FPGA-Aware Techniques for Rapid Generation of Profitable Custom Instructions

Alok Prakash[1], Siew-Kei Lam[1], Christopher T. Clarke[2] and Thambipillai Srikanthan[1]

[1]School of Computer Engineering

NTU, Singapore, 637553.

{alok0001, assklam, astsrikan}@ntu.edu.sg

[2]Department of Electronic and Electrical Engineering,

University of Bath, United Kingdom, BA27AY.

eesctc@bath.ac.uk

ABSTRACT

Instruction set extension of FPGA based reconfigurable processors provides an effective means to meet the increasingly strict design constraints of embedded systems. We have shown in our previous works [20][21] that the usage of FPGA architectural constraints for pruning the design space during enumeration of custom instructions/patterns not only leads to notable reduction in the time taken to identify custom instructions but can also result in the selection of profitable custom instructions when the area is highly constrained. However when area constraint is relaxed, the previously proposed methods failed to perform better than traditional methods. In this paper, we propose a heuristic to identify profitable custom instructions for designs with *arbitrary area constraints*. The proposed heuristic relies on a new pruning criterion to enumerate patterns with high size-to-hardware-area ratio. We also proposed a suitable algorithm to select profitable custom instructions from the enumerated patterns. The proposed template selection algorithm takes advantage of the FPGA area-time measures of the enumerated patterns, which can be easily inferred from the FPGA-aware enumeration strategy. Experimental results show that the proposed methods in this paper result in custom instructions that achieve an average performance gain of 76.23 % over current state-of-the-art approaches.

Keywords

Custom Instruction Extension, Enumeration, Selection, Application-specific architectures.

1. INTRODUCTION

Embedded system designers are consistently facing the ever-increasing time-to-market (TTM) constraints while being pushed to deliver products with longer life cycle. These contradictory set of constraints has favored FPGA-based systems which are characterized by high flexibility, low TTM etc. Recently, soft-core processors that can be synthesized on FPGAs have gained a lot of popularity due to their immense flexibility and performance benefits. Reconfigurable processors, such as NIOS II[1] from Altera, Microblaze [26] from Xilinx, etc., provide the capability to extend the instruction set for high performance systems. Although a lot of research has been done in the area of instruction set extension, commercial FPGA tools still lack a consolidated framework for automatic generation of custom instructions from a given application code. There are typically two major steps in custom instruction generation namely, custom instruction (pattern) identification and custom instruction (template) selection. A commonly adopted approach in pattern identification is pattern enumeration, which tries to identify all the legal custom instruction patterns within an application. A template selection algorithm is then employed to select a set of non-overlapping patterns for final implementation based on certain constraints such as area and performance.

While custom instruction generation is a well-researched problem, the existing literature typically ignores the implementation constraints of the custom instructions on the target FPGA hardware during the process of identifying and selecting custom instructions. In [20], [21], we showed that neglecting the FPGA architecture characteristics at the early stages of custom instruction generation leads to solutions that cannot be mapped efficiently on the FPGA architecture. We also introduced the concept of FPGA-aware enumeration of custom instruction candidates wherein the pattern enumeration phase uses the target architecture information as additional pruning constraints. In particular, our enumeration algorithm identified only patterns that can be fully mapped onto a single logic block of the target FPGA architecture. We define a logic block as a set of 32 FPGA basic logic elements with the same hardware configuration that implements a 32-bit wide custom instruction. The method proposed in [20], [21] was devised based on the observation that custom instructions are generally dominated by small and frequently occurring patterns. Hence, when the available FPGA area is limited, it is sufficient to identify smaller and frequently occurring patterns that can efficiently utilize the FPGA space as they provide more performance gain per unit area compared to the larger and less frequently occurring patterns. While our work in [20], [21] performed exceptionally well in area-constrained designs, they do not scale well when the area constraint is relaxed. This is due the fact that our previous work focuses on maximizing the resource utilization of FPGA space with limited resources and this leads to pruning of larger patterns from the design space that have relatively lower performance gain per unit area. Hence, this strategy may not lead to the best results when the hardware area constraint is relaxed and the application contains large patterns.

In this paper, we propose a **new enumeration strategy** to identify profitable custom instructions for FPGAs with **arbitrary area constraints**. This is achieved by identifying (enumerating) large patterns that exhibit a high degree of instruction level parallelism, which can be mapped efficiently onto the FPGA logic blocks. We also propose a **new selection heuristic** to facilitate the **selection of large custom instructions with low critical path** for FPGA designs with arbitrary area constraint.

It is noteworthy that our previous work in [20], [21] is still relevant for many embedded designs that typically have very strict area constraints. The work proposed in this paper can be used in a wider range of designs especially given that FPGAs are becoming commonplace in high performance embedded systems. It should also be noted that while techniques for Pattern Enumeration have received considerable attention in the past, to the best of our knowledge, the existing Pattern Enumeration techniques, aside from our own work in [20] and [21], have never taken the implementation constraints of FPGA architecture into account. The novelty of our work lies in the fact that we have used these constraints during enumeration so as to make a more *informed-enumeration* and thereby achieving much higher performance from the selected custom instructions with less FPGA area utilization. The experiments undertaken in this work relies on a low-end Xilinx Virtex II FPGA in which the basic logic element is a 4-input LUT. The results obtained for this FPGA showed that high performance custom instructions can still be realized using a low-end FPGA device when architecture aware methods are employed to generate custom instructions. We expect to achieve even higher performance on modern FPGAs (with more efficient LUT architecture) with the proposed methods.

The rest of the paper is organized as follows. Section 2 gives a comprehensive survey of the related work in this area. In Section 3, we briefly discuss the methods proposed in our previous work [20] and [21] and highlight their limitation. Section 4 explains the proposed pattern enumeration algorithm while Section 4.2 explains the pattern selection algorithm. In Section 5, we provide the experimental results to demonstrate the benefits of the proposed method and Section 6 concludes the paper.

2. RELATED WORK

A typical custom instruction generation methodology follows a two-step process. In the first step called template identification, valid patterns are identified in an application DFG. A pattern 'P' is designated valid if it satisfies certain constraints related to the structure of the pattern (e.g. convexity), input-output constraints of custom instructions and operation types.

In the second step, the template selection algorithm selects a small set of non-overlapping patterns for implementation on the target FPGA device. In the past decade, there has been tremendous amount of research in custom instruction enumeration and selection methods, making it difficult to list them extensively. In the following subsections, we will discuss existing state-of-the-art works in the area of pattern identification and pattern selection. In some of these methodologies, design space exploration is performed after the selection step to select the right set of custom instructions based on the available area and performance constraint.

2.1 Pattern Identification

A branch and bound algorithm was proposed by Atasu et al. in [3] which identified all the valid patterns from application Data flow graph (DFG) while discarding those that violated the micro-architectural constraints of the FPGA. While their algorithm performed significantly better than the contemporary state of the art techniques, its computation complexity quickly gets out of hand with increasing size of the DFG. Pothineni et al. described a method to identify maximal convex patterns from a DFG, without any I/O constraints [16]. They claimed that these patterns, which are not bounded by general I/O constraints, provide up to 50% performance benefits. They also claimed that such an approach leads to reduction in the enumeration tool runtime.

In [17] the authors provided a new algorithm for faster enumeration of legal patterns. They proposed an iterative process which reduced their algorithm runtime by up to two orders of magnitude when compared to the existing work. They enumerated the patterns in increasing order of size, while relating to the patterns with (n+1) nodes to the patterns with n nodes.

Atasu et. al. in [2] enumerated only maximal convex patterns without considering Input-Output constraints and achieved a performance increment of an order of magnitude, while having an extremely fast tool runtime. They defined a maximal convex pattern as a convex subgraph that cannot be grown further by adding more nodes into it while maintaining its convexity.

In [6] and [12] Chen et al. and Li et al. proposed techniques for fast enumeration of custom instruction candidates, that are especially well suited for large DFGs. Their algorithm required orders of magnitude less time compared to state-of-the-art methods. Li et. al. [12] enhanced the work in [6] for the special case of single-output constraint. Due to its fast runtime, we have extended the algorithm described in [6] and [12] to incorporate additional hardware cognizant pruning constraints.

2.2 Pattern Selection

After enumerating all the legal patterns from the DFG, pattern selection chooses a set of non-overlapping patterns for final implementation. This step typically uses graph-covering algorithms to choose a set of non-overlapping patterns with maximal cover.

There is a lot of research in the area of Pattern Selection [5], [7], [8], [9], [13], [19] etc.. In [9] Cong et al. presented a complete framework for custom instruction enumeration, selection as well as mapping. The extended instruction set identified by their algorithms enabled them to achieve up to 3.73x performance gain compared to the basic instruction set.

Clark et al. presented a complete design framework for automatic generation of patterns within a DFG and a compiler framework to take advantage of such custom units [7], [8]. The custom instruction candidates were annotated with their latency and hardware area from a stored hardware library. This information was used in the selection process which gives preference to the selection of patterns with largest speedup over area ratio. In [13] the authors used a similar approach, but the area and latency of the patterns were obtained by synthesizing them on a 0.13µm CMOS process.

Bonzini et al. in [5] proposed a recurrence-aware covering algorithm that achieved tangible improvement over existing methods. Their work focuses on identifying patterns, which were both large and recurring. This is an effective approach as it often provides better results than heuristics that either gives more importance to only frequency or only the pattern size. We have also used a similar heuristic in our pattern selection phase for our previous work in [20][21]. Yu et al. in [27] have proposed an efficient and scalable method for enumeration of disjoint pattern from an application DFG.

Guo et al. in [10] proposed a fast and effective method for pattern selection. The authors created a conflict graph, which is an undirected graph where every node in the graph corresponds to a pattern. An edge in the conflict graph between two pattern nodes represents an overlap between the corresponding patterns in the original DFG. The conflict graph enables the selection of non-overlapping patterns for the final implementation. Their selection strategy relies on choosing the maximal independent set (MIS) in the conflict graph. This strategy is also employed in our work. Wang et al. in [24] showed that the practical complexity of computing the MIS set for a given graph is of polynomial order and the complexity is governed by the size of the graph. In the worst-case scenario, the MIS algorithm is an NP-complete problem.

The work in [19] proposed a custom instruction generation framework. This framework incorporated a hardware area estimation technique to evaluate the required area to implement a custom instruction by clustering the pattern candidates. Their approach partitions every pattern into sub-graphs so that each sub-graph could be completely mapped onto a single logic block of the target FPGA. This approach was used to estimate the hardware implementation area for the selected patterns. The FPGA logic block is considered as the atomic unit of area.

Bohm et al. in [4] proposed a methodology for automatic selection of extended instructions in an ASIP. They proposed a GCC based tool-chain to automatically exploit the extended instructions generated by their custom instruction generation methodology.

Traditionally memory operations, which are generally excluded from custom instructions and thereby treated as invalid nodes, act as one of the pruning constraints. A few recent works have proposed to include memory operations in custom instructions [11], [28] by introducing local memory elements. This allowed enumeration and selection of larger patterns compared to other approaches thereby resulting in better performance.

Micro-architectural properties of the underlying hardware, e.g. the number of I/O ports available for custom instruction implementation, are typically used as a fundamental set of constraints during enumeration of custom instructions. In [20], [21], we proposed a tighter set of constraints for pattern enumeration in order to identify the most profitable pattern candidates. This led to generation of more profitable custom instructions. However, their performance was limited to designs with strict area constraint. In this work, we overcome this limitation and propose strategies to generate profitable custom instructions for any given area constraint.

3. Design with strict area constraints – Our previous work [20, 21].

In this section, we will introduce and discuss our previous work, which targets systems with strict area constraint. In Section 4, we will show how this was extended to cater to designs with arbitrary area constraints. Figure 1 shows the proposed methodology for instruction set extension and customization. Trimaran [23] is used as the compiler for the C-application and to generate an Intermediate Representation (IR) for pattern enumeration. The Trimaran compiler was also used to profile the application in order to identify the most frequently executed basic blocks. The IR consists of basic operations that can be classified into three main groups: 1) logical operations e.g. *and, or, xor* etc., 2) shift operations e.g. *shl, shr* etc. and 3) arithmetic operations e.g. *add, sub, mult, div* etc. It can be observed that the proposed methodology in Figure 1 incorporates the architecture information at the pattern enumeration as well as selection phase.

3.1 FPGA-Aware Pattern Enumeration

As defined before, the pattern enumeration phase identifies all the legal patterns within a DFG. A pattern is legal if the micro architecture of the FPGA is capable of efficiently supporting that pattern (e.g. I/O ports, convexity, etc.) [3, 9]. Theoretically, a DFG with 'n' nodes can have up to 2^n number of possible patterns. The enumeration algorithm uses these micro-architectural constraints to discard illegal patterns. Additionally, a pattern containing either memory (such as LOAD/STORE) or control flow operations (such as BRANCH) are also termed as illegal.

However, the number of possible legal patterns can still be exponentially large after taking into consideration these micro-architectural constraints.

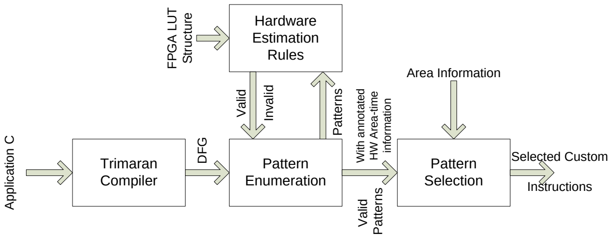


Figure 1. Our methodology for instruction set extension [20, 21].

In this work we used the pattern enumeration technique proposed in [6], which has been shown to be orders of magnitude faster than other existing algorithms for large DFGs. Specifically, the enumeration algorithm in [6] has been modified to take into account the additional pruning constraints so that all enumerated patterns fit exactly into a single logic block of the target FPGA.

The additional pruning constraints are based on the hardware estimation rule-sets that are proposed in [19]. In particular, the work in [19] estimates the area-time of custom instructions by partitioning every pattern into smaller sub-graphs (clusters) so that each cluster could be completely mapped onto exactly one FPGA logic block (i.e. a set of n logic elements in the FPGA that are similarly configured to create a function that processes n-bit inputs). This approach enabled rapid estimation of the number of logic blocks as well as the critical path that is required to implement every pattern in the target FPGA. The authors in [19] used two sets of rules to partition a pattern into clusters. The first set of rules determine if an operation can be included in a cluster, whereas the second set evaluates whether the number of inputs and outputs of the cluster conform to the architecture constraints of the FPGA logic block.

We have incorporated these hardware estimation rule-sets as the pruning constraints alongside the traditional micro-architectural constraints, so as to identify only “clusters”. Since each “cluster” by definition fits into exactly one logic block of the target FPGA, we call such custom instructions Fine Grained Custom Instructions (FGCI) in our work. This ensures that every enumerated pattern can be implemented in a single logic block of the target FPGA architecture. It should be noted that the proposed enumeration strategy could be used for different FPGA families, as we would only need to change the hardware estimation rule-sets of [19] to conform to the architecture constraints of the target FPGA. Since these rule sets can be easily inferred from the target FPGA device’s data sheet, it is easy to re-target the hardware estimation rule-sets to a different FPGA model or manufacturer. In the next subsection, we will briefly describe the enumeration algorithm used in [6] and explain the modifications made in order to incorporate the target architecture information.

3.2. Modifications to the current state-of-art Enumeration Algorithm

The enumeration algorithm proposed by authors in [6, 12] works in a recursive manner to identify valid patterns (also called "cuts") from an application DFG. The terms used in the algorithm are defined as follows:

- G = Data Flow Graph of a basic block of an application.
- P = Existing Pattern
- u = Selected Node for merging to the existing pattern to form a new pattern.
- P’ = New pattern formed by merging pattern P and the selected node ‘u’ and other nodes to ensure convexity.
- Succ(G, u) = Successor nodes of u in the basic block DFG, G.
- Pred(G, u) = Predecessor nodes of u in the basic block DFG, G.
- Disc(G, u) = Disconnected nodes of u in the basic block DFG, G.
- r_g = Redundancy guarding node.

The various terms can be understood by the following example.

Let G(V,E) be a Data-Flow Graph. Node u $\in$ V can be classified into the following:

1) Predecessors of node u: Pred(G, u) is set of nodes n $\in$ V (n $\neq$ u), if there is a path in G from n to u.

2) Successors of u: Succ(G, u) = set of nodes n $\in$ V (n $\neq$ u), if there is a path in G from u to n.

3) Disconnected nodes of u: Disc(G, u) = set of nodes n $\in$ V, if there is neither a path from u to n nor from n to u}.
A pattern (denoted by P) obtained from an application DFG (denoted by G) is considered to be a valid pattern if it satisfies the following micro-architectural constraints:

i. P is convex;

ii. Number of Input ports in P ≤ IN_LIMIT;

iii. Number of Output ports in P ≤ OUT_LIMIT;

iv. P does not contain any invalid operations e.g. memory, branch operations.

The terms IN_LIMIT and OUT_LIMIT denote the number of available input and output ports respectively for the custom instructions in the target architecture. The enumeration algorithm proceeds as follows: The algorithm starts with an empty pattern P. In addition, a basic block of the application is considered as the initial Data-Flow Graph, G. It is also necessary to include a "redundancy guarding node" r_g in G to avoid enumerating the same pattern twice in the split function. The algorithm then recursively invokes the enumeration procedure, which consists of three important functions, *select_node*, *unite* and *split*. The *select_node* function, as the name suggests, takes the nodes(G-P) (i.e. the nodes in G that are not a part of the already selected pattern P) and selects a single node from this set and considers them for inclusion in the pattern. Function *unite* handles the positive branch whereby the selected node from the function *select_node* is used to create a new pattern P' that is the combination of the new node and P. As a result, the search process is accelerated since the number of nodes that needs to be clustered in the next iterations of the algorithm is reduced thereby limiting the search time. The unite function outputs a valid pattern P' in all cases.

The negative branch of this enumeration algorithm is handled by the *split* function, which caters to the scenario when the selected node could not be included in the existing pattern to form a valid pattern. In particular, the function *split* breaks up the current DFG G into one or two DFGs (e.g. G' and G'') based on the selected node. Therefore, this step reduces the depth of recursive search. The algorithm repeats by passing the new pattern (i.e.

P') and DFGs (G' and G'') as arguments to the three functions. Further details of this algorithm can be found in [6, 12].

In order to enumerate patterns in an FPGA-aware manner, further constraints are added to the *select_node* and *unite* functions. The modified *select_node* function utilizes the types of nodes that are already present in the existing pattern to identify another node that can be merged, while the new *unite* function checks for additional constraints while merging the selected node (or the pattern consisting of the selected node) with the existing pattern. For example, since the Virtex 2 or 4 FPGAs have a single carry chain along with their 4-Input LUT structure, as shown in Figure 2 [19] [25], they cannot implement more than one ADD/SUB operation in a single logic block, and therefore a valid pattern can only contain at most one ADD/SUB operation. To ensure this, the modified *select_node* function checks the existing pattern for ADD/SUB operations and will not return a new node which is an ADD/SUB operation, if the existing pattern already has such an operation. Similar changes were made to check for the other rules as well. The modified select_node function is called *mod_select_node* algorithm, the pseudocode and further details of which could be found in [20, 21].

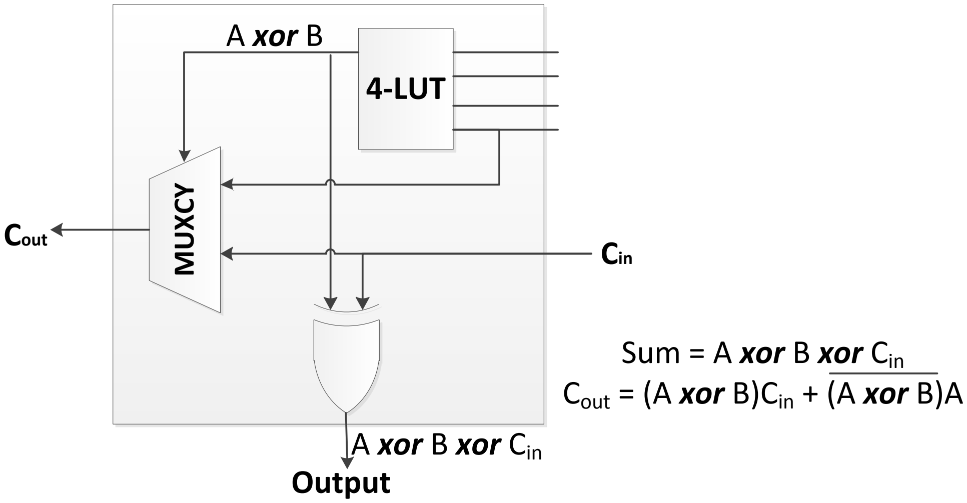

A **xor** B
4-LUT
MUXCY
C_{out}
C_{in}
Sum = A **xor** B **xor** C_{in}
C_{out} = (A **xor** B)C_{in} + $\overline{(A \textbf{ xor } B)}$A
A **xor** B **xor** C_{in}
Output

Figure 2. An Add operation in a 4-Input LUT, present in Xilinx 2 and 4 FPGAs

It is noteworthy that although we discussed the changes made specifically to the enumeration algorithms proposed in [6, 12], our concepts could be used with any enumeration algorithm. We chose enumeration method proposed in [6, 12] only because of its fast runtime compared to the rest of the algorithms. Our idea of enumerating only patterns that can fit into exactly one logic block can be easily implemented in other enumeration algorithms by incorporating the hardware estimation rule-sets of [19] as pruning constraints.

The strategies in [20, 21] were proposed specifically for tight area designs by **enumerating only small** and hence frequently occurring Fine Grained Custom Instructions. Therefore, such patterns are not suited for designs with large available hardware area. In the next section, we will discuss the strategy proposed in this work, which can cater to designs with arbitrary area constraint.

4. PROPOSED METHOD FOR DESIGNS WITH ARBITRARY AREA CONSTRAINTS

The method proposed in [20] and [21] can achieve significant performance benefits for designs with stringent area-constraints. However, it may not lead to the best results for designs with relaxed area constraints. The reason for such a behavior is explained by the fact that the discovery of large patterns that have a high degree of instruction level parallelism was intentionally omitted during the enumeration process of [20, 21]. Although these large patterns generally occur less frequently than the smaller ones, they can still contribute to notable performance gain, provided there is sufficient FPGA space to implement such large patterns. In this paper, we denote the method described in Section 3 ([20] & [21]) as the Fine Grained Custom Instructions (FGCI) approach. The enumeration technique proposed in this paper (denoted as Coarse-Grained Custom Instructions (CGCI) approach) not only enumerates the smaller patterns as the FGCI approach, but also identifies large profitable patterns that exhibit a high FPGA performance-area ratio. An example of such a pattern is shown in Figure 3. We will use this example to highlight the limitation of our previous work. It can be observed that the pattern consists of four FGCIs in accordance with the concepts explained in [18], [19], [20] and [21]. These four FGCIs are then implemented as four individual custom instructions, each with a critical path of 1 FGCI. We use the following equation to calculate the number of clock cycles saved using these extended instructions [19].

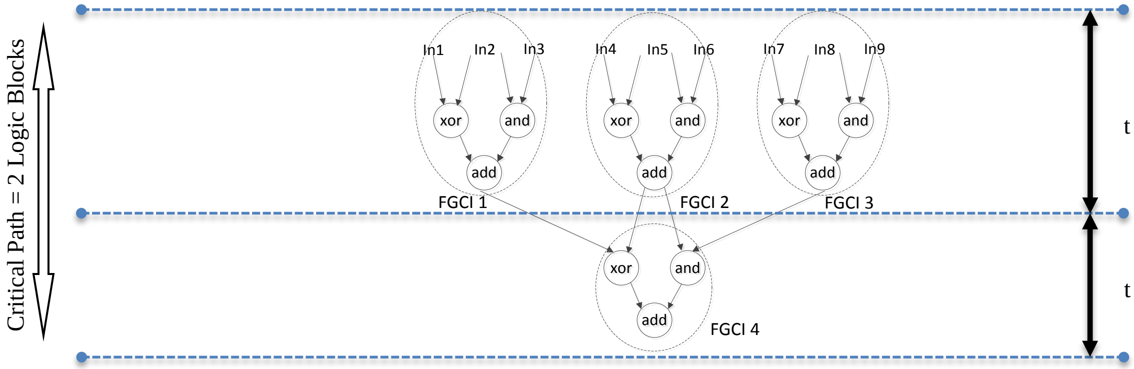


Figure 3. An example of a large pattern with relatively small critical path

$$cycle\ savings = \left[\sum_{1}^{n} \underbrace{(NN * DO)}_{\text{SW TIME}} - \sum_{1}^{n} \underbrace{(CLP * DO)}_{\text{HW TIME}}\right] \qquad [1]$$

where *NN* denotes number of nodes (or operations) in a custom instruction, *DO* is the number of dynamic occurrences (number of times the custom instruction is called) of the custom instruction during execution, *n* is the total number of selected custom instructions and *CLP* is the critical path of the custom instruction.

Using equation (1) and assuming that the dynamic occurrence of the pattern is 10, the performance achieved is:

$$\left[\sum_{1}^{4} (3 * 10) - \sum_{1}^{4} (1 * 10)\right] = 80 \text{ clock cycles}$$

However, it can be observed from Figure 3 that three of the FGCIs can be implemented in parallel, and hence the critical path of the pattern is only 2 logic blocks. If the pattern in Figure 3 can be identified as a single custom instruction (as opposed to four smaller custom instructions), a higher performance gain can be achieved, i.e. from (1), the performance will be:

$$\left[\sum_{1}^{1}(12 * 10) - \sum_{1}^{1}(2 * 10)\right] = 100 \text{ clock cycles}$$

Hence, we can see that while it is prudent to enumerate small and frequently occurring patterns for area constrained designs, the concept does not hold true when the area constraint is relaxed.

We define Coarse-grained Custom Instruction or CGCI as a combination of FGCI, where more than one of the constituent FGCIs can be implemented in parallel. This in turn means that the critical path of CGCIs is always smaller than the number of FGCIs. Since every FGCI can be implemented in 1 logic block, the number of FGCIs in a CGCI is equivalent to the number of Logic blocks required to implement it.

4.1 Pattern Enumeration for CGCI

We have changed the enumeration algorithm in [6][12] to include patterns that exhibit a high degree of instruction parallelism and at the same time, can be efficiently mapped onto the FPGA architecture. While this algorithm looks similar to the one proposed in our previous work, the critical differences are highlighted below. To avoid confusion and for the sake of clarity, we present and explain the entire algorithm including the ones in [20, 21]. In particular, apart from enumerating FGCIs, the new algorithm also identifies patterns in which the number of FGCIs is greater than the critical path of the pattern. These patterns are annotated with the hardware area and critical path (both in terms of number of FGCIs) during the enumeration phase. Again, it should be noted that the area of a CGCI (in number of logic blocks) is basically the number of FGCIs in it, whereas its critical path is obtained by determining the maximum schedule length of the custom instruction, e.g. (2*t) in Figure 3, where “t” is the latency of a logic block.

Algorithm 1 describes the new enumeration algorithm that incorporates the necessary steps to evaluate the number of FGCIs and the critical path of every new pattern after merging the selected node with the existing pattern (along with the other nodes needed to meet the convexity constraint [3]]). The basic enumeration algorithm used here has been adapted from the work in [6] and [12] and explained previously in Section 3.

Just as we explained in Section 3.1, a new pattern P’ is formed by adding a new node ‘u’ to the existing pattern P as well as the other nodes needed to ensure the convexity of the generated pattern. The extra nodes needed for convexity depends upon the set to which the selected node belongs to. For example, if ‘u’ is the element of Pred(G, P), then all the nodes in the intersection of set Succ(G, u) and the set Pred(G, P) must also be added and so on. In the current state-of-the-art methods any new pattern formed by this step needs to be verified against the traditional micro-architectural constraints such as the number of Input/Output nodes etc. However, in our case, it is also checked against the additional rules (Lines 7 and 10 in Algorithm 1) to ensure the enumeration of only the most profitable custom instruction candidates. These additional checks rely on two values obtained previously in

Lines 5 and 6 in Algorithm 1, to ensure that only profitable custom instruction candidates that are large and exhibit high degree of parallelism are enumerated. These two values are:

1) *The number of FGCIs in the pattern (Line 5 in Algorithm 1 and Algorithm 1a)*. In order to obtain this, we first enumerate all the sub-patterns of a pattern. Each sub-pattern is then evaluated using the *cluster_evaluation* algorithm to obtain a set of valid FGCIs. The FGCIs adhere to the rules in *cluster_evaluation* that was proposed in [21] (shown here in Algorithm 2), which ensures that a pattern implemented in a single logic block of the target FPGA is a valid FGCI. We then select a set of non-overlapping FGCIs that *"cover"* the entire pattern using the conflict graph and Maximum Independent Set (MIS) methods proposed in [10]. These concepts have been explained in detail later in Section 4.2 for the selection of Coarse Grained Custom Instructions (CGCIs). The number of valid FGCIs required to cover the pattern gives the area measure of the pattern.

2) *The critical path of the pattern in terms of number of FGCIs (Line 6 in Algorithm 1)*. Since we know the FGCIs covering the pattern P' from Step "1", it is easy to calculate the critical path of P' based on the schedule time step of FGCIs in pattern P', based on the level of parallelism present in the pattern P'. As an example, consider the pattern in Figure 3. The three FGCIs at the first scheduling level can be implemented in parallel and therefore the time required to execute them is equivalent to the time for a single logic block. The last FGCI in Figure 3 needs the output of the first three FGCIs and hence it is executed in the second time step. Therefore the critical path of the entire patter is 2 logic blocks.

In order to verify a pattern for a valid FGCI the *cluster_evaluation* algorithm basically works by first arranging all nodes in a given pattern in a topological order and then traversing the pattern in that order, checking for various rules needed to ensure that all the operations of the pattern can fit into a single logic block. For example, since a logic block in Virtex 2/4 (even the newer FPGA such as Virtex 5/6/7) has single carry chain, it does not allow patterns with two arithmetic operations. Similarly, since the last stage of the LUT (Figure 2) is the partial sum (A ***xor*** B **xor** C), all *logical* or *shift* operations must be executed first to generate the required operands (i.e. A and B) for the *add* operation. Therefore, the algorithm does not allow a pattern with *logical* or *shift* operation after an arithmetic operation to pass as a valid cluster. The variables such as *has_arith_op* etc. are used to denote if a pattern has an arithmetic operation etc. The I/O constraints are also verified to ensure that patterns can indeed be implemented in a single logic block. Therefore, the algorithm verifies the pattern against the rules needed to implement a pattern in a single logic block of the target FPGA fabric.

It should be noted again that the approaches proposed in this paper is targeted towards a low end device with only 4-inputs LUT with a single carry chain such as the ones in Xilinx Virtex 2 or 4. The ideas presented in this paper can be easily extended to current and future generation of devices such as Xilinx Virtex 5/6/7 that contain 6-input LUTs with similar carry chain connections [29] that can either implement a single 6-input function or dual 5-input functions. The larger LUTs in newer FPGA devices typically enables more functionality in a single Logic block, which potentially leads to better performance-area ratio. Therefore, in order to target a different FPGA device, we need to change the number of Input/output ports allowed in a single cluster or FGCI thereby allowing the concept itself to be applied to any FPGA family across different vendors.

Once the number of FGCIs and the critical path of a pattern are defined, we proceed with the remaining steps in Algorithm 1. A pattern consisting of a single FGCI, is identified as a valid pattern if it satisfies the I/O constraints (specific to the target FPGA device). On the other hand, a pattern with multiple FGCIs is chosen if and only if the number of FGCIs in the pattern is greater than its critical path i.e. patterns similar to the example pattern in Figure 3, discussed before. This strategy leads to the enumeration of the most profitable set of patterns for the given application on a particular architecture.

It is noteworthy that the proposed enumeration algorithm identifies custom instructions wherein their FPGA area-time measures can be inferred without the need for time-consuming hardware implementation. The area-time in our case was calculated using cluster_evaluation function at its core.

In the next subsection, we propose a selection heuristic that favors these large patterns when the FPGA area is unconstrained, while selection smaller patterns for area-constrained design.

Algorithm 1 Proposed Enumeration Algorithm

```
unite_new(P, G, $r_g$, u) {
$P' = \left\{ \begin{cases} P \cup \{u\} \cup (Succ(G,u) \cap Pred(G,P)), if\ u \in Pred(G,P) \\ P \cup \{u\} \cup (Pred(G,u) \cap Succ(G,P)), if\ u \in Succ(G,P) \\ P \cup \{u\} \qquad\qquad ,if\ u \in Disc(G,P) \end{cases} \right\}$

if((u∈Disc(G,P))&&(Succ(G,u)∩Succ(G,P)=={})&&(OUT_LIMIT==1))
        return;

$r_g$=$N_a$N;
Count the number of FGCIs in the pattern P' using count_fgci;
Evaluate the critical path of the pattern P' using based on scheduling of FGCIs;
if ((number of FGCIs(P')==1)&& in_check(P') &&out_check(P')){          //function to check FGCI node rules and I/O constraints
            patterns.add(P');
            enumerate(P', G, $r_g$);
  }
elseif $(number\_of\_FGCIs)_{pattern\ P'}$>$(critical\_path)_{pattern\ P'}$){
            patterns.add(P');
            enumerate(P', G, $r_g$);
  }
else
        return
```

Algorithm 1a Count FGCIs

```
count_fgci (Pattern P){
Enumerate every sub-pattern in Pattern P.
Evaluate every sub-pattern for valid FGCI using cluster_evaluation.
Find the set of FGCIs that completely “covers” the pattern P using conflict graph and MIS methods.
number_of_FGCI = number of sub-patterns required to cover the pattern P.
return number_of_FGCI.
}
```

Algorithm 2 Cluster Evaluation

```
cluster_evaluation(pattern){
Sort the nodes in the pattern in topological order.
bool has_arith_op, has_logical_op, has_shft_op =FALSE;
 //variables has_arith_op, has_logical_op & has_shft_op show
 //whether or not a pattern has arithmetic, logical or shift
 //operations respectively

for every node in the pattern{
        if node == arith_op
            if has_arith_op==TRUE; //any prior node in the pattern // is an arithmetic operation
                return 0; //cluster not possible; cluster cannot contain //two arithmetic ops.
            else
                has_arith_op = TRUE;
            end if
        else if node == logical_op
            if has_arith_op==TRUE; //any prior node in the pattern is //an arithmetic operation
                return 0; //cluster not possible; a logical operation // cannot appear after an arithmetic operation
            else
                has_log_op =TRUE;
            end if
        else if node == shift_op
            has_shft_op =TRUE;
        end if
        }
return 1; //pattern is a valid cluster
}
```

4.2 PATTERN SELECTION FOR CGCI

Using the method proposed in the last section, we get the most profitable set of patterns in a given application. This set basically comprises of two types of patterns, each of which are specifically useful for different area constraint scenarios. The smaller patterns known as FGCIs, with critical path of 1 LUT perform very well when the area is highly constrained. As we showed in [20] and [21], when the area constrained is high, it is sufficient to identify such small patterns, which usually occur very frequently. On the other hand, the second type of patterns enumerated using the proposed method are the ones with critical path more than one, but which exhibit high instruction level parallelism. These patterns known as CGCIs, help us to achieve high performance in designs with relaxed area constraints.

Although the set of patterns identified contain all the profitable patterns in an application, a suitable pattern selection algorithm is a must to choose the right set of patterns for different area constraint. We propose a selection heuristic that promotes small patterns when the area constraint is high, but allows large patterns for implementation in designs with sufficient FPGA area. In order to achieve this goal, the enumerated patterns were grouped together using “VFLib” [22] to find isomorphic patterns. A group of isomorphic patterns was then represented as a corresponding template. Our selection algorithm follows a similar basic step described in [10] where a conflict graph is used to select a set of non-overlapping patterns. A conflict graph is an undirected graph where every node in the graph corresponds to a pattern. An edge in the conflict graph between two nodes represents an overlap between the corresponding patterns in the DFG. Algorithm 3 shows the proposed pattern selection algorithm. Once the conflict graph is created, we use the steps similar to the one described in [20] and [21] to compute the local Maximum Independent Set (MIS) to find a set of vertices in every template of the conflict graph that are mutually non-adjacent. Here we propose a new heuristic defined in equation 2 to compute the local MIS weight:

$$MIS\ Wt. = Frequency \times \left[\frac{No.of\ Nodes}{Critical\ path}\right]^3 \quad [2]$$

Algorithm 3 Pattern Selection

```
pattern_selection{
remaining_area=area_constraint;
Compute local MIS of patterns with area less than the area_constraint;
Compute weight of these patterns based on a heuristic (no.of nodes*Number of local MIS)/(Critical Path);
Sort all the templates in decreasing order of MIS weight;
Find the template with the largest MIS weight;

if (Area of this template ≤ remaining_area)
8               Choose this template for implementation;
9               remaining_area=(remaining_area)-(area of selected template);
10          Remove the overlapping patterns from the conflict graph;
11      Restore the patterns temporarily removed previously from the Conflict graph;
12      Recompute the local MIS of the patterns left in the conflict graph and Proceed to Step 4;

13  else if (Area of this template > remaining_area)
14          Choose the template with the next largest MIS weight;
15      if (Area of this template ≤ remaining_area)
16              Proceed with Step 8;
17          else
18              if No templates are left then
19                  exit;
20              calculate Performance with the selected templates;
21              else
22                  Proceed with Step 14;
23              end if
24          end if
25  end if
}
```

The new heuristic has two important differences from the heuristics used in [5] [10] [19] as well as our previous work in [20] and [21]. Firstly, we divide the existing heuristic (which favors large and frequently occurring patterns) by the critical path of the pattern. This enables a selection preference for large patterns with a high degree of parallelism when the area constraint of the design is relaxed. At the same time, since the critical path of a FGCI, by our definition is 1, the new heuristic also caters to the selection of smaller and frequently occurring patterns that can efficiently utilize the FPGA space when the area constraint of the design is tight. The second difference is the extra power of 3 to the second term in equation 2. This was found empirically to reinforce the selection of larger patterns with smaller critical path.

After calculating the local Maximum Independent Set, all the templates are sorted according to their MIS weight. Next, the algorithm starts with the template with the largest MIS weight to ensure constraint-aware pattern selection. This template is selected if and only if it does not violate the current area constraint. In case the available area is not sufficient to implement this pattern, it moves on to the template with second largest MIS weight and continues till it finds a suitable template and chooses it. After selecting a particular template for implementation, all the patterns in the maximum independent set of the selected template are implemented as custom instructions while also removed from the conflict graph. We also updated the remaining area constraint and recalculated the MIS weight for the remaining templates in the conflict graph. The algorithm is reiterated till either the area constraint is violated or the templates are exhausted in the conflict graph.

5. EXPERIMENTS AND RESULTS

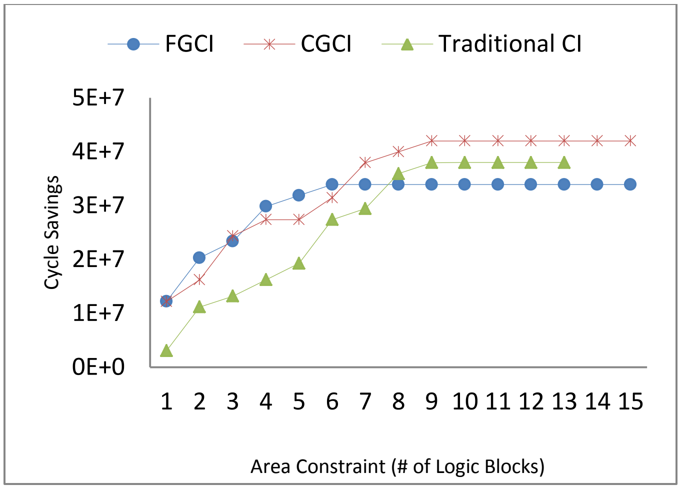
In this section, we compare the results obtained using the proposed FPGA-aware custom instruction generation approach with our previous work [21] and [20] and the existing state-of-the-art custom instruction generation approaches from [5][10][19][6]. We denote the existing state-of-the-art approaches as the traditional method. In traditional method, legal patterns are first enumerated, and this is followed by a pattern selection stage that attempts to select large and recurring custom instructions. The enumeration algorithm used in the traditional approach is from [6]. The pattern selection step uses the greedy selection approach described in [5] and [10]. We have also implemented a design exploration step for the traditional approach that relies on solving the knapsack problem [9][8] to identify the most profitable set of custom instructions (from the selected patterns) that meet the given area constraint. It should be noted that all the three different methods produce different sets of custom instructions based on what is deemed profitable by the corresponding heuristics.

We have observed an average reduction of 5.25% in the number of patterns enumerated using the proposed approach when compared to the traditional method. The reduction in the number of patterns enumerated by the proposed method also translates to the reduction in the enumeration runtime, though it is not very significant since our algorithm has to check for additional constraints during enumeration.

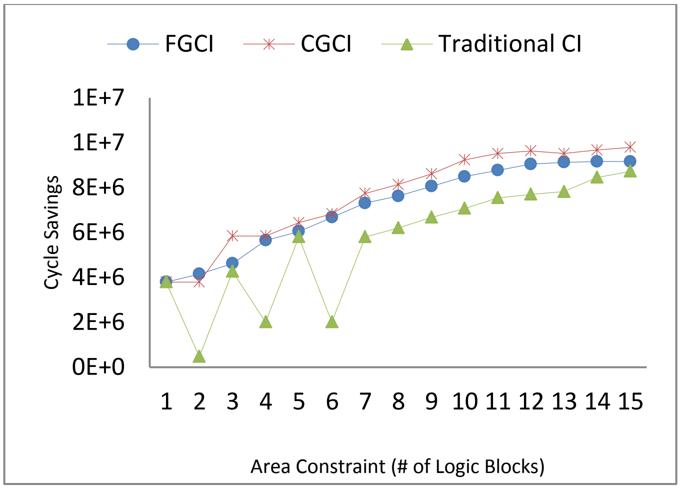
Now we compare the cycle savings achieved by using the proposed CGCI approach as opposed to the traditional approach, as well as our previously proposed FGCI. The performance achieved by using each of the three extended instructions was estimated using equation 1, which gives a fairly accurate estimation of the performance gained by using these extended instructions [19]. As mentioned in Section 4.1, we targeted our design for the Xilinx Virtex 2 FPGA, but the concepts and algorithms can be easily retargeted to any FPGA family. It should be noted that even a low end FPGA such as the Virtex 2 FPGA shows the advantage of our methodology to create designs that are more efficient. Better and newer FPGA families can achieve even better performance. For example, the newer FPGAs like Xilinx Virtex 5/6/7 or the Altera Stratix IV/V have adaptive-LUT structures with more number of inputs, I/O etc. and therefore can fit in larger patterns with more I/O, thus providing better performance per unit area. The frequency of operation of the processor after the addition of the extended instructions generated by any methodology is likely to fall compared to the base processor, especially in the case of instructions with larger critical path. Although the results in this work do not consider this effect of reduced processor frequency, it is still a fair comparison since the frequency would reduce by implementing any large custom instructions from any enumeration methodology.

It should be noted that the addition of custom instructions using any methodology is normally done under the presumption that the advantage of adding the custom instructions offsets the penalty paid because of reduced frequency. Moreover, the larger custom instructions can be implemented as a multi-cycle custom instruction in the newer FPGAs as part of the processor pipeline, thereby eliminating the penalty. In addition, our method is particularly efficient in this regard compared to the traditional method, since we ensure the **critical path of the extended instruction is as low as possible**.

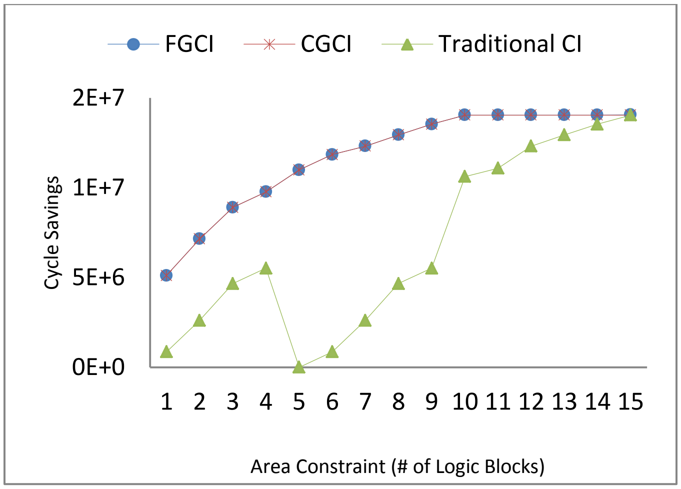
Figure 4 shows performance achieved in terms of clock cycles saved by using the custom instructions obtained by the various methods. The cycle counts were calculated using equation 1. Figures 4.1 to 4.6 show three trend lines, one for each of the three methods, Traditional in green, FGCI in blue and CGCI in red. The X-axis represents the FPGA area in terms of logic blocks. As defined before, a logic block is a set of 32 FPGA LUT with the same hardware configuration to implement a 32-bit operation. The number of logic blocks was limited to 15 as we experimentally found out that



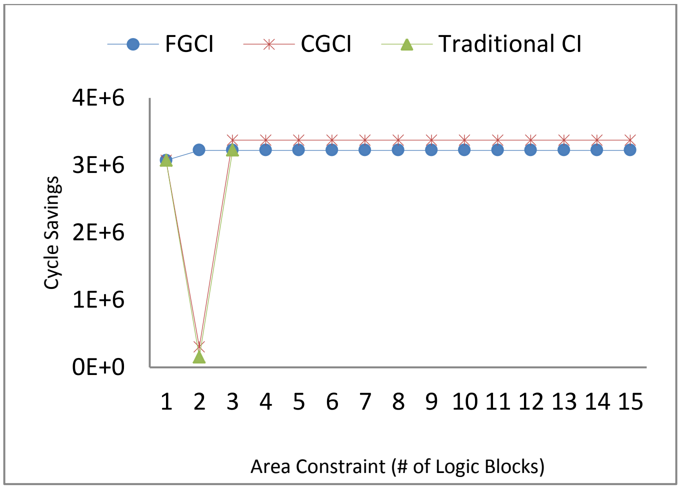
4.1 SHA



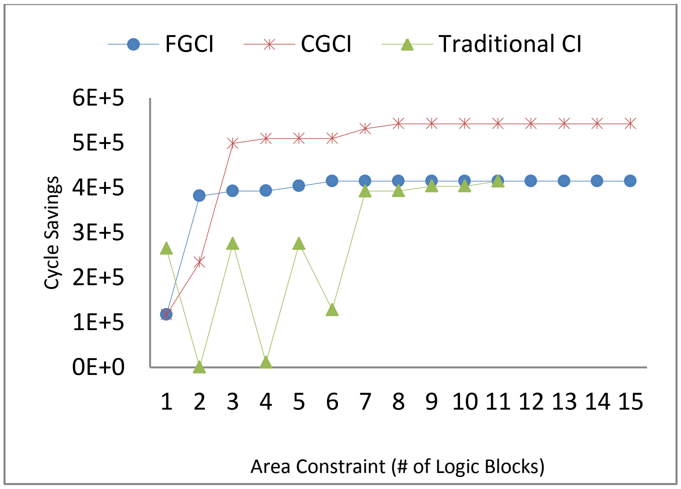
4.2 BLOWFISH



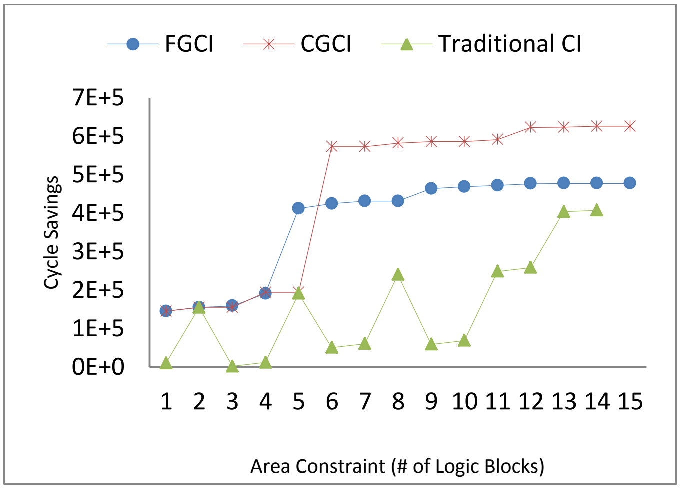
4.3 RIJNDAEL



4.4 QSORT



4.5 PATRICIA



4.6 CJPEG

most of the applications do not exhibit significant performance increment beyond that. This is also evident by the trend lines that tend to plateau as the area is increased to 15 logic blocks. This in turn means that we never implement more than 15 custom instructions in any case. It should be remembered that in this work we have defined a logic block as a set of 32 FPGA basic logic elements with identical hardware configuration that implements a 32-bit wide custom instruction. Table 2 shows the number of custom instructions implemented by the various methodologies. The numbers in the bracket show the area in logic blocks of these custom instructions.

A large number of custom instructions may lead to reduced clock frequency due to delay in the arbitrator logic between the soft-core processor and custom hardware. However, as shown in Table 2, the maximum number of custom instructions implemented for an application is only 15. Based on earlier investigation in the influence of the number of custom instructions to the delay of arbitrator logic that was reported in [30], we find that number of selected custom instructions in the proposed method will not have any notable impact on the clock frequency. In addition, we have also not taken into consideration the decrease in clock frequency due to the arbitrator logic in the conventional method and therefore providing a fair comparison in all cases.

Table.2. Number of Custom Instructions (and their area) implemented using the various methodologies.

Application	Traditional C.I.(Area)	FGCI.(Area)	CGCI.(Area)
SHA	13(13)	9(9)	10(12)
BLOWFISH	11(15)	15(15)	11(15)
RIJNDAEL	14(15)	15(15)	14(15)
QSORT	2(3)	2(2)	2(3)
PATRICIA	8(15)	8(8)	8(12)
CJPEG	9(14)	15(15)	10(15)

The sample applications were taken from two widely used Embedded Benchmark suites. They are representative applications from various domains; SHA, Blowfish, Rijndael are from the Security suite of MiBench[14]; QSORT from the Automotive; Patricia from Networking; and lastly CJPEG, a multimedia application from the MediaBench 2[15] benchmark suite.

The Y-axis shows the estimated clock cycles *saved* as calculated by equation 1. It can be observed that the CGCIs proposed in this work consistently perform better than both the traditional methods. It also overcomes in limitation of FGCI in applications such as SHA, BLOWFISH and PATRICIA, where the FGCI could not outperform the traditional method during relaxed area constraints. Evidently, the proposed method performs significantly better than the traditional method with an average improvement of over 76.23%. In applications such as Rijndael and Qsort, the CGCI method underperforms slightly when compared to FGCI. This is attributed to the fact that these applications predominantly consist of smaller and frequently occurring patterns where the FGCI method has been shown to perform exceptionally well.

Table 3 shows the average performance per unit area (area measured in number of logic blocks) achieved by the three methods. Like before, the performance is measured in terms of number of cycles saved by using the custom instructions generated by the various methods. From the figures 4.1 to 4.6, we found the area constraint where each of the methods provides maximum performance. We then divided this maximum performance with the corresponding area to get the average performance per unit area. It is evident that the proposed method outperforms the traditional method in all cases, while also being better when compared to the FGCI approach [20, 21] in almost all cases (and comparable with the remaining ones). This proves that the proposed CGCIs achieve the best performance per unit area and therefore are the most profitable set of custom instructions.

Table.3. Performance achieved per unit area using the proposed method against the Traditional Method and FGCI.

Application	Traditional C.I.	FGCI	CGCI
SHA	3167337.6	2824816	3501402
BLOWFISH	581967.13	610934	653067
RIJNDAEL	936244.87	937560	936260
QSORT	1074334.3	1074334	1124335
PATRICIA	37684.81	69088.8	67801
CJPEG	14285.92	31825.9	41725.7

Figure 5 shows the absolute speedup achieved when the proposed CGCIs and the traditional CIs were used as extended instructions when compared to the base processor implementation. The relatively low speedup in Rijndael and Cjpeg is attributed to the fact that we have assumed every instruction in the base processor takes exactly one clock cycle giving the base processor an unrealistic advantage. In reality, we expect the speedup to be higher. As the objective of this work is to demonstrate that the proposed method can lead to better solutions than the existing methods, we have refrained from performing detailed cycle level simulations to show a more realistic speedup over the base processor for the time being.

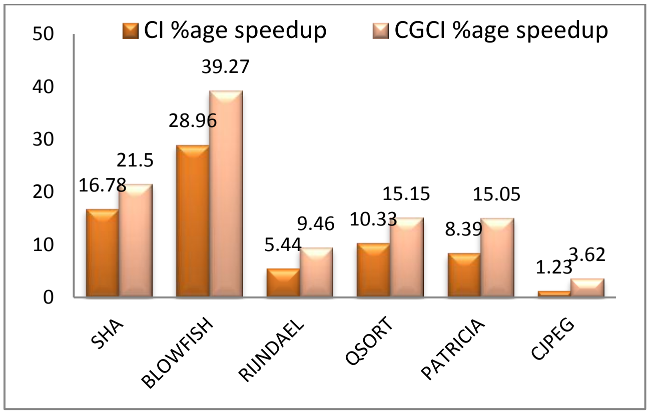


Figure 5. Average speedup achieved by using the proposed CGCIs against Traditional CIs, as extended instructions in the Applications

6. CONCLUSION

Instruction set extension, while being very useful for high performance embedded systems, needs careful selection of the extended instructions for achieving high performance per unit area. This work proposes new enumeration and selection heuristics to identify the most profitable custom instruction candidates for designs with arbitrary area. The proposed enumeration approach identifies large patterns that have relatively smaller critical path. Such patterns have been shown to achieve higher performance in designs with relaxed area constraints when compared to the existing work. The proposed template selection heuristic enables a selection preference for large patterns with a high degree of parallelism when the area constraint of the design is relaxed. Based on the results of our experiments on benchmark application from the MiBench and MediaBench Benchmark suites, we show that the proposed method achieves significant performance improvement of 76.23% on average over the current state-of-the-art approach. Our simulation results show that the proposed method can achieve a maximum speedup of 39.27% with an average speedup of 17.34% when compared to the base processor implementation.

REFERENCES

[1] Altera corp. NIOS II processors.(http://www.altera.com/products/ip/processors/nios2/ni2-index.html).

[2] Atasu, K., Mencer, O., Luk,W., Ozturan, C., Dundar, G. 2008. *Fast custom instruction identification by convex subgraph enumeration*. In: Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on. (July 2008) 1–6.

[3] Atasu, K., Pozzi, L., and Ienne, P. 2003.*Automatic application- specific instruction-set extensions under microarchitectural constraints.* In DAC '03: Proceedings of the 40th annual Design Automation Conference, pages 256–261, New York, NY, USA, 2003. ACM.

[4] Bohm A.C. Murray X. Qu M. Zuluaga O. Almer, R.V. Bennett and N.P. Topham.2009. *An end-to-end design flow for automated instruction set extension and complex instruction selection based on gcc*. In Proc. 1st International Workshop on GCC Research Opportunities (GROW'09), 2009.

[5] Bonzini, P., Pozzi, L. 2008. *Recurrence-aware instruction set selection for extensible embedded processors*. IEEE Trans. Very Large Scale Integr. Syst. 16(10) (2008) 1259–1267.

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

[7] Clark, N., Zhong, H., Mahlke, S. 2003. *Processor acceleration through automated instruction set customization*. In: MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, IEEE Computer Society (2003) 129.

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

[9] Cong, J., Fan, Y., Han, G., Zhang, Z. 2004. *Application-specific instruction generation for configurable processor architectures*. In: FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays, New York, NY, USA, ACM (2004) 183–189.

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

[11] Kluter, T., Burri, S., Brisk, P., Charbon, E. and Ienne, P. 2010. *Virtual Ways: Efficient Coherence for Architecturally Visible Storage in Automatic Instruction Set Extensions*. High Performance Embedded Architectures and Compilers, Lecture Notes in Computer Science Y. Patt, P. Foglia, E. Duesterwald et al., eds., pp. 126-140: Springer Berlin / Heidelberg, 2010.

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

[13] Li, T., Jigang, W., Lam, S.K., Srikanthan, T., Lu, X. 2009. *Efficient heuristic algorithm for rapid custom-instruction selection*. In: ICIS ’09: Proceedings of the 2009 Eigth IEEE/ACIS International Conference on Computer and Information Science, Washington, DC, USA, IEEE Computer Society (2009) 266–270.

[14] Mibench version 1.0(http://www.eecs.umich.edu/mibench/).

[15] Mediabench consortium(http://euler.slu.edu/ fritts/mediabench/).

[16] Pothineni, N., Kumar, A. and Paul, K. 2007. *Application specific datapath extension with distributed i/o functional units*. In VLSI Design, 2007. Held jointly with 6th International Conference on Embedded Systems., 20th International Conference on, pages 551–558, Jan. 2007.

[17] Pothineni, N., Kumar, A., Paul, K. 2008. *Exhaustive enumeration of legal custom instructions for extensible processors*. In: VLSI Design, 2008. VLSID 2008. 21st International Conference on. (Jan. 2008) 261–266.

[18] Siew-Kei Lam, Wen Li, and T. Srikanthan. High level area estimation of custom instructions for fpga-based reconfigurable processors. In Information, Communications & Signal Processing, 2007 6th International Conference on, pages 1–5, Dec. 2007..

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

[20] Prakash, A., Lam, S.K., Singh, A.K. and Srikanthan, T.: Architecture-Aware Custom Instruction Generation for Reconfigurable Processors. International Symposium on Applied Reconfigurable Computing, Bangkok, Thailand, March 2010, pp. 414-419.

[21] Prakash, A., Lam, S.K., Clarke, C.T. and Srikanthan, T.: Instruction Set Customization For Area-Constrained FPGA Designs. International SOC Conference, Taipei, Taiwan, September 2011.

[22] The vflib graph matching library, version 2.0 (http://amalfi.dis.unina.it/graph/db/vflib-2.0/doc/vflib.html).

[23] Trimaran compiler (http://trimaran.org/).

[24] Wang, P. and Bohacek, S. 2008. *On the practical complexity of solving the maximum weighted independent set problem for optimal scheduling in wireless networks*. In Proceedings of the 4th Annual international Conference on Wireless internet (Maui, Hawaii, November 17 - 19, 2008). ACM International Conference Proceeding Series. Institute for Computer Sciences Social-Informatics and Telecommunications Engineering, ICST, Brussels, Belgium, 1-9.

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

[26] Xilinx microblaze(http://www.xilinx.com/tools/microblaze.htm).

[27] Yu, P., Mitra, T. 2007. *Disjoint Pattern Enumeration for Custom Instructions Identification*. Field Programmable Logic and Applications, 2007. FPL. International Conference on, pp.273-278, 27-29 Aug. 2007.

[28] Zuluaga, M., Kluter, T., Brisk, P., Topham, N., Ienne, P. 2009. Introducing control-flow inclusion to support pipelining in custom instruction set extensions. SASP, pp.114-121, 2009 IEEE 7th Symposium on Application Specific Processors, 2009.

[29] Xilinx 7 series Overview, online, (*http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf*)

[30] Lam, S.K., Shoaib, M., Srikanthan, T.:Modeling Arbitrator Delay-Area Dependencies in Customizable Instruction Set Processors. IEEE Third International Workshop on Electronic Design, Test and Applications (DELTA), January 2006, pp. 237-242.