Morphable Structures for Reconfigurable Instruction Set Processors

Lam Siew-Kei, Deng Yun, and Thambipillai Srikanthan

Centre for High Performance Embedded Systems
Nanyang Technological University
Nanyang Drive, SINGAPORE
{assklam, pg10831880, astsrikan}@ntu.edu.sg

Abstract. This paper presents a novel methodology for instruction set customization of RISPs (Reconfigurable Instruction Set Processors) using morphable structures. A morphable structure consists of a group of hardware operators chained together to implement a restricted set of custom instructions. These structures are implemented on the reconfigurable fabric, and the operators are enabled/disabled on demand. The utilization of a predefined set of morphable structures for instruction set customization dispenses the need for hardware synthesis in design exploration, and avoids run-time reconfiguration while minimizing the reconfigurable area. We will describe the two stages of the methodology for constructing the morphable structures, namely template generation and identification of a maximal unique pattern set from the templates. Our preliminary studies show that 23 predefined morphable structures can sufficiently cater to any application in a set of eight MiBench benchmark applications. In addition, to achieve near-optimal performance, the maximum required number of morphable structures for an application is only 8.

1 Introduction

Future embedded systems 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 of embedded systems. 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. However, reconfigurability of the RISP incurs an overhead that can hamper its ability to outperform conventional instruction set processors.

In this paper, we introduce a methodology for instruction set customization on RISPs that relies on a set of morphable structures to implement the custom instructions. A one-time effort is required to identify a unique set of morphable structures from a subset of enumerated custom instruction instances. The pattern enumeration method introduced in [6] is combined with graph isomorphism [7] to identify unique custom instruction instances from a set of embedded applications. The process of selecting a subset of custom instruction instances or templates 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 morphable structures are then constructed by using a subgraph isomorphism method to combine the selected templates into a set of maximal unique structures. These morphable structures are then characterized to obtain their hardware performance and cost models to be used for future design exploration. We show that the total number of unique morphable structures generated from eight applications in the MiBench embedded benchmark suite [8], is only 23. Moreover, a maximum of only 8 morphable structures are required for a particular application. This restricted set of morphable structures is sufficient to achieve high performance gain, while keeping the reconfigurable logic area low.

The availability of a predefine set of morphable structures dispenses the need for hardware implementations during design exploration. This can significantly increase the efficiency of the custom instruction selection process. The main goal of custom instruction selection is to determine viable custom instruction candidates from the application DFG (Data Flow Graph) to be implemented on a morphable structure. The custom instruction selection process in the methodology employs template matching that utilizes the restricted set of templates for improved efficiency. In this paper, we will not discuss the custom instruction selection process and limit the scope to the construction of morphable structures.

In the following section, we discuss some previous work in the areas of RISP and instruction set customization. In Section 3, we describe the notion of morphable structures on RISP and in Section 4, present our methodology for instruction set customization using these structures. Section 5 presents the experimental results, and the paper concludes with some consideration on future directions.

2 Background

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 [9] requires a reconfiguration time that is projected to contribute up to 16% of an application's total execution time. In [10], 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 [11] requires 80μs to change an instruction on their proprietary programmable logic. In order to maximize the efficiency of hardware execution, commercial RISPs [12], [13] often provide a large reconfigurable area to accommodate all the custom instructions of an application. These custom instructions are implemented on the reconfigurable fabric prior to execution to avoid run-time reconfiguration. However, these processors are likely to violate the tight area constraints imposed by most embedded systems.

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 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 [6].

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 [6] 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 generation and matching 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.

The methodology proposed in this paper differs from previously reported work as it aims to identify a predefine set of morphable structures that can implement custom instructions of numerous applications. Unlike existing methods (i.e. [6], [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. The enumerated patterns are used to generate a set of morphable structures, which are then 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. [10]). Our preliminary studies show that only a small number of morphable structures can sufficiently cater to eight embedded applications, while providing comparable performance gain with existing techniques. In addition, this study opens up new possibilities for area-efficient designs of commercial RISPs [12], [13], as the proposed methodology provides an insight to the reconfigurable area needed for efficient custom instruction implementations.

3 Instruction Set Customization Using Morphable Structures

A morphable structure consists of a group of operators that are chained together to implement a restricted set of custom instructions. These operators are derived from the set of primitive operations in the processor’s instruction set. Fig. 1a) illustrates an example of a morphable structure and Fig. 1b) 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 the morphable structure by enabling and disabling the necessary operators. Operators that are disabled allow the input operand to bypass the primitive operation, and directly routed to the output port.

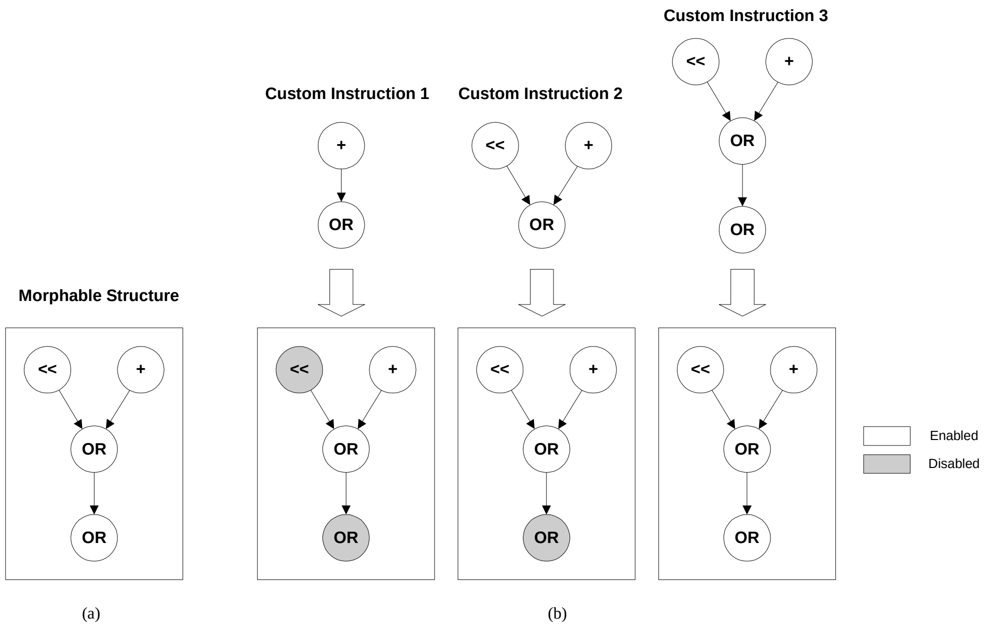


Fig. 1. Implementation of custom instructions on a morphable structure

Fig. 2 shows an example of a RISP, which is four-wide VLIW (Very Long Instruction Word) architecture that has been extended with a reconfigurable fabric for implementing custom instructions. The morphable structures (denoted as MS) 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 configurable platform [13] are commonly used to facilitate the sharing of register files or memory between the processor core and other peripherals. In the example RISP, 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 morphable structures will affect the required area of the reconfigurable fabric. In addition, since the complexity of the arbitrator logic is dependent on the number of connections to the morphable structures, it is imperative to keep the number of morphable structures tractable.

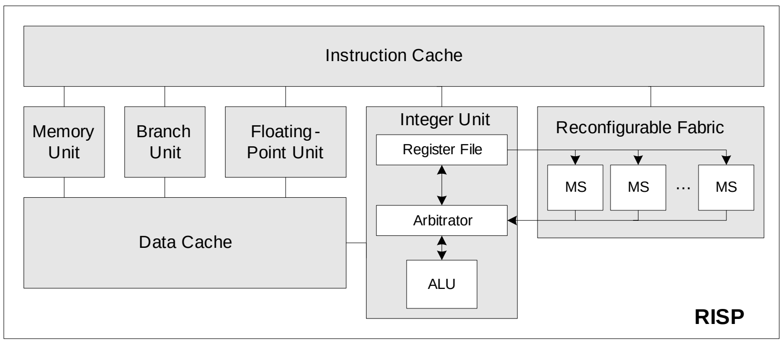


Fig. 2. Implementing morphable structures in a RISP

The morphable structures are pre-defined from a set of custom instruction instances obtained by enumerating a number of embedded applications. Since the morphable structures 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 [18], [19]. It is noteworthy that although a substantial amount of effort is required to obtain the morphable structures, this process is performed only once. In a later section, we will describe an approach to obtain the maximal unique set of morphable structures in a tractable manner.

The advantage of using morphable structures stems from the availability of a predefine set of morphable structures that can lead to rapid design exploration without a time-consuming hardware synthesis flow to evaluate the feasibility of the custom instruction candidates. This is possible as the morphable structures can be pre-characterized to facilitate area-time estimations of the custom instructions on hardware. In addition, a minimal set of morphable structures can be mapped onto the reconfigurable logic prior to the application execution to avoid run time reconfiguration. The reconfigurable logic space to accommodate the morphable structures is also minimized, as the number of morphable structures that are specific to a particular application is reasonably small.

4. Proposed Methodology

Fig. 3 illustrates an overview of the proposed methodology for instruction set customization using morphable structures.

The proposed methodology consists of several key stages, but in this paper, we limit the discussion to the construction of morphable structures. This comprises of two stages: template generation and identification of morphable structures. It is noteworthy that these two stages along with the hardware characterization of morphable structures is a one-time process, whereas custom instruction selection performs template matching to select the custom instructions for each new application. The selected custom instructions, and the corresponding morphable structures are passed to the compiler and hardware synthesis flow, which are beyond the scope of this paper. In the following sub-sections, we will provide more detailed descriptions of the two stages to construct morphable structures.

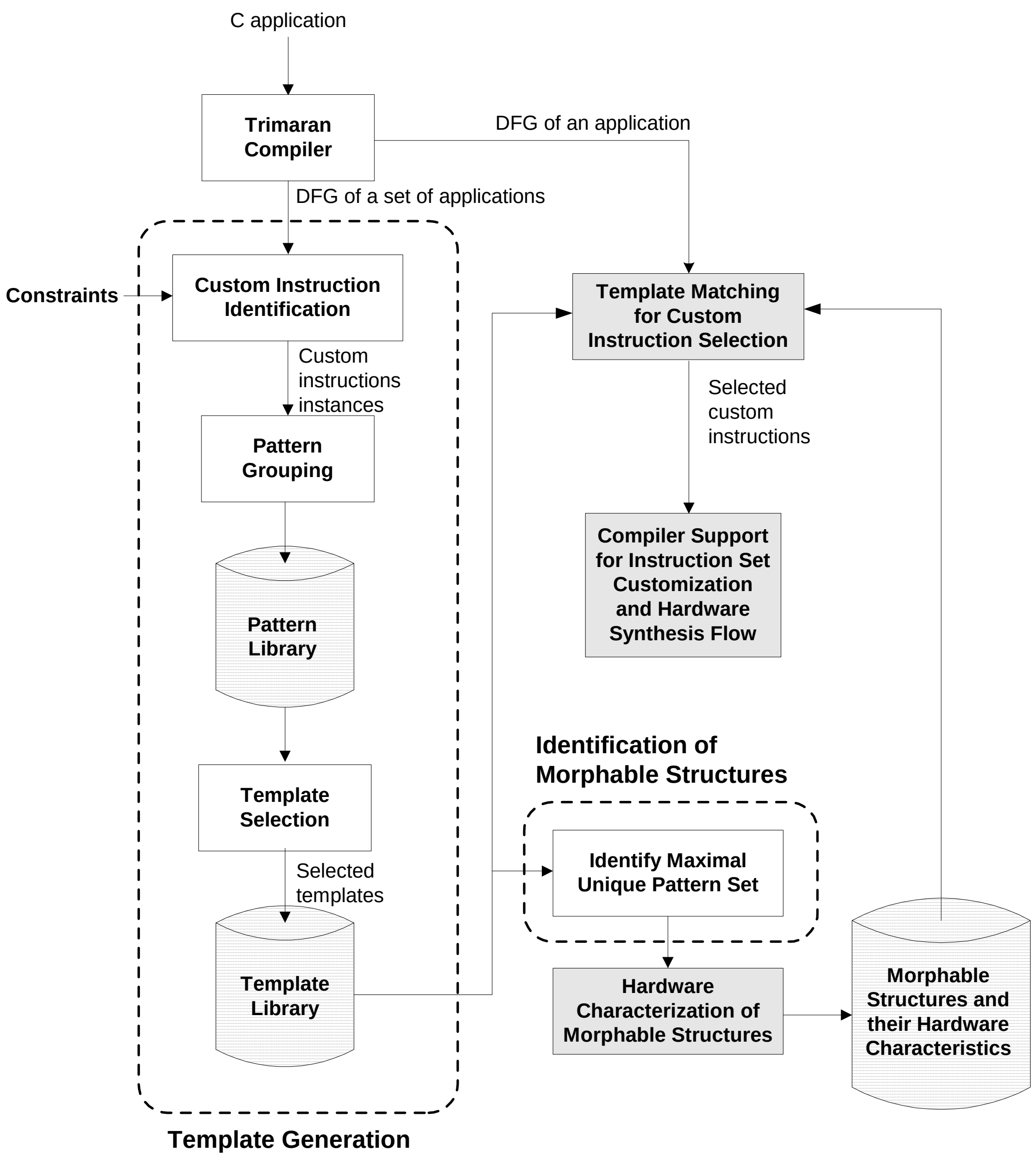

C application
Trimaran Compiler
DFG of an application
DFG of a set of applications
Constraints
Custom Instruction Identification
Custom instructions instances
Pattern Grouping
Pattern Library
Template Selection
Selected templates
Template Library
Template Generation
Template Matching for Custom Instruction Selection
Selected custom instructions
Compiler Support for Instruction Set Customization and Hardware Synthesis Flow
Identification of Morphable Structures
Identify Maximal Unique Pattern Set
Hardware Characterization of Morphable Structures
Morphable Structures and their Hardware Characteristics

Fig. 3. Overview of proposed methodology

4.1. 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 form the basic structures to construct the morphable structures, and secondly the templates are used to select custom instruction candidates from a given application.

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 large applications, it does not affect the custom instruction selection process.

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

4.1.1 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 [6] to identify all the custom instruction instances from the standard application set. As mentioned earlier, the method in [6] 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 [20] 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 [21] has shown that input-output ports more than this range results in little performance gain when no memory and branch operations are allowed in the custom instructions.
4. Only convex sub-graphs [6] 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.

4.1.2 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 [22] 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. Fig. 4 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.

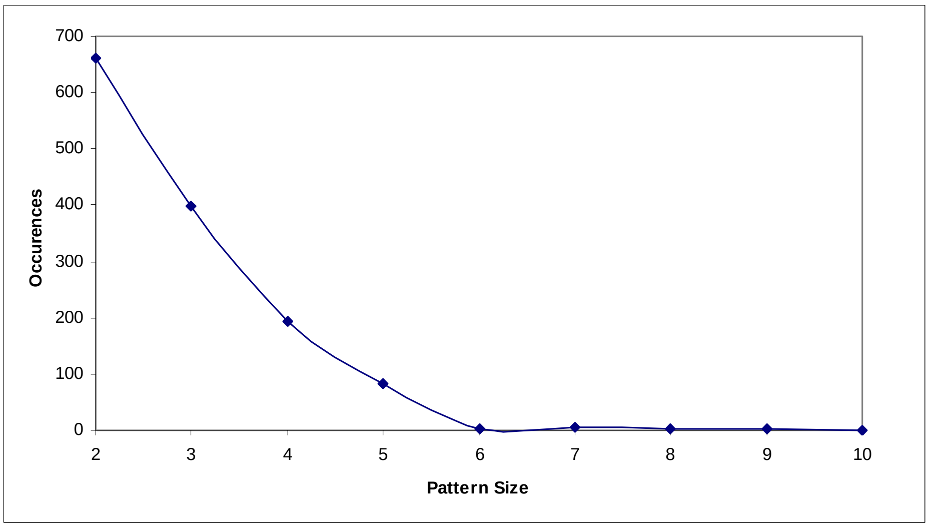


Fig. 4. Static occurrences and the pattern size of the unique patterns

4.1.3 Template Selection

In this step, a subset of templates is selected from the pattern library to reduce the complexity of the custom instruction selection process. This is necessary as the number of templates influences the computational complexity of the template matching process in custom instruction selection. In addition, restricting the number of templates can also lead to more efficient construction of morphable structures.

Although, custom instructions with small pattern sizes are likely to appear frequently in the embedded applications (see Fig. 4), 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 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.

$$Gain(p) = \frac{Performance\ Gain\ (p)}{Pattern\ size\ (p)} \tag{1}$$

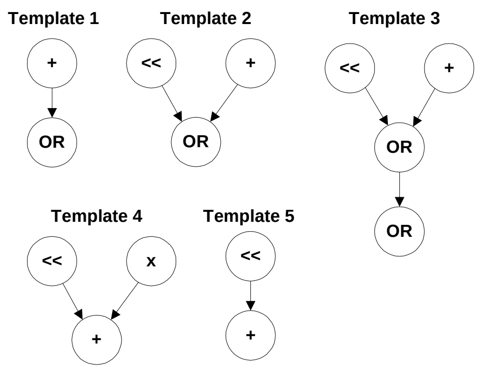
$$Performance\ Gain\ (p) = \frac{T_{SW}(p)}{T_{HW}(p)} \tag{2}$$

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 pattern size of the custom instructions implicitly associate the occurrences of patterns in the gain as smaller patterns are likely to occur more frequently in embedded applications.

4.2 Identification and Characterization of Morphable Structures

The main task in this stage is to identify a unique set of morphable 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.

Fig. 5 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.



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

Fig. 5. Identifying a maximal unique set of patterns from the templates

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 [22] to find a maximal unique pattern set from the selected templates.

The morphable structures are then characterized to obtain their hardware performance and cost models to be used during custom instruction selection.

5 Experimental Results

In this section, we present experimental results to evaluate the benefits of our proposed methodology. We have selected a total of eight benchmarks from the MiBench embedded benchmark suite [8] as the standard application set. 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.

Table 1 shows the results obtained from the custom instruction identification process and pattern grouping. 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.

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

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

As can be observed from Table 1, a total of 82 templates can be used for custom instruction selection. 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.

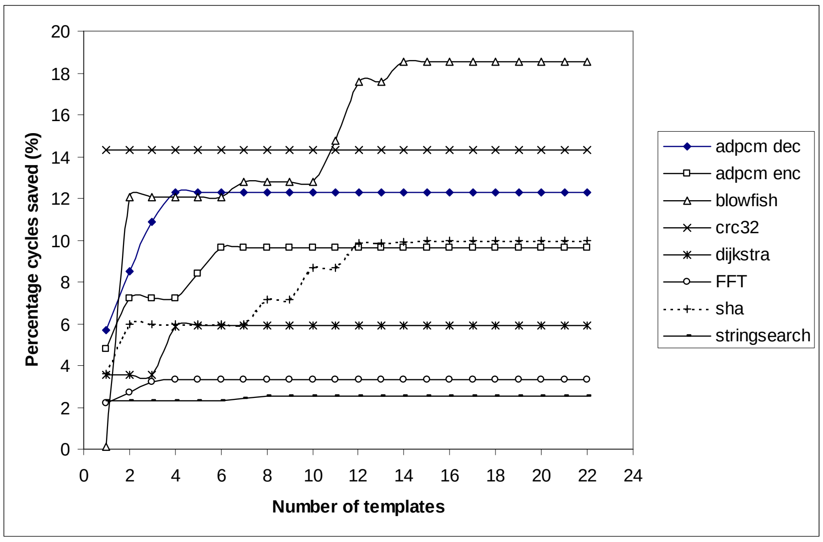


Fig. 6. Percentage cycles saved with varying number of selected templates used for template matching

Fig. 6 shows the percentage cycles saved that can be achieved with varying number of templates used for template matching. 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.

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

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 60 templates with highest gain values have been selected based on the approach described in Section 4.1.3. These templates consist of various pattern sizes (i.e. 2, 3, 4, 5, 6), which is necessary to accommodate to the different embedded applications. For example in Fig. 7, although the performance gain in most benchmark applications is contributed by small custom instructions (i.e. 2), larger custom instructions form a significant portion of the performance gain in certain benchmarks (i.e. sha).

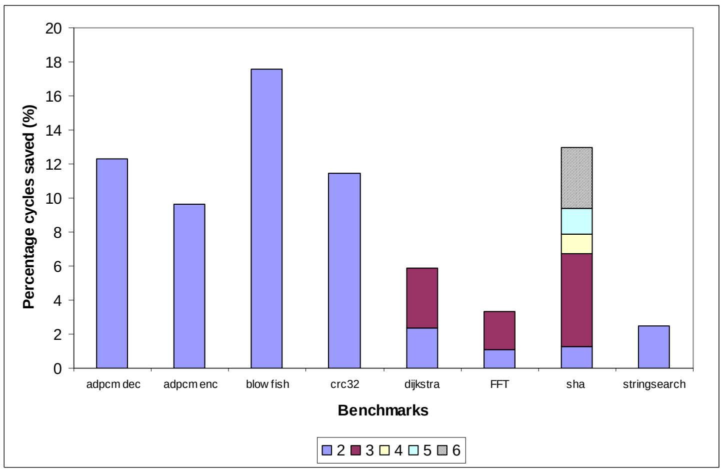


Fig. 7. Percentage cycles saved contributed by varying pattern sizes of the templates

Fig. 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. It can be observed from Fig. 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 60), provides comparable results with the application-centric approach. It is noteworthy that the proposed methodology executes much faster as it only performs the pattern enumeration process once. Moreover, as mentioned earlier, the employment of morphable structures dispenses the need for hardware syntheses flow in design exploration, and can give rise to area efficient implementations.

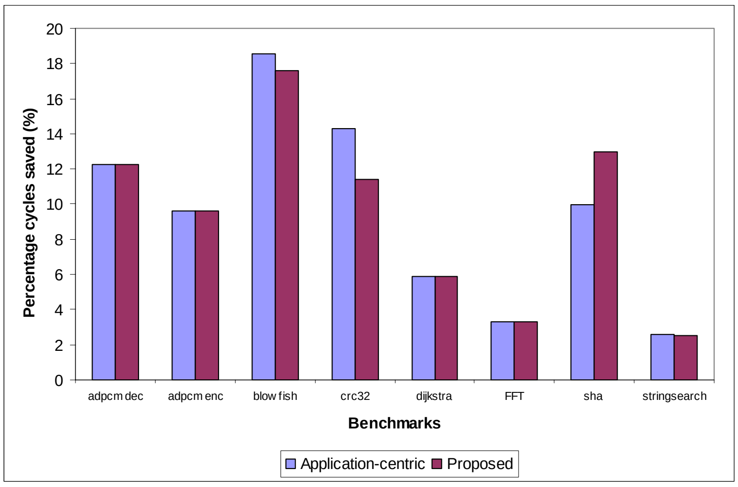


Fig. 8. Performance comparison of the proposed method with an application-centric approach

Table 2 shows the number of morphable structures required for each application and the total pattern sizes. As can be observed, the average number of morphable structures and the average number of operations for the eight applications is only 5.125 and 18.75 respectively. The maximum number of morphable structures is 8 with 45 operations, which is required by the sha application. These results imply that the reconfigurable area on the RISP can be kept small to cater to efficient custom instruction implementations.

Table 2. The number of morphable structures and the total pattern sizes for each application

Benchmarks	**Required number of morphable structures**	**Sum of Pattern Sizes of the morphable structures**
adpcm dec	5	19
adpcm enc	6	18
blowfish	6	21
crc32	2	8
dijkstra	5	14
FFT	6	15
sha	8	45
stringsearch	3	10
Average	**5.125**	**18.75**

Fig. 8 shows the 23 morphable structures that have been constructed and the corresponding applications that they cater to.

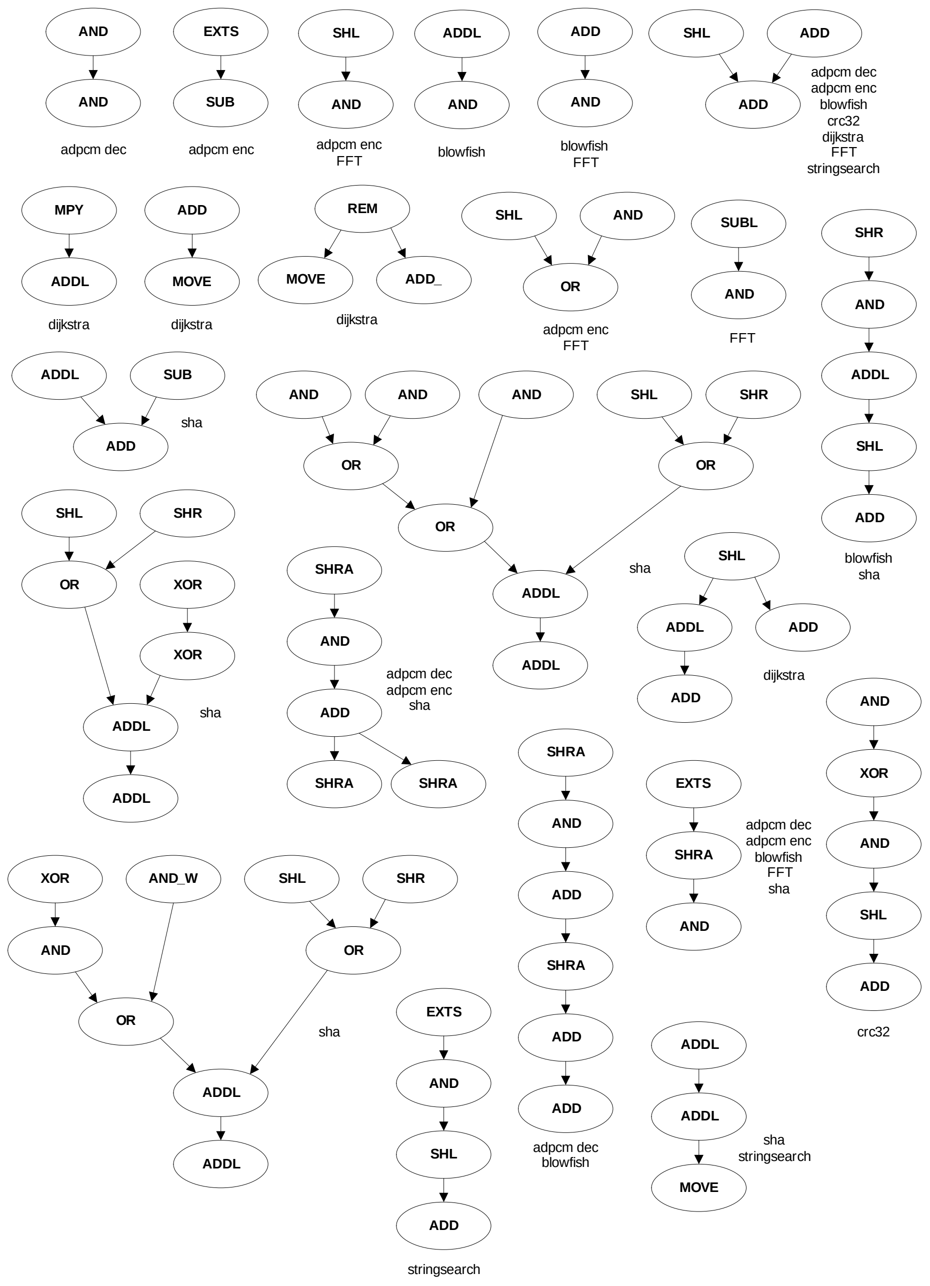


Fig. 9. The set morphable structures constructed through the proposed methodology

6 Conclusion

We have proposed a methodology for instruction set customization on RISPs that uses morphable structures. The advantage of using morphable structures stems from the availability of a predefine set of morphable structures that can lead to rapid design exploration without a time-consuming hardware synthesis flow to evaluate the feasibility of the custom instruction candidates. In addition, the reconfigurable logic space to accommodate the morphable structures can also be minimized, as the number of morphable structures that are specific to a particular application is very small. The experimental results show that 23 predefined morphable structures can sufficiently cater to any application in a set of eight MiBench benchmarks, and the average number of morphable structures per application is only 5.125 in order to achieve high performance gain. Future work includes validation of the methodology on a larger standard application set, and defining more effective criteria for the construction of morphable structures.

References

1. Dutt, N., Choi, K.: Configurable Processors for Embedded Computing, Computer, Vol. 36, No. 1, (2003) 120-123
2. Henkel, J.: Closing the SoC Design Gap, IEEE Computer, Vol. 36, No. 9 (2003) 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) 847-862
6. 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) 256-261
7. Ohlrich, M., Ebeling, C., Ginting, E., Sather, L.: SubGemini: Identifying Subcircuits Using a Fast Subgraph Isomorphism Algorithm, Proceedings of the 30th ACM/IEEE Design Automation Conference (1993) 31-37
8. 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) 3-14
9. Wirthlin, M.J., Hutchings, B.L.: A Dynamic Instruction Set Computer, Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (1995) 99-107
10. 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) 92-101
11. Flaherty, N.: On the Chip or On the Fly, IEE Review, Vol. 50, No. 9 (2004) 48-51
12. Xilinx Platform FPGAs: http://www.xilinx.com
13. Altera Nios Soft Core Embedded Processor: http://www.altera.com
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) 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) 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) 183-189
18. Sassone, P.G., Wills, D.S.: On the Extraction and Analysis of Prevalent Dataflow Patterns, Proceedings of the Workshop on Workload Characterization (2004)
19. Spadini, F., Fertig, M., Patel, S.: Charaterization of Repeating Dynamic Code Fragments, Technical Report CRHC-02-09, University of Illinois, Urbana-Champaign (2002)
20. Trimaran: An Infrastructure for Research in Instruction-Level Parallelism: http://www.trimaran.org
21. Yu, P., Mitra, T.: Characterizing Embedded Applications for Instruction-Set Extensible Processors, Proceedings of the 41st IEEE/ACM on Design Automation Conference (2004) 723-728
22. 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) 1172-1177