6.89	0.68	1.86
3F1S	13.86	17.58	1.65	5.16	0.57	1.47
6F1S	20.28	12.86	1.21	3.08	0.33	0.79
6F2S	22.43	7.11	1.92	1.58	0.65	0.41
8F1S	19.39	10.84	1.18	2.40	0.30	1.18
2S1F	22.44	**20.76**	**6.95**	**7.50**	**1.48**	**2.20**
4S2F	18.86	11.62	2.27	4.22	0.87	1.25
5S3F	13.98	8.02	1.79	2.34	0.68	0.75
6S4F	11.41	5.84	1.53	1.50	0.56	0.47
tF	12.12	17.52	2.80	6.15	0.69	2.11
tS	6.02	1.67	0.38	0.05	0.19	0.08
1(F/S)	7.30	6.09	0.88	2.01	0.29	0.48
2(F/S)	9.88	3.47	0.82	0.89	0.30	0.29
3(F/S)	8.82	2.74	0.73	0.51	0.27	0.19
1S	13.01	11.63	1.21	3.07	0.40	0.92
2S	15.62	8.63	2.09	2.53	0.67	0.75
3S	13.74	7.25	2.04	2.13	0.63	0.65
1F	results equal to 2S1F					
2F	18.47	12.43	2.24	4.49	0.85	1.31
3F	14.34	9.20	1.80	2.67	0.68	0.84
no-pruning	4.14	0.76	0.19	0.05	0.00	0.00

Table IV: Ratio of speedup to identification time for the pruning algorithms (Table II). The `no-pruning` row corresponds to Table III.

particular for the exponential SC and UN algorithms it would be important to focus on few or even a single BB to maintain a high speedup to identification time ratio.

The `L` results for the SC and UN are not as good as for other algorithms which make use of the frequency data (`nSmF`, `nFmS`), but they are still satisfactory. In particular for the MM algorithm which allows for targeting more than one loop-BB while keeping a good speedup to identification time ration the results are in the top range for the smaller embedded benchmarks where fewer loop-BBs exist.

Hence, the `L` algorithms work best with linear complexity algorithms, where the inaccuracy of identifying loop-BBs is overcompensated by the fast runtimes. In particular this is the case for the `@` loop algorithms which show higher speedup to identification time ratios than the `L` algorithms that do not exclude BBs which are never executed.

The results for the `nLm*` algorithm are mediocre. Perhaps the applications do not have the necessary modular design or it was destroyed during the compiler optimizations.

It is interesting to see how well the `nF` and the `nS` algorithms work. The `nF` algorithm eliminates not executed BBs, which is a clear advantage over the `nS` algorithm, especially when $n = 1$. For $n > 1$ the difference is disappearing, since `nS` has a higher chance to identify used BBs. However, for such cases the `@npSmL` algorithm should be used as it performs even better.

The `mSnF` algorithm on average provides better results than the `S` and `F` separately and should be used therefore. Otherwise, `nF` should be chosen over the `nS` algorithm.

VI. Conclusions

In this work we have studied approaches for pruning the design space for a just-in-time ASIP specialization process. Additionally, we compared the suitability of three state-of-the art ISE algorithms for just-in-time processor specialization. We have shown that for JIT systems where the reducing the runtime for identification is as important as speedup, linear complexity ISE algorithms deliver the best performance. Simultaneously, this work has explored the potential of the Woolcano reconfigurable ASIP architecture. We have described and evaluated our tool-flow with a broad range of complete applications from embedded and general-purpose computing and have shown that a speedup up to $44\times$ for a SPEC2006 application is feasible.

References

[1] 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)*. USA: CSREA Press, Jul 2009, pp. 319–322.

[2] 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.

[3] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *DAC '03: Proceedings of the 40th annual Design Automation Conference*. New York, NY, USA: ACM, 2003, pp. 256–261.

[4] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction-set extensible processors," in *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. New York, NY, USA: ACM, 2004, pp. 69–78.

[5] C. Alippi, W. Fornaciari, L. Pozzi, and M. Sami, "A DAG-based design approach for reconfigurable vliw processors," in *Proc. Design, Automation and Test in Europe Conf. (DATE)*. New York, NY, USA: ACM, 1999, p. 57.

[6] H. P. Huynh, J. E. Sim, and T. Mitra, "An efficient framework for dynamic reconfiguration of instruction-set customization," in *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. New York, NY, USA: ACM, 2007, pp. 135–144.

[7] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. 2004 Int. Symp. on Code Generation and Optimization (CGO)*. IEEE Computer Society, Mar 2004, pp. 75–86.

[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)*. USA: CSREA Press, Jul. 2010, pp. 319–322.

[9] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1209–1029, Jul 2006.

[10] R. Meeuws, Y. Yankova, K. Bertels, G. Gaydadjiev, S. Vassiliadis, A. Heterogeneous, and S. Development, "A quantitative prediction model for hardware/software partitioning," in *Proc. 17th Int. Workshop on Field Programmable Logic and Applications (FPL)*, 2007, pp. 735–739.