Architecture-Aware Custom Instruction Generation for Reconfigurable Processors

Alok Prakash, Siew-Kei Lam, Amit Kumar Singh and
Thambipillai Srikanthan (Senior Member IEEE)

Center for High Performance Embedded Systems
School of Computer Engineering
Nanyang Technological University
email: {alok0001,assklam,amit0011,astsrikan}@ntu.edu.sg

Abstract. Instruction set extension is becoming extremely popular for meeting the tight design constraints in embedded systems. This mechanism is now widely supported by commercially available FPGA (Field-Programmable Gate Array) based reconfigurable processors. In this paper, we present a design flow that automatically enumerates and selects custom instructions from an application DFG (Data-Flow Graph) in an architecture-aware manner. Unlike previously reported methods, the proposed enumeration approach identifies custom instruction patterns that can be mapped onto the target FPGA in a predictable manner. Our investigation shows that using this strategy the selection process can make a more informed decision for selecting a set of custom instructions that will lead to higher performance at lower cost. Experimental results based on six applications from a widely-used benchmark suite show that the proposed design flow can achieve significantly higher performance gain when compared to conventional design approaches.

1 Introduction

The rapidly increasing time-to-market pressure and demand for higher performance is consistently pushing FPGA-based systems to the forefront of embedded computing. One of most popular ways of using this FPGA space, is by adding Custom Instructions to the system. Although a lot of research has been done in the area of custom instruction generation, commercial FPGA tools still lack a consolidated framework for automatic implementation of custom instructions for a given application code. There are typically two major steps in custom instruction implementation namely, custom instruction (pattern) *identification* and custom instruction (pattern) *selection*. A commonly adopted approach in pattern identification is pattern enumeration, which tries to identify all the legal custom instruction patterns within an application. Pattern selection then selects a set of non-overlapping patterns for final implementation based on certain constraints such as area and performance. In certain design flows, a design exploration step, which takes into consideration the area required by each pattern, is undertaken to propose a set of custom instructions for a given performance area constraint.

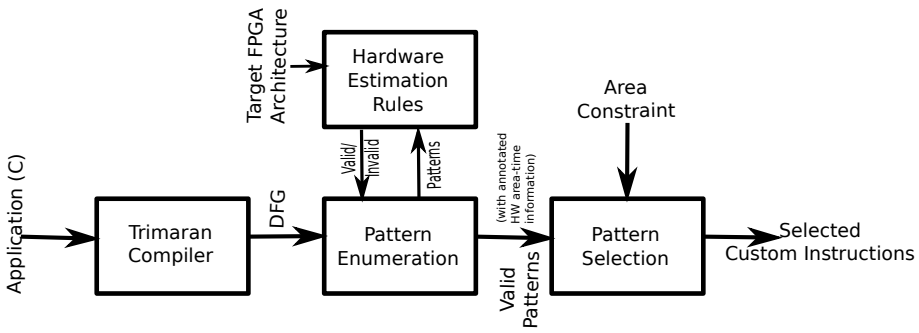


Fig. 1. Proposed Instruction Set Extension Methodology

There has been a number of previous work in the area of pattern identification and selection. Atasu et al. in [1] described a branch and bound algorithm that enumerates all the legal patterns from an application DFG, while pruning off the ones, which violate the fundamental micro-architectural constraints. Their method produced significantly better performance when compared to the state of the art techniques at that time. However, the complexity of the branch and bound algorithm in their work increases rapidly with the size of the DFG.

Clark et al. presented a complete design framework for automatic identification of pattern candidates within a DFG and a compiler framework to take advantage of such custom units [2]. In their selection stage, a greedy heuristic was used to select the candidate with largest (speedup/area) ratio before selecting any other patterns.

Chen et. al. presented a novel algorithm for rapid identification of custom instructions for extensible processors which was especially useful for large DFGs [3]. Li et. al. [4] proposed further enhancements in Chen's algorithm and thus achieving up to 50% faster tool runtime for enumerating patterns with single-output constraint.

Lam et. al. [5] presented a custom instruction generation design flow that incorporates a novel way for estimating the area of custom instructions in the final hardware by *clustering* the candidate patterns [5].

All the enumeration algorithms described above take the micro-architectural properties of the underlying hardware as a fundamental set of constraints, for example the number of input or output ports available for custom instruction implementation. Although these constraints must be used during enumeration, we propose a custom instruction generation strategy that incorporates the target architecture information in the pattern enumeration and selection phases as shown in Fig. 1.

In particular, the proposed pattern enumeration approach generates only patterns that can be fully mapped onto the logic elements of the target FPGA architecture. As the hardware area-time of the custom instruction patterns from the enumeration phase are implied, effective pattern selection can be performed to choose a set of custom instructions that can lead to higher performance at

lower cost when compared to conventional design approaches that focus on selecting large and recurring custom instructions.

The rest of the paper is organized as follows. Section 2 discusses the proposed pattern enumeration algorithm, while section 3 explains the pattern selection phase. Section 4 deals with the experiments and results. Section 5 concludes this paper.

2 Pattern Enumeration

Pattern enumeration is defined as the process of identifying all the legal patterns (subgraphs) within a DFG. The legal patterns of an application must obey a set of constraints that is compatible with the micro-architectural constraints of the underlying architecture [1], [3]. These micro-architectural constraints are used to prune away the illegal patterns. In addition, memory and branch operations are generally not allowed in a custom instruction.

The proposed pattern enumeration technique in this paper extends the method in [3] and [4], which have been shown to be orders of magnitude faster than other existing algorithms for large DFGs. In particular, we have incorporated additional constraints, which ensures that the enumerated patterns can be mapped onto the target FPGA in a predictable manner. In the following sub section we will describe the proposed architecture-aware pattern enumeration method.

2.1 Architecture-Aware Pattern Enumeration

In order to perform architecture-aware enumeration, we have introduced additional architecture specific constraints during pattern enumeration phase to limit the number of enumerated patterns to those that would lead to efficient mapping on the target FPGA and achieve high performance. Similar to [5], we have used Trimaran to compile the C Code of the benchmark application. The output of the Trimaran compiler is called a "Rebel" file which is used as the input to our enumeration stage. The "Rebel" file obtained from Trimaran gives the application representation using the basic operations of the Trimaran instruction set architecture. We mentioned before that we have used the enumeration algorithm from [3] and [4] in our work. This enumeration algorithm works in the following manner: It searches for valid patterns, starting with an empty pattern P, the convex DFG G of a basic block and a redundancy guarding node referred to as r_g. The algorithm recursively invokes the enumeration procedure, which consists of three functions, *select_node*, *unite* and *split*. Function *select_node* returns a selected node from the remaining node set *(G-P)*. The function *unite* handles the condition of addition of valid nodes into the pattern to create a new pattern, while the *split* function handles the situation where the selected node cannot be added to the pattern. The function *unite* can merge multiple nodes if the node selected by the function *select_node* is of high quality. It is in this *unite* step, wherein we can check the extra constraints for our purpose during the merging of a selected node with the existing pattern. Meanwhile, the function *split*

can decompose the current DFG *G(V,E)* into one or two sub-graphs by splitting them when necessary, thus reducing the depth of recursive search. Using this methodology the output of the *unite* phase always produces a valid pattern which is ready for the selection stage. The details of the algorithm can be found in [3] and [4].

Lam et. al. in [5] showed a novel method for estimating the area of a given pattern. They applied various architecture specific constraints to ensure that a candidate pattern can be fully implemented within a logic group (a set of FPGA logic elements with the same hardware configuration) on the Xilinx Virtex 4 FPGA. In particular, after the pattern selection phase, they partitioned each of the selected pattern into clusters, wherein each cluster could be implemented in one logic group. This enabled them to estimate the total number of logic groups required to implement the selected patterns on the target architecture. A detail description of the constraints used by them for *Clustering* can be found in [5]. We incorporated these rules along with the previously mentioned micro-architecture constraints, during our enumeration phase in order to enumerate only "clusters". This ensured that all the patterns we enumerated could be implemented in the logic blocks of the FPGA. It is noteworthy that although we have demonstrated the proposed approach for the Virtex 4 FPGA, our idea is generic enough to be used with other FPGA families.

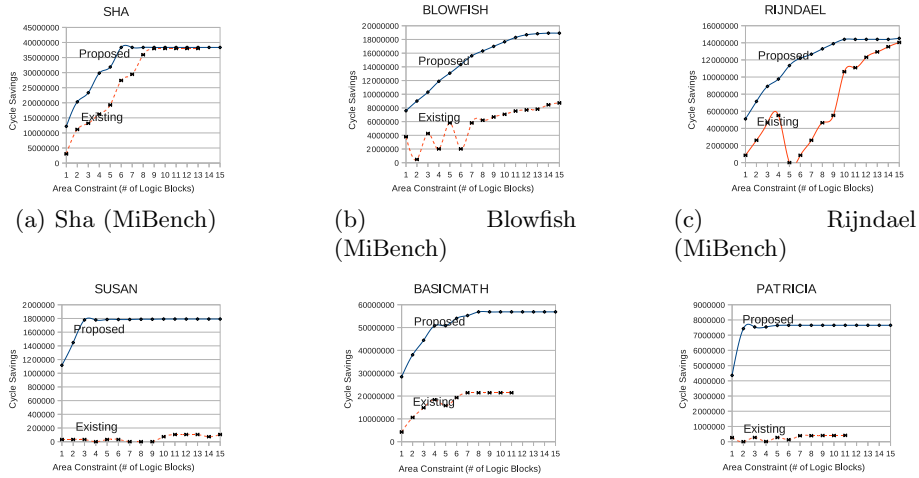
Once the valid clusters are enumerated, we begin the selection phase. In this phase, we select the most profitable patterns to be implemented as custom instructions. In the next section we will explain the pattern selection step in detail.

3 Pattern Selection

After enumerating all the possible legal patterns from the enumeration phase, we perform graph-subgraph isomorphism using "VFLib" to find the similar patterns and group them together. Each pattern group is defined as a template, whereby each template corresponds to a set of similar patterns. In the next step, pattern selection is performed to find a set of non-overlapping patterns for final implementation.

We have used the approach described in [6] for pattern selection. A conflict graph is created based on the enumerated patterns.The objective of creating this conflict graph is to facilitate choosing a set of non-overlapping patterns for final implementation.

Once the conflict graph is created, we use the steps described in [5] to compute the local Maximum Independent Set (MIS) in order to obtain a set of non-overlapping patterns within each template. This aims to find the largest set of vertices in every template of the conflict graph that are mutually non-adjacent. The process of computing the local MIS relies on a similar heuristic used in [6] and [7], which aims to select large and recurring custom instructions. The metric for this heuristic is calculated based on the product of the number of nodes in a template and the number of corresponding patterns in the DFG.



(a) Sha (MiBench) (b) Blowfish (MiBench) (c) Rijndael (MiBench)

(d) Susan (MiBench) (e) Basicmath (MiBench) (f) Patricia (MiBench)

Fig. 2. Performance vs. Area Curves for Benchmark Applications.

After the local MIS is calculated, the templates are arranged in decreasing order of their MIS weight. In order to perform constraint-aware pattern selection, the template with the largest MIS weight is evaluated first. It is selected for implementation if the area constraint is not violated. If this template violates the area constraint, we move on to evaluate the template with the next largest MIS weight and so on, unless a suitable template is found and selected. Once a template is selected, the corresponding patterns of the template are implemented as custom instructions. These selected patterns are then removed from the conflict graph, and the remaining area for custom instruction implementation is updated. The algorithm repeats to find a new set of local MIS for each template. The algorithm terminates when either the conflict graph becomes empty or the area constraint is violated.

4 Experiments and Results

In this section, we compare the results obtained using our proposed architecture-aware custom instruction design flow with a conventional approach that aims to select large and recurring custom instructions without considering how they can be mapped onto the target FPGA. The conventional methodology has been implemented using the methods described in [3], [4] for enumeration, [5], [6] for selection and [8] for design space exploration.

The cycle savings after using the extended instruction set has been calculated using the following formula [5] for both the conventional and proposed methods:

$$cycle\ savings = [\sum_1^n Num\_of\_Nodes * Dynamic\_Occurance] - [\sum_1^n Critical\_LUT\_Path * Dynamic\_Occurance] \quad (1)$$

The first term in the above equation 1 represents the software execution time of a custom instruction whereas the second term depicts the time in number of clock cycles if the custom instruction is executed in the hardware.

The graphs in Fig. 2 clearly show the advantage of the proposed architecture-aware custom instruction generation process. This methodology is especially useful for Area-Constrained FPGA designs.

5 Conclusion

In this paper, we proposed a design flow for automated custom instruction generation that takes into account the target FPGA architecture in both the custom instruction enumeration and selection phase. In particular, the proposed enumeration approach incorporates a set of rules that identifies custom instruction patterns which can be fully mapped onto the logic elements of the FPGA architecture. As the hardware area-time of the candidates are available after the enumeration process, the selection process can effectively choose a set of custom instructions that lead to high performance at low cost. Experimental results based on six applications from the MiBench benchmark suite show significant performance improvement of up to 5 orders of magnitude over an existing approach.

References

[1] Atasu, K., Pozzi, L., Ienne, P.: Automatic application-specific instruction-set extensions under microarchitectural constraints. In: DAC '03: Proceedings of the 40th annual Design Automation Conference. (2003) 256–261

[2] Clark, N., Zhong, H., Mahlke, S.: Automated custom instruction generation for domain-specific processor acceleration. Computers, IEEE Trans. on **54**(10) (Oct. 2005) 1258–1270

[3] Chen, X., Maskell, D.L., Sun, Y.: Fast identification of custom instructions for extensible processors. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on **26**(2) (2007) 359–368

[4] Li, T., Jigang, W., Deng, Y., Srikanthan, T., Lu, X.: Fast identification algorithm for application-specific instruction-set extensions. In: Electronic Design, 2008. ICED 2008. International Conference on. (Dec. 2008) 1–5

[5] Lam, S.K., Srikanthan, T.: Rapid design of area-efficient custom instructions for reconfigurable embedded processing. J. Syst. Archit. **55**(1) (2009) 1–14

[6] Guo, Y., Smit, G.J., Broersma, H., Heysters, P.M.: A graph covering algorithm for a coarse grain reconfigurable system. In: LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems, ACM (2003) 199–208

[7] Bonzini, P., Pozzi, L.: Recurrence-aware instruction set selection for extensible embedded processors. IEEE Trans. VLSI Syst. **16**(10) (2008) 1259–1267

[8] Cong, J., Fan, Y., Han, G., Zhang, Z.: Application-specific instruction generation for configurable processor architectures. In: FPGA '04, ACM (2004) 183–189