Rapid Evaluation of Custom Instruction Selection Approaches with FPGA Estimation

SIEW-KEI LAM, THAMBIPILLAI SRIKANTHAN
Centre for High Performance Embedded Systems
Nanyang Technological University
Nanyang Drive, Singapore 637553
and
CHRISTOPHER T. CLARKE
Department of Electronic and Electrical Engineering,
The University of Bath,
Bath BA2 7AY, United Kingdom

The main aim of this paper is to demonstrate that a fast and accurate FPGA estimation engine is indispensable in design flows for custom instruction (template) selection. The need for a FPGA estimation engine stems from the difficulty in predicting the FPGA performance measures of selected custom instructions. We will present a FPGA estimation technique that partitions the high-level representation of custom instructions into clusters based on the structural organization of the target FPGA, while taking into account general logic synthesis principles adopted by FPGA tools. In this work, we have evaluated a widely-used graph covering algorithm with various heuristics for custom instruction selection. In addition, we present an algorithm called Refined Largest Fit First (RLFF) that relies on a graph covering heuristic to select non-overlapping superset templates, which typically incorporate frequently used basic templates. The initial solution is further refined by considering overlapping templates that were ignored previously to see if their introduction could lead to higher performance. While RLFF provides the most efficient cover compared to the ILP method and other graph covering heuristics, FPGA estimation results reveals that RLFF leads to the worst performance in certain applications. It is therefore a worthy proposition to equip design flows with accurate FPGA estimation in order to rapidly determine the most profitable custom instruction approach for a given application.

Categories and Subject Descriptors: C.0 [**Computer Systems Organization]:** General - *Systems specification methodology*; C.0 [**Computer Systems Organization**]: General - *Instruction set design (e.g., RISC, CISC, VLIW)*
Additional Key Words and Phrases: Customizable processors, ISA extension, Approximation algorithms

---

1. INTRODUCTION

Reconfigurable processors offer the possibility of extending the basic instruction set of the microprocessor by introducing custom functional units on the reconfigurable space (e.g. Field Programmable Gate Arrays (FPGAs)) to implement custom instructions. A custom instruction typically encapsulates multiple primitive operations that constitute the critical portion of the application. The corresponding code segments associated with the custom instructions are implemented in hardware (i.e. Reconfigurable Functional Unit (RFU)). Reconfigurable computing platforms such as Stretch [STRETCH], NIOS II [ALTERA] and MicroBlaze [XILINX] create an environment in which this capability to extend the instruction set is enshrined in the basic architecture of the processor. They provide a toolset to allow the designer to modify the instruction set in order to meet the competing demands that exist in the embedded computing device product space.

It is possible through instruction set customization to transfer the burden of creating the custom instructions for a given application from the designer to an automated process. There are two major tasks in this automated process: firstly custom instruction identification and secondly custom instruction selection. *Custom instruction identification* detects a set of instances from the application Data Flow Graph (DFG) that satisfies certain constraints (e.g. number of inputs-outputs, convexity etc.). *Custom instruction selection* evaluates the relative area, power consumption and/or speed of the identified custom instructions and attempts to select a set that best meets the objectives of that particular design process. In this paper, the terms 'template' and 'custom instruction' are used interchangeably. Hence, template selection refers to custom instruction selection.

We aim to demonstrate the importance of a fast and accurate FPGA estimation engine in design flows for template selection. Existing methods often rely on crude strategies for estimating the hardware performance measures of custom instructions, which may lead to adverse decisions in template selection for FPGA implementation. In addition, the quality of results from different template selection approaches is influenced by the application, micro-architecture constraints and compiler infrastructure used in the design flow. The presence of an accurate FPGA estimation engine will therefore aid in the rapid determination of the most suitable template selection approach for a given scenario.

In this work, we evaluated a widely-used graph covering algorithm with various heuristics for template selection. We also present an approximate algorithm for template selection, called Refined Largest Fit First (RLFF) that is accomplished in two steps. The first step employs a graph covering algorithm to select a set of non-overlapping templates. The covering algorithm relies on a heuristic to select superset templates that tend to incorporate frequently used basic templates. This strategy is supported by our findings, which show that a significant number of frequently occurring templates are consumed by larger templates. In the second step, the initial solution is further refined by evaluating the benefits of selecting other templates that overlap with the existing solution. Performance analysis shows that RLFF outperforms other well-known heuristics and an Integer Linear Programming (ILP) approach, in terms of number of nodes covered. However, FPGA estimation results reveal that RLFF leads to the worst performance in certain applications. This affirms the need for an accurate FPGA estimation engine for template selection.

The observations of the templates' characteristic (i.e. most of the frequently occurring templates are consumed by larger templates) were initially reported in our preliminary

work in [LAM 2006a] for a restricted set of applications. In this paper, we performed proper template classification in order to study the statistical properties of the templates, and validated this characteristic with a larger set of applications. In addition, based on our findings, we present a graph covering strategy (i.e. RLFF) that maximizes the number of nodes covered.

We will present a FPGA estimation technique that partitions the high-level representation of custom instructions into clusters based on the structural organization of the target FPGA, while taking into account general logic synthesis principles adopted by FPGA tools. A preliminary version of this technique has been presented in [LAM 2009]. Based on this preliminary technique, we have shown that area-time efficient custom instructions can be achieved by merging clusters in order to maximize the utilization of FPGA logic blocks [LAM 2011]. In this paper, we extend our previously reported FPGA estimation technique by formulating delay and area estimation models that take into account the characteristics of the operations to accurately estimate the critical path and FPGA area utilization of the clusters. We also evaluated the accuracy of the proposed estimation technique on an extensive set of custom instructions from a large number of benchmark applications. Experimental results show that the average estimated critical paths of 150 custom instructions from sixteen applications using the proposed method are only within 3% of those obtained using hardware synthesis. In addition, the average estimated area utilization using the proposed method are within 1% of those obtained from FPGA implementation results.

This paper starts with a discussion of the existing methods for template selection and FPGA estimation. Section 3 describes the various template selection approaches that have been used in our evaluation. This include a novel graph covering algorithm that provides the most efficient cover compared to the ILP method and other widely-used graph covering heuristics for template selection. In Section 4, we present the proposed FPGA estimation strategy. Experimental results are shown in Section 5 to demonstrate the benefits of incorporating the proposed FPGA estimation method in design flows for template selection. We conclude the paper in Section 6.

2. RELATED WORK

Template selection evaluates the area, speed, and/or power consumption of template instances and uses these metrics to select the subset that best meets the objectives of the design. Our work published in [LI] has shown that exact algorithms for template selection

are prohibitive for large sized problems. Hence, approximate solutions are often used for template selection.

In [KASTNER], the approach proposed was to maximize the number of covered nodes whilst utilizing a minimal template set through the use of a covering algorithm. In [CLARK 2003][CLARK 2005][POZZI], greedy selection policies were employed to heuristically select a subset of templates. The work in [ATASU][CONG] formulated the template selection process as a knapsack problem. Each template instance is associated with a performance gain and area cost. A dynamic programming algorithm is then employed to select a subset of the instances that maximizes the performance gain subjected to an area cost bound. ILP based methods for template selection have also been discussed in [LEE 2002][GALUZZI][YU]. The method presented in [GUO] employs a graph-covering algorithm, formulated as the Maximum Independent Set (MIS) problem, on a conflict graph to maximize the number of covered nodes using a minimum number of templates. The work in [BONZINI] proposed a hybrid algorithm for recurrence-aware template selection that combines a greedy covering algorithm and an exact branch and bound algorithm that operates on a restricted problem space. Even though the technique in [BONZINI] has restricted the problem size for the exact algorithm, it still requires a runtime in the order of seconds for certain applications. [YAZDANBAKHSH] investigated the effects of selecting local and global templates using graph covering algorithms, and reported that locally selected templates lead to better results in terms of performance and performance per area.

Instruction set customization research is heavily focused on the selection of near-optimal sets of custom instructions from a larger candidate population. These approaches do not incorporate an accurate method for assessing the impact of the architecture and its resultant constraints and this has an impact on the achievable performance gains from the selected custom instructions. A number of approaches make use of pre-computed area-time values resulting from synthesis in a standard cell design flow in [CLARK 2003][CLARK 2005][LI][POZZI][YU]. These values form the basis of cumulative area-time values for custom instruction candidates. A similar approach using operator throughput as the major metric combines the individual throughput values to form an overall throughput estimate for the custom instruction. In the constrained environment of FPGA based implementation however, these approaches are less effective. The alternative approach taken by some authors (e.g. [ATASU][SUN 2004][YAZDANBAKHSH]) of including full hardware synthesis within the custom

instruction selection process is more effective but very slow in the design stage, making design exploration a significant challenge.

It is possible to take a high-level approach to area-time estimation based solely on algorithmic representations of the design. Physical hardware implementation is not required in these methods which results in considerable reductions in the computational effort required to generate an estimate. The high level techniques do not require a gate level implementation of the custom instruction giving them a considerable advantage over technology mapping based approaches such as [CHEN] and [JOEY].

A formula based upon register properties and the operator in use has been reported as an approach for area estimation [NAYAK]. The parameters for this formula are derived from pre-synthesized register transfer level (RTL) descriptions of the relevant operators and register configurations. This approach to pre-characterization of the area leads to error estimates below 16% when compared to implementation using commercial synthesis engines. Execution traces have also been used to generate DFGs [BJURÉUS]. Area-time estimates are then generated based upon the frequency of occurrence of the operations observed. Feeding this data into a performance model that has been pre-characterized for the target FPGA results in area estimates that are within 10% of actual implementation results. A two level model for area estimations in a System-C environment has been reported [BRANDOLESE]. The upper level of the model takes the input code and provides the lower level model with a list of intermediate variables which can be used to estimate the number of Look-Up Tables (LUTs) and Flip Flops via a set of equations proposed by the authors. Re-tuning of the equations is needed each time there is a change in either the design tools or the target FPGA family. Reported area estimate errors average approximately 17% for this approach. High-level SA-C codes are used in [KULKARNI] to create a DFG. This DFG is then used to calculate data-path area estimates using a formulaic approach. Resource consumption characterization combined with heuristic pattern analysis yield area estimates within 5% of the actual implemented areas. The heuristics account for typical synthesis engine optimizations. Area-time estimates of RTL solutions are created in a two-step process in [BILAVARN]. The process starts with a structural exploration step to create a range of possible RTL solutions. The second step maps the RTL candidates to area-time estimates for the given architecture based upon a characterization file for the target FPGA. The characterization file includes a range of information on the target architecture such as basic operator and memory timing from the device data sheet and operator block synthesis results. The mean error in area estimation for this approach over a range of FPGA vendors was 18%.

The separation of the methods discussed above from the constraints imposed by the FPGA architecture may lead to unreliability in the estimates produced by those methods. Synthesis optimization can exacerbate this limitation. Pre-characterization is also employed by these methods limiting the range of potential solutions considered. Hence, there is a need to develop more reliable FPGA estimation techniques in order to facilitate template selection.

2.1 Our Contributions

This work aims to show the essentiality of an accurate FPGA estimation for template selection, which is often taken for granted in existing work. We first present a novel strategy for the rapid selection of custom instructions based on a graph covering approach. This strategy is motivated by our investigations which reveal that a majority of frequently executed custom instructions are consumed by larger custom instructions. Comparisons with previously reported approximate strategies show that our technique selects custom instructions with highest performance gain (in terms of number of nodes covered). In order to rapidly evaluate the FPGA performance measures of template selection techniques, we propose a strategy that permits the estimation of the achievable clock rate of systems that incorporate those templates. These estimates include parameterized target FPGA data allowing the estimates to take these architectural constraints into account without requiring a pre-characterization step. The major input to the estimation process is the Intermediate Representation (IR) generated by an ANSI-C compiler. This makes it directly applicable to the majority of embedded applications. The proposed technique is applicable for direct implementation on existing, commercially available FPGA architectures but includes parameterization that allows it to be applicable to multiple FPGA families from most device vendors. This parameterization also provides applicability to future architectures. Finally, we demonstrate the necessity and effectiveness of the proposed FPGA estimation technique for evaluating the performance gain of custom instructions that are selected using various template selection approaches. The proposed approach can be used to rapidly determine if custom instructions should be introduce for fine-grained acceleration in programmable system-on-a-chip alongside with other means of acceleration (e.g. co-processors) [SUN 2007].

3. GRAPH COVERING FOR TEMPLATE SELECTION

The following describes the problem formulation of template selection. This approach is based upon the concept of graph-covering:

Given an application DFG G, a unique set of templates $T = \{T_1, T_2, \ldots, T_i\}$ and the template instances of each template T_i, $I_i = \{I_{i,1}, I_{i,2}, \ldots, I_{i,j}\}$, find a subset of the set I that covers G. Figure 1(a) shows an example with three templates (i.e. T_1, T_2 and T_3) and nine instances in a DFG. An efficient cover can be achieved by selecting a set of non-overlapping instances that maximizes the number of covered nodes. Existing work often assumes that a template selection approach that result in an efficient cover can lead to higher performance gain as the instances cover a larger number of operations.

The covering algorithm that we have adopted is based on the conflict graph approach that was presented in [GUO]. A conflict graph is an undirected graph $G_u(V_u,E_u)$. Each vertex represents a template instance $I_{i,j}$ that is associated with a unique template T_i. An edge $e \in E_u$ between two instances signifies that the instances have at least one overlapping node. The number of nodes in an instance $I_{i,j}$ is denoted as $size(I_{i,j})$. Figure 1(b) shows the conflict graph for the example in Figure 1(a). In this example, we assume $size(I_{1,j}) = 5$, $size(I_{2,j}) = 9$, and $size(I_{3,j}) = 1$.

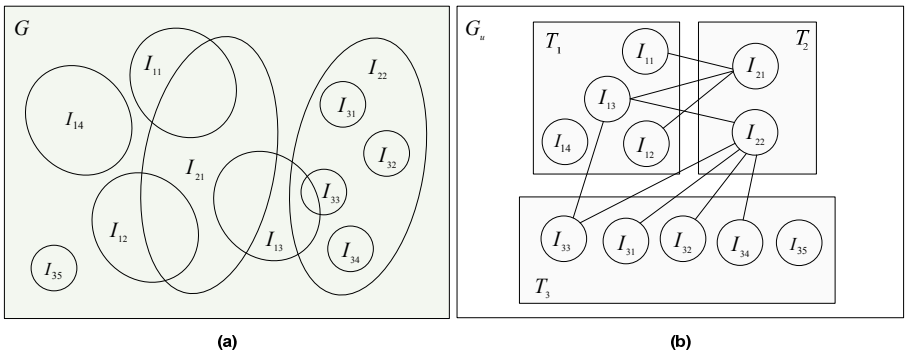


Figure 1: (a) Template instances in a DFG G, (b) Conflict graph of G

The covering algorithm starts by taking all the template instances and constructing a conflict graph with them. For each unique template T_i the Maximum Independent Set (MIS) (referred to hereinafter as MIS_i) is the largest subset of instances in T_i for which those instances do not share any common edges (they are mutually non-adjacent). This is established using an iterative approach. The term $size(MIS_i)$ is used in this paper to indicate the number of instances that are contained within MIS_i . The computation of the MIS can be implemented in time linear in the number of vertices and edges of G_u by

using a simple heuristic, which has been shown to provide good solutions [HALLDÓRSSON].

The MIS_i with the largest objective function ($w(MIS_i)$) is then selected. All instances that match the selected MIS then become selected instances. After selection, these instances and their neighbors can be removed from the conflict graph. This algorithm is repeated until the conflict graph is empty. Details of the algorithm are described in Figure 2. NON_i refers to the non-overlapping nodes of the selected instances in T_i. As shown in line 7 of Algorithm 1, the NON_i of each selected template T_i is stored in C. In the worst case, the number of iterations required by the selection algorithm is equivalent to the number of vertices in G_u.

The choice of objective function (i.e. $w(MIS_i)$) will have significant impact on the template selection process. We will discuss the objective functions that are analogous to commonly used heuristics in existing template selection methods, before presenting a novel strategy for template selection based on the graph covering approach. The description of the objective functions and our strategy in the subsequent sections will refer to the example in Figure 1.

Algorithm 1

```
1.  TEMPLATE-SELECTION (C, G_u) {
2.      G_u' = G_u
3.      while G_u' ≠ φ {
4.          Find MIS_i of each template group T_i in G_u'
5.          Compute w(MIS_i) for each MIS_i
6.          Select MIS_i with the largest objective function
              (corresponding T_i is the selected custom instruction)
7.          Store nodes corresponding to the selected MIS_i (i.e. NON_i) in C
8.          Delete NON_i and the adjacent nodes from G_u'
9.      }
10. }
```

Figure 2: Pseudo code of conflict graph based template selection

3.1 Most-Frequently-Fit-First (MFF)

This objective function for MFF is: $w(MSI_i) = size(MSI_i)$. It aims to select a set of frequently occurring templates, as the algorithm will select the MIS with the largest number of instances first. This approach has a similar objective to the work in [KASTNER]. Figure 3(a) shows the final covering solution with MFF for the example in Figure 1 after two iterations. In the first iteration, MIS_3 will be selected as it has the

largest objective function (i.e. $size(MIS_3)$ = 5). MIS_1 is then selected in the subsequent iteration. The result of the algorithm is the selection of instances $I_{1,1}$, $I_{1,2}$, $I_{1,4}$, $I_{3,1}$, $I_{3,2}$, $I_{3,3}$, $I_{3,4}$ and $I_{3,5}$ that are associated with the templates T_1 and T_3. The gain (measured in terms of the total number of nodes covered) is 20.

3.2 Most-Frequent-Largest-Fit-First (MLFF)

The objective function for MLFF is: $w(MSI_i) = size(v_x) \times size(MSI_i)$, which takes into account both the frequency of template occurrence and the size of the templates. This objective function is analogous to the heuristic used in [GUO][BONZINI]. Figure 3(b) shows the covering solution of MLFF for the example in Figure 1 after two iterations. MIS_1 is selected in the first iteration and this is followed by selection of MIS_3 in the subsequent iteration. The result of the algorithm is the selection of T_1 and T_3 (corresponding to $I_{1,1}$, $I_{1,2}$, $I_{1,3}$, $I_{1,4}$, $I_{3,1}$, $I_{3,2}$, $I_{3,4}$ and $I_{3,5}$) which covers 24 nodes.

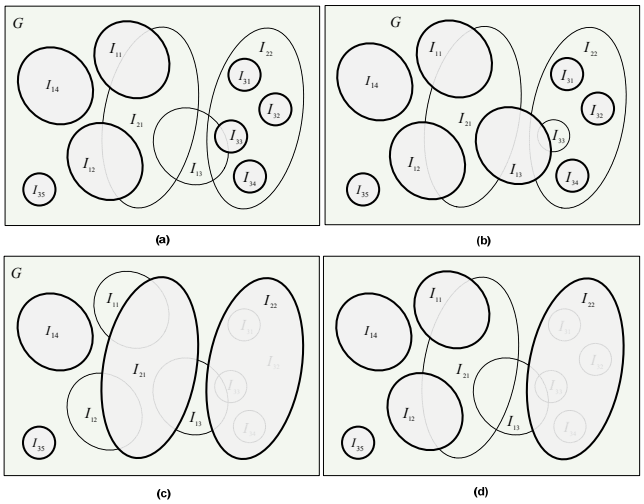


Figure 3: (a) MFF based selection (Gain = 20), (b) MLFF based selection (Gain = 24), (c) LFF based selection (Gain = 24), (d) Refining LFF (Gain = 25)

3.3 Largest-Fit-First (LFF)

This objective function for the LFF heuristic is: $w(MSI_i) = size(v_x)$. The LFF approach attempts to select the MIS with the largest instances first. Where there are small templates which have a large number of instances (e.g. $I_{3,j}$), the MFF has a tendency to select these in preference to other templates with a large node count such as $I_{2,2}$, and the LFF strategy attempts to remove this small template bias. Figure 3(c) shows the covering solution of LFF for the example in Figure 1 after three iterations. MIS_2 is selected in the

first iteration as it has the largest instances (i.e. $size(I_{2,j})$ = 9). This is followed by MIS_1 and MIS_3 in the subsequent iterations. The result of the algorithm is the selection of T_1, T_2 and T_3 (consisting of instances $I_{1,4}$, $I_{2,1}$, $I_{2,2}$, and $I_{,35}$) which covers 24 nodes. From Figure 3(c), it can be observed instances $I_{3,1}$, $I_{3,2}$, $I_{3,3}$ and $I_{3,4}$ are consumed by instance $I_{2,2.}$ More generally, where a large proportion of the instances that belong to frequently occurring templates are covered by larger instances, the LFF approach can lead to a larger number of covered nodes. However, partial overlap of a frequently occurring template by a larger instance will cause the frequently occurring template to be discarded. In the experimental section, we will present statistical analysis of the spatial locality of templates in a number of applications to justify the feasibility of the LFF approach.

3.4 Refining LFF

In this sub-section, we present a novel strategy to increase the number of nodes covered for template selection using the graph covering approach.

Algorithm 2

```
1.  RLFF-BASED-TEMPLATE-SELECTION {
2.      C = φ
3.      Build conflict graph G_u from DFG G_i
4.      TEMPLATE-SELECTION (C, G_u) with LFF objective function
5.      REFINE-SELECTION (C, G_u)
6.  }
```

Algorithm 3

```
1.  REFINE-SELECTION (C, G_u) {
2.      G_u' = G_u
3.      Calculate gain of each selected template T_i based on NON_i in C
4.      Sort the selected T_i in ascending gain
5.      for each selected T_i starting with the lowest gain {
6.          Remove corresponding NON_i from C
7.          Identify all neighboring vertices of MIS_i in G_u' and store in N, where
              the nodes corresponding to the vertices in N ∉ C
8.          Find MIS of N (MIS_N)
9.          Calculate gain of MIS_N
10.         if gain(MIS_N) > gain (NON_i)
11.             Store nodes associated with MIS_N in C
12.         else restore NON_i in C
13.     }
14. }
```

Figure 4: Pseudo code of RLFF algorithm

The LFF based covering algorithm can be further refined by evaluating the benefits of replacing the initial solution with the non-selected template instances that overlap with the selected instances. This is achieved by evaluating the instances of each selected template (in order of ascending template gain) in the initial solution to check whether they should be replaced by the overlapping instances. For example, in Figure 3(d), the initial selected instance $I_{2,1}$ in Figure 3(a) is replaced with the instances $I_{1,1}$ and $I_{1,2}$ as they lead to higher gain (i.e. $gain(I_{1,1}) + gain(I_{1,2}) > gain(I_{2,1})$). We denote the LFF algorithm with refinements as RLFF.

The RLFF algorithm is described in Algorithm 2 and 3 of Figure 4. First the gain of NON_i of each selected template T_i is calculated (line 3 of Algorithm 3). Starting from the selected template T_i with the lowest NON_i gain, the corresponding NON_i is temporary removed from C. The non-selected instances that overlap with the selected instances T_{ij} are identified and stored in N (Line 7). Note that the instances in N must not overlap with any selected instances $I_{k,j}$ where $k \neq i$. The MIS of N is then computed to find a maximal non-overlapping set of instances in N (line 9). If the gain of MIS_N is larger than the gain of NON_i, then the instances of N replace the initial instances of T_i (line 11). Otherwise, the original selected instances in T_i are restored (line 12). This process is repeated until all the selected templates T_i in the initial solution have been considered. As can be observed from Figure 3(d) (based on the example in Figure 1), the proposed method leads to the highest gain among the various objective functions (i.e. gain = 25) on the given example.

3.5 Feasibility Study of the LFF and RLFF Approach

In Section 3.3, we explained that the LFF approach can lead to a larger number of covered nodes if (and only if) most of the instances belonging to frequently occurring templates are entirely consumed by large instances. This also applies to the RLFF approach. In order to investigate the feasibility of the LFF and RLFF approach, we analyzed the spatial locality of the template instances in sixteen benchmark applications. Our experiments are based on applications that are obtained from the widely-used MediaBench [LEE 1997], MiBench [GUTHAUS], and EEMBC [EEMBC] benchmark suites.

Exhaustive template enumeration has been carried out using the method proposed in [POZZI] which combines a tree search with a constraint violation based pruning process. A pre-register allocation IR is used to prevent false dependencies from influencing the experimental results. This IR is generated using Trimaran [TRIMARAN] which performs

loop unrolling and loop pipelining to expose the instruction-level parallelism of the application. The same constraint set as described in [LAM2009] is used in these enumeration experiments. In particular, only integer operations are allowed in the template instance. Including memory accesses in custom instructions can lead to non-deterministic latencies and increased complexity. In addition, custom instructions with floating-point operations often do not lead to notable speedup [YU]. Maximum number of input ports is 5 and maximum number of output ports is 2. Previous work [YU] has shown that input-output ports more than this range result in little performance gain. Finally, only convex sub-graphs are allowed in template instances to ensure a feasible schedule exists when the sub-graph is collapsed into a custom instruction.

Templates can be divided into: 1) *Superset templates*, 2) *Basic templates*, and 3) *Others*. A superset template consists of large template instances that cannot be entirely consumed by other instances or subsume one or more basic template instances. In Figure 1(a), T_2 is a superset template as both its instances ($I_{2,1}$ and $I_{2,2}$) are not entirely consumed by other template instances. In addition, $I_{2,2}$ subsumes template instances $I_{3,1}$, $I_{3,2}$, $I_{3,3}$ and $I_{3,4}$. A basic template consists of template instances that do not subsume any template instances or are entirely consumed by superset templates. In Figure 1(a), T_3 is a basic template as its instances $I_{3,1}$, $I_{3,2}$, $I_{3,3}$ and $I_{3,4}$ are entirely consumed by $I_{2,2}$. Template T_1 in Figure 1(a) falls under the category 'Others' as none of its instances ($I_{1,1}$, $I_{1,2}$, $I_{1,3}$ and $I_{1,4}$) subsumes other instances or are entirely consumed by other instances.

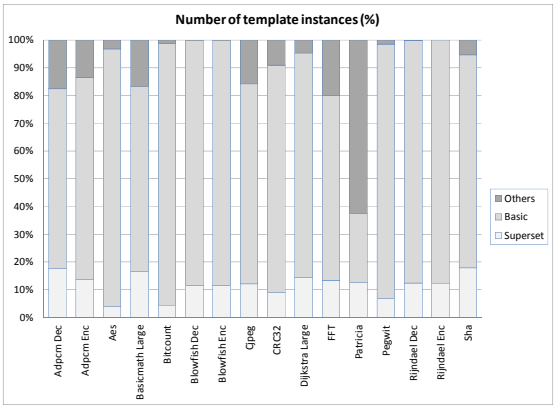


Figure 5: Percentage of different template types

Figure 5 shows the percentage of instances of the three template types for the sixteen benchmark applications. The statistics imply that a significant number of frequently occurring templates (i.e. basic templates) are contained within the superset templates. In

particular, an average of 77.4% of the template instances, are basic templates that are consumed within the superset templates in the sixteen applications. Hence, employing LFF and RLFF for template selection can lead to the selection of large templates that are also likely to subsume frequently occurring templates.

3.6 Efficiency of Graph Covering Heuristic

In this sub-section, we will compare the performance of the selected templates that are obtained using MFF, MLFF, LFF and RLFF. In addition, we have implemented an ILP method for template selection in order to evaluate the quality of the solutions obtained using the proposed methods.

Let's define T_i, for i = 1,2, ..., n, as a unique template, where n is the total number of templates obtained from template enumeration. A template T_i can have n_i number of instances in the application denoted by $I_{i,1}$, $I_{i,2}$, …, $I_{i,ni}$. Each instance has an execution frequency of $F_{i,j}$. Let $size(I_{i,j})$ denote the number of nodes in instance $I_{i,j}$. We define binary variables $x_{i,j} \in \{0,1\}$, which is equal to 1 if instance $I_{i,j}$ is selected and 0 otherwise. Finally, $G_u(V_u,E_u)$ is the conflict graph, where $(I_{i,j},I_{k,l}) \in E_u$ if the instances $I_{i,j}$ and $I_{k,l}$ overlaps. The objective function of ILP for template selection is formulated by maximizing the total performance gain (in terms of number of nodes/operations):

$$\max : \sum_{i=1}^{n} \sum_{j=1}^{n_i} \left(x_{i,j} \times F_{i,j} \times size(I_{i,j}) \right) \qquad (1)$$

We optimize the objective function under the constraint that none of the selected instances overlap.

$$x_{i,j} + x_{k,l} \leq 1 \qquad \left(I_{i,j}, I_{k,l}\right) \in E_u \qquad (2)$$

The ILP-based method, although capable of producing optimal solutions, can have extremely long runtime due to the complexity of the problem. As shown in Table 1, the ILP-based method does not produce optimal solutions for half of the applications considered even after 1 hour of runtime. On the other hand, the runtime of MFF, MLFF, LFF and RLFF are less than 1 second for all the applications considered. In the following experiments, we have obtained results for the ILP approach after 15 minutes of runtime. The optimality of the solutions produced by the ILP approach is the same as that in Table 1. While it may be possible to obtain better solutions when the ILP-based method is executed for a longer time, this will not be acceptable in view of the tight time-to-market pressures faced by the industry.

Figure 6 shows the number of selected templates that are obtained using the various approaches. The average number of selected templates obtained using MFF, MLFF, ILP, LFF and RLFF is 8.8, 9.1, 11.2, 11.6 and 12.3 respectively. It is evident from this set of results that LFF and RLFF lead to the selection of a larger number of templates compared to the existing approaches. The results also show that template selection based on the frequency of occurrence of the templates (e.g. MFF and MLFF) can lead to the disposal of a notable number of overlapping templates.

Table I. Optimality of ILP based template selection after 1 hour

Application	Optimal Solution
Adpcm Dec	Yes
Adpcm Enc	Yes
Aes	No
Basicmath Large	Yes
Bitcount	No
Blowfish Dec	No
Blowfish Enc	No
Cjpeg	No
CRC32	Yes
Dijkstra Large	Yes
FFT	Yes
Patricia	Yes
Pegwit	No
Rijndael Dec	No
Rijndael Enc	No
Sha	Yes

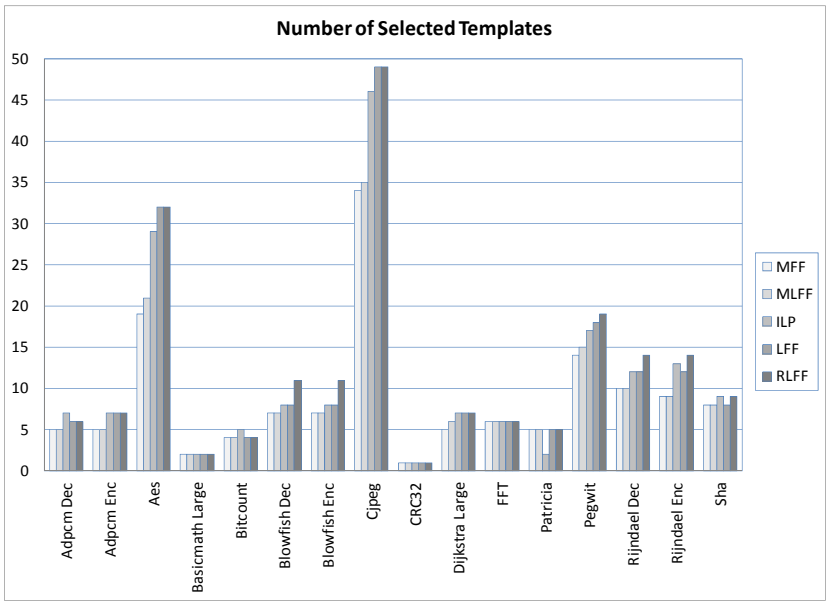


Figure 6: Number of selected templates

In this sub-section, the performance gain is reported in terms of the number of nodes covered in the various graph covering approaches. This metric is analogous to the reduction in the number of Instruction Set Architecture (ISA) operations executed on the base processor. Figure 7 shows the gain contribution of the selected templates that have been grouped according to their size, in terms of number of nodes (as labeled on the charts), for MFF, MLFF, ILP, LFF and RLFF respectively. For example in Adpcm Dec, only templates of size 2 are selected using MFF and MLFF, while templates of size 2 and 3 are selected using ILP, LFF and RLFF. The results for Blowfish Enc are the same as Blowfish Dec, and hence are not shown.

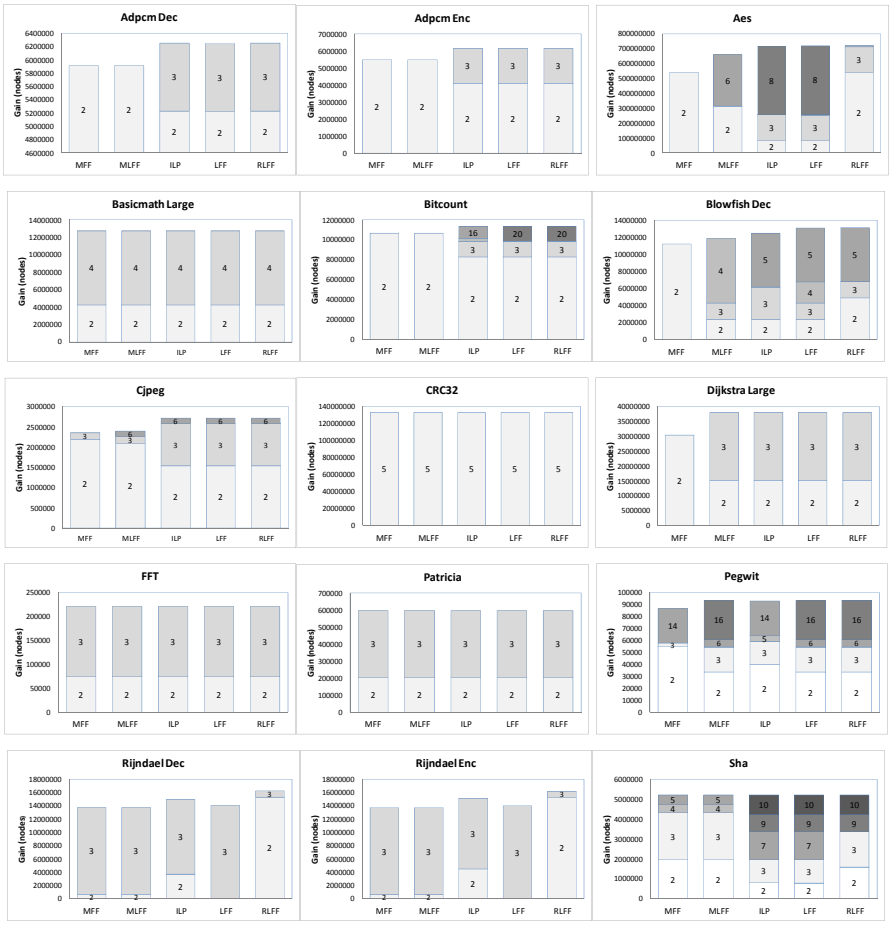


Figure 7: Percentage performance contribution of selected templates

When compared to the MFF and MLFF approaches, LFF and RLFF generally leads to the selection of more templates (see Figure 6) with larger number of nodes (see Figure 7), which has resulted in higher gain. For Aes, Rijndael Dec and Rijndael Enc, RLFF leads

to the selection of smaller templates compared to those selected using MFF and MLFF approaches. However, as shown in Figure 6, the RLFF approach leads to the selection of more templates and hence, the overall gain of RLFF is still higher than MFF and MLFF for these applications. In general, the selected templates of LFF have the largest number of nodes.

On average, the LFF method outperforms the MFF and MLFF method by 9.0% and 4.6% respectively. For Aes and Dijkstra Large, LFF outperforms MFF by over 32% and 24% respectively. The LFF approach also outperforms the MFF and MLFF methods by over 10% for a number of applications. Except for two applications (i.e. Rijndael Dec and Rijndael Enc), LFF either performs more favorably or is comparable with the ILP approach. For the Rijndael Dec and Rijndael Enc applications, the percentage performance gain difference between the LFF and ILP approach is less than 10%.

The RLFF method outperforms the LFF method by over 15% in Rijndael Dec and Rijndael Enc. On average, the RLFF method outperforms the MFF and MLFF method by 11.0% and 6.6% respectively. In addition, when compared to the ILP method, the proposed RLFF approach has a higher number of nodes covered in several applications (e.g. Blowfish Dec, Blowfish Enc, Rijndael Dec and Rijndael Enc), and is comparable in the remaining ones. It is noteworthy that LFF and RLFF can execute in a fraction of the time that is required by the ILP method.

Based on the experiments discussed in this section, it can be deduced that template selection based on frequently occurring templates (using the MFF and MLFF approach) can result in the disposal of a notable number of overlapping templates that are likely to have a larger number of operations. This concurs with our earlier analysis which shows that a significant number of frequently occurring templates (i.e. basic templates) are contained within the superset templates. Hence, we can conclude that template selection strategies that give preference to the selection of large templates can lead to better results in all the applications considered. In addition, the experimental results show that the RLFF method leads to the highest gain (in terms of number of nodes covered).

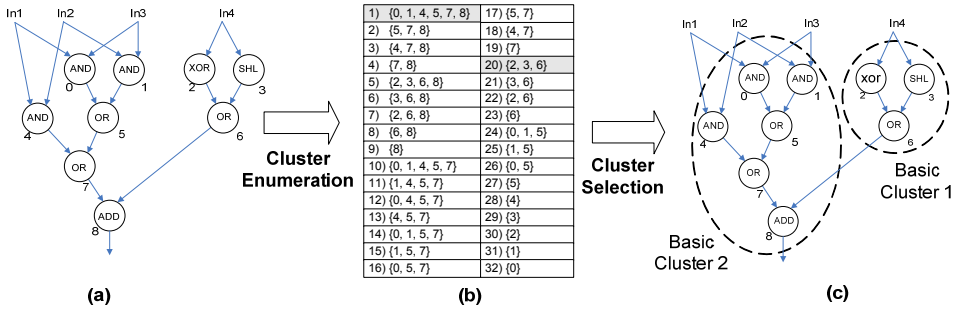
4. PROPOSED FPGA ESTIMATION TECHNIQUE

In this section, we proposed the cluster generation technique for FPGA estimation.

Definition 1: A template (custom instruction) can be defined as a directed graph $G_i = (V_i, E_i)$ for $i = 1,2,...,n$ and n is the number of selected templates, where:

- A vertex $v \in V_i$ for $1 \le i \le n$ is a primitive integer operation in a compiler's IR. Each vertex is associated with at most two input ports and one output port. These operations can be categorized as 1) arithmetic i.e. addition (*ADD*), subtraction (*SUB*), multiplication (*MUL*), division (*DIV*), 2) logical (*AND*, *OR*, *XOR*), and 3) relational e.g. logical/arithmetic shift by a constant/non-constant (*SHL*, *SHR*, *SHRA*).
- An arc $e = (u, v) \in E_i$, indicates a data transfer from vertex *u* to vertex *v*, whereby the output port of *u* is connected to one of the input ports of *v*.

Definition 2: A cluster $C_i^j = \left(V_i^j, E_i^j\right)$ is a sub-graph of a template G_i, which can be implemented either: 1) on a set of FPGA logic blocks with the same configuration, or 2) using FPGA embedded IP cores. In particular, multipliers, dividers and shift by non-constant operations are implemented using widely available IP cores such as DSP slices in the Xilinx devices [XILINX 2012]. None of the clusters in G_i overlap, i.e. $V_i^j \cap V_i^k = \phi$ and $E_i^j \cap E_i^k = \phi$ for $j \ne k$. In addition, $\bigcup_{j=1}^{c} V_i^j = V_i$ and $\bigcup_{j=1}^{c} E_i^j = E_i$, where *c* is the number of clusters in G_i.



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 8: (a) Cluster enumeration, (b) cluster instances, (c) cluster selection

Cluster generation partitions the templates into a cluster set, where the template is entirely covered by that cluster set. The process of generating clusters can be further sub-divided into a cluster enumeration step and cluster selection step. These steps are illustrated in Figure 8, where Figure 8(a) shows an example template with nine vertices (primitive operations). In cluster enumeration, the template is decomposed into connected sub-graphs that are realizable in the logic block architecture of target FPGA. These become the cluster instances. Typical FPGA targets can replicate the single logic block in a tightly coupled group to provide the multi-bit (usually 32 bit) architecture required for

the custom instruction. The tight coupling provides inter-bit routes for carry propagation in the case of primitive operations such as *ADD*. Figure 5(b) shows the 32 cluster instances that will be enumerated in our example for the case $K = 4$. From this point onwards, we refer to a set of logic blocks that is configured to implement a cluster as a single logic block, without any loss of generality. A set of clusters is then selected to effectively cover the template in order to meet a certain criteria. For example in Figure 8(c), clusters 1 and 20 are selected from the enumerated set such that the number of clusters required to cover the data-path is minimized. Two FPGA logic blocks will be required to realize the template in Figure 8(a).

The cluster generation algorithm follows a set of rules to establish mapping capability within a single logic block. In general, operators can almost be combined with each other so long as the combination of the operators does not overload the carry propagation circuitry, the input or output capability of the logic block. These rules have been verified using a standard synthesis process on VHDL implementations of the basic clusters. Further detail on the rule sets is given in [LAM 2009].

The experiments described in this paper make use of a 4-input logic block that is seen in many FPGA architectures such as the Xilinx Virtex-2 and Virtex-4 families [XILINX 2007, XILINX 2008]. However, the use of the variable K to indicate the number of direct inputs to the logic block makes the approach applicable in situations where the number of inputs is different. There is an assumption in our work that the logic blocks will be accompanied by a fast carry propagation structure that is standard on all modern FPGAs.

4.1 Critical Path Estimation

Figure 9 shows the critical path delay estimation of a template that has been partitioned to clusters. Note that the critical path is the path with the maximum number of clusters from the input buffer to output buffer. The timing characteristics for the various logic and interconnect is also shown in Figure 9 with their default values, which are obtained empirically or from data sheets of the target device. In the example, the target device is Xilinx Virtex-4 xc4vlx40-10ff1148. In general, the critical path delay estimation model of a template is shown in Eq. (3), where m is the number of basic clusters in the critical path of the template ($m = 3$ in the example).

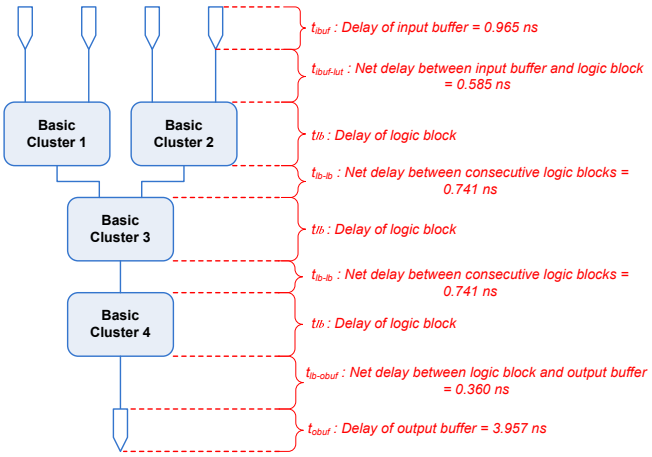


Figure 9: Example of critical path delay estimation of template

$$T_{template} = t_{ibuf} + t_{buf-lut} + \sum_{i}^{m} t_{lb}^{i} + \sum_{i}^{m-1} t_{lb-lb}^{i} + t_{lb-obuf} + t_{obuf} \tag{3}$$

The delay of a logic block i (i.e. t_{lb}^{i}) consists of two components, i.e. the delay of the LUT (i.e. t_{lut}^{i}) and the delay of the carry chain. The estimation model of t_{lb}^{i} is shown in Eq. (4), where x_i indicates the existence of an *ADD* operation in cluster i:

$$t_{lb}^{i} = t_{lut}^{i} + x_i \cdot \left(t_{muxcy\_so} + n_{cc}^{i} \cdot t_{muxcy\_cio} + t_{xorcy\_cio} \right)$$
$$where \quad x_i = \begin{cases} 1 & if \ ADD \ exists \ in \ cluster \ i, \\ 0 & else \end{cases} \tag{4}$$

The parameters t_{muxcy_so}, t_{muxcy_cio} and t_{xorcy_cio} correspond to the delay of the multiplexer and XOR components in the carry-chain structure with the following values (obtained from the data sheets): 0.366ns, 0.044ns and 0.360ns respectively. n_{cc}^{i} is the number of multiplexers in the carry-chain path (excluding the first and last one) and is assumed to be 30. The estimation model in Eq. (4) can be easily verified from Figure 10, which shows the delay path of an addition operation.

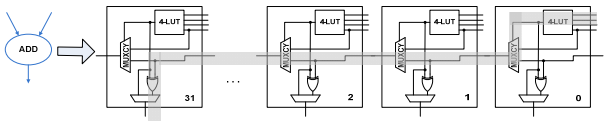


Figure 10: Delay path of an addition operation

The estimation model in Eq. (3) and Eq. (4) must be extended to take into consideration consecutive basic clusters with addition operations in the critical path. Figure 11 illustrates the delay path of two consecutive addition operations (where each adder is in a separate cluster). It can be observed that the carry chain delay of the second addition operation partially overlaps with the first and hence, should not be included in the estimation model.

In order to take into account the partial overlapping carry chain delay of the second addition operation, the estimation model in Eq. (3) is modified as shown in Eq. (5), where the modified logic block delay consists of two components (i.e. t^i_{lb-1} and t^i_{lb-2}) as shown in Eq. (6) and Eq. (7). *a* denotes the number of adder groups, where each adder group consists of either one disjoint cluster with *ADD* operation or several consecutive clusters with *ADD* operations.

$$T_{template} = t_{ibuf} + t_{buf-lut} + \sum_{i}^{a} t^i_{lb-1} + \sum_{i}^{m-a} t^i_{lb-2} + \sum_{i}^{m-1} t^i_{lb-lb} + t_{lb-obuf} + t_{obuf} \quad (5)$$

$$t^i_{lb-1} = t^i_{lut} + x_i \cdot \left(t_{muxcy\_so} + n^i_{cc} \cdot t_{muxcy\_cio} + t_{xorcy\_cio} \right) \quad (6)$$

$$t^i_{lb-2} = t^i_{lut} + x_i \cdot \left(t_{muxcy\_so} + t_{xorcy\_cio} \right) \quad (7)$$

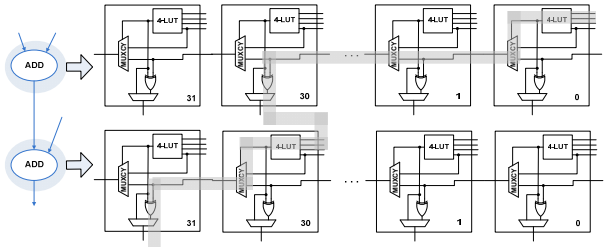


Figure 11: Delay path of two consecutive additions

The estimation model for the logic block delay can be further extended to take into account the effective shift-right constant offset that occurs between two consecutive additions. In cases where the cluster consists of more than one shift operation preceding the *ADD* operation, the effective shift right offset must be computed. This can be calculated using existing bit-width analysis approaches such as that proposed in [MAHLKE].

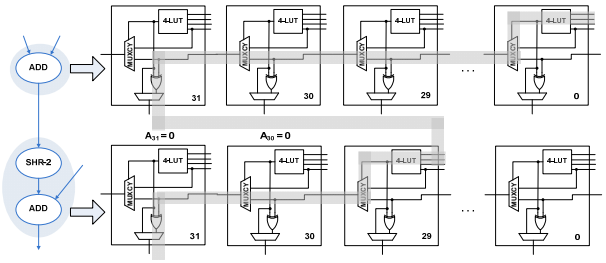


Figure 12: Delay path of two consecutive additions with a right shift operation

Figure 12 illustrates this scenario, whereby the first cluster consists of an *ADD* operation, while the second cluster consists of a shift-right-constant operation (by a factor of two) followed by an *ADD* operation. It can be observed that due to the shift operation, the full result of the first addition must be obtained and shifted to the right by two bits, before being fed to the second basic cluster. This incurs additional carry chain delay in the second cluster. In order to take into account the addition delay incurred by shift-right-by-constant operations between two consecutive *ADD* operations, the estimation model for the logic block delay ($t_{lb\text{-}2}$) in Eq. (7) is extended as shown in Eq. (8) to incorporate a new parameter (i.e. n^i_{shr}), which denotes the effective shift right constant offset of the basic cluster *i*.

$$t^i_{lb-2} = t^i_{lut} + x_i \cdot \left(t_{muxcy\_so} + \left(n^i_{shr} - 1\right) \cdot t_{muxcy\_cio} + t_{xorcy\_cio}\right) \qquad (8)$$

4.2 Area Estimation

The estimated number of logic blocks is equivalent to the number of selected clusters that are obtained after the cluster selection step. This estimation is undertaken with the assumption that the eventual hardware operators must cater to operands with the maximum bit-width. However, this may lead to high inaccuracies as commercial FPGA tools are capable of inferring the appropriate data-path widths of hardware operators, which processes operands that occupy a limited segment of the maximum bit-width.

In this sub-section, we describe a more accurate method for estimating the area utilization of the custom instructions on FPGAs based on the cluster generation process. In particular, the proposed method is capable of inferring the appropriate data-path widths of clusters, by analyzing the logic shift offsets in the clusters. We consider two cases based on the shift-left-by-constant and shift-right-by-constant operations.

In the first case, we consider shift-left-by-constant operations that occur before or after a logical or arithmetic operation in a basic cluster. Figure 13 shows two examples of

this situation, where a shift left operation (by a constant factor of two) occurs after and before an *ADD* operation. It can be observed from Figure 13 that the delay path consist of only 30 logic blocks (we assume the maximum bit-width is 32) in both examples. In the first example, since the two Most Significant Bits (MSBs) of the addition result will be shifted out, the FPGA synthesis tool will recognize the redundancy in computing the addition for the two operand MSBs. Hence, only the last 30 bits of the operands will be computed. In the second example, since the operands are shifted left by two, only 30 logic blocks will be required to compute the effective bit-width of the operands. Therefore, the number of required logic blocks of a cluster is equivalent to the maximum bit-width (i.e. 32) minus the effective left shift offset of the cluster.

In the second case, we consider shift-right-by-constant operations that occur before a logical or arithmetic operation in a cluster. Figure 14 shows an example of this case where a shift right operation (by two) occurs before an *ADD* operation in a cluster. Similar to the discussion for the first case, since the operands are shifted right by two, only 30 logic blocks will be required to compute the effective bit-width of the operands. Therefore, the number of required logic blocks of a cluster is equivalent to the maximum bit-width (i.e. 32) minus the effective right shift offset of the cluster, when the right shift operation occurs before the logical or arithmetic operations in the cluster.

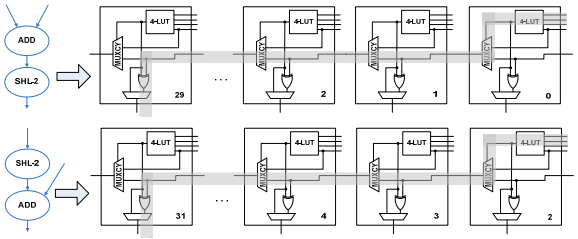


Figure 13: Delay path of cluster with shift-left-by-constant and addition

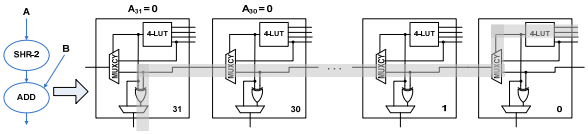


Figure 14: Cluster with shift-right-by-constant preceding an addition

The area estimation model of a template (in terms of number of basic logic elements) is shown in Eq. (9), where k is the number of basic clusters. n_{sh}^{i} denotes the effective shift offset of basic cluster i for the two cases described above. Note that the effective shift offsets must be calculated when the basic cluster has multiple shift-by-constant

operations. This can be computed using existing bit-width analysis methods (e.g. [MAHLKE]).

$$A_{template} = \sum_{i}^{k} \left(32 - n_{sh}^{i}\right) \qquad (9)$$

4.3 Delay-Area Estimation Example

The custom instruction in Figure 8 is used as an example to demonstrate the viability of the proposed delay-area estimation models. Figure 15 shows the critical path estimation of the custom instruction that has been partitioned into two clusters. Figure 16 shows the FPGA synthesis timing report of the custom instruction. The synthesis report clearly shows that the delay path of the custom instruction implementation on FPGA spans across two logic blocks, i.e. a 2-input LUT for the first logic block and a 4-input LUT for the second logic block. In addition, the delay of the second logic block comprises of the carry-chain delay. This is consistent with the results of the proposed clustering generation process.

Based on Eq. (5), (6) and (8), the critical delay is estimated as 9.044ns (less than 1% estimation error). For area estimation, the effective shift offsets is 4 (we assume a constant offset of 2 for both the shift operations). Hence, the number of basic logic elements is estimated correctly as 60 ((32-4) + 32).

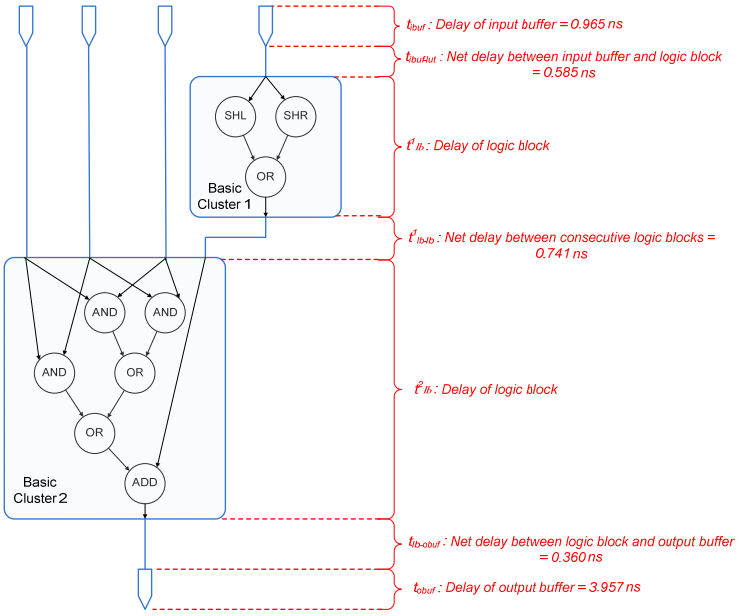


Figure 15: Critical path delay estimation of example in Figure 8

```
                          Gate    Net
Cell:in->out      fanout  Delay   Delay  Logical Name (Net Name)
----------------------------------------  ------------
 IBUF:I->O             1   0.965   0.585  pIn4_0_IBUF (pIn4_0_IBUF)
 LUT2:I0->O            1   0.195   0.741  v6_or0000<2>1 (v6_or0000<2>)
 LUT4:I0->O            1   0.195   0.000  Madd_pOut_lut<2> (Madd_pOut_lut<2>)
 MUXCY:S->O            1   0.366   0.000  Madd_pOut_cy<2> (Madd_pOut_cy<2>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<3> (Madd_pOut_cy<3>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<4> (Madd_pOut_cy<4>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<5> (Madd_pOut_cy<5>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<6> (Madd_pOut_cy<6>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<7> (Madd_pOut_cy<7>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<8> (Madd_pOut_cy<8>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<9> (Madd_pOut_cy<9>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<10> (Madd_pOut_cy<10>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<11> (Madd_pOut_cy<11>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<12> (Madd_pOut_cy<12>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<13> (Madd_pOut_cy<13>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<14> (Madd_pOut_cy<14>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<15> (Madd_pOut_cy<15>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<16> (Madd_pOut_cy<16>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<17> (Madd_pOut_cy<17>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<18> (Madd_pOut_cy<18>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<19> (Madd_pOut_cy<19>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<20> (Madd_pOut_cy<20>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<21> (Madd_pOut_cy<21>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<22> (Madd_pOut_cy<22>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<23> (Madd_pOut_cy<23>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<24> (Madd_pOut_cy<24>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<25> (Madd_pOut_cy<25>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<26> (Madd_pOut_cy<26>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<27> (Madd_pOut_cy<27>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<28> (Madd_pOut_cy<28>)
 MUXCY:CI->O           1   0.044   0.000  Madd_pOut_cy<29> (Madd_pOut_cy<29>)
 MUXCY:CI->O           0   0.044   0.000  Madd_pOut_cy<30> (Madd_pOut_cy<30>)
 XORCY:CI->O           1   0.360   0.360  Madd_pOut_xor<31> (pOut_31_OBUF)
 OBUF:I->O                 3.957          pOut_31_OBUF (pOut<31>)
----------------------------------------
Total                      8.970ns (7.284ns logic, 1.686ns route)
                           (81.2% logic, 18.8% route)
```

Basic Cluster 1

Basic Cluster 2

Figure 16: FPGA synthesis timing report

5. EXPERIMENTAL RESULTS

In this section, we will first introduce the target reconfigurable processor platform. Experimental results is presented to show the accuracy of the proposed area-time estimation technique which relies on the cluster generation process and delay-area estimation models discussed in Section 4.1 and 4.2. Next, we demonstrate how the cluster generation can be employed for multi-cycle custom instructions. We then reexamine the performance of the custom instruction selection approaches by using the proposed cluster generation technique on the same set of application benchmarks that were employed in Section 3.6. Finally, we perform experiments that take into account the area constraint of the FPGA to further reinforce the need for a fast and accurate FPGA estimation engine in design flows for custom instruction selection.

5.1 Target Reconfigurable Processor Model

The target reconfigurable model, which is shown in Figure 17, is a four-wide Very Long Instruction Word (VLIW) architecture that has been extended with an RFU for implementing multi-cycle custom instructions. The target model provides coupling logic between the integer unit and RFU. Hence, the RFU only facilitates custom instruction

implementations of integer type operations. We assumed that the number of available input/output ports in the RFU is 5 and 2 respectively.

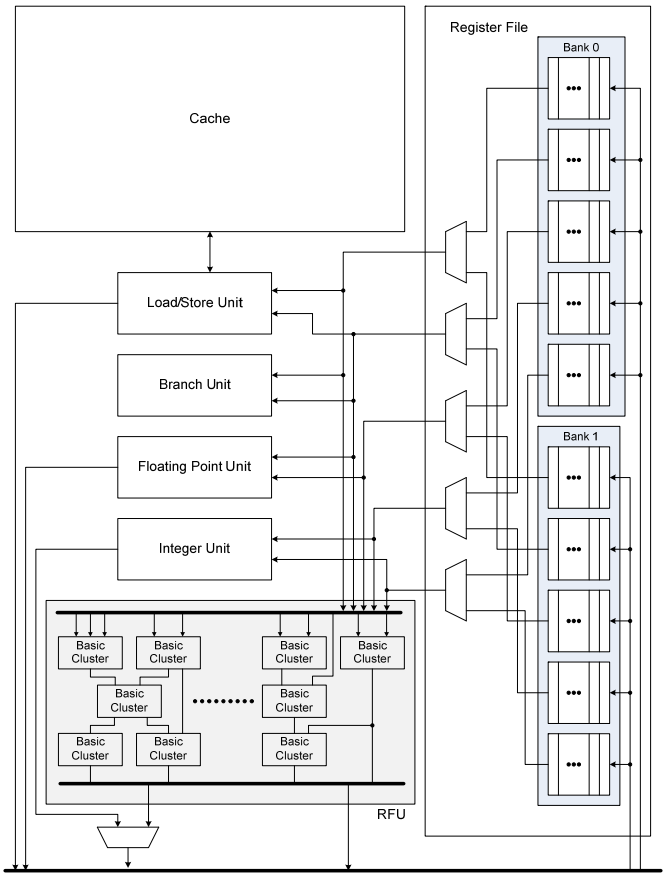


Figure 17: Target reconfigurable processor model

We assume that a multi-ported register file [RIXNER] is used to enable simultaneous accesses of the custom instruction to the register file. The number of read/write ports of the register file corresponds to the operand number of the custom instructions and hence, no communication latency will be introduced [CONG 2006]. The work in [SAGHIR] has demonstrated that low-latency multi-ported register file can be implemented using block RAMs that are available in high-density FPGAs.

While the overhead of data transfer in tightly-coupled scheme does not pose a major bottleneck to the system performance and is often ignored during performance evaluation [BARAT], the delay of the multiplexer logic that is required to select the desired custom instruction result can significantly affect the system's performance. As discussed in [LAM 2006], this logic increases with the number of custom instructions. Hence, the

outputs of the custom instructions are multiplexed to meet the two output port constraint of the RFU. In particular, single output custom instructions are multiplexed to the primary output port of the RFU, and dual-output custom instructions are multiplexed to the primary and secondary output ports of the RFU.

Table 2 shows the critical path delay of the custom instructions and the delay contributed by the output multiplexer in the critical path for all the applications considered. The results are based on custom instructions selected using the LFF method which produces large number of instructions (see Figure 6). It can be observed that the delay contribution of the multiplexer is very small for all the applications (average difference of less than 10%). Hence, the delay of the multiplexers is not likely to affect the achievable clock rate of the custom instructions.

Table II. Critical path and multiplexer delay

Application	Critical path (ns)	Multiplexer delay (ns)	Difference (%)
Adpcm Dec	10.64	1.46	13.72
Adpcm Enc	10.20	0.97	9.46
Aes	19.69	1.76	8.91
Basicmath Large	6.79	0.56	8.18
Bitcount	16.47	0.58	3.50
Blowfish Dec	13.16	1.51	11.50
Blowfish Enc	13.16	1.51	11.50
Cjpeg	23.90	2.16	9.02
CRC32	8.19	0.36	4.40
Dijkstra Large	8.63	0.57	6.60
FFT	10.26	1.51	14.75
Patricia	11.61	3.74	32.24
Pegwit	16.38	0.53	3.22
Rijndael Dec	13.11	1.51	11.54
Rijndael Enc	23.04	0.57	2.47
Sha	11.47	0.97	8.42

5.2 FPGA Estimation Results

In this sub-section, we present experimental results to show the reliability of the proposed cluster generation process for FPGA estimation. Column 3/4 of Table 3 shows the average/maximum percentage error of the proposed critical path delay estimation technique with respect to the synthesis results of equivalent hand-crafted designs for 150 custom instructions from sixteen benchmark applications. Only custom instructions that do not contain complex operations (e.g. multiplication, division, shift-by non-constant) are considered. These complex operations can be implemented using FPGA embedded IP cores. Xilinx ISE Version 11.2 is used as the synthesis engine to create implementations of the hand-crafted VHDL designs. A Xilinx Virtex-4 xc4vlx40-10ff1148, which incorporates logic blocks with 4-input LUTs is used as the experimental target.

The proposed FPGA estimation technique executes cluster enumeration and cluster selection algorithms to generate the internal structure for each custom instruction using the estimation models shown in Eq. (5), Eq. (6) and Eq. (8). It can be observed that the proposed technique has an average/maximum percentage delay error of only 2.85% and about 11% respectively for the 150 custom instructions. In addition, except for Basicmath Large, the average percentage delay error for each application is within 4%. The absolute error of critical path estimation is less than 1ns.

Table III. Average/maximum error of critical path delay and area estimation for 150 custom instructions

Application	Number of Custom Instructions	Critical Path Delay (% Error)		Area (% Error)	
		Average	Maximum	Average	Maximum
Adpcm Dec	6	2.89	7.54	0.00	0.00
Adpcm Enc	7	1.45	6.19	0.00	0.00
Aes	27	3.30	11.08	0.54	5.93
Basicmath Large	2	9.63	11.08	0.00	0.00
Bitcount	4	3.13	2.30	0.00	0.00
Blowfish Dec	8	2.20	3.52	0.27	2.13
Blowfish Enc	8	2.20	3.52	0.27	2.13
Cjpeg	29	1.71	11.08	0.44	3.23
CRC32	1	1.00	2.08	0.00	0.00
Dijkstra Large	5	1.45	7.08	1.65	4.92
FFT	5	3.03	7.70	0.00	0.00
Patricia	4	1.27	2.30	0.81	3.23
Pegwit	14	2.29	11.08	0.79	3.33
Rijndael Dec	12	3.56	11.08	0.18	2.13
Rijndael Enc	11	3.44	8.18	0.19	2.13
Sha	7	3.04	11.08	1.23	3.23

Column 5/6 of Table 3 shows the average/maximum percentage area error in each of the benchmark applications. Area errors are calculated by comparing the logic block usage estimate of our proposed area estimation technique with the synthesis device utilization summary of the respective hand-crafted VHDL design. It can be observed that the proposed technique has an average and maximum percentage error of only 0.40% and less than 6% respectively for the 150 custom instructions. The maximum average percentage error of each application is within 2%. In addition, it can be seen that the proposed method can estimate the number of logic blocks without any error for several applications (e.g. Adpcm Dec, Adpcm Enc, Basicmath Large, Bitcount, CRC2 and FFT). These results demonstrate the reliability of the cluster generation process for high-level area estimation of custom instructions on FPGA.

These results are very encouraging as the critical delay and area can be rapidly estimated using just information relating to primitive operations. Design conversion to

VHDL and subsequent synthesis are not required. Cluster enumeration and selection become millisecond timescale operations using this approach which enables the use of this estimation technique in rapid design exploration during instruction set customization.

5.3 Multi-Cycle Custom Instructions

Multi-cycle custom instruction implementation is necessary to maintain a high clock rate especially for custom instructions with long critical path. The cluster generation process can be employed to partition custom instructions into multiple states in an architecture-aware manner to provide for a high degree of estimation accuracy.

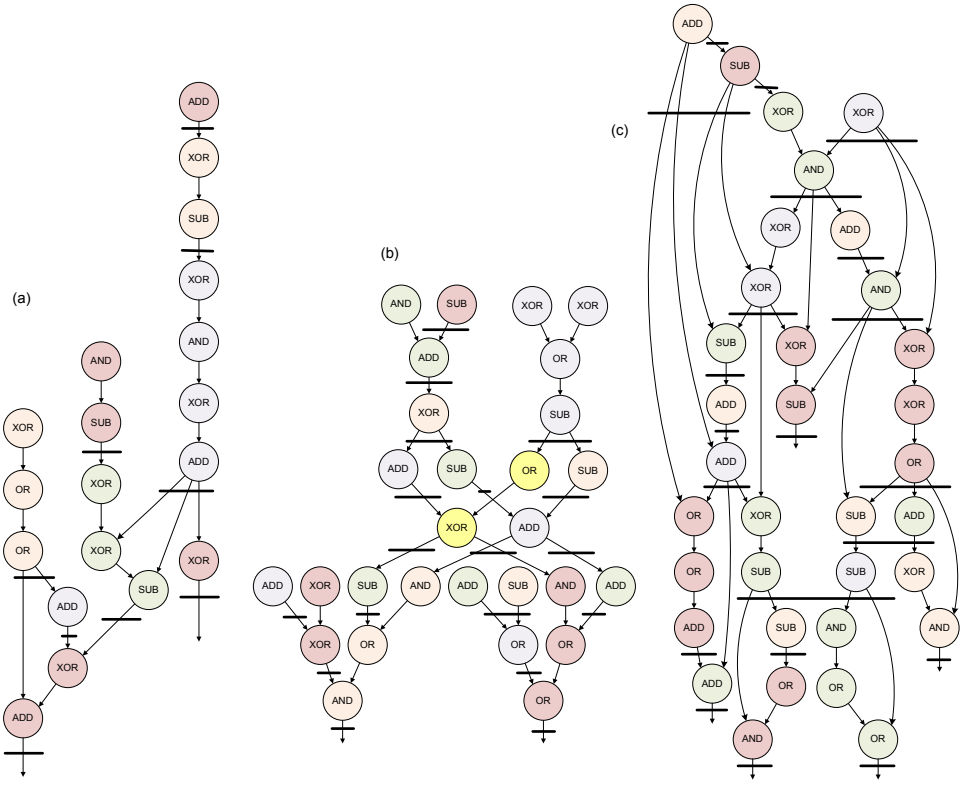


Figure 18: Multi-cycle custom instructions for data flow structures extracted from DFGs of a) JPEG Smooth Downsample, b) Auto Regression Filter, and c) Elliptic Wave Filter

In this section, we evaluate the viability of the cluster generation technique for multi-cycle custom instructions using three large data flow structures that have been extracted from [EXPRESS]. These data structures are part of DFGs obtained from real application functions i.e. JPEG Smooth Downsample, Auto Regression Filter and Elliptic Wave Filter. In order to investigate the effects of the number of arithmetic and logical operations on the estimation accuracy, we have randomly generated the operations based

on four arithmetic-to-logical operation ratios for each data flow structure (i.e. 1:1, 1:2, 1:3, 1:4).

Figure 18 shows the results of cluster generation on the three data flow structures with arithmetic-to-logical operation ratio 1:1. In these examples, register states (represented by the bold lines) are inserted after each basic cluster. Connected operations are labeled with the same color if they belong to the same basic cluster. Based on the results of cluster generation, the register states are explicitly specified in the RTL codes. As such, the synthesis tool will produce custom instruction implementations with deterministic latencies. This enables the cluster generation process to accurately estimate the latencies of the multi-cycle custom instructions by determining the critical path with the largest number of state registers.

Table IV shows the area estimation results for the multi-cycle custom instructions with randomly generated arithmetic-to-logical operation ratio (column 3). It can be observed that the average area estimation error is less than 7% for these large custom instruction data-paths. These results demonstrate the viability of the proposed cluster generation technique for generating multi-cycle custom instructions with high degree of estimation accuracy.

Table IV. Estimation results of large multi-cycle custom instructions

Source Function	Number of nodes/edges	Operation Ratio	Latency (Cycles)	Area (Logic blocks)		
				Actual	Estimated	Error (%)
JPEG Smooth Downsample	19/20	1:1	5	313	288	7.99
		1:2	5	313	288	7.99
		1:3	4	286	256	10.49
		1:4	4	254	224	11.81
Auto Regression Filter	28/31	1:1	7	607	576	5.11
		1:2	7	607	576	5.11
		1:3	7	543	512	5.71
		1:4	6	480	480	0.00
Elliptic Wave Filter	34/48	1:1	10	735	704	4.22
		1:2	9	671	640	4.62
		1:3	8	512	544	6.25
		1:4	8	512	544	6.25

5.4 Speedup Evaluation with Hardware Estimation

The experimental results in Section 3.6 reveal that RLFF can rapidly select custom instructions with highest performance (in terms of number of nodes covered). However, the number of nodes covered (or number of operations in the ISA that can be mapped to hardware) does not provide a realistic measure of the effective performance gain that can

be achieved by the custom instructions. This is due to the fact that the hardware latencies of the custom instructions have not been taken into consideration.

We now reexamine the performance of the custom instruction selection approaches by using the proposed cluster generation technique to estimate the custom instruction latencies on FPGAs. The performance of an application with custom instruction extension for application A can be calculated in terms of *SCS* (Software Cycle Savings) as shown in Eq. (10). *SCS(A)* is defined as the number of software clock cycle savings due to the migration of the native instructions of the processor to hardware for application A. In Eq. (10), G_i for i = 1,2, ..., n is a custom instruction, where n is the total number of custom instructions obtained from template selection for application A, $F(G_i)$ is the execution frequency of instruction G_i in application A, $TS(G_i)$ denotes the number of nodes covered in instruction G_i using the template selection methods discussed in the previous chapter, and r is the ratio of the clock frequency of the RFU and the base processor (r is chosen based on the area-optimized configuration of the soft-core processor in [MATTSON]). $T_{template}(G_i)$, which is the estimated critical path delay of G_i is b, which is the multi-cycle latency of G_i.

$$SCS(A) = \sum_{i}^{b} F(G_i) \cdot \left(TS(G_i) - r \cdot T_{template}(G_i)\right) \qquad (10)$$

We compare the speedup obtained from the various template selection heuristics with ILP, where the objective function is formulated to maximize *SCS(A)*. In particular, the ILP objective function in Eq. (1) is modified to include the estimated latency of the instances as shown in Eq. (11) under the non-overlapping constraint in Eq. (2).

$$\max : \sum_{i=1}^{n} \sum_{j=1}^{n_i} \left(x_{i,j} \times F_{i,j} \times (size(I_{i,j}) - T_{template}(I_{i,j}))\right) \quad (11)$$

Figure 19 shows the percentage speedup of the estimated performance calculated using *SCS* with respect to the performance of the baseline processor that is obtained from Trimaran's simulator. The result for the ILP approach is obtained after 15 minutes of runtime. Contrary to the results in Figure 7, where the RLFF approach performs favorably across all the applications considered, the results in Figure 19 which incorporate FPGA estimation, shows that RLFF has the worst performance in the following applications: AES, Blowfish Dec, Blowfish Enc, Rijndael Dec, Rijndael Enc and Sha. In addition, the ILP approach is unable to produce better results than the LFF method in several applications in the given time. These results clearly demonstrate the importance of an accurate and rapid estimation technique for template selection.

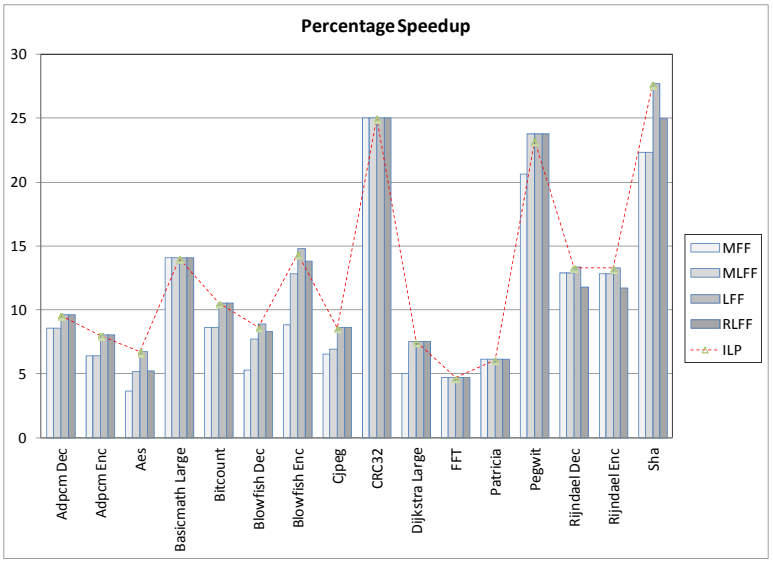


Figure 19: Percentage speedup

5.5 Speedup Evaluation with Area Constraint

In this section, we evaluate the performance of the MFF, MLFF, LFF and RLFF approaches by taking into consideration the FPGA area constraint. This is achieved by employing a greedy algorithm that solves the well-known 0-1 knapsack problem [PISINGER] to select a set of profitable custom instructions that meets the area constraints for each application. The greedy algorithm aims to choose a set of custom instructions (from MFF, MLFF, LFF and RLFF) to maximize the *SCS* while ensuring that the total area utilization of the custom instructions does not exceed the resource constraint. We have used the proposed area estimation technique to predict the total area utilization of the custom instructions. We have also extended the ILP formulation in Eq. (11) and Eq. (2) to take into consideration the estimated area of template instances and the area constraint R as shown in Eq. (12). This ILP formulation has been presented in [Yu].

$$\sum_{i=1}^{n}(S_i \times A_{template}(T_i)) \le R$$

$$where\ S_i = \begin{cases} 1 & if \sum_{j=1}^{n_i} x_{i,j} > 0 \\ 0 & otherwise \end{cases} \qquad (12)$$

Figure 20 shows the experimental results, where the *x*-axes denote the range of area constraints (in terms of number of basic clusters) for each application. We have not shown the results for applications where the template selection approaches do not exhibit any variations. Results for Blowfish/Rijndael Enc are the same as that for Blowfish/Rijndael Dec, and hence are also not shown.

It can be observed that in general, the performance increases for all the template selection approaches when the area constraint is relaxed. This is reasonable as more custom instructions can be implemented when the available FPGA resources increase. We can also observe that the MFF and MLFF approaches lead to better performance in many cases when the area constraint is tight. This is attributed to the fact that custom instructions selected using the MFF and MLFF approach are usually small (Figure 7) and hence they can better utilize a restricted FPGA space. As the area constraint is relaxed, template selection approaches e.g. LFF, RLFF and ILP, which produces large custom instructions, become more favorable.

It is noteworthy that the point at which one template selection approach becomes more favorable than another cannot be effectively determined without the knowledge of the hardware area utilization of the custom instructions. In addition, the RLFF method (which has been shown in Section 3.6 to be able to select custom instructions with largest number of operations) consistently underperforms under varying area constraints compared to most of the other template selection approaches for AES, Rijndael Dec and Rijndael Enc. The ILP approach, which typically takes hours to complete for each application (restricted to 15 minutes runtime for each design point) consistently underperforms particularly when the area constraint is tight when compared to the other template selection approaches that can execute in the order of seconds.

These experiments clearly show the need for an accurate delay-area estimation technique to facilitate fast design space exploration for custom instruction selection.

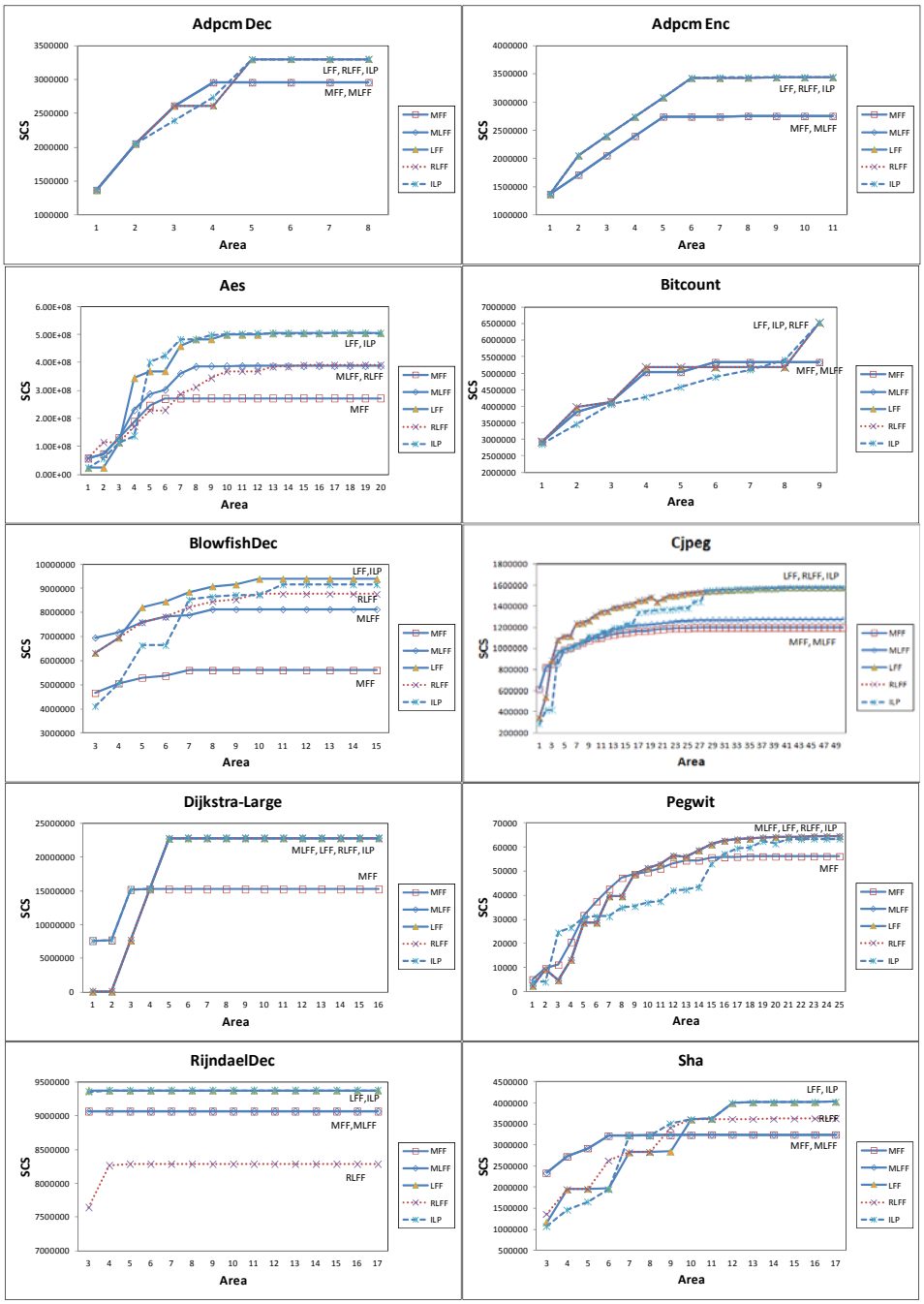


Figure 20: Percentage speedup with area constraint

6. CONCLUSION

We have proposed a novel cluster generation strategy that partitions the custom instructions into a set of clusters such that the clusters can be efficiently mapped onto the LUT and carry-look-ahead structure of the FPGA logic blocks. We presented delay

estimation models, which take into account the anomalies incurred by consecutive addition operations and shift right operations to accurately estimate the critical path of the basic clusters. Experimental results show that the average estimated critical paths of 150 custom instructions from sixteen applications using the proposed method are within 3% of those obtained using hardware synthesis. The short execution time of this estimation process in comparison to a full hardware synthesis makes is particularly attractive. The inclusion of strategies to account for logic shift offsets in basic clusters permits accurate estimation of the area utilization of the custom instructions on FPGA.

In order to demonstrate the benefits of the proposed FPGA estimation technique for template selection, we devised an efficient strategy for the rapid selection of high performance custom instructions for reconfigurable processors based on the graph covering approach. The RLFF selection strategy is based on our findings that over 77% of the template instances are high frequency basic templates that are incorporated into larger templates. The RLFF approach has been shown to benefit from cases in which overlapping templates could be re-introduced to maximize performance. Comparisons with widely used approximate strategies such as MFF, MLFF and ILP show that the RLFF select custom instructions that covers the most number of nodes for the applications considered.

When the performance of the template selection approaches are re-examined using the proposed cluster generation technique, the new experimental results reveal that RLFF performs less favorably than the other approaches for certain applications. This is due to the fact that the templates selected using RLFF incur hardware latencies that are under-compensated by the number of clock cycle savings. This clearly demonstrates the necessity and effectiveness of the proposed delay estimation technique for template selection. Finally, the proposed estimation technique can be incorporated in the template selection process in order to select FPGA efficient custom instructions.

REFERENCES

ALTERA. NIOS II Processors. Online: http://www.altera.com/products/ip/processors/nios2/ni2-index.html.

ATASU, K., ÖZTURAN, C., DÜNDAR, G., MENCER, O., AND LUK, W. 2008. CHIPS: Custom Hardware Instruction Processor Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 27, No. 3*, 528-541.

BARAT, F., LAUWEREINS, R., AND DECONINCK, G., 2002. Reconfigurable Instruction Set Processors from a Hardware/Software Perspective. IEEE *Transactions on Software Engineering, Vol. 28, No. 9*, 847–862.

BILAVARN, S., GOGNIAT, G., PHILIPPE, J.-L., AND BOSSUET, L. 2006. 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*, 1950–1968.

BRANDOLESE, C., FORNACIARI, W., AND SALICE, F. 2004. An Area Estimation Methodology for FPGA based Designs at SystemC Level. *Proceedings of Design Automation Conference*, 129–132.

BJURÉUS, P., MILLBERG, M., AND JANTSCH, A. 2002. FPGA Resource and Timing Estimation from Matlab Execution Traces. *Proceedings of the International Symposium on Hardware Software Codesign*, 31–36.

BONZINI, P., AND POZZI, L. 2008. Recurrence-Aware Instruction Set Selection for Extensible Embedded Processors. *IEEE Transactions on Very Large Scale Integration Systems, Vol. 16, No. 10*, 1259-1267.

CHEN, D., AND CONG, J. 2004. DAOmap: A Depth-Optimal Area Optimization Mapping Algorithm for FPGA Designs. *IEEE International Conference on Computer-Aided Design*, 752–759.

CLARK, N.T., ZHONG, H., AND MAHLKE, S.A. 2003. Processor Acceleration Through Automated Instruction Set Customization. *Proceedings of the 36th IEEE/ACM International Symposium on Microarchitecture*.

CLARK, N.T., ZHONG, H., AND MAHLKE, S.A. 2005. Automated Custom Instruction Generation for Domain-Specific Processor Acceleration. *IEEE Transactions on Computers, Vol. 54, No. 10*, 1258-1270.

CONG, J., FAN, Y., HAN, G., AND ZHANG, Z. 2004. Application-Specific Instruction Generation for Configurable Processor Architectures. *Proceedings of the ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, 183-189.

CONG, J., HAN, G., AND ZHANG, Z. 2006. Architecture and Compiler Optimizations for Data Bandwidth Improvement in Configurable Processors, *IEEE Transactions on Very Large Scale Systems*, Vol. 14, No. 9, pp. 986-997.

EEMBC. The Embedded Microprocessor Benchmark Consortium. Online: http://www.eembc.org/home.php.

GALUZZI, C., PANAINTE, E.M., YANKOVA, Y., BERTELS, K., AND VASSILIADIS, S. 2006. Automatic Selection of Application-Specific Instruction-Set Extensions. *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis*, 160-165

EXPRESS. ExPRESS Benchmarks. Online: http://express.ece.ucsb.edu/benchmark/

GUO, Y., SMIT, G.J.M., BROERSMA, H., AND HEYSTERS, P.M. 2003. A Graph Covering Algorithm for a Coarse Grain Reconfigurable System. *Proceedings of the ACM SIGPLAN Conference on Language, Compiler, and Tool for Embedded Systems*, 199-208.

GUTHAUS, M.R., RINGENBERG, J.S., ERNST, D., AUSTIN, T.M., MUDGE, T., AND BROWN, R.B. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. *IEEE International Workshop on Workload Characterization*, 3-14,

HALLDÓRSSON, M., AND RADHAKRISHNA, J. 1994. Greed is Good: Approximating Independent Sets in Sparse and Bounded-Degree Graphs. *Proceedings of the Annual ACM Symposium on Theory of Computing*, 439-448.

JOEY, Y.L., CHEN, D., AND CONG, J. 2006. Optimal Simultaneous Mapping and Clustering for FPGA Delay Optimization. *Proceedings of Design Automation Conference*, 472-477.

KASTNER, R. KAPLAN, A., MEMIK, S.O., AND BOZORGZADEH, E. 2002. Instruction Generation for Hybrid Reconfigurable Systems", *ACM Transactions on Design Automation of Embedded Systems, Vol. 7, No. 4*, 605-627.

KULKARNI, D., NAJJAR, W.A., RINKER, R., AND KURDAHI, F.J. 2006. Compile-Time Area Estimation for LUT-based FPGAs. *ACM Transactions on Design Automation of Electronic Systems*, *Vol. 11, No. 1*, 104–122.

LAM, S.K., KRISHNAN, B.N., AND SRIKANTHAN T., 2006a. Efficient Management of Custom Instructions for Run-Time Reconfigurable Instruction Set Processors. *IEEE International Conference on Field Programmable Technology*, 261-264.

LAM, S.K., SHOAIB, M., AND SRIKANTHAN T. 2006B. Modeling Arbitrator Delay-rea Dependencies in Customizable Instruction Set Processors. *IEEE International Workshop on Electronic Design, Test and Applications*.

LAM, S.K., and SRIKANTHAN, T. 2009. Rapid Design of Area-Efficient Custom Instructions for Reconfigurable Embedded Processing. *Journal of Systems Architecture*, Vol. 55, No. 1, 1-14.

LAM, S.K., SRIKANTHAN, T. AND CLARKE C.T. 2011. Architecture-Aware Technique for Mapping Area-Time Efficient Custom Instructions onto FPGAs. *IEEE Transactions on Computer*s, *Vol. 60, No. 5,* 680-692

LEE, C., POTKONJAK, M., AND MANGIONE-SMITH, W.H. 1997. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. *Proceedings of the 13th Annual IEEE/ACM International Symposium on Microarchitecture,* 330-335.

LEE, J.-E., CHOI, K., and DUTT, N. 2002. Efficient Instruction Encoding for Automatic Instruction Set Design of Configurable ASIPs. *IEEE/ACM International Conference on Computer-Aided Design*, 649-654

LI T., WU J., LAM S.K. AND SRIKANTHAN T. 2010. Selecting Profitable Custom Instructions for Reconfigurable Processors, *Journal of Systems Architecture, Vol. 56, No. 8*, 340-351.

MAHLKE, S., RAVINDRAN, R., SCHLANSKER, M., SCHREIBER, R. AND SHERWOOD, T. 2001. Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20, No. 11*, 1355-1371.

MATTSON, D., AND CHRISTENSSON, M. 2004. Evaluation of Synthesizable CPU Cores, M.S. thesis, Chalmers University of Technology, Gothenburg, Sweden.

NAYAK, A., HALDAR, M., CHOUDHARY, A., AND BANERJEE, P. 2002. Accurate Area and Delay Estimators for FPGAs", *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 862–869.

PISINGER, D. 1994. A Minimal Algorithm for the Multiple Choice Knapsack Problem. Technical Report 94-25, DIKU, University of Copenhagen, Denmark.
POZZI, L., ATASU, K., AND IENNE, P. 2006. 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*, 1209-1229.
RIXNER, S., DALLY, W.J., KHAILANY, B., MATTSON, P., KAPASI, U.J., AND OWENS, J.D. 2000. Register Organization for Media Processing. *Sixth International Symposium on High-Performance Computer Architecture*, pp. 375-386.
SAGHIR, M.A.R., AND NAOUS, R. 2007. A Configurable Multi-Ported Register File Architecture for Soft Processor Cores. *International Workshop on Applied Reconfigurable Computing*, pp. 14-25.
STRETCH. S6000 Family Software Configurable Processors. Online: http://www.stretchinc.com/products/s6000.php.
SUN, F., RAVI, S., RAGHUNATHAN, A., AND JHA, N.K. 2004. Custom-Instruction Synthesis for Extensible-Processor Platforms. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 23, No. 2*, 216-228.
SUN, F., RAVI, S., RAGHUNATHAN, A., AND JHA, N.K. 2007. A Synthesis Methodology for Hybrid Custom Instruction and Coprocessor Generation for Extensible Processors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 11, 2035-2045.
TRIMARAN. An Infrastructure for Research in Instruction-Level Parallelism, Online: http://www.trimaran.org.
XILINX. 2007. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. DS083 (Version 4.7).
XILINX. 2008. Virtex-4 FPGA User Guide. User Guide UG070 (Version 2.6).
XILINX. 2012. 7 Series DSP48E1 Slice User Guide. User Guide UG479 (Version 1.3).
YAZDANBAKHSH, A., SALEHI, M.E., SAFARI, S., AND FAKHRAIE, S.M., 2010. Locality Considerations In Exploring Custom Instruction Selection Algorithms. 2nd Asia Symposium on Quality Electronic Design. 157-162.
YU, P., AND MITRA, T. 2004. *Characterizing Embedded Applications for Instruction-Set Extensible Processors. Proceedings of the 41st IEEE/ACM on Design Automation Conference*, 723-728

Received October 2011.
Revised 27 September 2012