INSTRUCTION SET CUSTOMIZATION FOR AREA-CONSTRAINED FPGA DESIGNS

Alok Prakash\(^1\), Siew-Kei Lam\(^1\), Christopher T. Clarke\(^2\) and Thambipillai Srikanthan\(^1\) (SMIEEE)

\(^1\)School of Computer Engineering
NTU, Singapore.
\{alok0001, assklam, astsrikan\}@ntu.edu.sg
\(^2\)Department of Electronic and Electrical Engineering, University of Bath, United Kingdom.
eesctc@bath.ac.uk

ABSTRACT

Custom instructions are commonly used to meet the strict design constraints in high performance systems. This paper proposes a FPGA-aware custom instruction enumeration and selection technique for area-constrained designs that maximizes the logic utilization of the available FPGA space. Experimental results show that the proposed technique achieves a performance gain (execution time) of up to 3 orders of magnitude while leading to an average runtime reduction of over 31\% and 50\% in the enumeration and selection phases respectively.

I. INTRODUCTION

Designers are being constantly pushed to increase flexibility, features and functionalities in embedded systems, while meeting the tight time-to-market constraints. This has led to the increasing popularity of FPGA-based systems. Existing FPGA soft core processors such as NIOS II from Altera [1], Microblaze from Xilinx [2] etc. have a high degree of configurability and can be tailored to meet application-specific requirements through instruction set customization. Instruction set customization aims to add custom instructions to the default instruction set architecture to achieve faster execution and lower power consumption.

To date, commercial FPGA design tools still lack a consolidated framework to identify suitable custom instructions for a given application. There are typically two major steps in instruction set customization namely, custom instruction identification and custom instruction selection. A common approach in custom instruction identification is enumeration, which identifies all the legal custom instruction instances (patterns) within an application. Custom instruction selection then typically selects a set of non-overlapping patterns for final implementation based on certain constraints such as area and performance.

In this paper, we propose a strategy to enumerate patterns from an application in an FPGA aware manner. Existing work in Pattern Enumeration takes into account the macro-architectural constraints of the FPGA such as the Input/Output constraints of the FPGA logic blocks etc. While these constraints are mandatory, we incorporate additional constraints based on the architecture of the FPGA logic blocks. In particular, our enumeration algorithm identifies only patterns that can be fully mapped onto a single logic block of the target FPGA architecture. Such patterns have been termed as 'Clusters' in this work. The proposed pattern selection algorithm aims to select clusters that comprises of the largest set of instructions which can be fully mapped on a single logic block of the targeted FPGA architecture. This approach ensures that we achieve maximum performance using minimum FPGA area. Therefore, the proposed method leads to the selection of custom instructions with notable performance gain when compared to those obtained using conventional approaches for FPGA designs with tight area constraints. We will show that the proposed strategy can also lead to significant runtime reduction in the pattern enumeration and selection phases.

To the best of our knowledge, our work in [20] is the first to take into account the implementation constraints of the FPGA architecture during Pattern Enumeration. This approach enabled us to make more informed pattern enumeration, which has resulted in higher performance from the selected Custom Instructions while using much lesser FPGA area. This paper extends our previous work in [20] and provides detailed algorithms for the proposed method. The method proposed in this paper also overcomes the limitations of the work in [20], which ignored certain practical constraints while calculating the performance gain. This has led to more realistic results which demonstrate the benefits of our approach over the conventional methods.

Section II reviews the related work in custom instruction identification and selection. The proposed pattern enumeration strategy is discussed in Section III, and Section IV describes the custom instruction selection method that we have adopted. Experimental results are shown and discussed in Section V. We conclude the paper in Section VI.

II. RELATED WORK

Atasu et al. in [3] described a branch and bound algorithm that enumerates all the legal patterns from an application Data-Flow Graph (DFG), while pruning off the ones that violate the fundamental micro-architectural constraints. The computational complexity of their method has been shown to increase rapidly with the size of the DFG. Jason Cong et al. described a comprehensive set of algorithms for custom instruction enumeration, selection as well as mapping in [4]. The extended instruction set identified by their algorithms enables them to achieve a maximum of 3.73X performance gain compared to the basic instruction set.

The method presented by Atasu et al. in [5] enumerated only maximal convex patterns and achieved...
a performance increment of an order of magnitude, while having an extremely fast tool runtime. Clark et. al. presented a complete design framework for automatic identification of pattern instances within a DFG and a compiler framework to take advantage of such custom units [6]-[7]. Each pattern instance was annotated with the latency and area information obtained from a hardware library. In the selection stage, a greedy heuristic was used to select instances with largest speedup/area ratio. The work in [8] used a similar approach, where the area and latency of every pattern candidate was estimated by synthesizing them on a 0.13µm CMOS process.

Chen et. al. presented a novel algorithm for rapid identification of custom instructions for extensible processors [9]. Their method performed exceptionally well in large DFGs. Li et. al. [10] proposed further enhancements in Chen's algorithm and achieved up to 50% faster tool runtime for enumerating patterns with single-output constraint. The work in [11] presented a custom instruction generation design flow that incorporates a novel way for estimating the area of custom instructions in the final hardware by clustering the candidate patterns. Their approach partitions every pattern into sub-graphs so that each sub-graph could be completely mapped onto a set of FPGA logic elements with the same hardware configuration. This approach enabled them to estimate the number of logic elements required for implementing the patterns on the target FPGA.

In all of the work discussed above, the authors have used the underlying micro-architectural information during the enumeration phase, albeit in a limited way. For example, constraints such as the number of available input or output ports, the convexity of the patterns, etc., are commonly used as to prune the search space during enumeration. In addition to these mandatory constraints, we propose to incorporate additional constraints that encapsulate the target FPGA logic block structure so as to enumerate only the most profitable patterns as candidates for the selection stage. This approach can help to reduce the tool runtime of the enumeration algorithm by pruning off patterns, which do not lead to efficient mapping onto the FPGA architecture.

III. FPGA-AWARE PATTERN ENUMERATION

Figure 1 shows the proposed methodology for instruction set extension customization. We have used Trimaran [13] to compile the C-application and generate an Intermediate Representation (IR) for pattern enumeration. We have also used Trimaran 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.

A. FPGA-Aware Pattern Enumeration

Pattern enumeration is defined as the process of identifying all the legal patterns within a DFG. The legal patterns of an application must obey a set of constraints that is compatible with the micro-architectural constraints (e.g. I/O ports, convexity, etc.) of the underlying architecture [3][9]. Theoretically, a DFG with ‘n’ nodes can have up to 2^n number of possible patterns. The micro-architectural constraints are used to prune away the illegal patterns. In addition, memory and branch operations are generally not allowed in a custom instruction. However, the number of possible legal patterns can still be exponentially large after taking into consideration these micro-architectural constraints.

The pattern enumeration technique proposed in this paper extends the enumeration method in [10], which has been shown to be orders of magnitude faster than existing algorithms for large DFGs. In particular, we have incorporated additional constraints, which ensure that the enumerated patterns can be mapped onto the target FPGA in a predictable manner.

The additional pruning constraints are based on the hardware estimation rule-sets that are proposed in [11]. In particular, the work 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 (a set of FPGA logic elements with the same hardware configuration). 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. Two sets of rules have been used 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 along with the conventional micro-architectural constraints, in order to enumerate only “clusters”. This ensures that every enumerated pattern can be implemented in a single logic block of the target FPGA architecture. It is noteworthy that the proposed enumeration strategy can be used for different FPGA families. In the next subsection we will briefly describe the enumeration algorithm used in [10] and explain the modifications made in order to incorporate the target architecture information.

B. Modifications to Existing Enumeration Algorithm

The enumeration algorithm proposed in [10] works in a recursive manner to identify valid patterns (also called
"cuts") from an application DFG. 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 \( P \) \( \leq \) IN\_LIMIT;
iii. Number of Output ports in \( P \) \( \leq \) 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 I/O ports available for the custom instructions in the target architecture. The enumeration algorithm works in the following manner: It starts with an empty pattern \( P \), the DFG \( G \) of a basic block and a "redundancy guarding node" referred to as \( r_g \). The algorithm then recursively invokes the enumeration procedure, which consists of three important functions, \( \text{select_node} \), \( \text{unite} \) and \( \text{split} \). Function \( \text{select_node} \) selects a node from the remaining node set \( (G-P) \) (set of nodes in \( G \) that excludes the nodes in pattern \( P \)) to be considered for inclusion in \( P \). The function \( \text{unite} \) handles the situation where the selected node is included into \( P \) to create a new pattern \( P' \). In particular, the function \( \text{unite} \) can merge a pattern, which consists of the selected node, with the existing pattern \( P \). As a result, the search process is accelerated. The output of the \( \text{unite} \) phase always produces a valid pattern \( P' \). On the other hand, the \( \text{split} \) function handles the situation where the selected node cannot be added to the pattern \( P \) to create a new legal pattern. In particular, the function \( \text{split} \) decomposes the current DFG \( G \) into one or two DFGs (e.g. \( G' \) and \( G'' \)) based on the selected node. 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. Details of this algorithm can be found in [10].

In order to perform FPGA-aware pattern enumeration, we have incorporated additional constraints in the \( \text{select_node} \) and \( \text{unite} \) functions. The new \( \text{select_node} \) returns a new node based on the types of nodes that are already present in the existing pattern, while the new \( \text{unite} \) 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-4 FPGA cannot implement more than one ADD/SUB operation in a single logic block, a valid pattern can only contain at most one ADD/SUB operation. To ensure this, the modified \( \text{select_node} \) 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 \( \text{select_node} \) function is called \( \text{mod_select_node} \) and the pseudocode is shown in Algorithm 1.

The first step in \( \text{mod_select_node} \) function checks the nodes in the existing pattern \( P \) and classify them into specific categories e.g. arithmetic, logic operations, etc. The remaining \( \text{mod_select_node} \) function works similar to the way described in [10], except that it now incorporates additional checks to return suitable nodes to the new \( \text{unite} \) function for valid pattern (cluster) generation. In particular, it checks the existing pattern for I/O violations. If the I/O constraints are violated, the function selects a suitable node from the successor or predecessor set (for output and input violation respectively), in order to resolve these violations. The \( \text{get_output_resolve_nodes} \) and \( \text{get_input_resolve_nodes} \) functions return suitable nodes to resolve the output and input violations in \( S_0 \) and \( S_1 \) respectively. These selected nodes are verified against the cluster generation rule-set before being returned as valid nodes. If no such nodes can be found, then the function returns a NOT\_FOUND value and the search branch is terminated. On the other hand, if there is no I/O violation in the existing pattern, \( \text{mod_select_node} \) function selects a suitable node from the disconnected set keeping in mind the hardware estimation rule-sets. If a suitable disconnected node can be found, it is selected and returned. Otherwise, the function selects a suitable node from the connected set and returns it if the node conforms to the cluster generation rule-set.

---

### Algorithm 1 Modified select_node

\[
\text{mod_select_node}(P, G, rg) \{ \\
1 \text{Classify the nodes of the pattern } \"P\" \text{ based on node operations} \\
2 \text{flag}=0; // flag used to avoid re-calculating the resolvable nodes } \\
3 \text{if } (|\text{OUT}(P)| > \text{OUT\_LIMIT} \&\& |\text{IN}(P)| > \text{IN\_LIMIT}) \{ /\text{denotes both I/O } \\
4 \text{flag == 0) }
5 \text{S_0 := get_output_resolve_nodes();} \\
6 \text{S_1 := get_input_resolve_nodes();} \\
7 \text{if } (|\text{S_0}| + |\text{S_1}| != 0) \\
8 \text{return NOT\_FOUND; } \\
9 \text{flag := 1; } \\
10 \text{if } (|\text{OUT}(P)| > \text{OUT\_LIMIT}) \{ /\text{denotes output violation in P } \\
11 \text{S_0 := get_output_resolve_nodes();} \\
12 \text{if } (|\text{S_0}| != 0) \\
13 \text{return a node in } \text{S_0} \text{ or an invalid node in Succ(G, P)} \text{which does not violate the cluster rules; } \\
14 \text{else } \\
15 \text{return NOT\_FOUND; } \\
16 \} \text{if } (|\text{IN}(P)| > \text{IN\_LIMIT}) \{ /\text{denotes input violation in P } \\
17 \text{flag == 0) } \\
18 \text{S_1 := get_input_resolve_nodes();} \\
19 \text{if } (|\text{S_1}| != 0) \\
20 \text{return a node in } \text{S_1} \text{ or an invalid node in } \text{Pred(G, P)} \text{which does not violate the cluster rules; } \\
21 \text{else } \\
22 \text{return NOT\_FOUND; } \\
23 \} \text{if } (|\text{Disc}(G, P) | != 0) \{ /\text{denotes that there is no I/O violation in P } \\
24 \text{return a node in } \text{Disc}(G, P) \text{which does not violate the cluster rules; } \\
25 \} \text{if } (|\text{Pred}(G, P) \cup \text{Succ}(G, P) | != 0) \\
26 \text{return a node in } (\text{Pred}(G, P) \cup \text{Succ}(G, P)) \text{ which does not violate the cluster rules; } \\
27 \text{else } \\
28 \text{return NOT\_FOUND; } \\
\}
\]

*Note: if \((r_g \text{\_A\_NODE})\), each of these sets is intersected with \text{Pred}(G_r, r_g) where \(G_r\) is the DFG containing all the nodes of the basic block.*
Algorithm 2 Modified unite

```plaintext
mod_unite(P, G, r, u) {
    P = \begin{cases} 
    \{P \cup (u) \subseteq \text{Succ}(G, u) \cap \text{Pred}(G, P), \text{if } u \in \text{Pred}(G, P) \} \\
    \{P \cup (u) \subseteq \text{Pred}(G, u) \cap \text{Succ}(G, P), \text{if } u \in \text{Succ}(G, P) \} \\
    \{P \cup (u), \text{if } u \in \text{Disc}(G, P) \}
    \end{cases}
    
    1 \text{ if } ((u = \text{Disc}(G, P)) \text{ and } \text{Succ}(G, u) \cap \text{Pred}(G, P) = \emptyset) \text{ and } \text{OUT\_LIMIT} = 1)
    \text{return;}
    
    \text{r} = 0;
    \text{return;}
```

The new `unite` function also checks the existing pattern before merging the pattern consisting of the selected node with the existing pattern. Using the same example as before, if the existing pattern already contains an ADD/SUB operation, the `unite` function will not merge the existing pattern with a pattern that consists of an ADD/SUB operation. This limits the number of patterns that can be enumerated. The other rules were similarly incorporated in the new `unite` function. We call this modified `unite` function as `mod_unite` and the pseudo code is shown in Algorithm 2. The `mod_unite` function deviates from the algorithm described in [10] by incorporating an additional check for valid cluster generation.

Algorithm 3 Cluster Evaluation

```plaintext
cluster_evaluation(pattern) {
    \text{1. Sort the nodes in the pattern in topological order.}
    \text{2. bool has_arith_op, has_logical_op, has_shift_op = FALSE;}
    \text{//variables has_arith_op, has_logical_op & has_shift_op show}
    \text{//whether or not a pattern has arithmetic, logical or shift}
    \text{//operations respectively}
    \text{3. for every node in the pattern{
        \text{if node == arith_op
        \text{if has_arith_op = TRUE; //any prior node in the pattern
            //is an arithmetic operation
        \text{return 0; //cluster not possible; cluster cannot contain
            //two arithmetic ops.
        \text{else
            has_arith_op = TRUE;}
        \text{end if}
        \text{else if node == logical_op
        \text{if has_logical_op = TRUE; //any prior node in the pattern is
            //an arithmetic operation
        \text{return 0; //cluster not possible; a logical operation
            //cannot appear after an arithmetic operation
        \text{else
            has_log_op = TRUE;
        \text{end if}
        \text{end if}
    \text{19 return 1; //pattern is a valid cluster}
    \text{}}}
```

This additional check is achieved by a new function `cluster_evaluation`, which is described in Algorithm 3. This pseudo code, along with the required I/O constraints validation, can be used to test all the node constraints mentioned in [11] for a given pattern. It basically works by first arranging all nodes in a given pattern in a topological order and then traversing the pattern in that order. It does not allow patterns with two arithmetic operations or a pattern with logical 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.

In the final step, the patterns enumerated from the `mod_unite` function are tested for I/O constraints conformity. This step is especially useful for patterns containing "Shift Operations" since the inputs to a Shift Operation are handled differently in a FPGA [19].

IV. FPGA-AWARE PATTERN SELECTION

The aim of pattern selection is to select the most profitable patterns to be implemented as custom instructions. After enumerating all the possible legal patterns from the enumeration phase, graph isomorphism is performed using “VFLib” [14] to group isomorphic patterns. Each pattern group is defined as a template, whereby each template corresponds to a set of isomorphic patterns. Next, pattern selection is performed to find a set of non-overlapping patterns.

Algorithm 4 Pattern Selection

```plaintext
pattern_selection() {
    \text{1. remaining_area = area_constraint;}
    \text{2. Compute local MIS of patterns with area less than the area_constraint;}
    \text{3. Compute weight of these patterns based on a heuristic (no.of nodes*Number of local MIS);}
    \text{4. Sort all the templates in decreasing order of MIS weight;}
    \text{5. Find the template with the largest MIS weight;
    \text{6 if (Area of this template \leq remaining_area)
        \text{Choose this template for implementation;}
        \text{7. remaining_area} = (remaining_area)-(area of selected template);}
    \text{8. Remove the overlapping patterns from the conflict graph;
    \text{9. Restore the patterns temporarily removed previously from the Conflict graph;
    \text{10. Recompute the local MIS of the patterns left in the conflict graph and Proceed to Step 4;}
    \text{11 else if (Area of this template \geq remaining_area)
        \text{Choose the template with the next largest MIS weight;}
        \text{12. if (Area of this template \leq remaining_area)
            \text{Proceed with Step 8;}
        \text{13 else if (Area of this template > remaining_area)
            \text{Exit;}
        \text{14 else if node == shift_op
            \text{has_shift_op} = TRUE;
        \text{15 end if}
        \text{16 else if node == logical_op
            \text{has_logical_op} = TRUE;
        \text{17 end if}
        \text{18 else if node == arith_op
            \text{has_arith_op} = TRUE;
        \text{19 end if}}
    \text{19 return 1; //pattern is a valid cluster}
    \text{}}
```

This additional check is achieved by a new function `cluster_evaluation`, which is described in Algorithm 3. This pseudo code, along with the required I/O constraints validation, can be used to test all the node constraints mentioned in [11] for a given pattern. It basically works by first arranging all nodes in a given pattern in a topological order and then traversing the pattern in that order. It does not allow patterns with two arithmetic operations or a pattern with logical 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.
We have extended the approach described in [15] for pattern selection. Algorithm 4 shows the proposed pattern selection algorithm. A conflict graph is first created based on the enumerated 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. The conflict graph facilitates the choosing of a set of non-overlapping patterns for final implementation.

Once the conflict graph is created, we use the steps described in [15]-[16] 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. The computation of the local MIS relies on a similar heuristic used in [15] and [16], which aims to select large and recurring custom instructions. This heuristic is calculated as the product of the number of nodes in a template and the number of the corresponding pattern occurrence in the DFG:

$$MIS\ Wt. = (\text{No. of Nodes} \times \text{Frequency}) \ (1)$$

After the local MIS is calculated, the templates are arranged in decreasing order of their MIS weight. In order to perform constraint-aware pattern selection, the template with the largest MIS weight is evaluated first. It is selected for implementation if the area constraint is not violated. If the selection of a particular template violates the area constraint, we move on to evaluate the template with the next largest MIS weight and so on, until a suitable template is found and selected. Once a template is selected, the corresponding patterns of the template are implemented as custom instructions. These selected patterns are then removed from the conflict graph, and the remaining area for custom instruction implementation is updated. The algorithm repeats to find a new set of local MIS for each template. The algorithm terminates when either the conflict graph becomes empty or the area constraint is violated.

V. EXPERIMENTS AND RESULTS

In this section, we compare the results obtained using the proposed FPGA-aware custom instruction design methodology with a conventional approach. In the conventional approach, 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 conventional approach was taken from [9] and [10]. The pattern selection step uses the greedy selection methodology described in [15]. We have also implemented a design exploration step in the conventional approach that relies on solving the knapsack problem [4] and [7] to identify the most profitable set of custom instructions (from the selected patterns) that meet the given area constraint.

Figure 2 shows the reduction in the runtime of the proposed algorithm over the conventional method in the Enumeration and Selection phases. The runtime in this work has been measured using the ‘gettimeofday’ function in C and the time periods are shown in microseconds. It is noteworthy that although the time saved by enumerating lesser number of patterns is partially offset by the need to evaluate additional FPGA-aware constraints, there is still a significant saving in the overall runtime of the Enumeration Stage.

The percentage reduction in the runtime of pattern selection using the method discussed in Section IV is also shown in Figure 2. In particular, when compared to the conventional approach, the proposed method achieves an average runtime reduction of about 50% and a maximum runtime reduction of about 88%.

Next, we compare the actual time saved (in # of clock cycles) by using the proposed approach against the conventional method. We have tabulated these results in Table 1. The number of clock cycles saved by using these extended instructions for every area constraint (1 to 15) are calculated using the following equation.

$$cycle\ savings = \sum_{i=1}^{n} (NN \times DO) - \sum_{i=1}^{n} (CLP \times DO)$$

where NN denotes number of nodes (or operations) in a custom instruction, DO is the number of occurrences of the custom instruction, n is the total number of selected custom instructions and CLP is the critical path of the custom instruction. The first term in the above equation represents the software execution time of a custom instruction whereas the second term depicts the number of clock cycles required to execute the custom instruction in the hardware. Hence equation 1 gives a very accurate estimation of the number of clock cycles saved using the Extended Instruction Set Architecture. The average time saved for both the methods were obtained by averaging the time saved over varying area constraint starting from
to high performance at low cost. We have demonstrated in our experimental results that when the available FPGA area is limited, it is prudent to choose 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. Experimental results based on six applications from two widely used benchmark suites, MiBench [17] and MediaBench [18], show significant performance improvement of up to 3 orders of magnitude over the conventional approach. Future work in this area will explore the usefulness of the proposed approach in designs with larger available area.

VI. CONCLUSION

In this paper, we proposed a design methodology for automated custom instruction generation that takes into account the underlying target FPGA architecture in both the custom instruction enumeration and selection phases. The proposed enumeration approach incorporates a set of hardware estimation rule-sets that identifies custom instruction patterns which can be fully mapped onto the logic blocks of the FPGA architecture. These additional pruning constraints lead to significant reduction in the enumeration runtime. In addition, the proposed strategy restricts the problem size of the selection process without sacrificing on the quality of the final solution. As the hardware area-time of the candidates is available after enumeration runtime. In addition, the proposed strategy restricts the problem size of the selection process without sacrificing on the quality of the final solution. As the hardware area-time of the candidates is available after enumeration runtime. In addition, the proposed strategy restricts the problem size of the selection process without sacrificing on the quality of the final solution.

REFERENCES


[17] MiBench version 1.0(http://www.eecs.berkeley.edu/mibench/).

