Rapid Design of Area-Efficient Custom Instructions for Reconfigurable Embedded Processing

Siew-Kei Lam[*] *and Thambipillai Srikanthan*[+]

Centre for High Performance Embedded Systems
Nanyang Technological University
50 Nanyang Drive, Research TechnoPlaza
3rd Storey, BorderX Block,
SINGAPORE 637553

Tel. No.: 65-67906643
Fax No.: 65-67920774
Email: [*]assklam@ntu.edu.sg, [+]astsrikan@ntu.edu.sg

Abstract

RISPs (Reconfigurable Instruction Set Processors) are increasingly becoming popular as they can be customized to meet design constraints. However, existing instruction set customization methodologies do not lend well for mapping custom instructions on to commercial FPGA architectures. In this paper, we propose a design exploration framework that provides for rapid identification of a reduced set of profitable custom instructions and their area costs on commercial architectures without the need for time consuming hardware synthesis process. A novel clustering strategy is used to estimate the utilization of the LUT (Look-Up Table) based FPGAs for the chosen custom instructions. Our investigations show that the area costs computations using the proposed hardware estimation technique on 20 custom instructions are shown to be within 8% of those obtained using hardware synthesis. A systematic approach has been adopted to select the most profitable custom instruction candidates. Our investigations show that this leads to notable reduction in the number of custom instructions with only marginal degradation in performance. Simulations based on domain-specific application sets from the MiBench and MediaBench benchmark suites show that on average, more than 25% area utilization efficiency (performance/area) can be achieved with the proposed technique.

Keywords: Area estimation, design exploration, FPGA, look-up table, reconfigurable logic

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 flexibility to meet the shrinking TTM (Time-To-Market) window. In recent years, configurable processors [1]-[4] have emerged to offer the possibility of extending the instruction set for a specific application by introducing custom functional units within the processor architecture. This provides an efficient mechanism to meet the growing performance and TTM demands. While configurable processors have been proven successful for features sizes below 90nm, rising developing costs for ASIC designs tend to favor reconfigurable approaches [5].

A RISP (Reconfigurable Instruction Set Processor) consists of a microprocessor core that is tightly coupled with a RFU (Reconfigurable Functional Unit) [6]. Commercially available reconfigurable processors include the Altera Nios II [7], Xilinx MicroBlaze [8], and Stretch [9] processors. As opposed to loosely couple schemes where data is communicated between the microprocessor and RFU through a shared memory, the tightly coupled scheme employs the internal register files for data transfer. Similar to configurable processors, the RISP facilitates critical parts of the application to be implemented in hardware using a specialized instruction set.

It has been shown that circuits implemented on an FPGA (Field Programmable Gate Array) are about 3 to 5 times slower, and about 35 times larger than the equivalent standard-cell implementation [10]. However, FPGAs are becoming more popular than their ASIC counterparts as the increasing NRE costs of ASIC begin to outweigh the per-unit-cost of FPGAs for high-volume applications [11]. This is corroborated by the increasing adoption of re-configurable technologies such as FPGAs in high-volume designs [12]. In addition, recent FPGA architectures are often viewed as SoC (System-On-a-Chip) designs as they incorporate a large range of IP (Intellectual Property) cores. It has been projected that by 2010, more than 40% of all FPGA designs will contain a microprocessor [13]. Logic suppliers are also driven towards embedding FPGA cores in SoC designs to address TTM and mitigate design risk issues [14].

Hence, it is envisioned that RISPs will play an important role in future embedded SoC platforms due to its promising ability to overcome the technological and market challenges [15]. Even though RISPs has lower performance, area and power efficiency than its configurable counterparts [1][3], the design flexibility of RISPs in the presence of reconfigurable logic leads to off-the-shelf products that can be customized for each application. This eliminates the need for ASIC tape-out for each design, thereby eliminating the need to manage exorbitant NRE (Non-Recurring Engineering) costs of configurable processors. This is increasingly preferred by designers who develop products for uncertain markets and shorter product life cycles. For example, the design flexibility of reconfigurable logic is especially attractive for applications in ubiquitous computing with evolving standards, which require frequent functionality updates [16]. The major challenges to increase the proliferation of RISPs lie in the development of supporting compilation and computer-aided design tools that enable rapid design exploration and efficient mapping of applications on such platforms [17].

This paper presents a framework that enables rapid design exploration for RISPs, which incorporate reconfigurable structures that are similar to commercially available technologies (i.e. LUT-based FPGAs with coarse-grained arithmetic units). In particular, the proposed framework can effectively select custom instructions that maximize the area utilization of the reconfigurable space without compromising on the performance gain. We envisage that the reconfigurable space available for custom instructions will be limited in future embedded SoC designs, particularly due to tighter design constraints of embedded systems. Hence, we believe that strategies for reducing the area utilization of FPGAs will be an important step for satisfying the design constraints of systems consisting of reconfigurable space. The proposed framework can also be used for determining the optimal size of FPGAs to be embedded in cost and power sensitive SoC platforms.

The remainder of this paper is organized as follows: In the following section, we discuss some existing work and state the main contributions of this paper. Section 2 describes the architecture model of the RFU. In Section 3 and Section 4, we describe the main stages of the proposed framework. Section 5 provides experimental results for a set of application domains to demonstrate that considerable area savings can be achieved using the proposed framework with marginal loss in performance gain. Finally we conclude in Section 6.

1.1. Related Work

Instruction set customization is defined as a process to automatically generate custom instructions from an application in order to meet certain design objectives. Existing work in instruction set customization generally consists of two steps: 1) Custom instruction identification and 2) Custom instruction selection.

Custom instruction identification can be loosely described as a process of detecting a group 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). This step generates a set of custom instruction candidates, which will be evaluated for custom instruction implementation. In [18], an approach that combines template matching and generation have been proposed to identify sub-graphs based on recurring patterns. Other approaches [19][20] rely on heuristics to identify good custom instruction candidates while discarding less promising ones. The pattern enumeration method proposed in [21] 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.). Other pattern enumeration approaches for custom instruction identification have been presented in [22][23][24].

Custom instruction selection evaluates the custom instruction candidates in terms of their performance, area or power, and selects a subset of them that meets the design constraints. In [18], a covering algorithm was presented to select a minimal set of templates that maximizes the number of covered nodes. The templates are custom instruction candidates that are derived from the custom instruction identification process. The authors analyzed the trade-off between the number of templates and the percentage of node coverage. It was observed that increasing the number of templates in the covering algorithm will lead to notable increase in the number of covered nodes only up to a certain point. After which, employing more templates in the covering algorithm will not significantly impact the number of nodes covered. This implies that selecting larger number of custom instruction candidates may not necessary lead to better performance gain. Although this is an interesting observation, the work in [18] however, have not studied the effect on the actual hardware resources that is incurred when varying number of templates are selected.

In order to facilitate effective custom instruction selection, rapid design exploration must be undertaken without delaying the short TTM requirements for embedded systems. Rapid design exploration can be achieved with the presence of a fast and accurate method to estimate the performance-cost mapping of custom instructions on hardware. While previously reported design flows for instruction set customization have focused on efficient algorithms for custom instruction identification and selection, they do not incorporate an effective technique for area-time estimation that takes into account the architectural constraints of commercial FPGAs. For example, the estimation process in [19][21][25] is obtained by pre-computing the area-time of the custom instruction operations using standard-cell design tools. The area and delay of a custom instruction is then derived by summing up the pre-computed area-time values of the corresponding operations. In a similar manner, the delay estimation strategy in [23] predicts the relative speedup of the custom instructions on FPGA by utilizing a rough approximation of the throughput of each instruction. While these approaches may provide reasonable estimations for standard-cell implementations, they do not lend themselves well towards FPGA estimations. This is due to the fact that these methods do not take into consideration FPGA optimization strategies that maximize the resource utilization of the programmable logic structures. Other reported design flows (e.g. [20]) incorporates a hardware synthesis flow to facilitate the selection of custom instruction candidates that maximizes performance under a given area constraint.

In this paper, we incorporate high-level FPGA area estimation in the design flow to facilitate rapid design exploration for RISPs. The estimation is directly performed on the high-level algorithmic representations of an application (e.g. Data-Control Flow Graphs, C-language, Matlab, etc.) without the need for time consuming hardware design entry and implementation. It is worth mentioning that high level estimation techniques differ from existing technology mapping approaches for area-time FPGA

optimizations (e.g. [26][27]) as the latter relies on the availability of gate-level representations of the applications.
The work in [28] estimates the FPGA data-path area by using a formula, which is a function of the operator and register properties that are derived from the RTL code (generated from MATLAB). The number of CLBs (Configurable Logic Blocks) that corresponds to the operators is obtained by pre-characterizing the area that is consumed by each operator type and size. The area estimation error is within 16% of those reported by the commercial FPGA implementation tools. The work in [29] derives area-time estimation from a DFG that is generated from an execution trace (obtained from simulating a MATLAB program), which contains information on the type and frequency of the operations. A FPGA performance model is used to estimate the area-time of the operations in the DFG. The performance model incorporates information of the operations which includes the characterized FPGA area-time measures. Accuracy of the estimation is within 10% of actual implementations.

The authors in [30] presented a two-level model to estimate the area of System-C designs. The high-level model analyzes the System-C description and estimates the number of intermediate variables. The low-level model substitutes these high-level variables into a set of equations to estimate the number of LUTs (Look-Up Tables) and FFs (Flip Flops). The proposed models must be re-tuned for a new set of benchmarks, when tool changes and for different target devices. For the applications and models considered, the authors reported an average error of about 17% for the LUT estimation.

The work presented in [31] estimates the FPGA data-path area (in terms of LUTs) from a DFG that is generated from high-level SA-C codes. The estimation method relies on a formula that is derived from characterizing the resource consumption of all DFG nodes. In order to take into account some synthesis optimizations in the estimation for improved accuracy, some heuristics are employed on patterns that are frequently optimized by the synthesis tool. However the work does not consider more complex optimizations that aim to maximize the FPGA resource utilization. The area estimation error is within 5%.

A recently reported work in [32] performs area-delay estimations of the RTL solution using a two-step approach. In the first step, a structural exploration is performed to obtain several RTL solutions. In the second step, the area-time estimation of mapping the RTL solutions is undertaken. The physical mapping estimation relies on a FPGA characterization file for a target device. The information of the FPGA characterization file is obtained from the data-sheet of the target device, and from synthesis of basic operators. Experimental results performed on FPGA devices from different vendors reported an average area estimation error of 18%.

Recently, run-time reconfiguration has been investigated for cost effective realizations on FPGA based RISPs [33]-[35]. These works have demonstrated the benefits of runtime reconfiguration on the JPEG and H.264 encoder/decoder. The feasibility of runtime reconfiguration on RISPs depend largely on the type of application and the ability of the compiler to extract custom instructions that can mitigate the high reconfiguration overhead of existing FPGA architectures. For example, the DISC (Dynamic Instruction Set Computer) processor proposed in [36] requires a reconfiguration time that is projected to contribute up to 16% of an application’s total execution time. The Stretch S6000 processor requires 20 µs to change an instruction on their proprietary programmable logic [9]. Partial reconfiguration on the Xilinx Virtex FPGA is accomplished in the order of milliseconds [35]. This high reconfiguration overhead could suppress the benefits of dynamically reusing the FPGA hardware resources, if the reconfiguration overhead cannot be compensated by the speedup obtained from the acceleration of custom instructions.

1.2. Main Contributions

This paper presents a rapid design exploration framework to select custom instructions for RISPs. Unlike previously reported approaches that are targeted towards the eventual porting of a configurable processor (e.g. Xtensa processor [20]) with custom instructions to ASIC, our approach targets commercial fine grained reconfigurable architectures (i.e. Xilinx). We believe that appropriate design methodologies will pave the way for the emergence of off-the-shelf processors that consist of fine grained FPGA architectures to facilitate custom instruction acceleration. As opposed to the methods in [37][38], the proposed techniques targets reconfigurable structures that are similar to commercially available technologies (i.e.

LUT-based FPGAs with coarse-grained arithmetic units), and hence they can be more readily integrated with existing hardware synthesis tools. The contributions in this work include the following:

1. We will show that while a set of domain specific applications can exhibit large number of custom instruction instances, only a fraction of it is required for evaluation in the custom instruction selection process. Although the results in this paper are based on domain-specific application sets, the framework can also be adopted for application-centric solutions.
2. In order to facilitate rapid design exploration, a novel clustering strategy is proposed to estimate the area utilization of the RFU without the need for lengthy hardware synthesis such as that required in [20]. In contrast to existing high-level estimation methods [28]-[32], the proposed technique takes into account the FPGA architectural constraints and synthesis optimizations for maximizing the utilization of the FPGA resources. It also does not rely on a pre-characterization step, which limits the scope of representing all possible combinations of the design under examination. In addition, unlike the methods in [28]-[31], the proposed area estimation strategy is performed on the intermediate representation of ANSI C applications, which are commonly employed in embedded applications. It is noteworthy that the proposed strategy are targeted towards state-of-the-art FPGA architectures (e.g. the Xilinx Virtex devices), and can be easily extended for newer and future FPGA devices that are likely to incorporate similar programmable logic elements.
3. The rapid estimation strategy enables the use of a simple approach for selecting a reduced set of candidates that will lead to high area-time efficiency. When compared to existing commercial tools (e.g. Mimosys Clarity [39]) that are capable of automatically identifying only a single performance-optimal solution set, the proposed framework facilitates the exploration of the design space for selecting custom instructions that meets the area-time constraints of the application. We show that the proposed approach can achieve a significant area reduction of over 27% with an efficiency (performance/area) gain of over 25% for a wide range of applications.

2. Target Architecture Model

The target architecture model of a RISP is shown in Figure 1(a), which is a four-wide VLIW (Very Long Instruction Word) architecture that has been extended with a RFU for implementing custom instructions. The RFU obtains the input data from the register file, and outputs the results to the interconnection network, which facilitate the sharing of register files between the functional units and custom logic.

Figure 1(b) illustrates the proposed FPGA-based RFU, which consists of a 2D array of programmable logic elements that are interconnected by the switches and routing bus. Each logic element is a simplified version of a logic element that is found in the commercially available Xilinx FPGA devices [40]. As illustrated in Figure 1(c), each logic element is composed of a K-input LUT (K-LUT) and fast carry-logic (K is 4 in the example). The fine-grained programmable LUTs allow any function of up to K inputs to be implemented, providing for generic logic realization. The fast carry-logic aims to speed up carry-based computations, such as addition, parity, etc. [41]. The logic elements can be organized in groups of 32 to implement a sequence of 32-bit wide operations, which could comprise of logical, shift and addition/subtraction operations, with up to K operands.

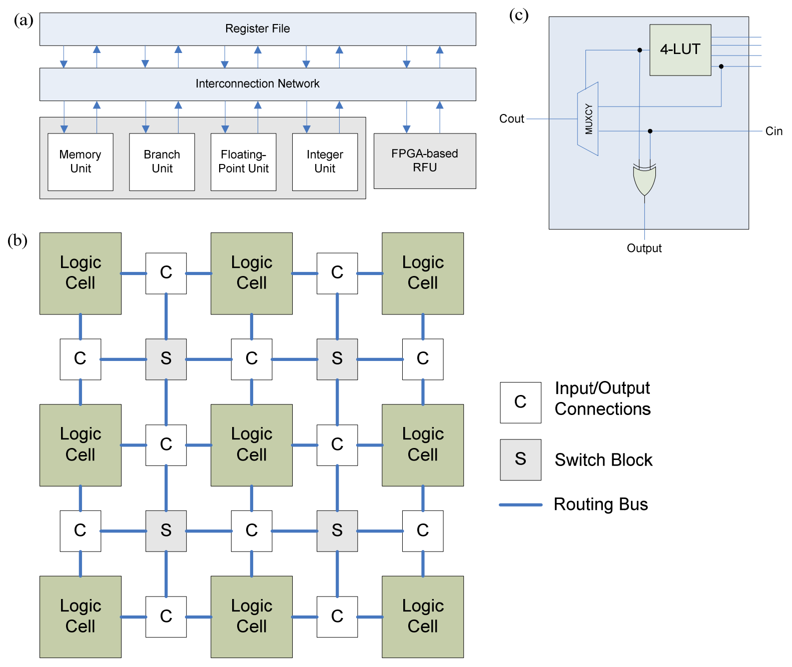


Figure 1: RISP model

Increasingly, commercial FPGAs are incorporating coarse-grained functional blocks to significantly improve the density, speed and power of the device. For example, the Xilinx Virtex device contains embedded 18x18-bit multiplier units. Although not shown in Figure 1, the RFU can also incorporate 32-bit CGAUs (Coarse-Grained Arithmetic Units) to facilitate high-speed complex arithmetic operations such as multiplication and division that cannot be efficiently mapped onto the fine-grained logic blocks.

In the following section, we will describe a methodology to explore the area-time trade-offs of the FPGA-based RFU in Figure 1. Given a set of applications, the proposed framework in the following section can rapidly estimate the number of logic elements and critical path delays to realize the custom instructions on the RFU. This enables one to quickly decide on a suitable RISP for a given problem, or determine a new set of custom instructions that meet the area-time constraints.

3. Design Exploration Framework

The proposed framework in Figure 2 consists of three stages: 1) Pattern Library Generation, 2) Template Library Generation, and 3) Hardware Generation.

In the Pattern Library Generation stage, a pattern enumeration method is combined with graph isomorphism to identify unique custom instruction instances from a set of embedded applications. Application profiling is also performed to compute the frequency of occurrences of the custom instruction instances bases on an input data-set. This is a one-time effort that is required for the second stage of the framework to select a set of templates from the custom instruction instances based on some user-specified criteria. These templates form a set of potential custom instruction candidates to be mapped on the RFU. In the Template Library Generation stage, the effects of the number of templates used on the potential performance gain are rapidly explored. This performance estimation is made possible through a pattern matching and selection process to identify a set of custom instructions (from the selected templates) based on the runtime application characteristics obtained from application profiling.

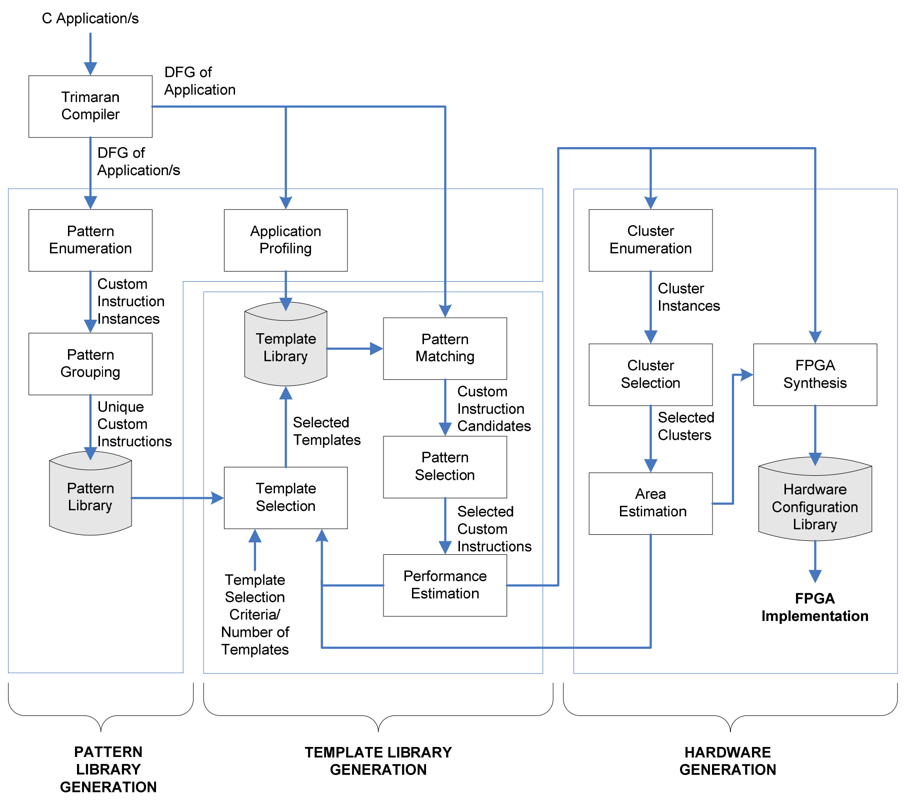
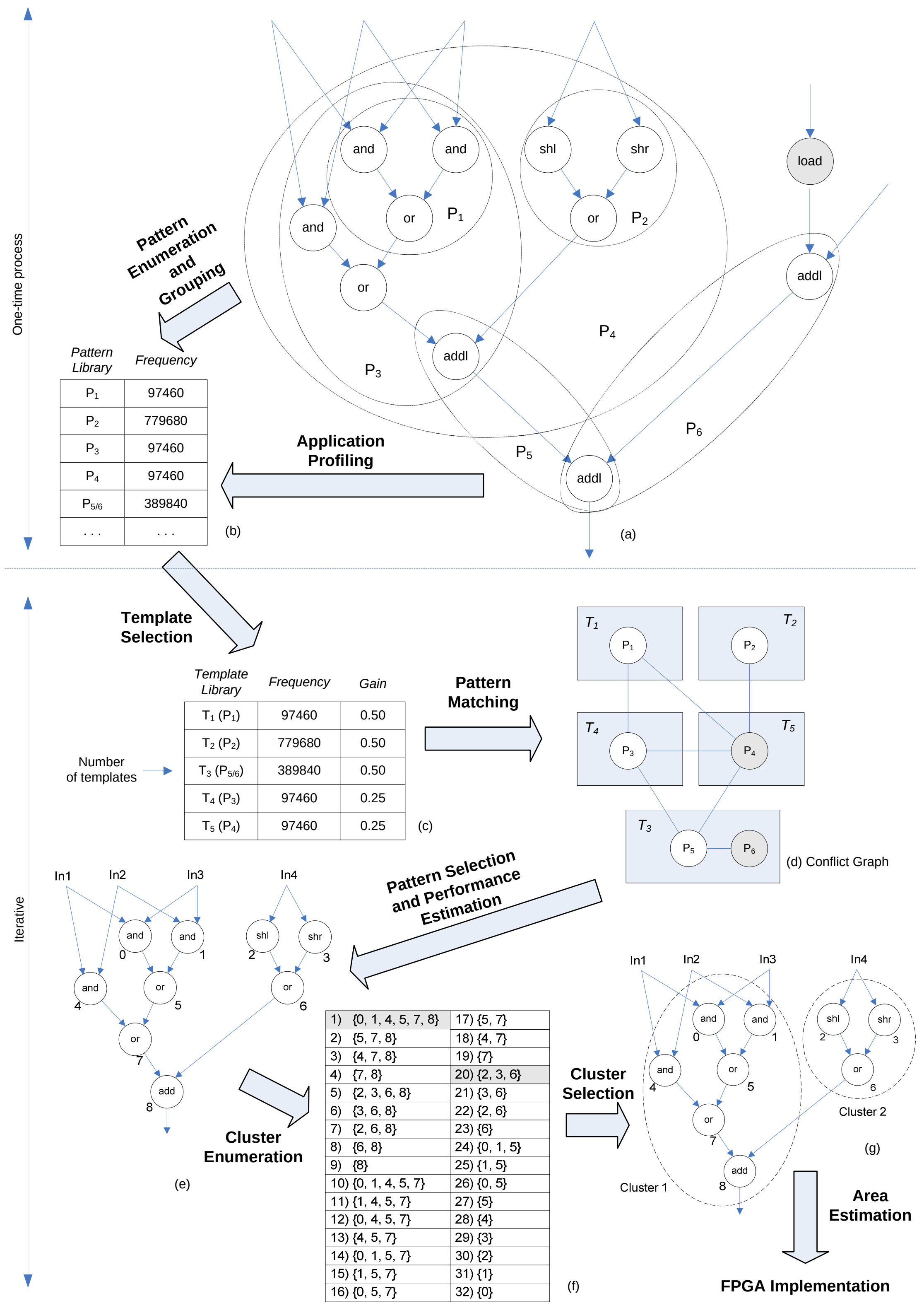


Figure 2: Overview of framework

In the Hardware Generation Stage, a cluster enumeration process for a given K value is performed to identify all the cluster instances from the selected custom instructions. Each cluster comprises of a custom instruction data-path that can be mapped onto a group of 32 LUTs or a CGAU. A heuristic based approach is then employed to select a set of clusters that aim to minimize the reconfigurable resources. An area estimation model is used to evaluate the hardware requirements of the selected clusters when they are realized on the RFU. If the selected clusters do not meet the constraints of the system, a new set of templates is then obtained and the design exploration repeats until the constraints are met. It is noteworthy that the design exploration process is performed without actual hardware synthesis. Finally, the hardware configurations are generated using FPGA synthesis tools.

In the following sections, we will describe the steps in the framework based on the example in Figure 3.



Pattern Library	Frequency
P_1	97460
P_2	779680
P_3	97460
P_4	97460
$P_{5/6}$	389840
. . .	. . .

Template Library	Frequency	Gain
T_1 (P_1)	97460	0.50
T_2 (P_2)	779680	0.50
T_3 ($P_{5/6}$)	389840	0.50
T_4 (P_3)	97460	0.25
T_5 (P_4)	97460	0.25

1) {0, 1, 4, 5, 7, 8}	17) {5, 7}
2) {5, 7, 8}	18) {4, 7}
3) {4, 7, 8}	19) {7}
4) {7, 8}	20) {2, 3, 6}
5) {2, 3, 6, 8}	21) {3, 6}
6) {3, 6, 8}	22) {2, 6}
7) {2, 6, 8}	23) {6}
8) {6, 8}	24) {0, 1, 5}
9) {8}	25) {1, 5}
10) {0, 1, 4, 5, 7}	26) {0, 5}
11) {1, 4, 5, 7}	27) {5}
12) {0, 4, 5, 7}	28) {4}
13) {4, 5, 7}	29) {3}
14) {0, 1, 5, 7}	30) {2}
15) {1, 5, 7}	31) {1}
16) {0, 5, 7}	32) {0}

Figure 3: Example of instruction set customization with the proposed framework

3.1. Pattern Library Generation

In the first stage of the framework, a one-time effort is required to construct a pattern library that consists of a set of unique custom instruction patterns. Pattern Library Generation is divided into two steps: 1) Pattern enumeration, and 2) Pattern grouping.

3.1.1. Pattern Enumeration

The objective of this step is to enumerate the custom instruction pattern instances from the application's DFG. For example Figure 3(a) shows a DFG and the corresponding custom instruction pattern instances (P_1, P_2, …, P_6). Each of the pattern instances is a sub-graph of the DFG that satisfies a set of constraints that have been imposed in the pattern enumeration algorithm. Note that for simplicity, we have only shown six enumerated pattern instances in this example. The pattern instances are stored in the pattern library (Figure 3(b)) and their frequency of occurrences based on an input data-set is obtained from application profiling.

We have adopted the pattern enumeration algorithm in [21] to identify all the custom instruction pattern instances from the given application set. As mentioned earlier, the method in [21] 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 [43] IR (Intermediate Representation) for the enumeration process. 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. Including memory accesses in custom instructions can lead to non-deterministic latencies and increased complexity in the RFU [44]. In addition, custom instructions with floating-point operations often do not lead to notable speedup [25].
2. Each custom instruction instance must be a connected sub-graph as we assume the parallelism in custom instructions associated with disconnected sub-graphs are not exploited in the target architecture.
3. Maximum number of input ports is 5 and maximum number of output ports is 2. Previous work [25] has shown that input-output ports more than this range results in little performance gain. It is noteworthy that even if the actual number of input-output ports of the RFU is lesser than the imposed constraints, existing techniques

that exploits pipelining and multi-cycle register file access can be employed to efficiently map the custom instructions on the RFU [45]. For simplicity, we assume that pipelining and multi-cycle register file accesses are not supported in the target architecture.

4. Only convex sub-graphs are allowed in the custom instructions instance to ensure a feasible schedule exists when the sub-graph is collapsed into a custom instruction [21].
5. An operation that feeds an input to the custom instruction instance must execute before the first operation in that instance, to avoid dependency violation when the instance is realized as a custom instruction.

In the subsequent sections, we will describe efficient techniques for selecting custom instructions from the set of enumerated custom instruction instances. It is worth mentioning that more recent methods for pattern enumeration (e.g. [24]) can be readily incorporated into the proposed framework in order to accelerate the custom instruction identification process.

3.1.2. Pattern Grouping

The custom instruction pattern 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. We have used the graph isomorphism method in the vflib graph-matching library [46] for the pattern grouping process. Graph isomorphism is performed for each pattern instance in the pattern library with the rest of the pattern instances. For example, in Figure 3(b), pattern instances P_5 and P_6 are isomorphic, and hence they are grouped. The unique custom instruction patterns are stored in the pattern library to be used for subsequent stages of the framework.

The pattern enumeration, pattern grouping and application profiling process in our framework needs to be performed only once on a set of applications.

3.2. Template Library Generation

The contents of the pattern library form an initial set of templates, where a template refers to an instance of a computational pattern with repeated occurrence [18]. This initial set of templates is first stored in the template library. Design exploration is then performed to identify a reduced set of templates, which will be evaluated in the custom instruction selection process. Although it is desirable to limit the number of templates to reduce the hardware costs, we need to ensure that the resulting performance gain is not heavily compromised. As such, we employ a pattern matching and selection approach to estimate the performance gain when varying number of templates is used. The main tasks of the Template Library Generation stage is 1) Template selection, 2) Pattern matching and selection, and 3) Performance estimation.

3.2.1. Template Selection

We employ a heuristic approach for template selection, which accounts for the performance gain and area utilization of the custom instruction in hardware. Each pattern *p* in the pattern library is assigned a gain as shown in (1), where the speedup obtained by mapping *p* on hardware is calculated as shown in (2). $T_{SW}(p)$ denotes the number of clock cycles taken for the custom instruction *p* to run on a processor. $T_{HW}(p)$ denotes the number of clock cycles taken for the custom instruction *p* in hardware, and we estimate this by the length of the critical path in the custom instruction sub-graph. For example, $T_{HW}(p) = 5$ for custom instruction sub-graph *p* 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. *Pattern size(p)* denotes the size of the custom instruction *p* and is estimated by the number of primitive operations of *p*. If we assume that each operation for the custom instruction *p* takes 1 software clock cycle, then $T_{SW}(p) = \mathit{Pattern\ size}(p)$, and hence $\mathit{Gain}(p) = 1/T_{HW}(p)$.

$$\mathit{Gain}(p) = \frac{\mathit{Speedup}(p)}{\mathit{Pattern\ size}(p)} \qquad (1)$$

$$\mathit{Speedup}(p) = \frac{T_{SW}(p)}{T_{HW}(p)} \qquad (2)$$

The templates in the template library are then sorted in decreasing gain and varying range of templates (each range includes the templates with highest gain) is iteratively selected for performance evaluation. In the iteration example in Figure 3(c), five templates (T_1, T_2, …, T_5) that correspond to patterns (P_1, P_2, …, P_6) are selected for evaluation. Based on the performance estimation results, a range of templates that lead to insignificant performance degradation (compared to the case when all the templates are used for performance evaluation) will be selected and stored in the template library. The performance estimation is facilitated by pattern matching and selection.

3.2.2. Pattern Matching

The problem of pattern matching can be described as follows: Given an application DFG that is represented as a directed labeled graph $G_d(V,E)$ and a set of templates in the template library, where each template is a directed graph $T_i(V,E)$, find every sub-graph of G_d that is isomorphic to T_i. This problem is essentially equivalent to the sub-graph isomorphism problem, which is simplified due to the directed edges. We have used the vflib graph-matching library [46] to identify all the pattern matches in the application DFG.

3.2.3. Pattern Selection

We have adopted the technique presented in [47] for pattern selection, which is based on the conflict graph approach. A conflict graph is an undirected graph $G_u = (V,E)$, where each vertex $v \in V$ is a match that is a member of the template set associated with T_i for $1 \le i \le t$ and t is the number of templates in the template library with the highest gain as described in (1). We denote the template set associated with T_i as S_i. An edge $e \in E$ between two matches v_x and v_y signify that the matches have one or more nodes in common. The number of nodes in a match v_x is denoted as *size(v_x)*.

The algorithm first constructs a conflict graph from the template matches, and then iteratively computes the MIS (Maximum Independent Set) of each template set to select the custom instructions. The MIS of template set S_i, denoted as MIS_i is defined as the largest subset of vertices in S_i that are mutually non-adjacent. The adjacent matches of the MIS within each template set are temporarily removed.

In order to facilitate custom instruction selection, an objective function *w(MIS$_i$)* is computed in each iteration. We used the objective function in (3), where $v_x \in S_i$, to give preference to the selection of larger matches. The algorithm proceeds to select *MIS$_i$* with the largest objective function and the matches corresponding to the selected MIS are chosen as custom instructions. These matches and their adjacent neighbors are then permanently removed from the conflict graph. The rest of the matches are restored and the algorithm repeats until the conflict graph is empty.

$$w(MSI_i) = size(v_x) \qquad (3)$$

Figure 3(d) shows an example of a conflict graph and the patterns P_4 and P_6 are selected as custom instructions. It is worth mentioning that the conflict graph needs only to be built once using the full range of templates in the template library. In the subsequent iterations, where a reduced set of templates are used for pattern matching and selection, the conflict graph can be easily modified by temporarily removing the irrelevant nodes/edges.

3.2.4. Performance Evaluation

The estimated performance based on the selected custom instructions is calculated for varying number of templates used in pattern matching and selection. The performance estimation is reported in terms of percentage of *SCS* (Software Cycle Savings) for application *A*, which is computed as shown in (4), where p_i for i = 1 to n, represent the n custom instructions selected for the application *A*, $F(p_i)$ is the execution frequency of the custom instruction p_i in application *A*, and $T_{SW}(A)$ denotes the number of clock cycles of application *A*. The values of $F(p_i)$ and $T_{SW}(A)$ are obtained from application profiling.

$$SCS(A)\% = \frac{\sum_{i=1}^{n} T_{SW}(p_i) \times F(p_i)}{T_{SW}(A)} \times 100\% \qquad (4)$$

4. Hardware Generation

This stage incorporates a method to estimate the area costs and critical path delays of the selected custom instructions in the template library when they are realized on the FPGA-based RFU.

The hardware estimation process consists of the following steps 1) Cluster enumeration, 2) Cluster selection, and 3) Area Estimation. Steps 1) and 2) are illustrated in Figure 3, where Figure 3(e) shows an example custom instruction data-path. The cluster enumeration process decomposes the data-path into a list of clusters instances, where each instance represents a connected sub-graph that can be realized with a single K-LUT based logic element or a CGAU. Since the data width of the custom instruction data-path is 32-bits, each cluster can be essentially mapped to a set of 32 logic elements with the same configuration. As shown in Figure 3(f), there are 32 enumerated cluster instances for K = 4. A set of clusters is then selected to effectively cover the original data-path in order to meet a certain criteria. For example in Figure 3(g), clusters 1 and 20 are selected from the enumerated set such that the number of clusters required to cover the data-path is minimized. In this example, two 4-input LUTs will be required to realize the data-path in Figure 3(e). An area estimation model is then used to evaluate the hardware requirements for implementing the selected clusters on the RFU.

The steps discussed above facilitate rapid design exploration to identify a suitable RFU or to select a new set of templates with different selection criteria. The hardware estimation process does not require actual hardware synthesis as it directly maps the high-level operations of the custom instruction data-paths onto the FPGA. The high-level mapping is based on certain rule-sets that take into account the size of the LUT for mapping logical and relational operations, and the carry-logic architecture for mapping addition/subtraction operations. The validity of the proposed rules for the correct mapping of each cluster to a single logic element has been verified by implementing the clusters in VHDL and synthesizing them Xilinx ISE Foundation Version 6.1.03i [42].

It is worth mentioning that the proposed approaches in this paper are targeted towards FPGAs that consist of logic elements with LUT of any arbitrary input K and a carry-logic as shown in Figure 1(c). Hence, although the experiments in this paper is based on FPGA devices which consist of logic elements with 4-input LUT (similar to Xilinx Virtex-2 and Virtex-4 devices), the proposed approaches can be easily extended for newer and future FPGA architectures. For example, the recent Virtex-5 devices incorporates 6-input LUTs with similar carry-chain primitives that can be used to implement any 6-input function or two dual-output 5-input functions [48]. The proposed method can be used to estimate the hardware area-time of these devices by specifying the appropriate value of K (e.g. K = 5 or 6). Further optimization techniques can be incorporated to enable the mapping of two 5-input functions onto a single logic element in the Virtex-5 devices. It has also been predicted that future FPGAs are likely to incorporate similar programmable logic elements that incorporate K-LUTs and carry-logic [15].

4.1. Cluster Enumeration

The primitive operations of the Trimaran IR [43] can be categorized into 1) logical operations (e.g. *and*, *or*, *xor*), 2) shift operations (e.g. *shl*, *shr*) that shift the data to the left/right by a constant, and 3) arithmetic operations (e.g. *add*, *sub*, *mult*, *div*). We have adopted the pattern enumeration algorithm in [21] to identify all the cluster instances from the selected templates. The method employs a binary tree search approach that prunes unexplored sub-graphs from the search space if they violate a certain set of rules. Valid cluster instances, comprising of connected sub-graphs where each node is a primitive operation, must comply with a set of rules that enable them to be mapped onto the logic elements or CGAUs. We will describe these rules with the help of the example data-path in Figure 4(a), and Figure 4(b) that shows the implementation of an *add* operation using a programmable logic element similar to those found in the 4-input LUT based Xilinx Virtex device [40].

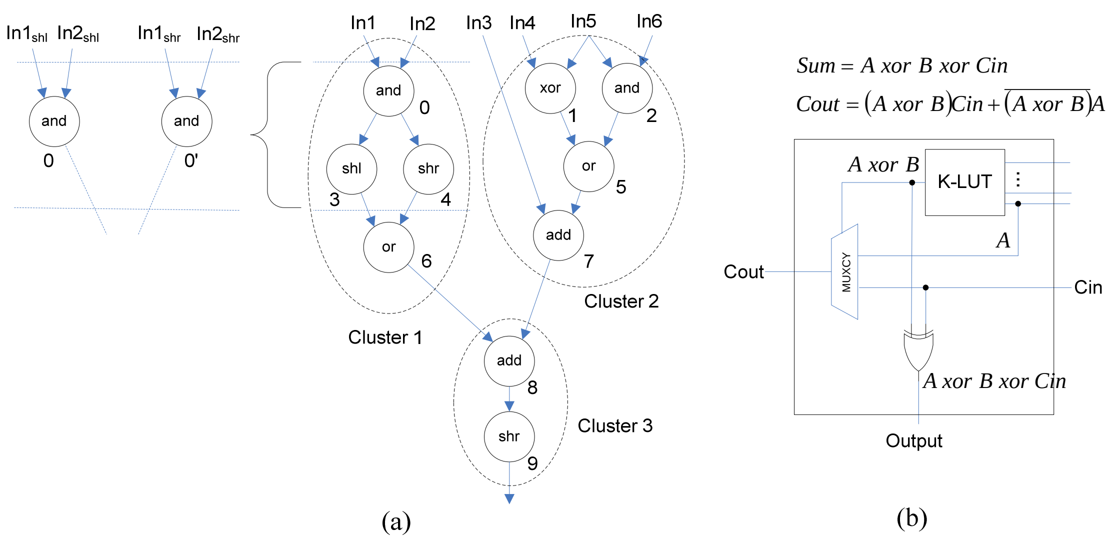


Figure 4: (a) Custom instruction with 3 clusters, and (b) implementing an *add* operation in a logic element

Two sets of rules are used to determine a valid cluster. The first set of rules is used to evaluate whether an operation can be included in the cluster during enumeration process. An operation can be included in a cluster if:

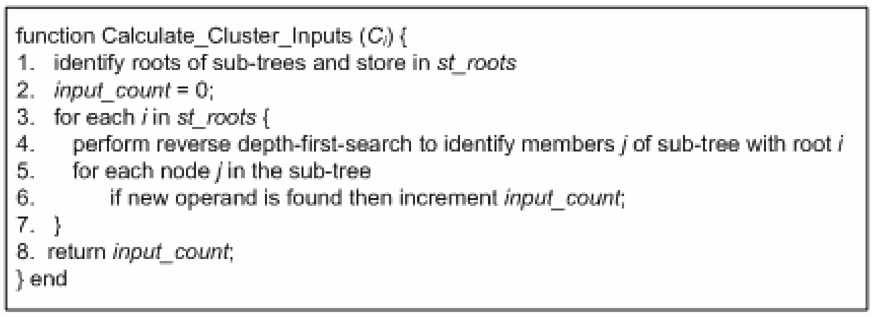
1. *It is a logical or shift operation and the cluster does not consist of any arithmetic operations* (e.g. Cluster 1 in Figure 4(a)). The programmable K-LUTs (K-input LUTs) can implement non-arithmetic functions of up to K inputs.
2. *It is a logical operation that is executed before a single add/sub operation in the cluster* (e.g. Cluster 2 in Figure 4(a)). The *add/sub* operation can be mapped onto the LUT-based logic elements to exploit the fast carry chains as shown in Figure 4(b) [49]. Since the output of the LUT is the partial sum ($A \oplus B$), all logical operations must be executed first to generate the required operands (i.e. *A* and *B*) for the *add* operation.
3. *It is a shift operation and the cluster contains a single add/sub operation* (e.g. Cluster 3 in Figure 4(a)). The shifted value of an addition/subtraction result by a constant can be realized by configuring the routing architecture to feed the suitable data range as inputs to another logic element.
4. *It is an add/sub operation and the cluster does not contain any other arithmetic operations*. A LUT-based logic element can only implement a single *add/sub* operation effectively by exploiting the fast carry chain. Other arithmetic operations (e.g. multiplication, division, etc.) are mapped to the CGAUs.

The second set of rules is used to evaluate the permissible inputs/outputs of the cluster:

1. *The total number of cluster inputs is at most K with only one output*. The maximum number of input ports and output ports for a LUT is K and 1 respectively.
2. *One of the inputs to the add/sub operation must be directly connected to an external input of the cluster* (e.g. Cluster 2 in Figure 4(a)). As shown in Figure 4(b), the carry-out (i.e. *Cout*) is selected from either the carry-in (i.e. *Cin* or an input operand (i.e. *A*) that is also fed directly to the input pin of the LUT.

If the cluster contains only logical and *add*/*sub* operations, the number of inputs can be easily derived by evaluating the external inputs to the cluster. For example in Figure 4(a), Cluster 2 and Cluster 3 consist of 4 inputs and 2 inputs respectively. However, clusters with shift operations must be evaluated differently. For example the hardware synthesis tool will not be able to map the two physical inputs (i.e. In1 and In2) of Cluster 1 in Figure 4(a) to only two LUT pins. The *shl* and *shr* operations will result in In1 and In2 being routed to four input pins in order to realize the required operands for node 6. To illustrate this, Figure 4(a) shows an equivalent representation of the *and-shl-shr* operations of the original data-path. It can be observed that node 0 can be duplicated in order to produce a pair of operands for node 6. The inputs to the duplicated and operations comprises of the shifted values of In1 and In2. It is evident from this representation that Cluster 1 requires 4 inputs (i.e. $In1_{shl}$, $In2_{shl}$, $In1_{shr}$, $In2_{shr}$) instead of 2.

Given a cluster in the form of a directed graph $C_i(V,E)$, the pseudo code in Figure 5 computes the number of inputs of the cluster and returns the result as *input_count*. The function identifies sub-trees in the cluster that are rooted at a shift operation or the output node. For example in Figure 4(a), the root nodes of Cluster 1 are nodes 3, 4, and 6. The function first identifies all the root nodes in the cluster C_i (line 1 in Figure 5) before finding the sub-tree members of the corresponding root nodes using the reverse depth-first-search algorithm (line 4). A sub-tree can only have one shift operation that must be a root node. Hence in Cluster 1, there are three sub-trees with the following node members: {3, 0}, {4, 0}, and {6}. Each of these sub-trees is evaluated independently and the variable *input_count* is incremented whenever a new operand is detected in one of the node members (lines 5-6 of Figure 5). In the example, two new operands will be detected for each of the sub-tree {3, 0} and {4, 0}. It is noteworthy that the approach discussed in this section can be easily adapted for different values of K.



```
function Calculate_Cluster_Inputs ($C_i$) {
1.  identify roots of sub-trees and store in st_roots
2.  input_count = 0;
3.  for each i in st_roots {
4.      perform reverse depth-first-search to identify members j of sub-tree with root i
5.      for each node j in the sub-tree
6.          if new operand is found then increment input_count;
7.  }
8.  return input_count;
} end
```

Figure 5: Pseudo code to compute number of inputs in a cluster

4.2. Cluster Selection

We have adopted the conflict graph approach described earlier to select a set of clusters to cover the selected custom instructions. The objective function in (3) is used to heuristically select clusters with large number of primitive operations, as we aim to minimize the required number of clusters of the RFU, which will indirectly maximize the hardware utilization.

4.3. Area Estimation

An area estimation model is used to rapidly evaluate the hardware area requirements of the selected clusters. We have employed the area model that is presented in [50] for a generic island-style FPGA architecture. The area of a RFU with K-input LUTs is shown in (5), where A_L^K and A_R^K corresponds to the area incurred by the logic elements and routing resources respectively.

$$A_{RFU}^K = A_L^K + A_R^K \quad (5)$$

The area of the logic elements consisting of K-input LUTs is shown in (6), where n_c corresponds to the number of clusters that are computed in the Cluster Selection step.

A_b represents the area to store 1 bit in the LUT and the term $A_b \cdot 2^K$ corresponds to the area required by the LUT as a K-LUT can store up to 2^K bits of information. A_f represents the remaining area of the logic and routing resources within a logic element.

$$A_L^K = n_c \cdot 32 \cdot \left(A_b \cdot 2^K + A_f\right) \quad (6)$$

The area model of the routing architecture is shown in (7). The area of each routing track must be considered in order to model the routing architecture area. Since, each routing track requires at least one bit of information to control the opening and closing of the programmable switch, the pitch of a routing track is approximated as the square root of the area required by a single bit (i.e. $\sqrt{A_b}$) [50]. The routing channel width is denoted as W_K.

$$A_R^K = n_c \cdot 32 \cdot \left(2 \cdot \sqrt{A_L^K} \cdot \sqrt{A_b} \cdot W_K + A_b \cdot W_K^2\right) \quad (7)$$

5. Experimental Results

We have evaluated the benefits of the proposed framework using sixteen benchmarks from the MiBench embedded benchmark [51] and MediaBench benchmark [52] suite. In this section, we provide experimental results for the three stages of the proposed framework: 1) Pattern Library Generation, 2) Template Library Generation, and 3) Hardware Generation. We will also report on the efficiency (performance/area) gain of the proposed approach.

5.1. Pattern Library Generation

Table 1 shows the results obtained from the Pattern Library Generation stage. The first two columns signify the corresponding domain and application names of the sixteen applications. The remaining columns report the number of custom instruction instances (derived from the pattern enumeration) and the number of unique patterns in the pattern library (after pattern grouping) for different application sets. The results in the third and forth columns are obtained using five domain-specific application sets (i.e. automotive-industrial, image, network, security and telecommunications). In the last two columns, results are obtained by using all sixteen applications as the application set. We denote this application set as the generic application set.

Domain	Application	Domain-Specific		Generic	
		CI Inst.	Unique Pat.	CI Inst.	Unique Pat.
Automotive/ Industrial	Basicmath Bitcount Qsort	121	51	2810	280
Image	Cjpeg Epic Mpeg2	446	133		
Network	Crc32 Dijkstra Patricia	36	26		
Security	BlowfishDec Pegwit RijndaelDec Sha	2156	102		
Telecomm	AdpcmDec AdpcmEnc FFT	51	24		

Table 1: Results of pattern library generation

It can be observed from Table 1 that most of the enumerated pattern instances are duplicates and can be grouped. For example, only about 10% and 5% of the custom instruction instances are unique in the generic and security application set respectively. The significant reduction in the custom instruction patterns after pattern grouping implies that custom instructions across various applications share common patterns. While it has been previously shown that domain-specific applications exhibit common dataflow sub-graph patterns [53], it is interesting to note that this property exists across custom instructions found in generic applications as well. The unique custom instruction patterns obtained from the pattern-grouping step are stored in the pattern library.

5.2. Template Library Generation

Figure 6 shows the performance estimation results for domain-specific and generic application sets.

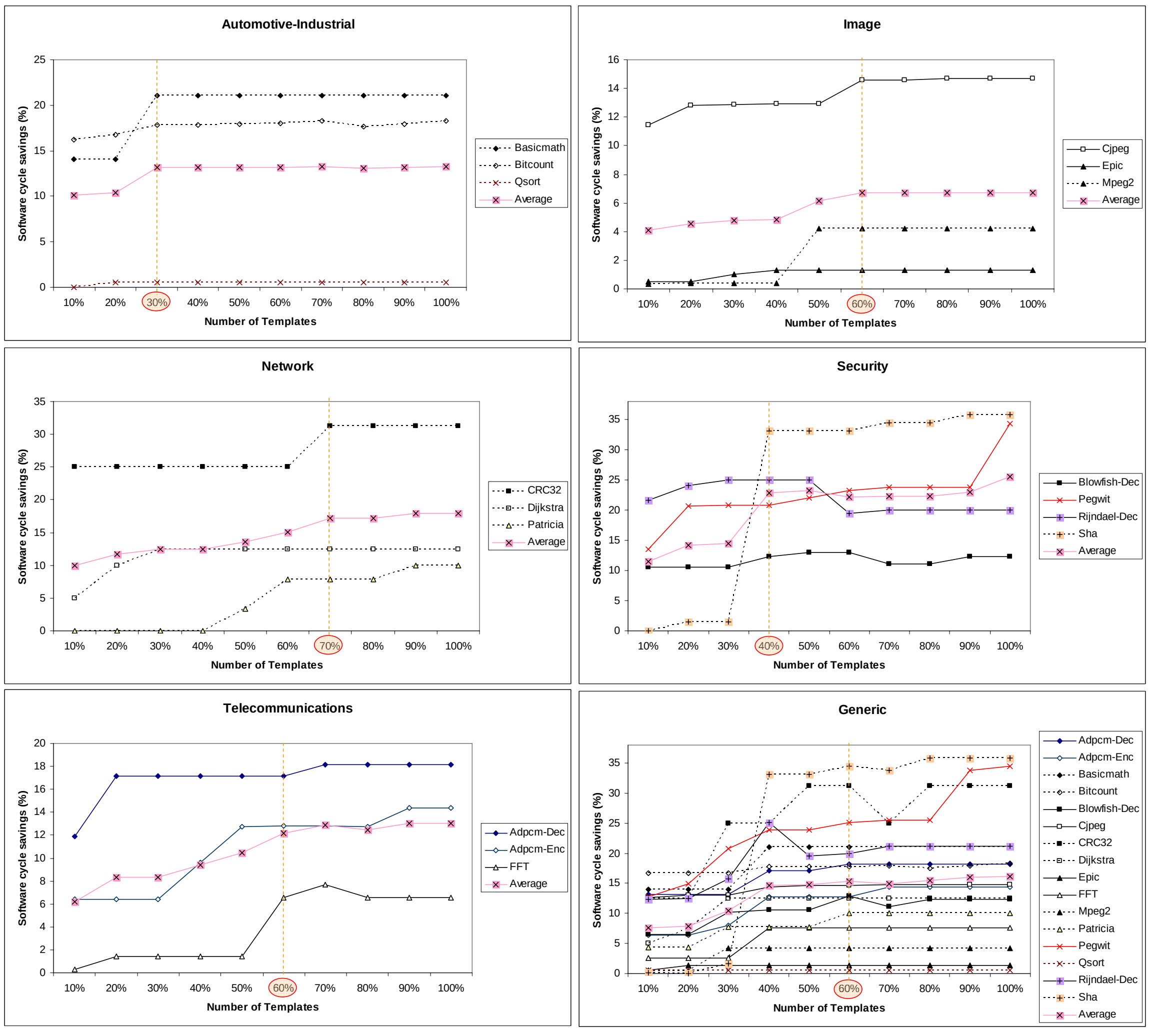


Figure 6: Performance estimation with varying number of templates

It is evident that increasing the number of templates for custom instruction selection will not lead to any notable gain after a certain point for each of the application sets. For example in the generic application set, selecting 60% of the original set of templates (168 templates) leads to negligible degradation in the average percentage software cycle savings for all sixteen applications. Similarly, selecting 30% (15 templates), 60% (80 templates), 70% (18 templates), 40% (41 templates) and 60% (14 templates) of the original set of templates in the automotive/industrial, image, network, security, and telecommunications application sets respectively, do not lead to significant degradation in the average percentage software cycle savings. Specifically, when compared to the case where the full set of templates are used for custom instruction selection, the average percentage software cycle savings difference for the automotive/industrial, image, network, security, telecommunications, and generic application sets are only 0.88%, 0.28%, 7.33%, 5.15%, 5.53%, and 2.92% respectively. These observations imply that it is possible to reduce the number of custom instructions for mapping onto the RFU without compromising heavily on the performance gain.

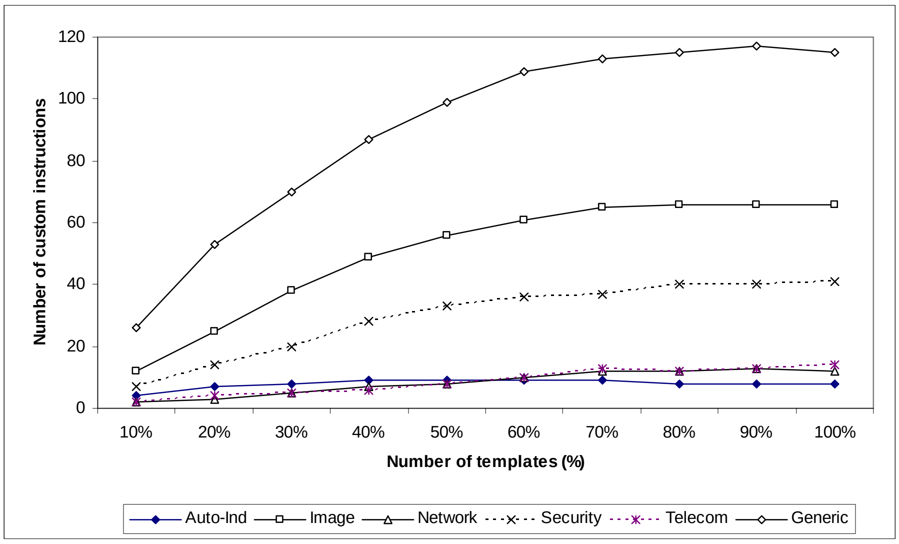


Figure 7: Number of selected custom instructions with varying number of templates

Figure 7 shows how the number of selected custom instruction varies with the number of templates used in the custom instruction selection process. It can be observed that the gradient of the plots are steeper for smaller number of templates and gradually flattens when higher number of templates is used for custom instruction selection. This implies that employing a reduced set of templates for custom instruction selection can lead to reduction in the area utilization of the FPGA-based RFU without compromising heavily on the performance gain.

5.3. Hardware Generation

Table 2 compares the hardware estimation with the synthesis results of equivalent hand-crafted designs for 20 custom instructions from the benchmark applications. The hand-crafted designs are realized using VHDL and synthesized with Xilinx ISE Foundation Version 6.1.03i [42] that is targeted for area optimization. The target FPGA device is the Xilinx Virtex-II 1000-bg575-4, which incorporates 4-input LUTs in the logic elements. Information pertaining to the custom instructions being evaluated (e.g. application source and size) are shown in columns 1 to 2.

For each custom instruction, we compare the number of logic elements obtained from the Cluster Generation stage (column 5) with the number of 4-input LUTs reported in the synthesis results (column 3). It can be observed that the automatic hardware generation results have an average of only 7.88% more hardware resources than the equivalent hand-crafted synthesized designs. These results are very encouraging as the number of logic elements is directly estimated from the high-level primitive operations and their dependences, without time-consuming hardware synthesis. It is noteworthy that in this paper, the hardware estimation process assumes that each cluster group requires 32 logic elements. In order to approximate the number of logic elements in each cluster group, the hardware estimation process can take into account the logic shift offsets and the effect of commonly used technology mapping techniques.

		Synthesized		**Estimated**	
Application	**Nodes**	**4-input LUT**	**Time (s)**	**Logic Elements**	**Time (ms)**
Basicmath	4	31	3	32	0.07
Bitcount	3	31	3	32	0.56
Bitcount	20	289	4	320	0.56
CRC32	5	30	3	32	0.19
Blowfish Dec	6	92	4	96	0.12
Blowfish Dec	6	96	4	96	0.12
Pegwit	5	125	3	128	2.76
Pegwit	18	152	6	160	2.76
Pegwit	3	60	3	64	2.76
Pegwit	9	161	4	160	2.76
Pegwit	5	90	4	96	2.76
Pegwit	6	48	3	96	2.76
Pegwit	7	97	3	96	2.76
Sha	3	96	4	96	0.15
Sha	8	96	5	96	0.15
Sha	4	32	3	32	0.15
Sha	10	64	4	64	0.15
Sha	9	96	4	96	0.15
Adpcm Enc	4	30	3	32	0.09
Adpcm Enc	3	32	3	32	0.09

Table 2: Comparing the hardware estimation results with synthesis results

The time taken for the hardware synthesis engine (column 4) and the average time to estimate each custom instruction (column 6) is also shown. The time taken for the estimation process is calculated as the average time to perform cluster enumeration and selection for each custom instruction data-path in a particular application. The synthesis and estimation process are both executed on a HP Workstation with two

2.66GHz processors and 2GB RAM. It is evident that the estimation process can be achieved significantly faster than the time taken to synthesize the custom instructions, hence facilitating rapid design exploration. It is noteworthy that the time required for designing and compiling the custom instructions using the commercial design flow have not been taken into account in column 4 of Table 2.

Based on (5), the estimated area of the RFU (in terms of μm^2) that is incurred due to the reduced set of templates that are chosen for custom instruction selection is shown in Figure 8. We have used the values for the constants A_b and A_f, and the average values for W_K that is provided in [54], where K = 4. The area of the RFU corresponding to the selected templates ('Reduced Templates') in Figure 6 is compared to the case when the full set of templates ('Full Templates') are used for custom instruction selection.

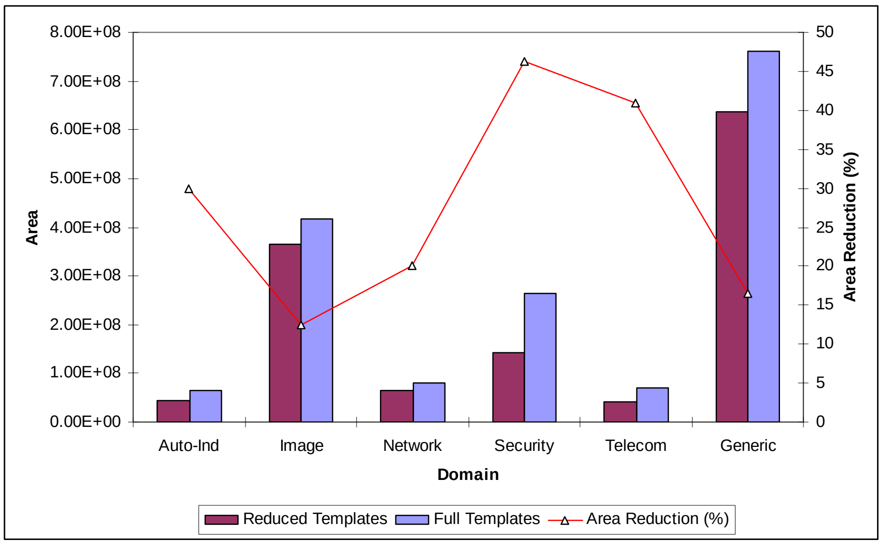


Figure 8: Estimated area when reduced set and full set of templates are used for template selection

It is evident that selecting a smaller set of templates can significantly reduce the number of required clusters to be implemented on the RFU. For example, compared to the case when the full set of templates are employed for custom instruction selection, the percentage reduction in the area when a reduced set of templates are selected are 30%, 12.4%, 20%, 46.3%, 40.9%, and 16.5% for the automotive/industrial, image, network, security, telecommunications, and generic application sets respectively. In particular, employing a reduced set of templates for template selection leads to an average area reduction of over 27%.

Finally, Table 3 reports on the efficiency that is obtained when the reduced set of templates and the full set of templates are used for template selection. The efficiency is calculated as shown in (8), where *ESCS* is the Effective Software Cycle Savings which accounts for the hardware execution delay of the custom instructions p_i ($T_C(p_i)$), where i = 1, … , n, and is calculated as shown in (9). $T_C(p_i)$ is obtained by calculating the number of clusters in the critical path of the custom instructions. For example, T_C of the custom instruction in Figure 4(a) is 2. Note that we have assumed that the delay of each cluster is equivalent to two software clock cycle executions. This is a reasonable assumption as the FPGA logic can generally execute at a significantly higher clock frequency than a commercially available soft processor core which is implemented on the same fabric [55].

$$Efficiency = \frac{ESCS}{A_{RFU}^{K}} \qquad (8)$$

$$ESCS = \sum_{i=1}^{n} F(p_i) \times (T_{SW}(p_i) - 2 \cdot T_C(p_i)) \qquad (9)$$

Domain	Reduced Templates			Full Templates		
	ESCS	**Area**	**Efficiency**	**ESCS**	**Area**	**Efficiency**
Automotive-Industrial	14664576	45182539.6	0.325	14814576	64546485.15	0.230
Image	1538018	364687641.1	0.004	1559449	416324829.2	0.004
Network	102859967	64546485.15	1.594	129603704	80683106.43	1.606
Security	19207719	142002267.3	0.135	22467162	264640589.1	0.085
Telecomm	5823638	41955215.34	0.139	6873517	71001133.66	0.097
Generic	146359636	635782878.68	0.230	176180091	761648524.7	0.231

Table 3: Comparing the efficiency when reduced set and full set of templates are used for template selection

It can be observed in Table 3 that the efficiency of the proposed method is higher than the case when the full set of templates is used in a number of application domains (with comparable results in the remaining domains). In particular, the average efficiency gain when a reduced set of templates is used is over 25%.

6. Conclusion

A design exploration framework for RISPs has been presented for the rapid selection of a minimal set of profitable custom instruction candidates. Simulations reveal that domain-specific applications share common custom instruction patterns, and hence domain-specific instruction set customization can lead to area efficient solutions. Experimental results for the generic, automotive/industrial, image, network, security and telecommunications application sets, show that the number of candidates for custom instruction selection can be significantly reduced by 30% to 70% with marginal degradation in the resulting performance gain. A novel clustering strategy for mapping the operations on the LUT based RFU is also proposed to estimate the reconfigurable resources for realizing the selected custom instructions. It has been shown that the runtime of the proposed estimation process is negligible when compared to the time taken for hardware synthesis. Experiments reveal that the estimated area costs are within 8% of those obtained using hardware synthesis. Finally, investigations based on domain-specific application sets from the MiBench and MediaBench benchmark suites show that the design exploration framework can lead to an average area reduction of 27%, and an average efficiency gain of over 25%.

References

[1] ARC Configurable Processor. Available: http://www.arc.com/

[2] N. Dutt and K. Choi, "Configurable Processors for Embedded Computing", *Computer*, Vol. 36, No. 1, January 2003, pp.120-123

[3] R.E. Gonzalez, "Xtensa: A Configurable and Extensible Processor", *IEEE Micro*, Vol. 20, No. 2, March-April 2000, pp. 60-70

[4] J. Henkel, "Closing the SoC Design Gap", *Computer*, Vol. 36, No. 9, September 2003, pp. 119-121

[5] N. Flaherty, "On the Chip or On the Fly", *IEE Review*, Vol. 50, No. 9, September 2004, pp. 48-51

[6] F. Barat, R. Lauwereins and G. Deconinck, "Reconfigurable Instruction Set Processors from a Hardware/Software Perspective", *IEEE Transactions on Software Engineering*, Vol. 28, No. 9, September 2002, pp. 847-862

[7] Altera: NIOS II Processors. Available: http://www.altera.com/products/ip/processors/-nios2/ni2-index.html

[8] Xilinx Platform FPGAs. Available: http://www.xilinx.com

[9] Video/Imaging Design Line, “Analysis: Stretch's Second-Gen Configurable Processor”, March 2007, Available: http://www.videsignline.com/howto/-videoprocessing/201311209

[10] I. Kuon and J. Rose, “Measuring the Gap between FPGAs and ASICs”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 26, No. 2, February 2007, pp. 203-215

[11] P. Garcia, K. Compton, M. Schulte, E. Blem and W. Fu, “An Overview of Reconfigurable Hardware in Embedded Systems”, *EURASIP Journal on Embedded Systems*, Vol. 2006, pp. 1–19

[12] P. Lysaght and P.A. Subrahmanyam, "Guest Editors' Introduction: Advances in Configurable Computing", *IEEE Design & Test of Computers*, Vol. 22, No. 2, March-April 2005, pp. 85-89

[13] W. Marx and V. Aggarwal, “FPGAs Are Everywhere – In Design, Test & Control”, NI Developer Zone, June 2008. Available: http://zone.ni.com/devzone/cda/pub/p/id/401

[14] Y. Tanurhan, “Logic Suppliers Seek Ways to Embed FPGAs”, EE Times India, March 2001. Available: http://www.eetindia.co.in/ART_8800387951_1800009_TA_7f011aab.HTM

[15] B. Zeidman, “The Future of Programmable Logic”, Embedded.com, 2003. Available: http://www.embedded.com/columns/technicalinsights/15201141?_requestid=477944

[16] M. Glesner, T. Hollstein, L.S. Indrusiak, P. Zipf, T. Pionteck, M. Petrov, H. Zimmer and T. Murgan, “Reconfigurable Platforms for Ubiquitous Computing”, *Proceedings of the 1st conference on Computing Frontiers*, April 2004, pp. 377-389

[17] T.J. Todman, G.A. Constantinides, S.J.E Wilton, O. Mencer, W. Luk and P.Y.K. Cheung, "Reconfigurable Computing: Architecture and Design Methods", *IEE Proceedings on Computers and Digital Techniques*, Vol. 152, No. 2, March 2005, pp. 193-207

[18] R. Kastner, A. Kaplan, S.O. Memik and E. Bozorgzadeh, "Instruction Generation for Hybrid Reconfigurable Systems", *ACM Transactions on Design Automation of Embedded Systems*, Vol. 7, No. 4, October 2002, pp. 605-627

[19] N.T. Clark, H. Zhong and S.A. Mahlke, "Automated Custom Instruction Generation for Domain-Specific Processor Acceleration", *IEEE Transactions on Computers*, Vol. 54, No. 10, October 2005, pp. 1258-1270

[20] F. Sun, S. Ravi, A. Raghunathan and N.K. Jha, "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

[21] L. Pozzi, K. Atasu and P. Ienne, "Exact and Approximate Algorithms for the Extension of Embedded Processor Instruction Sets", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 25, No. 7, July 2006, pp. 1209-1229

[22] P. Yu and T. Mitra, "Scalable Custom Instructions Identification for Instruction-Set Extensible Processors", *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, September 2004, pp. 69-78

[23] J. Cong, Y. Fan, G. Han and Z. Zhang, "Application-Specific Instruction Generation for Configurable Processor Architectures", *Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, February 2004, pp. 183-189

[24] X. Chen, D.L. Maskell and Y. Sun, “Fast Identification of Custom Instructions for Extensible Processors”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 26, No. 2, February 2007, pp. 359-368

[25] P. Yu and T. Mitra, "Characterizing Embedded Applications for Instruction-Set Extensible Processors", *Proceedings of the 41st IEEE/ACM on Design Automation Conference*, June 2004, pp. 723-728

[26] D. Chen and J. Cong, "DAOmap: A Depth-Optimal Area Optimization Mapping Algorithm for FPGA Designs", *IEEE International Conference on Computer-Aided Design*, November 2004, pp. 752-759

[27] J. Lin, D. Chen, and J. Cong, "Optimal Simultaneous Mapping and Clustering for FPGA Delay Optimization", *Proceedings of Design Automation Conference*, July 2006

[28] A. Nayak, M. Haldar, A. Choudhary and P. Banerjee, “Accurate Area and Delay Estimators for FPGAs”, *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, March 2002, pp. 862-869

[29] P. Bjureus, M. Millberg and A. Jantsch, “FPGA Resource and Timing Estimation from MATLAB Execution Traces”, *Proceedings of the International Symposium on Hardware/Software Codesign*, May 2002, pp. 31-36

[30] C. Brandolese, W. Fornaciari and F. Salice, “An Area Estimation Methodology for FPGA Based Designs at SystemC-Level”, Proceedings of the 41st Design Automation Conference, 2004, pp. 129-132

[31] D. Kulkarni, W.A. Najjar, R. Rinker and F. J. Kurdahi, “Compile-Time Area Estimation for LUT-based FPGAs”, *ACM Transactions on Design Automation of Electronic Systems*, Vol. 11, No. 1, January 2006, pp. 104-122

[32] S. Bilavarn, G. Gogniat, J.L. Philippe and L. Bossuet, “Design Space Pruning Through Early Estimations of Area/Delay Tradeoffs for FPGA Implementations”, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 25, No. 10, October 2006, pp. 1950-1968

[33] H.P. Huynh, J.E. Sim and T. Mitra, “An Efficient Framework for Dynamic Reconfiguration of Instruction-Set Customization”, Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, September 2007, pp. 135-144

[34] L. Bauer, M. Shafique, S. Kramer and J. Henkel, "RISPP: Rotating Instruction Set Processing Platform", *ACM/IEEE/EDA Design Automation Conference*, June 2007, pp. 791-796

[35] L. Bauer, M. Shafique and J. Henkel, “Efficient Resource Utilization for an Extensible Processor through Dynamic Instruction Set Adaptation”, *5th Workshop on Application Specific Processors*, October 2007, pp. 39-46

[36] M.J. Wirthlin and B.L. Hutchings, "A Dynamic Instruction Set Computer", *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1995, pp. 99-107

[37] N.T. Clark, J. Blome, M. Chu, S.A. Mahlke, Stuart Biles and Krisztian Flautner, "An Architecture Framework for Transparent Instruction Set Customization in Embedded

Processors", *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005

[38] S. Yehia, N.T. Clark, S.A. Mahlke and K. Flautner, "Exploring the design space of LUT-based transparent accelerators", *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, September 2005, pp. 11-21

[39] J. Brown and M. Epalza, "Automatically Identifying and Creating Accelerators Directly from C Code", Xcell Journal, Issue 58, 2006. Available: http://www.xilinx.com/publications/xcellonline/xcell_58/index.htm#Letter

[40] Xilinx Data Sheet, "Virtex 2.5V FPGA Detailed Functional Description", DS003-2, Version 2.8.1, December 2002

[41] K. Compton and S. Hauck, "Reconfigurable Computing: A Survey of Systems and Software", *ACM Computing Surveys*, Vol. 34, No. 2, June 2002, pp. 171- 210

[42] Xilinx ISE Foundation. Available: http://www.xilinx.com/ise/logic_design_prod/ foundation.htm

[43] Trimaran: An Infrastructure for Research in Instruction-Level Parallelism. Available: http://www.trimaran.org

[44] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne and N. Dutt, "Introduction of Local Memory Elements in Instruction Set Extensions", *Proceedings of the 41st Annual IEEE/ACM Design Automation Conference*, June 2004, pp. 729-734

[45] L. Pozzi and P. Ienne, "Exploiting Pipelining to Relax Register-File Port Constraints of Instruction-Set Extensions", Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems, 2005, pp. 2-10

[46] L.P. Cordella, P. Foggia, C. Sansone and M. Vento, "Performance Evaluation of the VF Graph Matching Algorithm", *Proceedings of the International Conference on Image Analysis and Processing*, September 1999, pp. 1172-1177

[47] Y. Guo, G.J.M. Smit, H. Broersma and P.M. Heysters, "A Graph Covering Algorithm for a Coarse Grain Reconfigurable System", *Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, June 2003, pp. 199-208

[48] Xilinx User-Guide, "Virtex-5 FPGA User-Guide", UG190, Version 3.2, December 2007

[49] Xilinx Application Note, "Design Tips for HDL Implementation of Arithmetic Functions", XAPP215, Version 1.0, June 2000

[50] H. Gao, Y. Yang, X. Ma and G. Dong, "Analysis of the Effect of LUT Size on FPGA Area and Delay Using Theoretical Derivations", *Proceedings of the Sixth International Symposium on Quality of Electronic Design*, March 2005, pp. 370-374

[51] M.R. Guthaus, J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, "MiBench: A Free, Commercially Representative Embedded Benchmark Suite", *IEEE International Workshop on Workload Characterization*, December 2001, pp. 3-14

[52] C. Lee, M. Potkonjak and W.H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *Proceedings of the 13th Annual IEEE/ACM International Symposium on Microarchitecture*, December 1997, pp. 330-335

[53] P.G. Sassone and D.S. Wills, “On the Extraction and Analysis of Prevalent Dataflow Patterns”, *Proceedings of the Workshop on Workload Characterization*, 2004

[54] J. Rose, R.J. Francis, D. Lewis and P. Chow, “Architecture of Field Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency”, *IEEE Journal of Solid-State Circuits*, Vol. 25, No. 5, October 1990, pp. 1217-1225

[55] R. Lysecky and F. Vahid, “A Study of the Speedups and Competitiveness of FPGA Soft Processor Cores using Dynamic Hardware/Software Partitioning”, *Proceedings of the Conference on Design, Automation and Test in Europe*, 2005, pp. 18-23