Towards Domain-Specific Instruction-Set Generation

Adithya Pulli[♮], Carlo Galuzzi[♮], Georgi Gaydadjiev[♯]
[♮] Delft University of Technology (NL) [♯] Chalmers University of Technology (SE)
c.galuzzi@tudelft.nl, georgig@chalmers.se

***Abstract*—Over the past years, a considerable amount of effort has been devoted to the definition and implementation of techniques for the optimization and acceleration of applications on various (reconfigurable) computing platforms. Among these techniques, the extension of a given instruction-set architecture with custom instructions has become a common approach. Custom instructions effectively reduce the dynamic instruction count, which, in turn, leads to an increase in performance. Traditionally, existing techniques address Instruction-Set Extension (ISE) on a per-application basis. Anyhow, when many applications have to be considered at the same time, ISE on a per-application basis is, clearly, less effective, as the custom instructions have, often, limited re-utilization across applications. To overcome this problem, we propose a new framework for the automatic generation of domain-specific ISEs. Experimental results show that, the proposed framework, evaluated on a number of applications from various domains, can effectively generate domain-specific instructions with high utilization factor across the targeted applications. At the same time, the generated instructions dramatically reduce the instruction count, 50% on average and upto 95% in special cases. This, in turn, can lead to a considerable improvement in performance.**

I. Introduction

Traditionally, embedded processors target single applications or small set of applications. Over the years, many techniques have been proposed and used in both industry and academia to specialize general purpose processors for the required application(s). One such technique consists in the augmentation of a processor core with special-purpose hardware to improve the application performance. To reduce both the design time and costs, the special-purpose hardware is often implemented on a configurable hardware, like the Field Programmable Gate Array (FPGA), coupled to a standard processor core [1]. This special purpose hardware can be in the form of either a custom functional unit, or a co-processor subsystem, or an accelerator, and it is exposed to the programmer as a Custom Instruction (CI). Supplementing a processor's instruction set with CIs has various advantages including reduced instruction cycles, improved performance, and reduced power consumption [2]. The identification of these CIs constitutes a major challenge, usually addressed in literature as the Instruction-Set Extension (ISE) problem. Over the years, many frameworks have been proposed for the automatic generation of CIs [2][3]. Anyhow, during the last decade, with the proliferation of electronic consumer gadgets, the embedded systems have changed from being highly application-specific to supporting a plethora of applications. The unique combination of increasing demand for computational resources and diversifying nature of the applications makes application-specific CIs less favourable. CIs should shift from application-specific to domain-specific, in a sense that each CI should have high utilization across the applications (in a domain), while still delivering the required performance improvements. This requirement for effective re-utilization of CIs brings in a new challenge in the CI search. Additionally, from an implementation perspective, if we target a reconfigurable architecture to implement the new CIs, the re-utilization of the CIs would lead to fewer reconfigurations of the custom functional unit on the FPGA. This, in turn, would result in reduced reconfiguration overheads in terms of both time and power. In this paper, we present a framework for the automatic identification and selection of coarse-grained domain-specific CIs. We formulate the instruction identification as a maximum common sub-graph problem and build our solution around the maximum clique problem. Once the candidate instructions are identified, instruction selection is formulated as an exact covering problem. Experimental results show that the dynamic instruction count can be reduced by 50% on average, and upto 95% in specific cases. More specifically the main contributions of this paper are the following:

- an instruction identification technique based on the maximum clique problem to boost the utilization of instructions across applications, so to identify 'real' domain-specific CIs;
- an instruction selection scheme based on the exact set covering problem for minimization of the dynamic instruction count, which can be reduced upto 95%.

The reminder of the paper is organized as follows. In Section II, we present a short overview of existing works in ISE. In Section III, we formalize the problem of domain-specific ISE and present the framework to optimally solve this problem. In Section IV, we detail our experimental setup and discuss the results. Finally, Section V concludes the paper.

II. Related Work

Existing methodologies for ISE usually address the customization (extension) problem in two steps: first instruction identification and, after that, instruction selection [2]. Instruction identification consists in the identification of a number of CIs (clusters of basic operations from the available instruction-set), which are either application- or domain-specific and can be mapped on the available (reconfigurable) computer system. Once a pool of candidate instructions is available, instruction selection narrows them down by selecting a subset of instructions, which optimize (maximize/minimize) specific metrics, such as, for example, a reduction in the application execution time and/or a reduction in power consumption. A large body of research has gone into automatic ISE [3], [2]. Early work in this direction totally side stepped the instruction identification by assuming existence of a library of templates (CIs) [4][5]. Anyhow, more recently many authors have developed methodologies for automatic identification of CIs under architectural constraints [6-12]. In [7], Atasu *et al.* proposed a single-cut identification algorithm, which maximize the gain per CI. They used a branch-and-bound algorithm with an efficient pruning technique based on the input/output (I/O) constraints for the identification of CIs. This approach is restricted by the number of register ports in a register file. To overcome this limitation, Pozzi *et al.* [13] presented a framework, which performs I/O serialization of the instructions, with higher I/O degree than the I/O degree of the register file in a processor. In their framework, instruction identification is performed using the single-cut identification algorithm from [7] with relaxed I/O constraints. This approach has two major drawbacks. Firstly, the runtime increases significantly because relaxing I/O constraints also relaxes the bounds on the single-cut identification algorithm. Secondly, it is hard to determine a relaxation factor that produces optimal results. In [12], Verma *et al.* proposed a processor-agnostic approach to ISE, which does not consider I/O degree during instruction identification. Unlike earlier approaches, which are based on exhaustive enumeration of sub-graphs, the authors formulated instruction identification as an identification of maximum cliques in a cluster graph. These approaches lead to the identification of large clusters of operations (CIs), which drastically increase the area if more than one application is considered. Our goal, instead, is to increase the utilization of the identified instructions, which, in turn, leads to a reduction of the overall area. In [8], Clark *et al.* presented a framework for automatic domain-specific instruction-set extension. The authors initially identify application-specific CIs and, later, use the techniques of wildcarding and subsumed graphs to generalize the identified instructions across the applications. Our approach to instruction identification is completely different. We consider the re-utilization of the CIs as a goal

in the instruction identification stage, which is formulated as a Maximum Common Sub-graph problem. In [10], Bonzini et al. address the generation of CIs by considering high utilization and better gain of the CIs, as we do in our work. Anyhow, they only consider the selection of CIs and leave their identification to the user. Rather, we address the full problem and we also propose an efficient instruction identification methodology, which identifies only the most *suitable* CIs available for the following instruction selection. As instruction selection is, in the general case, a well known NP-complete problem, by limiting the number of candidate instructions available for selection we obtain, as a result, an increase in scalability. In [11], Ahn *et al.* proposed an isomorphism aware instruction identification technique, which can improve the utilization of CIs. They used a canonic representation of graphs to more efficiently perform the isomorphism test. Their incremental template generation algorithm identifies connected sub-graphs with high utilization factor in the given DFG. This leads to the identification of one instruction at-a-time for a given application. Our methodology is rather different, as our identification algorithm simultaneously works with multiple applications and, in each iteration, identifies a set of CIs. Work in data path merging [14-16] is sometimes in-line with our technique for instruction identification. One of the main steps in data-path merging is to identify the common part that can be shared across applications. Our instruction identification technique is partly influenced by the data path merging algorithm proposed in [15]. This algorithm merges a set of input graphs into a super-graph. Unlike [15], we identify the maximum common sub-graph between every pair of graphs and use these sub-graphs as potential CIs. As different methodologies for CI generation use very different benchmarks and measures different metrics to evaluate their efficiencies, it is very difficult to effectively compare different methodologies. As a result, we consider, as a metric to evaluate the efficiency of our generation and selection processes, the instruction count referred to a single-issue processor model, which can be coupled with a reconfigurable hardware to implement the CIs. As the implementation in hardware is architecture dependent, we do not consider, at the moment, specific optimizations for the architecture in use.

III. Algorithms for Domain-Specific ISE

A. Motivational Example. Fig. 1 shows the dataflow graphs of the basic blocks of three different applications from [17]. They are simple but realistic examples. In the first stage, our framework identifies CIs, which are common across the applications. We consider two input DFGs at-a-time while identifying these instructions. As a result, for the three DFGs, we identify instructions in three stages: first, we identify common instructions across *ewf* and *fir*, then across *ewf* and *arf* and, finally, across *fir* and *arf*. After the instructions are identified for each pair of graphs, they are consolidated into a set, which contains the potential CIs (Fig. 1d). When we consider the utilization of the identified CIs in the applications, we can notice that some of these instructions may never be utilized. This happens as each node in the DFG can be covered by multiple CIs. We use this as a motivation for our instruction selection stage, where we cover each DFG with the identified CIs and select only those ones, which have a better utilization. Since our covering aims at reducing the dynamic instruction count, *the final set of CIs is pareto optimal*, i.e. for the given instruction utilization our CIs can produce maximum benefit (a reduction in the dynamic instruction count).

B. Problem Description. Before continuing further, it is necessary to introduce some definitions and concepts necessary to understand the proposed methodology.

Def 1. *A **Data Flow Graph** (DFG) $G(V, E)$ is a directed acyclic graph, where vertices represent basic operations and edges represent data dependencies. A sub-graph $G'(V', E')$ of a graph $G(V, E)$ is said to be an **induced sub-graph**, if $\forall u, v \in V', e = (u, v) \in E$ iff $e \in E'$.*

Def 2. *Given two graphs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$, the **Maximum Common Sub-graph** (MCS) of G_1 and G_2, $MCS(G_1, G_2)$, is a graph G', such that G' is an induced sub-graph of both G_1 and G_2, and there is no other sub-graph with greater number of vertices.*

In our framework, the goal of the instruction identification stage is to maximize the utilization of the CIs. The CIs with maximum utilization are the sub-graphs, which are common across all the applications. As this formulation leads to very small common parts, we relax our identification scheme, and find the MCS for every pair of sub-graphs. This is a disconnected graph and its connected components are the CI candidates. After the MCSs are identified for every pair of input DFGs, the connected components of these graphs represent a complete list of the CI candidates, each of which, clearly, has utilization greater than 1. To identify the instructions never utilized by the applications, we cover each of the input DFG with the identified CIs and remove the instructions, which are never utilized after all the DFGs are covered. The DFG covering problem, which is the crux of instruction selection can then be stated as follows.

Problem 1. *Given a graph G and a set of graphs $S = \{S_1, S_2, .., S_n\}$, find $S' \subseteq S$ with smallest[1] $|S'|$ and such that the number of nodes in G, not covered by elements of S', is minimal.*

C. The Framework. As outlined in Section II, the framework partitions the CI generation process into CI identification and CI selection. Given a set of DFGs, usually derived from computation intensive basic blocks of a set of given applications, the framework identifies the CIs in multiple steps.

1) Instruction Identification. This corresponds to the solution of the MCS problem for every pair of DFGs. Fig. 2(a) shows the steps involved in finding the MCS between two given graphs. A Compatibility Graph (CG) of two given DFGs is an undirected graph, where vertices represent possible edge mappings. An edge $e_1 = (u, v)$ in G_1 can be mapped to an edge $e_2 = (p, q)$ in G_2 if and only if $f(u) = f(p)$ and $f(v) = f(q)$, where the function f returns the basic operation performed at the given node. An edge exists between two vertices in a CG if both mappings represented by the two vertices are compatible. For example, let us consider the two graphs G_1 and G_2 shown in Fig. 3a and Fig. 3b. The possible edge mappings between G_1 and G_2 include $(a_1, a_3) \rightarrow (b_1, b_3)$, $(a_2, a_3) \rightarrow (b_2, b_3)$, $(a_1, a_3) \rightarrow (b_6, b_7)$ and $(a_4, a_5) \rightarrow (b_4, b_5)$. These mappings are the vertices of CG (see Fig. 3c). Both mappings $(a_1, a_3) \rightarrow (b_1, b_3)$ and $(a_2, a_3) \rightarrow (b_2, b_3)$ can co-exist, as there is no conflict between them. As a result, they are connected by an edge in CG. Now, let us consider the mappings $(a_1 a_3) \rightarrow (b_6, b_7)$ and $(a_2, a_3) \rightarrow (b_2, b_3)$. These two mappings cannot co-exist as vertex a_3 in G_1 can be mapped to both b_3 and b_7 in G_2. As a result, there is no edge between $(a_1, a_3) \rightarrow (b_6, b_7)$ and $(a_2, a_3) \rightarrow (b_2, b_3)$ in CG (see Fig. 3c). The maximum clique of CG corresponds to the maximum set of mappings that can co-exist, which corresponds to the maximum common part of G_1 and G_2. We use a branch-and-bound based exact algorithm from [18] to solve the maximum clique problem. When the solution is not computable in a feasible time, we employ a greedy heuristic from [19] to solve the problem. The MCS of the two input graphs $G1$ and $G2$ is shown in Fig. 3d. As mentioned, this is a disconnected graph and each connected component corresponds to a potential CI.

2) Instruction Selection. The unused/less-utilized CIs are removed from the identified list of instructions in this phase. This is done by covering each DFG with the list of instructions and, then, by removing all CIs with a utilization factor below a threshold value defined by the user. The steps involved in the DFG covering are shown in Fig. 2(b). For each instruction in the CI list, the isomorphism operator returns the list of vertices in the DFG isomorphic to the CI. It is noteworthy that each CI can return more than one isomorphism. After the isomorphism

[1] Given a set A, $|A|$ represents the number of elements in A.

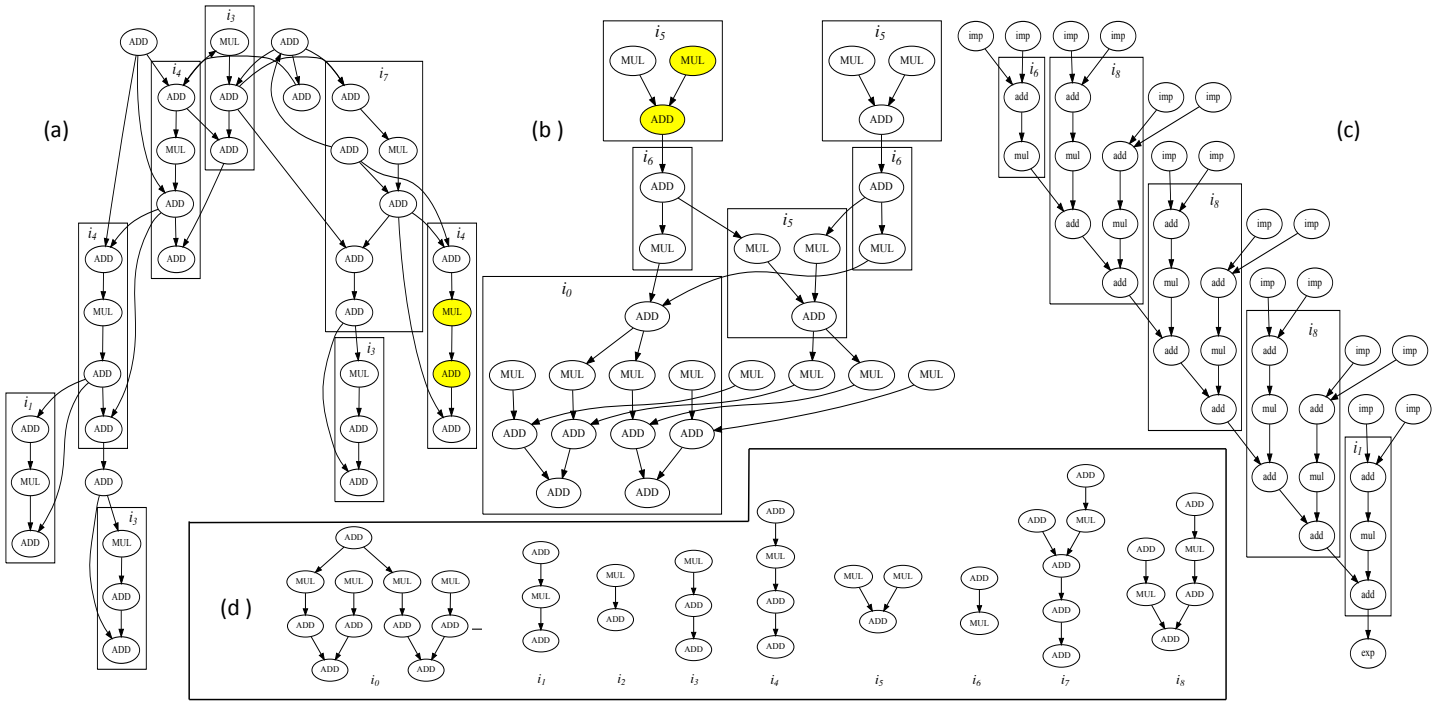

Fig. 1. Motivational example. Three DFGs from [17]: (a) Elliptic Wave Filter (ewf), (b) Finite Impulse Response Filter (fir), and (c) Auto Regression Filter (arf). The CIs generated by our framework are shown in (d).

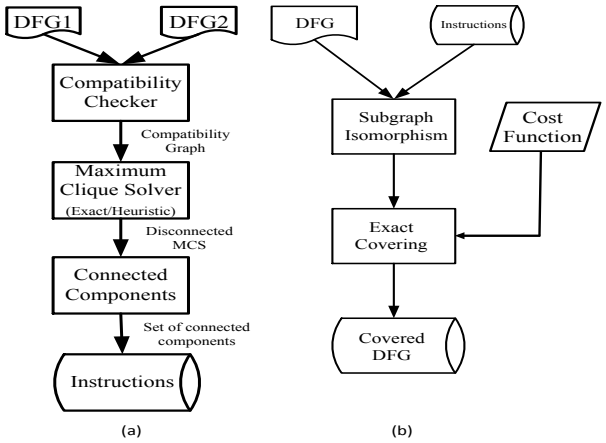

Fig. 2. (a) Steps to find the MCS of two input graphs and (b) Steps to cover a DFG with a set of instructions.

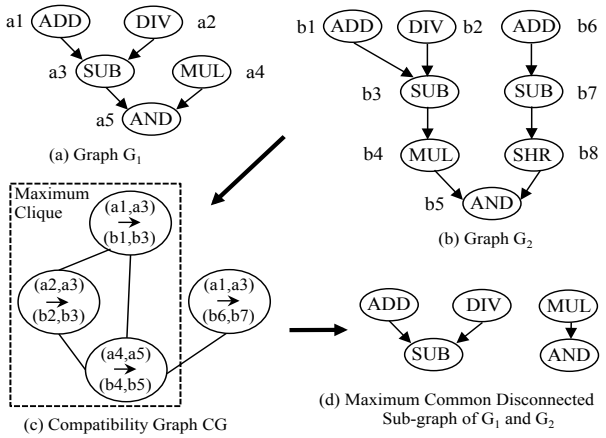

Fig. 3. Example for finding MCS of two graphs G_1 and G_2.

operator is applied to all the instructions and the DFG, we have a list of possible mappings to which we add also simple ALU operations. As a consequence, finding the cover of the DFG can now be formulated as an exact covering problem.

Def 3. *Given a set X and a collection S of its subsets, an **exact cover** of X is a subcollection $S^* \subseteq S$, such that each element in X is contained in exactly one element of S^*.*

The collection of subsets is the list of all isomorphisms. The set to cover is the set of vertices of the DFG. Each subset corresponds to a certain instruction saving and the exact cover with the maximum instruction saving is the one with minimum dynamic instruction count. We solve the exact cover problem using the Dancing Links implementation of AlgorithmX [20].

IV. Experimental set-up and Results

Our framework is implemented as a dedicated tool-chain and can work on any given set of DFGs. We have tested the framework with DFG benchmarks from [17], which are divided in three application domains and CIs are generated for each domain. Furthermore, we tested our framework by considering, at the same time, all applications from the different domains. We assume a simple single-issue processor model to evaluate the effectiveness of our CIs. These CIs can be implemented on an FPGA coupled to the processor. The performance benefit offered by these CIs depends on the implementation techniques. Therefore, we evaluate the effectiveness of our CIs on basis of the *dynamic instruction count*. The DFGs used to test our framework (Table I) typically represent the basic blocks inside loops, which constitutes most of the application's execution time. If we consider the nature of DFGs, in a simple single-issue processor model, the dynamic instruction count is directly proportional to the number of vertices in the DFG. By utilizing our CIs, a set of vertices in the original DFG can be collapsed into one vertex, which leads to a reduction in the dynamic instruction count. Fig. 4 shows the reduction in dynamic instruction counts for different application domains. The number of CIs identified for each of our experiments is shown in Table II. On average, we observe a 50% reduction in the dynamic instruction count, when the CIs identified by our framework are utilized. This reduction is higher, if the application can utilize larger CIs. For instance, a 95% reduction can be observed in *Interpolate Aux*. In this case, the entire DFG is covered by just 3 CIs. In most of the cases, the input DFG is completely covered by CIs. If not, some basic ALU operations are added to the list of operations. To further improve the utilization of instructions, we may remove the instructions with limited utilization factor (below a certain threshold value defined by the user) and repeat the covering process with the new set of CIs. When this threshold is set to 2, we observe a slight increase in the dynamic instruction count (see Fig. 4). Furthermore, the number of CIs is reduced, on average, by 33%, which, in-turn, can result in huge area and power savings by reducing, for example, run time and memory accesses. Fig. 5 shows the reduction in dynamic instruction

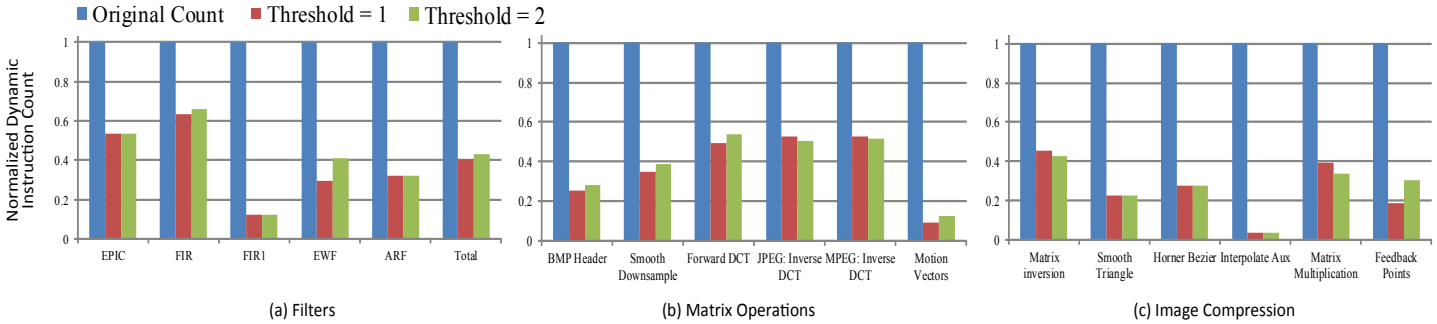

Fig. 4. Reduction in dynamic instruction count for various application domains.

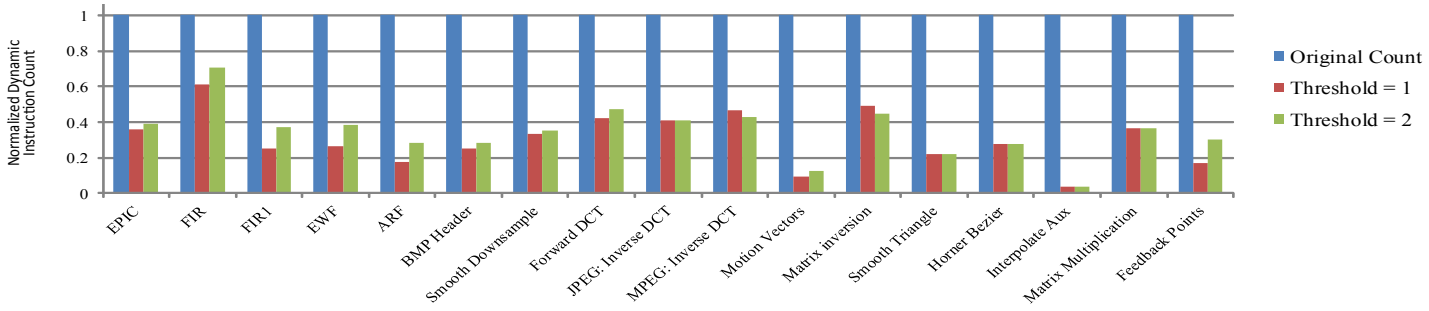

Fig. 5. Reduction in dynamic instruction count when all the applications are simultaneously given as input to our framework.

TABLE I. NUMBER OF NODES FOR THE TARGET APPLICATIONS [17].

Application	$\|V\|$	Application	$\|V\|$	Application	$\|V\|$
EPIC	56	BMP Header	106	Matrix inversion	333
FIR	44	Smooth Downsample	51	Smooth Triangle	197
FIR1	40	Forward DCT	134	Horner Bezier	18
EWF	34	JPEG: Inverse DCT	122	Interpolate Aux	108
ARF	28	MPEG: Inverse DCT	114	Matrix Multip.	109
		Motion Vectors	32	Feedback Points	53

TABLE II. NUMBER OF CIS PER APPLICATION DOMAIN.

Application domain	Threshold = 1	Threshold = 2	% reduction
Filters	13	9	30.7
Image Compression	36	25	30.5
Matrix Operations	24	19	20.8
All Applications	84	51	39.3

count, when all applications are considered simultaneously in the CIs generation process. It is interesting to see that dynamic instruction count is only slightly higher when we consider all the applications at the same time. In this case, the major disadvantage is the drastic increase of the number of generated CIs. 84 CIs need to be implemented if all the applications are simultaneously considered, whereas, the total number of CIs generated for each each domain is only 73. The reason for this is that, when we consider all the applications at the same time, the utilization factors of the CIs drastically decrease. The number of instructions with utilization factor smaller than 2 is 40% of the total instructions, when all the applications are considered at the same time. Instead, this figure is only 13%, 19% and 14% when the three domains are considered separately. If we set the utilization threshold in our framework to 2, the number of CIs is the same irrespective of weather all applications are considered simultaneously or the CIs are separately generated for each domain. Also, the reduction in dynamic instruction count is similar.

V. CONCLUSIONS

In this paper, we presented a framework for the automatic generation of domain-specific custom instructions, which can be utilized in various (reconfigurable) computing platforms, to improve the dynamic instruction count of the applications. The problem is divided into instruction generation and instruction selection. In the former, we identify instructions, which can potentially have high utilization across different applications. Later, custom instructions for hardware implementation are selected in such a way that, the dynamic instruction count is reduced. In our experiments, we considered various applications from different domains and we have shown that dynamic instruction count can be reduced 50%, on average, and upto 95% in specific cases.

REFERENCES

[1] S. Vassiliadis et al., "The molen polymorphic processor," *IEEE ToC*, vol. 53, no. 11, pp. 1363–1375, 2004.

[2] C. Galuzzi and K. Bertels, "The instruction-set extension problem: A survey," *ACM TRETS*, vol. 4, no. 2, pp. 18:1–18:28, 2011.

[3] P. Ienne and R. Leupers, *Customizable Embedded Processors: Design Technologies and Applications*. Morgan Kaufmann, 2007.

[4] C. Liem et al., "Instruction-set matching and selection for dsp and asip code generation," in *EUROASIC*, pp. 31–37, 1994.

[5] A. Alomary et al., "Peas-i: A hardware/software co-design system for asips," in *DAC*, pp. 2–7, 1993.

[6] H. Choi et al., "Synthesis of application specific instructions for embedded dsp software," *IEEE ToC*, vol. 48, no. 6, pp. 603–614, 1999.

[7] K. Atasu et al., "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *DAC*, pp. 256–261, 2003.

[8] N. Clark et al., "Automated custom instruction generation for domain-specific processor acceleration," *IEEE ToC*, vol. 54, no. 10, pp. 1258–1270, 2005.

[9] J. Cong et al., "Application-specific instruction generation for configurable processor architectures," in *FPGA*, pp. 183–189, 2004.

[10] P. Bonzini and L. Pozzi, "Recurrence-aware instruction set selection for extensible embedded processors," *IEEE TVLSI*, vol. 16, no. 10, pp. 1259–1267, 2008.

[11] J. Ahn and K. Choi, "Isomorphism-aware identification of custom instructions with i/o serialization," *IEEE TCAD*, vol. 32, no. 1, pp. 34–46, 2013.

[12] A. Verma et al., "Fast, nearly optimal ise identification with i/o serialization through maximal clique enumeration," *IEEE TCAD*, vol. 29, no. 3, pp. 341–354, 2010.

[13] L. Pozzi and P. Ienne, "Exploiting pipelining to relax register-file port constraints of instruction-set extensions," in *CASES*, pp. 2–10, 2005.

[14] P. Brisk et al., "Area-efficient instruction set synthesis for reconfigurable system-on-chip designs," in *DAC*, pp. 395–400, 2004.

[15] N. Moreano et al., "Efficient datapath merging for partially reconfigurable architectures," *IEEE TCAD*, vol. 24, no. 7, pp. 969–980, 2005.

[16] M. Stojilovic et al., "Selective flexibility: Creating domain-specific reconfigurable arrays," *IEEE TCAD*, vol. 32, no. 5, pp. 681–694, 2013.

[17] "DFG Benchmarks," http://express.ece.ucsb.edu/benchmark/, Feb. 2014.

[18] E. Tomita et al., "A simple and faster branch-and-bound algorithm for finding a maximum clique with computational experiments," *IEICE Transactions*, vol. 96-D, no. 6, pp. 1286–1298, 2013.

[19] M. Brockington and J. C. Culberson, "Camouflaging Independent Sets in Quasi-Random Graphs," in *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*. AMS, 1994, pp. 75–88.

[20] D. E. Knuth, "Dancing links," *Millenial Perspectives in Computer Science*, pp. 187–214, 2000.