Rapid Generation of Custom Instructions Using Predefined Dataflow Structures

[*]Siew-Kei Lam, [+]Thambipillai Srikanthan and [Ψ]Christopher T. Clarke

[*, +] Centre for High Performance Embedded Systems
Nanyang Technological University
50 Nanyang Drive, Research TechnoPlaza,
3rd Storey, BorderX Block, SINGAPORE
[*]Email: assklam@ntu.edu.sg
[*]Tel: +65 6790 6643
[*]Fax: +65 6792 0774

[Ψ]Department of Electronic and Electrical Engineering,
The University of Bath, Bath BA2 7AY,
UNITED KINGDOM

Abstract: Custom instruction generation is fast becoming popular as it provides an alternative means to realize application specific processors. In this paper, we propose an efficient methodology for rapid instruction set customization on RISPs (Reconfigurable Instruction Set Processors) using predefined sets of dataflow structures that are based on templates and reusable structures. A novel template selection strategy was employed to reduce the number of templates required for matching by up to 50%, while providing comparable performance with known approaches. It has been shown that custom instructions could be realized through instantiation of a reduced set of pre-designed reusable structures. Experimental results show that a small number of reusable structures can sufficiently cater to custom instruction generation to notably reduce the time required to realize them on configurable hardware. Moreover, based on our evaluations using MiBench benchmark suites, the reusable structures constitute to only 2% of all the custom instruction instances. The custom instructions generated with reusable structures were implemented in FPGA and it is evident that up to 14% area savings with comparable performance can be achieved when compared with conventional implementation approaches.

Keywords: Instruction customization, methodology, reconfigurable processors, FPGA

1. Introduction

Future embedded SoC (System-on-a-Chip) solutions will require a higher degree of customization to manage the growing complexity of the applications. At the same time, they must continue to facilitate a high degree of programmability to meet the shrinking TTM (Time-To-Market) window. Lately, extensible processors [1][2] have emerged to provide a good tradeoff between efficiency and flexibility. Many commercial processors (e.g. Xtensa from Tensilica [3], ARCtangent from ARC [4], etc.) offer the possibility of extending their instruction set for a specific application by introducing custom functional units within the processor architecture. This

application-specific instruction set extension to the computational capabilities of a processor, provides an efficient mechanism to meet the growing performance and TTM demands. However, the NRE (Non-Recurring Engineering) costs of redesigning a new extensible processor can still be quite high. This is exacerbated as the cost, associated with design, verification, manufacture and test of deep sub-micron chips, continue to increase dramatically with the mask cost.

A RISP (Reconfigurable Instruction Set Processor) [5] consists of a microprocessor core that has been extended with a reconfigurable fabric. Similar to extensible processors, the RISP facilitate critical parts of the application to be implemented using a specialized instruction set on reconfigurable functional units. The advantages of a RISP over the extensible processors stem from the reusability of its hardware resources in various applications without incurring high NRE costs. Due to this, RISP are more flexible than an extensible processor, which precludes post design flexibility. Although reconfigurability can also be harnessed to increase hardware reusability at run-time, the reconfiguration overhead can significantly hamper the RISP's performance. Hence, commercial RISPs (e.g. Altera NIOS II [6], Xilinx MicroBlaze [7], and Stretch processors [8]) often offer large platforms with various choices of programmable resources. As with extensible processors, custom instruction selection and implementation for the RISP must be realized rapidly to meet the tight TTM requirements.

In this paper, we introduce a methodology for instruction set customization on RISPs that relies on a set of predefined dataflow structures. The proposed methodology facilitates rapid instruction set customization by providing: 1) readily available templates for efficient template matching in the custom instruction generation stage; and 2) a set of pre-designed reusable structures that can be instantiated on demand in the implementation phase. Although not the focus of this paper, the methodology also dispenses the need for lengthy hardware synthesis during design exploration, as the reusable structures are pre-characterized to obtain accurate hardware estimation models. This significantly increases the efficiency of the custom instruction generation process. Apart from presenting practical solutions for rapid instruction set customization, this paper also offers an insight to reconfigurable area requirements and efficient resource utilization in commercial RISP platforms.

A one-time effort is required to identify the templates from a subset of enumerated custom instruction instances. The pattern enumeration method introduced in [9] is combined with graph isomorphism [10] to identify unique custom instruction instances from a set of embedded applications. The process of selecting a set of templates from the custom instruction instances is called template generation. We present a heuristic approach for selecting the templates from the enumerated patterns, and show that only a limited number of templates are required to achieve comparable results with known techniques. The architecture generation stage then constructs the reusable structures by using a sub-graph isomorphism method to combine the selected templates into a set of maximal unique structures. We show that the total number of reusable structures generated from eight applications in the MiBench embedded benchmark suite [11] is only 23. Moreover, a maximum of only 9 reusable structures are required for a particular application. When compared to conventional implementation practices, synthesis results on the FPGA (Field Programmable Gate Array) platform show that the reusable structures based implementation approach exhibit potential area savings, with less than 0.5ns of average critical path delay difference.

In the following section, we discuss some previous work in the areas instruction set customization and RISP. In Section 3, we present our methodology for instruction set customization. Section 4 and 5 describes the template and architecture generation stages in the methodology. Section 6 presents the experimental results, and the paper concludes with some consideration on future directions.

2. Background

For a given application, a RISP configuration that outperforms the conventional processors must be determined rapidly without delaying the short TTM requirements for embedded systems. However, automatically determining the right set of extensible instructions for a given application and its constraints remains an open issue [2]. The problem of custom instruction identification can be loosely described as a process of detecting a cluster of operations or sub-graphs from the application DFG (Dataflow Graph) that is to be collapsed into a single custom instruction to maximize some metric (typically performance). Previous works in custom instruction identification

can be broadly classified into the following four categories: 1) pattern matching [14], 2) cluster growing [15], 3) heuristic-based [16], and 4) pattern enumeration [9].

In [14], an approach that combines template matching and generation have been proposed to identify clusters of operations based on recurring patterns. The clusters identified with this approach are typically small and may not lead to a notable gain when implemented as custom instructions. The method proposed in [15] attempts to grow a candidate sub-graph from a seed node. The direction of growth relies on a guide function that reflects the merit of each growth direction. In [16], a genetic algorithm was devised to exploit opportunities of introducing memory elements during custom instruction identification.

The methods discussed above have demonstrated possible gains, but they can potentially miss out on identifying some good custom instruction candidates. The pattern enumeration method proposed in [9] employs a binary tree search approach to identify all possible custom instruction candidates in a DFG. In order to speed up the search process, unexplored sub-graphs are pruned from the search space if they violate a certain set of constraints (i.e. number of input-output ports, convexity, operation type, etc.). In [17], pattern enumeration is combined with pattern selection and mapping to identify the most profitable custom instructions in an application. Although these two approaches can lead to promising results, they can still become too time-consuming especially when dealing with large applications.

An inherent problem in RISP arises from the reconfiguration overhead that is incurred while reusing the hardware resources for various functions. For example, the DISC (Dynamic Instruction Set Computer) processor proposed in [12] requires a reconfiguration time that is projected to contribute up to 16% of an application's total execution time. In [13], a compiler tool chain was presented to encode multiple custom instructions in a single configuration to reduce the reconfiguration overhead and maximize the utilization of the resources. However, the compiler tool chain incorporates a hardware synthesis flow that hampers the efficiency of the design exploration process. In commercial RISPs, the run-time reconfiguration overhead is exacerbated by the fine-grained programmable structure. For example, the Stretch processor [8] requires 80μs to change an instruction on their proprietary

programmable logic. Hence, commercial RISPs [6][7] often offer large platforms with various choices of programmable resources to cater to the unforeseeable requirements of the applications. Inevitably, this leads to under-utilization of the reconfigurable area, which is not desirable for cost-effective solutions.

The methodology proposed in this paper differs from previously reported work in several ways. Firstly, unlike the application centric methodologies presented in [9][13][14][16][17][18], the proposed method identifies a set of predefined dataflow structures that can generate custom instructions for numerous applications. Secondly, unlike existing methods (i.e. [9][17]), which employs a time-consuming pattern enumeration process for each application, the proposed technique performs this process only once on a standard set of applications. Thirdly, the enumerated patterns are used to generate a set of reusable structures, which are characterized to obtain their hardware properties. The pre-characterized structures lead to substantial reduction in the design time, as it does not necessitate a lengthy hardware synthesis process during application mapping such as that required in existing methods (i.e. [13] [18]). Our preliminary studies show that only a small number of reusable structures can sufficiently cater to twelve embedded applications, while providing comparable performance gain with existing techniques. Finally, the reusable structures are pre-designed and instantiated when required by the application. To the best of our knowledge, our work is the first to present strategies to expedite implementation of custom hardware on the RISP.

3. Proposed Methodology

Figure 1 illustrates an overview of the proposed methodology for instruction set customization using predefined dataflow structures. The proposed methodology consists of three key stages, namely template generation, architecture generation, and custom instruction generation. It is noteworthy that the first two stages are a one-time process, whereas custom instruction generation performs template matching to select the custom instructions for each new application. In this paper, we limit the discussion to the template and architecture generation stages.

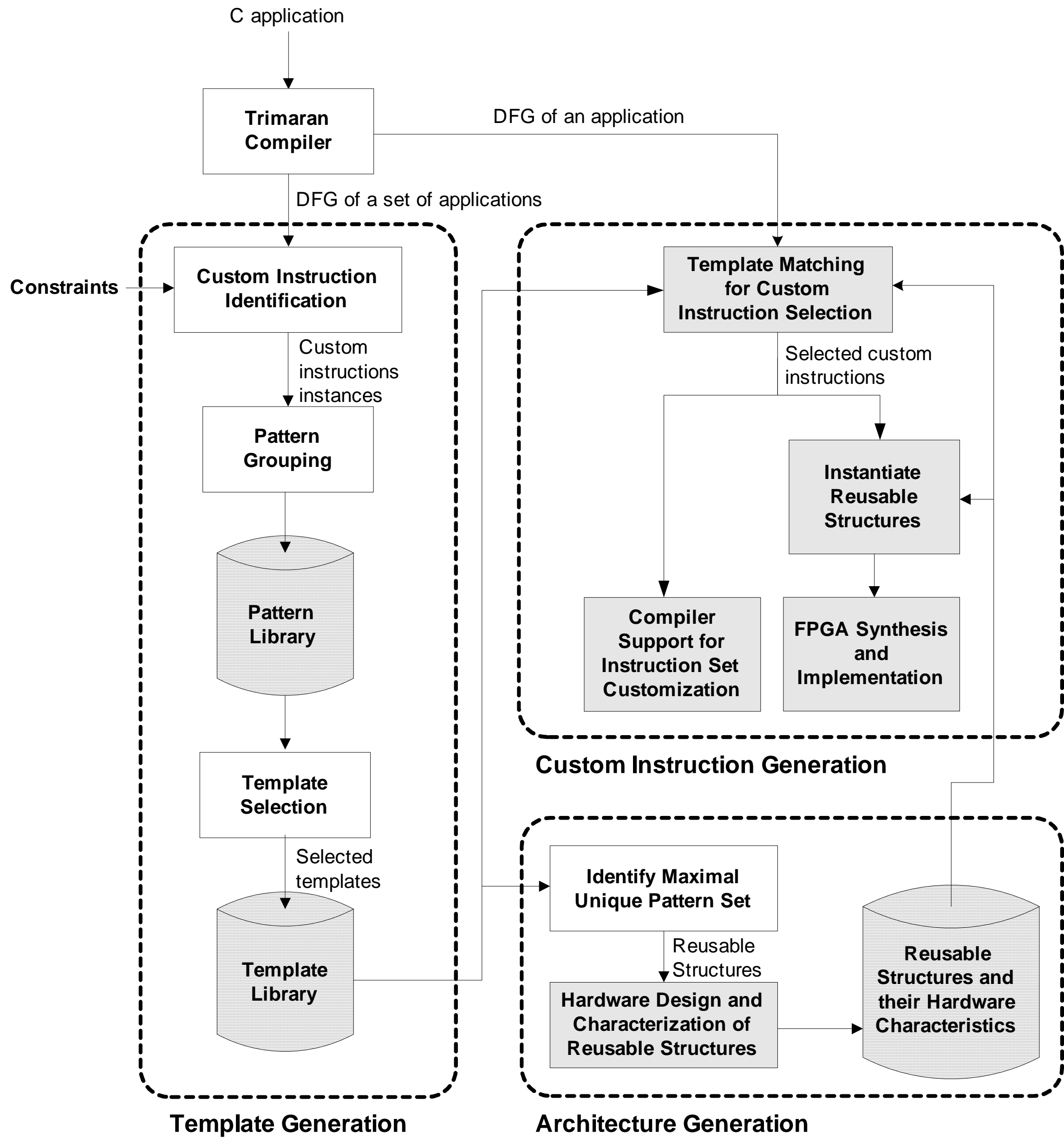


Figure 1: Overview of proposed methodology

In the template generation stage, the templates are constructed from a set of custom instruction instances obtained by enumerating a number of embedded applications. Since the templates are derived from the custom instruction instances, they are likely to implement a large variation of custom instructions in embedded applications. This is a plausible assumption as it has been shown that domain-specific applications exhibit common dataflow sub-graph patterns [19], [20]. In the next section, we will describe an approach to construct the templates in a tractable manner. In the architecture generation stage, a small set of reusable structures are constructed from the templates. The reusable structures are then modeled using hardware description

languages and stored in a component library. In addition, they are also characterized to obtain their hardware estimation models.

Finally in the custom instruction generation stage, template matching is employed to identify the custom instructions from a given application. The corresponding reusable structures of the selected custom instructions are instantiated and passed to the hardware implementation flow. The utilization of the predefined templates and reusable structures expedites the custom instruction generation stage. The application, which incorporates the custom instructions are also are passed to the compiler. The steps in this stage however are beyond the scope of this paper.

4. Template Generation

The main task of this stage is to perform template selection from a subset of custom instruction instances. The templates are used for two purposes. Firstly, the templates are used to select custom instruction candidates from a given application, and secondly the templates form the basic structures to construct the reusable structures.

It is noteworthy that compared to [17], the template generation process in our methodology is performed only once from a set of embedded applications. Let's denote this set of embedded applications as the standard application set. Hence, although this process can be time-consuming due to pattern enumeration of a large number of applications, it does not affect the custom instruction generation process.

The proposed approach for template generation is divided into three steps: 1) Custom instruction identification and 2) Pattern grouping, and 3) Template selection.

4.1. Custom Instruction Identification

The objective of this step is to enumerate the custom instruction instances from an application's DFG. We have modified the pattern enumeration algorithm in [9] to identify all the custom instruction instances from the standard application set. As mentioned earlier, the method in [9] employs a binary tree search approach that

prunes unexplored sub-graphs from the search space if they violate a certain set of constraints.

We have used the Trimaran [21] IR (Intermediate Representation) for custom instruction identification. In order to avoid false dependencies within the DFG, the IR is generated prior to register allocation. For the purpose of this study, we have imposed the following constraints on the custom instructions to increase the efficiency of the identification process:

1. Only integer operations are allowed in the custom instruction instance.
2. Each custom instruction instance must be a connected sub-graph.
3. Maximum number of input ports 5 and maximum number of output ports 2. Previous work [22] has shown that input-output ports more than this range results in little performance gain.
4. Only convex sub-graphs [9] are allowed in the custom instructions instance.
5. The operation that feeds an input to the custom instruction instance must execute before the first operation in the custom instruction instance.

4.2. Pattern Grouping

The custom instruction instances are subjected to pattern grouping, whereby identical patterns that occur in different basic blocks and applications are grouped to create a unique set of custom instruction patterns. Patterns are considered identical if they have the same internal sub-graphs, without considering their input and output operands. The static occurrences of each unique pattern are also recorded. We have used the graph isomorphism method in the vflib graph-matching library [10] for the pattern grouping process. Due to the limited size of the constrained custom instruction instances, the pattern-grouping step can be accomplished rapidly.

These unique custom instruction patterns are stored in the pattern library for the template selection process. Figure 2 presents the static occurrences and the corresponding pattern size of the unique patterns in the pattern library. The pattern size is calculated as the number of operations in the custom instruction. It can be observed that custom instructions with small pattern sizes occurs more frequently in

the set of embedded applications as compared to custom instructions with large pattern sizes.

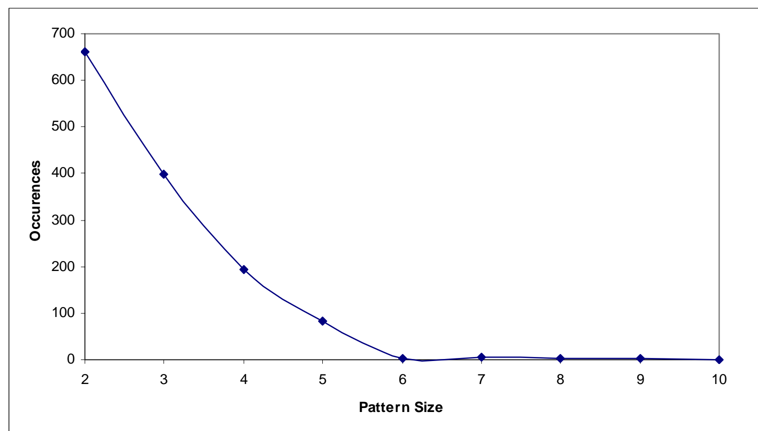


Figure 2: Static occurrences and the pattern size of the unique patterns

Table 1 shows the results obtained from the custom instruction identification process and pattern grouping using eight benchmarks from the MiBench embedded benchmark suite [11] as the standard application set. Although the pattern enumeration generates up to 1119 custom instruction instances, most of them can be grouped. After pattern grouping the number of unique patterns in the pattern library is reduced to 82 patterns. As can be observed from Table 1, a total of 82 templates can be used for custom instruction generation. Although it is desirable to limit the number of templates in order to increase the efficiency of template matching, we need to ensure that the resulting gain is not heavily compromised.

Benchmarks	Custom instruction instances
adpcm dec	17
adpcm enc	22
blowfish	990
crc32	10
dijkstra	18
FFT	6
sha	34
stringsearch	22
Total	**1119**
Number of patterns in the pattern library	**82**

Table 1: Results obtained from custom instruction identification and pattern grouping

4.3. Template Selection

In this step, a subset of templates is selected from the pattern library to reduce the complexity of the custom instruction generation stage. This is necessary as the number of templates influences the computational complexity of the template matching process.

Although custom instructions with small pattern sizes are likely to appear frequently in the embedded applications (see Figure 2), templates with larger pattern sizes should also be selected as they can lead to significant speedup in certain applications. We employ a heuristic approach for template selection, which account for the performance gain and area utilization of the custom instruction in hardware. Each pattern p is assigned a gain as shown in (1), where the $Performance\ Gain_{sas}$ (performance gain of the standard application set) obtained by mapping p on hardware is calculated as shown in (2). T_{SW} denotes the number of clock cycles taken for the custom instruction to run on a processor, and we assume each operation takes 1 clock cycle. T_{HW} denotes the number of clock cycles taken for the custom instruction in hardware, and we estimate this by the length of the critical path in the custom instruction sub-graph. For example, $T_{HW} = 5$ for a custom instruction sub-graph if the number of operations in the critical path is five. A DFS (Depth First Search) algorithm is employed to compute the length of the critical paths of each custom instruction sub-graph.

$$Gain(p) = \frac{Performance\ Gain_{sas}\ (p)}{Pattern\ size\ (p)} \qquad (1)$$

$$Performance\ Gain_{sas}\ (p) = \frac{T_{SW}(p)}{T_{HW}(p)} \qquad (2)$$

Patterns with higher gain values are selected as templates and stored in the template library. It is noteworthy that we do not consider the occurrences of the patterns in the selection decision as in [17]. This is because in our methodology, the templates are not used specifically for the standard application set. However, the preference for patterns with high occurrences is indirectly incorporated in the gain, as smaller patterns are likely to occur more frequently in embedded applications.

Figure 3 shows the percentage cycles saved that can be achieved with varying number of templates used for template matching. The baseline machine for the experiments is a four-wide VLIW (Very Long Instruction Word) architecture that can issue one integer, one floating-point, one memory, and one branch instruction each cycle. The simulation results are obtained through template generation and selection of each application independently. The percentage cycles saved for application *A* is computed as shown in (3), where p_i for $i = 1$ to k, represent the k custom instructions selected for the application *A*, dynamic occurrences (p_i) is the execution frequency of the custom instruction p_i in application *A*, and *SW Clock Cycles (A)* denotes the number of clock cycles of the application *A* that is reported from Trimaran.

$$Percentage\ cycles\ saved(A) = \frac{\sum_{i=1}^{k} Clock\ cycles\ saved(p_i) \times dynamic\ occurences(p_i)}{SW\ Clock\ Cycles(A)} \times 100 \qquad (3)$$

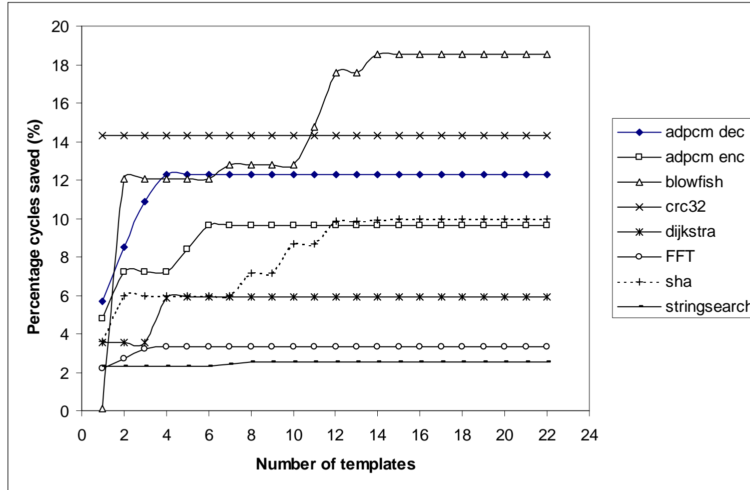


Figure 3: Percentage cycles saved with varying number of selected templates used

It can be observed that increasing the number of templates for matching will not lead to any notable gain after a certain point for each application. Hence, it is possible to reduce the number of templates for matching in order to achieve more efficient custom instruction selection, without compromising on the performance gain. A total of 40 general templates with highest gain values have been selected based on the gain computed in (1) and stored in the template library. As will be shown later, the

reduction of the number of templates for matching can lead to rapid custom instruction generation, while providing comparable results with known approaches.

5. Architecture Generation

5.1. Implementing Custom Instructions Using Reusable Structures

Figure 4 shows an example of a RISP, which is four-wide VLIW architecture that has been extended with a reconfigurable fabric for implementing custom instructions. The reusable structures (denoted as RS) implemented on the reconfigurable fabric obtain their input data from the integer unit's register file, and outputs the results to an arbitrator. High-speed arbitrators such as that found in the Altera Nios platform [20] are commonly used to facilitate the sharing of register files or memory between the processor core and other peripherals. In the example shown, the arbitrator is used to share the integer unit's register file between the ALU and custom logic. It is evident that the number and complexity of the reusable structures will affect the required area of the reconfigurable fabric.

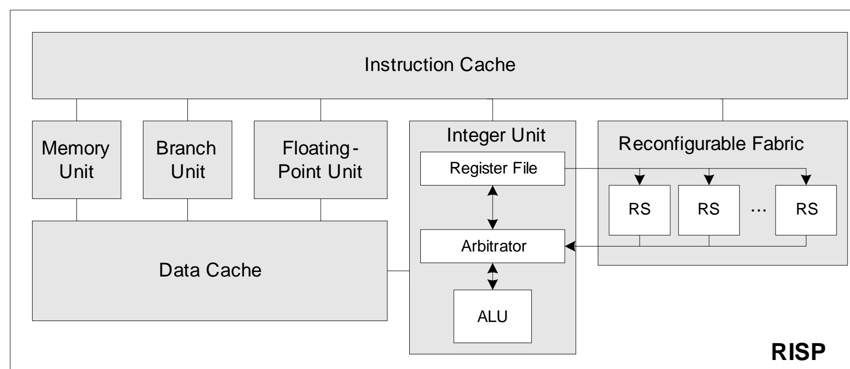


Figure 4: Implementing reusable structures on a RISP

As shown in Figure 5 a), a group of operators form a reusable structure that is employed to implement custom instructions. These operators are derived from the set of primitive operations in the processor's instruction set. Figure 5 b) describes how it can be used to implement three different custom instructions. Each of the custom instruction is sub-graph isomorphic to the structure. These custom instructions can be efficiently mapped onto a single reusable structure by taking advantage of the fact that most operations (i.e. ADD, SUB, AND as shown in the example) have an associated input that allows values to pass through the operators without changing. This is

achieved by setting one of the inputs to 0 or 1. For operators that do not possess this property (i.e. << in the example assuming the shifted value is a constant), a multiplexer is employed to allow the intermediate values to bypass the operator when the corresponding operation is not required.

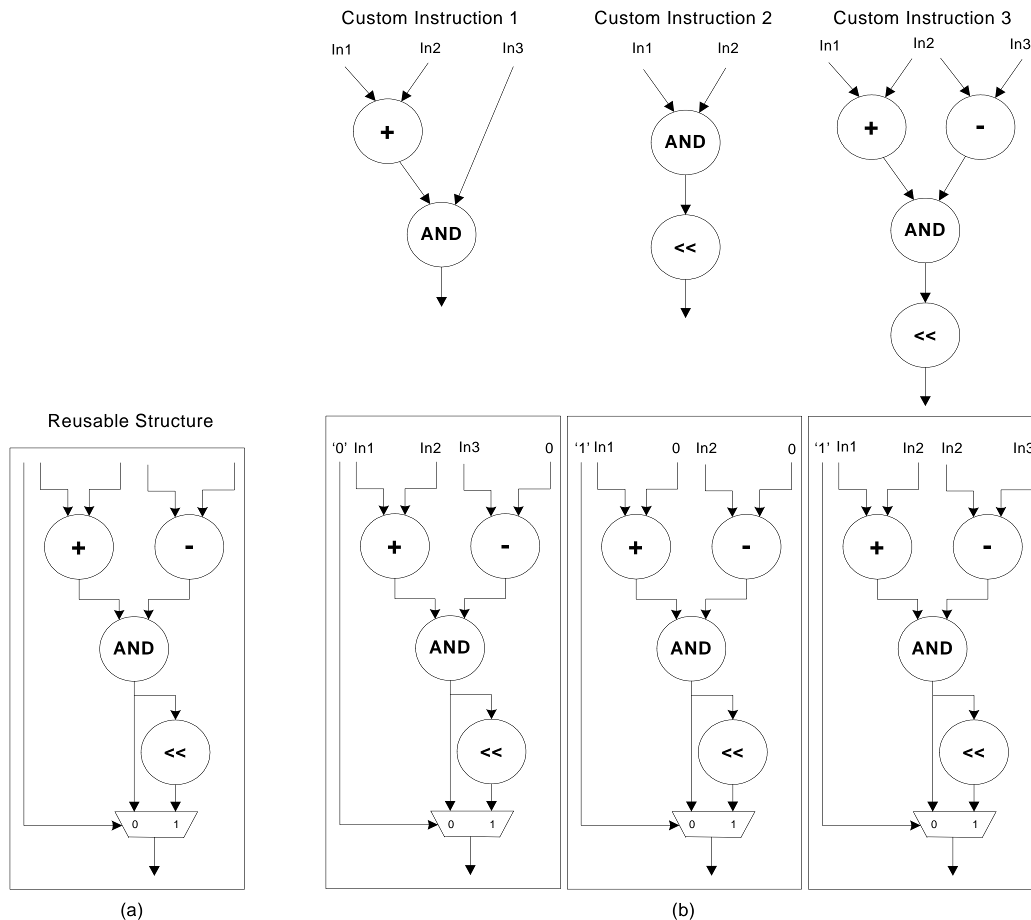
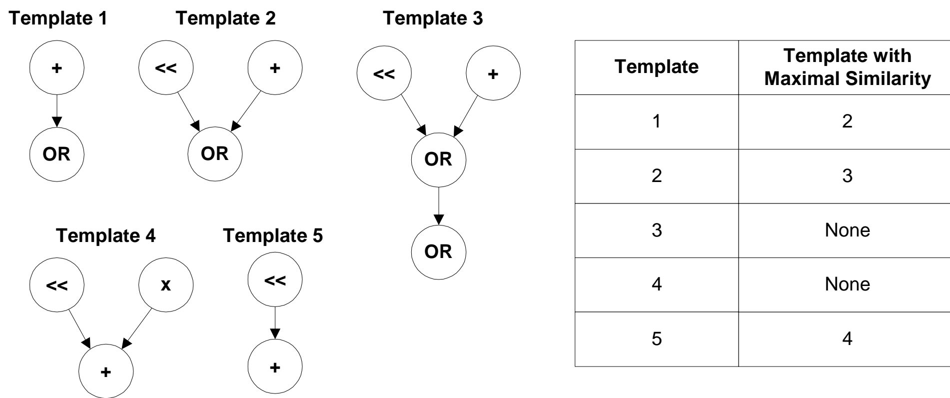


Figure 5: Implementation of custom instructions on a reusable structure

In our experiments, the reusable structures have been designed in VHDL (VHSIC Hardware Description Language) [23] and stored in a component library. When custom instructions have been selected from an application, the corresponding reusable structures are instantiated and their input and multiplexer select values are appropriately set in the main architecture. Since, all the reusable structures are pre-designed, implementation of the custom instructions can been realized rapidly. The main architecture consisting of the instantiated reusable structures are then subjected to the FPGA synthesis and implementation flow. It is noteworthy that any unused logic in the reusable structures resulting from the input and multiplexer select settings will be removed during FPGA synthesis.

5.2. Generation of Reusable Structures

The main task in the architecture generation stage is to identify a unique set of reusable structures from the template library. Specifically, we aim to find a maximal unique set of patterns that can cover all the templates. This is achieved by combining the larger sub-graphs in the template library, with smaller sub-graphs that are subsumed by it. The combination of subsumed sub-graphs is based on maximal similarity, which is defined as the minimal difference in the operations nodes of the two sub-graphs. Each pattern in the resulting maximal unique set cannot be subsumed by any other patterns in the set.



Template	Template with Maximal Similarity
1	2
2	3
3	None
4	None
5	4

Figure 6: Identifying a maximal unique set of patterns from the templates

Figure 6 illustrates the process of identifying a maximal unique set of patterns from the templates. In can be observed that although Template 1 can be subsumed by Template 2 and 3, it exhibit maximal similarity with Template 2. Hence, Template 1 is first combined with Template 2. Subsequently, Template 2 is combined with Template 3, and Template 5 is combined with Template 4. Finally, the remaining Templates 3 and 4 cannot be subsumed by each other and they formed the final set of maximal unique patterns. We can visually inspect that Templates 3 and 4 can cover Templates 1-5.

Combination of the subsumed patterns is equivalent to the sub-graph isomorphism problem. It is evident that this task is time consuming given the NP-completeness of the problem and the growing complexity of DFGs in modern embedded application. We have relied on the vflib graph-matching library [10] to find a maximal unique pattern set from the selected templates.

The reusable structures are then designed and characterized to obtain their hardware performance and cost models to be used during custom instruction selection. Figure 7 shows the 23 reusable structures that have been constructed from the standard application set and the corresponding applications that they cater to. It is noteworthy that the number of reusable structures is only 2% of the custom instruction instances.

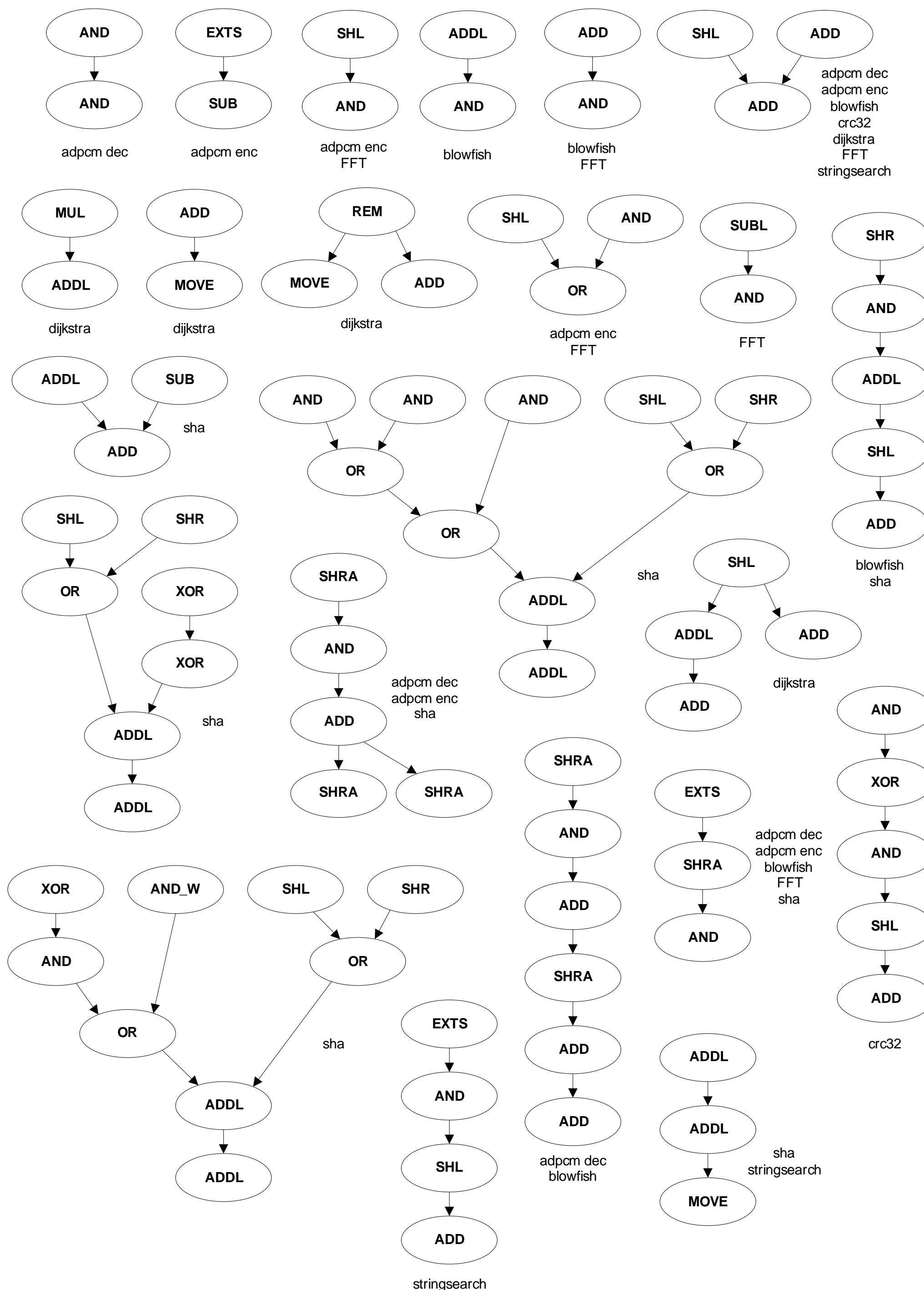


Figure 7: The set reusable structures constructed through the proposed methodology

6. Experimental Results

In this section, we present experimental results to evaluate the benefits of our proposed methodology. To illustrate the advantages of this approach, we chose a total of eight benchmarks from the MiBench embedded benchmark suite [11] as the standard application set (see Table 1). The baseline machine for the experiments is a four-wide VLIW architecture that can issue one integer, one floating-point, one memory, and one branch instruction each cycle.

Figure 8 compares the performance obtained by the proposed technique with an approach based on application-centric template selection. We denote the latter as an application-centric approach. The application-centric approach performs pattern enumeration on each application individually to select templates using a gain that combines speedup and the pattern occurrences, which is similar to the approach presented in [17]. In the application-centric approach, template matching is performed on the application using all the templates in the order of descending gain values. When a pattern match occurs, a custom instruction has been identified and the corresponding pattern is removed from the application DFG. The template matching process is repeated until there is no more pattern matches.

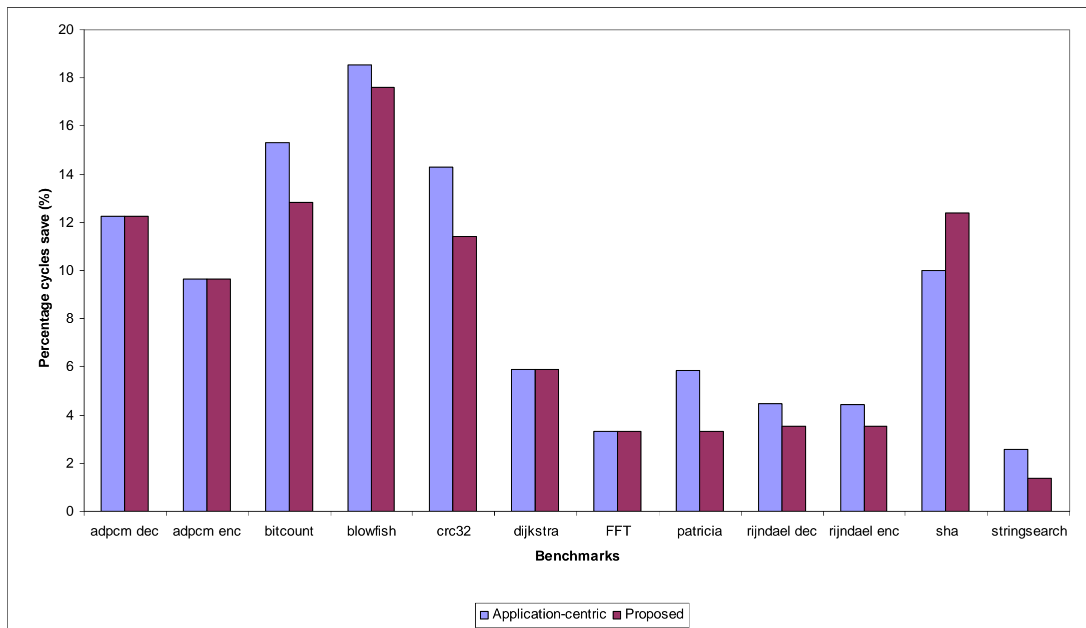


Figure 8: Performance comparison of the proposed method with an application-centric approach

It can be observed from Figure 8 that the proposed method, which employs the same strategy for template matching (except that the gain in (1) is used and the number of templates are restricted to 40), provides comparable results with the application-centric approach. It is noteworthy that comparable percentage cycle savings can still be obtained for applications (i.e. bitcount, patricia, rijndael dec, and rijndael enc from the MiBench embedded benchmark suite) that are not part of the standard application set. The average percentage cycle savings difference is only 0.7857%. Hence, it is demonstrated that rapid custom instruction selection can be achieved through significant reduction of the number of predefined templates (i.e. 50%) without compromising heavily on the performance gain.

It can be observed from Figure 8 that the proposed method leads to higher percentage cycle savings than the application-centric approach for the sha application. This is due to the random choice of templates used for matching in the application-centric approach, which may obviate highly profitable pattern matches when previous matched patterns are removed from the application DFG. This is avoided in the proposed method, as templates are matched in the order of descending gain values and hence, profitable pattern matches are more likely to occur earlier in the matching process.

Benchmarks	Required number of reusable structures	Sum of Pattern Sizes of the reusable structures
adpcm dec	5	19
adpcm enc	6	18
bitcount	3	9
blowfish	6	21
crc32	2	8
dijkstra	5	14
FFT	6	15
patricia	3	7
rijndael dec	9	36
rijndael enc	9	36
sha	8	45
stringsearch	3	10
Average	**5.42**	**19.83**

Table 2: The number of reusable structures and the total pattern sizes for each application

Table 2 shows the number of reusable structures (as illustrated in Figure 7) required for each application and the total pattern sizes. As can be observed, the average number of reusable structures and the average number of operations for the twelve applications is only 5.42 and 19.83 respectively. The rijndael applications require the

most reusable structures (i.e. 9), and the sha application requires reusable structures with the largest number of operations (i.e. 45). These results imply that the reconfigurable area on the RISP can be predetermined to cater to efficient custom instruction implementations.

Figure 9 and Figure 10 illustrates the area and latency comparisons of custom instruction implementations of the standard application set on the Xilinx Virtex xc2v40fg256-4 [7] FPGA device. The comparison is made between the conventional practice of implementing each of the selected custom instructions individually (denoted as Conv), and the implementation based on the reusable structures (denoted as RS). The results were obtained from Synplify Pro 7.5.1 [24], a state-of-the-art FPGA synthesis tool.

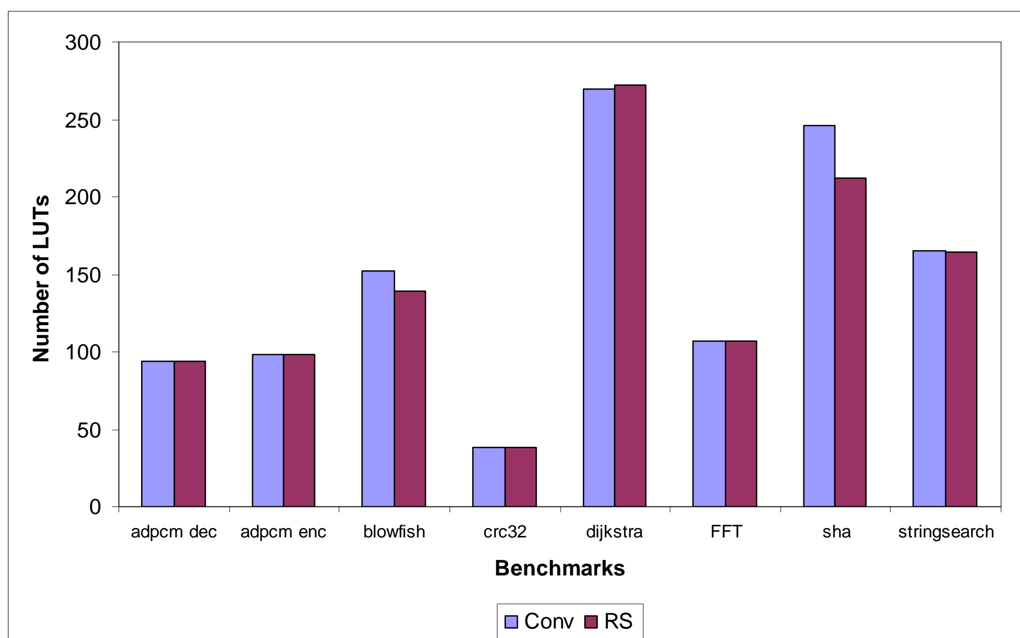


Figure 9: Area comparison of custom instructions implementation

It can be observed from Figure 9 that in general, the area consumed by the two implementations differs only marginally. Hence, although the pre-designed reusable structures typically have a larger area than the optimal custom instruction implementation, due to the inclusion of logic to generalize the structures, the redundant logic of the reusable structures are removed in the synthesis process. This results only in a marginal area difference between both the implementations. In addition, area savings are evident in applications (i.e. up to 14% for the sha

application) in which the reusable structures are exploited for multiple custom instruction implementations. This is due to the fact that a single reusable structure can be implemented in place of two or more custom instructions as described in Section 5.1.

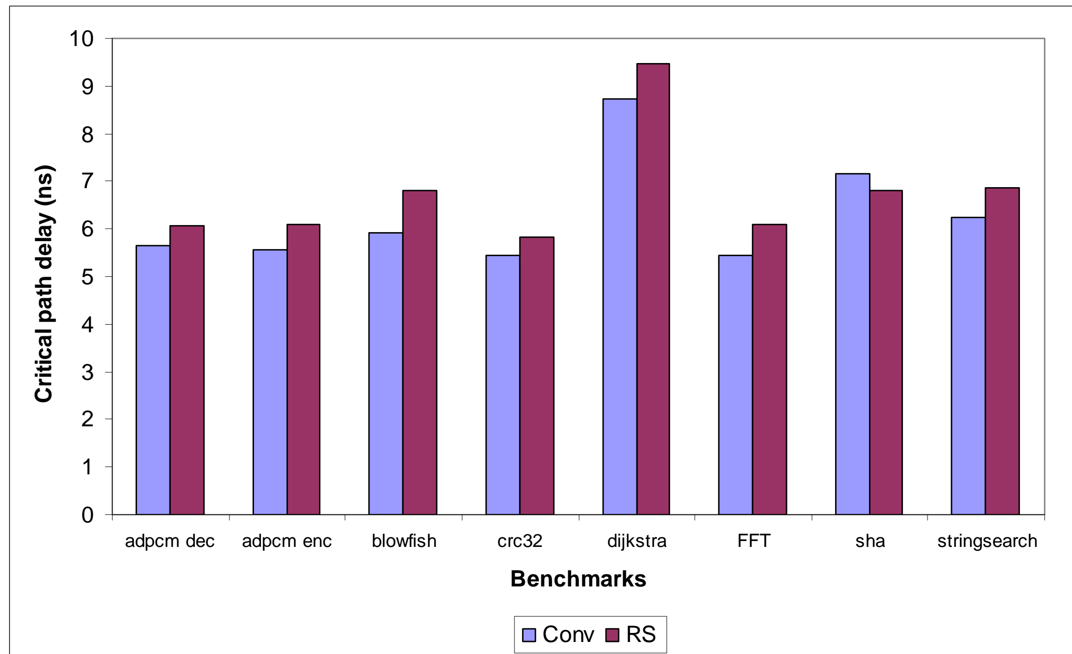


Figure 10: Critical path delay comparison of custom instructions implementation

Figure 10 compares the critical path delay of the two approaches on the Virtex device. It can be observed that the average delay difference of the two approaches is only 0.483 ns. Hence, it is evident that the implementation based on the reusable structures approach leads to rapid custom instruction realizations with potential area savings and comparable latency with the conventional implementation approach.

7. Conclusion

A novel methodology for instruction set customization of RISPs (Reconfigurable Instruction Set Processors) is presented in this paper. Our investigations show that the proposed methodology is superior to existing techniques for realizing custom instructions on RISPs both in terms of area savings and generation time. Unlike other approaches, it has been demonstrated that custom instruction enumeration needs to be performed only once on a set of applications to derive a unique set of templates. The

number of templates used for template matching is up to 50% lower without any notable performance degradations. Our studies show that only 2% of the custom instruction instances are required for the benchmarks suites examined. Synthesis results on the FPGA platform show that the reusable structures based approach exhibit potential area savings of up to 14%, with less than 0.5ns of average critical path delay difference when compared to conventional implementation practices. It is noteworthy that the proposed approach allows for a small set of reusable structures to be pre-designed and instantiated on the reconfigurable fabric to facilitate the rapid generation of custom instructions. This preliminary work has shown promising results to encourage further research in this area, which aims to promote on-line generation of custom instructions. Future work includes the identifying of a suitable standard application set to cater to a wider range of applications, devising more accurate models for the template gain values and incorporating efficient template selection schemes.

8. References

1. Dutt, N., Choi, K.: Configurable Processors for Embedded Computing, Computer, Vol. 36, No. 1, 2003, pp. 120-123
2. Henkel, J.: Closing the SoC Design Gap, IEEE Computer, Vol. 36, No. 9, 2003, pp. 119-121.
3. Xtensa Microprocessor: http://www.tensilica.com
4. ARCtangent Processor: http://www.arc.com
5. Barat, F., Lauwereins, R., Deconinck, G.: Reconfigurable Instruction Set Processors from a Hardware/Software Perspective, IEEE Transactions on Software Engineering, Vol. 28, No. 9, 2002, pp. 847-862
6. Altera Nios Soft Core Embedded Processor: http://www.altera.com
7. Xilinx Platform FPGAs: http://www.xilinx.com
8. Flaherty, N.: On the Chip or On the Fly, IEE Review, Vol. 50, No. 9, pp. 2004, 48-51
9. Atasu, K., Pozzi, L., Ienne, P.: Automatic Application-Specific Instruction-Set Extensions Under Microarchitectural Constraints, Proceedings of the 40th IEEE/ACM Design Automation Conference, 2003, pp. 256-261
10. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: Performance Evaluation of the VF Graph Matching Algorithm, International Conference on Image Analysis and Processing, 1999, pp. 1172-1177
11. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite, IEEE International Workshop on Workload Characterization, 2001, pp. 3-14
12. Wirthlin, M.J., Hutchings, B.L.: A Dynamic Instruction Set Computer, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines, 1995, pp. 99-107

13. Kastrup, B., Bink, A., Hoogerbrugge, J.: ConCISe: A Compiler-Driven CPLD-based Instruction Set Accelerator, Proceedings of the 7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 1999, pp. 92-101
14. Kastner, R., Kaplan, A., Memik, S.O., Bozorgzadeh, E.: Instruction Generation for Hybrid Reconfigurable Systems, ACM Transactions on Design Automation of Embedded Systems, Vol. 7, No. 4, 2002, pp. 605-627
15. Clark, N., Zhong, H., Mahlke, S.: Processor Acceleration Through Automated Instruction Set Customization, Proceedings of the 36th IEEE/ACM International Symposium on Microarchitecture, 2003
16. Biswas, P., Choudhary, V., Atasu, K., Pozzi, L., Ienne, P., Dutt, N.: Introduction of Local Memory Elements in Instruction Set Extensions, Proceedings of the 41st Annual IEEE/ACM Design Automation Conference, 2004, pp. 729-734
17. Cong, J., Fan, Y., Han, G., Zhang, Z.: Application-Specific Instruction Generation for Configurable Processor Architectures, Proceedings of the 12th International Symposium on Field Programmable Gate Arrays, 2004, pp. 183-189
18. Sun, F., Ravi, S., Raghunathan, A., Jha, N.K., "Custom-Instruction Synthesis for Extensible-Processor Platforms", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 23, No. 2, February 2004, pp. 216-228.
19. Sassone, P.G., Wills, D.S.: On the Extraction and Analysis of Prevalent Dataflow Patterns, Proceedings of the Workshop on Workload Characterization, 2004
20. Spadini, F., Fertig, M., Patel, S.: Characterization of Repeating Dynamic Code Fragments, Technical Report CRHC-02-09, University of Illinois, Urbana-Champaign, 2002
21. Trimaran: An Infrastructure for Research in Instruction-Level Parallelism: http://www.trimaran.org
22. Yu, P., Mitra, T.: Characterizing Embedded Applications for Instruction-Set Extensible Processors, Proceedings of the 41st IEEE/ACM on Design Automation Conference, 2004, pp. 723-728
23. Roth, C.H.: Digital Systems Design Using VHDL, PWS Publishing Company, 1998
24. Synplicity: http://www.synplicity.com/