Efficient Custom Instruction Identification with Exact Enumeration

Pan Yu and Tulika Mitra

Abstract

Extensible processors allow addition of application-specific custom instructions to the core instruction set architecture. These custom instructions are selected through an analysis of the program's dataflow graphs. The characteristics of certain applications and the modern compiler optimization techniques (e.g., loop unrolling, region formation, etc.) have lead to substantially larger dataflow graphs. Hence, it is computationally expensive to automatically select the *optimal* set of custom instructions. Heuristic techniques are often employed to quickly search the design space. In order to leverage full potential of custom instructions, our previous work [P. Yu and T. Mitra, *CASES*, 2004, p.69] proposed an efficient algorithm for *exact* enumeration of all possible candidate instructions (or patterns) given the dataflow graphs. But the algorithm was restricted to connected computation patterns. In this report, we describe efficient algorithms to generate all feasible patterns (connected and disjoint) based on our previous algorithm. More optimization techniques are used in both connected and disjoint pattern enumeration, resulting in further reduction of search space.

Keywords: ASIPs, customizable processors, custom instruction, instruction-set extensions, subgraph enumeration algorithm.

I. Introduction

The transition from desktop to embedded computing has made it crucial to design high performance, low cost embedded software/hardware systems within very short time-to-market window. The conventional approach of designing "hand-crafted" ASIC is too expensive and inflexible. On the other hand, general purpose processors, while inexpensive, are yet to meet the demanding performance requirement and usually consume too much power. These factors have resulted in the emergence of instruction-set extensible processors that consist of an existing processor core extended with application-specific *custom instructions*. These custom instructions execute on *custom functional units* (CFU) implemented in reconfigurable logic (as in Stretch S5 [4], NIOS from Altera [2] and Microblaze from Xilinx [17]) or ASIC (for example Lx [13] and Xtensa [14]). Application-specific instructions help simple embedded processors achieve considerable performance/energy efficiency. Moreover, the fact that the same set of custom instructions can benefit different programs from an application domain illustrates the flexibility of this approach [11], [13].

A custom instruction encapsulates the computation of a frequently executed subgraph of the program's *dataflow graph (DFG)*. A CFU is simply the hardwired datapath implementation of a custom instruction. Optimized hardwired CFUs help to improve performance through parallelism and chaining of operations. At the same time, custom instructions result in compact code size, less number of instruction fetches and decodes and elimination of temporary registers. All these factors reduces the total power consumption. When the same computation pattern appears elsewhere in the program or even in other programs, it can be converted to the same custom instruction and executed on the same CFU.

However, identifying the suitable set of subgraphs from a program's DFG to form a set of custom instructions that is optimal in performance, power and hardware cost (i.e., area) is not an easy problem. This problem involves two subproblems: (1) *custom instruction identification* – enumerate a set of candidate subgraphs from the program's DFG and (2) *custom instruction selection* – evaluate performance, power, area of each candidate and then select an optimal subset of them under various design constraints. In this paper, we put our emphasis on the discussion of the first problem. Interested readers can refer to [3], [11], [15], [21], [24] for various solutions to the second problem.

Enumerating all possible subgraphs of a given graph is intractable and computationally expensive. The number of subgraphs or patterns for a DFG is, in general, exponential in terms of the number of nodes in the DFG. However, some of these patterns are infeasible due to various microarchitectural constraints. Examples of such constraints include maximum number of input and output operands (due to restrictions on the number of register ports), area, and delay of each custom instruction. Moreover, a custom instruction is infeasible if it cannot be executed atomically (named as convexity constraint by [21] – see Section III-C for details).

Previous approaches either put very limiting constraints on the number of operands [12], [22] or use heuristics [7], [11] to explore the design space quickly. However, it has been shown [6], [24] that such approaches can significantly restrict the performance potential of using custom instructions. There are only two works [21], [25] targeting exhaustive enumeration of feasible patterns[1]. In [21], the algorithm walks through the enumeration space represented by a binary decision tree, and prunes unnecessary sub space effectively based on constraint violation of patterns. However, in the worst case, it will look at 2^N patterns where N is the number of nodes in the DFG. Therefore, scalability issues may occur when it deals with very large DFGs. Later, our previous work [25] addresses the scalability problem by presenting a fast pattern enumeration algorithm. Although the method is more scalable, it only produces the set of feasible connected patterns, while [21] generates the set of feasible connected and disjoint patterns. In this report, extensions of our previous algorithm is presented to deal with disjoint patterns. Moreover, although the algorithm structure for connected patterns remains similar, new pruning techniques are introduced to further reduce the combination space.

II. Related Work

The previous work in pattern enumeration can be classified according to the restrictions imposed on the feasibility of patterns and properties of generation process as follows:

Number of Operands: The maximum number of input and output operands of custom instructions is typically constrained due to length of instruction encoding and/or ports to register files. However, these restrictions can sometime lead to very efficient enumeration algorithms. For example, Pozzi et al. [22] has developed a linear time algorithm to identify the maximal Multiple Inputs Single Output (MISO) patterns. J. Cong et al. [12] enumerates all possible K-feasible MISO patterns (where K is the input operands constraint) through a single pass of the DFG. The problem of using Multiple Inputs Multiple Outputs (MIMO) patterns is that there can potentially be exponential number of them in terms of the number of nodes in the DFG. Arnold et al. [3] uses an iterative technique that replaces the occurrences of previously identified smaller patterns with single nodes to avoid the exponential blowup. Clark et al. [11] uses a heuristic algorithm that starts with small MIMO patterns and expands only the directions that can possibly lead to good patterns. Baleani et al. [7] uses another heuristic algorithm that adds nodes to the current pattern in topological order till input or output constraint is violated; it then starts a new pattern

[1]The method in [21] is an improved version of a previous one in [6] by the same authors

only with the node that caused the violation. All the last three algorithms only generate a subset of the candidate patterns that meet input, output, and convexity constraints. Therefore, they may miss opportunities to produce the globally optimal set of custom instructions. Other than ours, Pozzi's work [21] is the only known approach that exhaustively enumerates all possible patterns. However, scalability becomes a major obstacle when DFGs size increases.

Connectivity: A candidate subgraph (pattern) may contain one or more disjoint components. Including multiple components in a subgraph increases the potential to exploit parallelism and thus may provide better performance if the base architecture does not support instruction-level parallelism (ILP). On the other hand, doing so may not be beneficial for an ILP processor that would have been able to exploit this parallelism anyway. Under such context, custom instruction selection also needs to be considered carefully together with instruction scheduling to ensure reduction of critical path. [3], [7], [11], [12], [22], [25] identify subgraphs with only one component, while [9], [21] and [15] combine disjoint components.

Overlap: As the final set of selected custom instructions do not normally overlap in the DFG, [7], [22] do not consider overlapped candidate patterns (e.g., patterns $\{1, 2, 3\}$ and $\{2, 4\}$ in Fig. 1 (a) overlap at node 2, so only one of them will be enumerated). However, other works enumerate overlapped patterns as they may be used to produce a better optima considering pattern reuse, especially under tight area budget.

Explicitness: Two recent works [5], [20] use ILP formulation to generate a single best performing pattern in each iteration of their algorithms. In this way, all the patterns are potentially enumerated in a implicit manner and evaluated by the ILP solver. However, as only one pattern is generated, other patterns are lost. All other works identify patterns explicitly.

Order of pattern identification and selection: Most of the previous works take a two step approach where the first step identifies the set of candidate patterns and the second step does the selection. However, some heuristic algorithms, such as [11], combine the two steps. This way the likely bad patterns are eliminated on-the-fly, thereby reducing the time and storage complexity of the algorithm at the risk of missing the global optima.

Given a set of candidate patterns, various approaches have been proposed to select the optimal subset under different constraints. [21] proposes an optimal method to select N patterns. Both ILP-based [19], [24] and heuristic-based methods [11], [24] have been proposed to select patterns under area constraints. Finally, a dynamic programming approach has been proposed in [3] to select the optimal subset if there is no constraint on area or number of patterns.

We aim to enumerate all possible patterns (connected, disjoint, and possibly overlapped) that meet the input, output and convexity constraints. This provides the selection process an opportunity to find the globally optimal solution. Our approach is scalable both in terms of DFG size as well as number of input/output operands, and can be applied to large DFGs produced after modern compiler transformation techniques.

III. Custom Instruction Enumeration Problem

In this section, we formally define the custom instruction identification problem.

A. Dataflow Graph (DFG)

Given a program, custom instructions are identified on the dataflow graphs corresponding to the basic blocks. A **DataFlow Graph** $\mathrm{G(V,E)}$ represents the computation flow of data within a basic block. The nodes V represent the operations and the edges E represent the dependencies among the operations. $\mathrm{G(V,E)}$ is a directed acyclic graph (DAG). Node u is a predecessor of v if there exists a directed path $\{\mathrm{u}, \mathrm{x}_1, \ldots, \mathrm{x}_\mathrm{i}, \mathrm{v}\}$ between them, denoted as $\mathrm{u} \in$ **predecessors(v)**. Similarly, u is a successor of v if there exists a directed path $\{\mathrm{v}, \mathrm{x1}, \ldots, \mathrm{x}_\mathrm{i}, \mathrm{u}\}$ between them, denoted as $\mathrm{u} \in$ **successors(v)**. Note that $\mathrm{v} \in \mathrm{predecessor(v)}$ and $\mathrm{v} \in \mathrm{successor(v)}$.

The architectural constraints may not allow all types of operations to be included as part of a custom instruction. For example, memory access and control transfer operations are typically not included. Therefore, the nodes of the DFG are partitioned into valid nodes and invalid nodes.

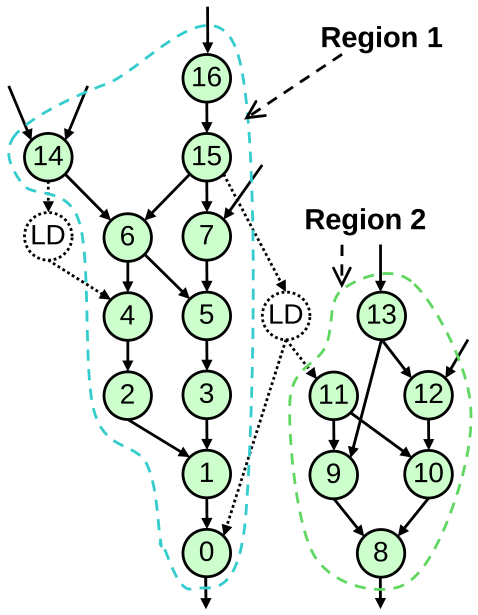

Fig. 1. An example dataflow graph. Valid nodes are numbered according to reverse topological order. Invalid nodes corresponding to memory load operations (LD) are unshaded. Two regions are separated by a LD operation.

A node in the DFG is a **valid node** if its corresponding operation can be included as part of a custom instruction; otherwise, it is an **invalid node**. An example DFG is shown in Fig. 1.

A DFG can be partitioned into multiple regions. Given a DFG $\mathrm{G(V, E)}$, we define a region $\mathrm{R(V', E')}$ as the maximal subgraph of G s.t. (1) $\mathrm{V'}$ contains only valid nodes, (2) there exists a path between any pair of nodes of $\mathrm{V'}$ in the undirected graph that underlies R, and (3) there does not exist any edge between a node in $\mathrm{V'}$ and a valid node in $\mathrm{(V - V')}$. Invalid nodes do not belong to any region. Fig. 1 shows a DFG divided into two regions by a memory load operation.

B. Patterns

Given a DFG, a **pattern** is a induced subgraph of the DFG. A pattern can be a possible candidate for custom instruction. For convenience, we represent a pattern by its set of nodes. A pattern P is **connected** if for any pair of nodes $\mathrm{\langle u, v\rangle}$ in p, there exists a path between u and v in the undirected graph that underlies the directed induced subgraph of p. A pattern is **disjoint** if it is not connected. The number of input and output operands of P are called **IN(p)** and **OUT(p)**, respectively. A node of p connected to an input (output) operand is called an input (output) node, and we denote p's input nodes and output nodes as **IN_SET(p)** and **OUT_SET(p)** respectively.

The following special patterns are of interest for custom instruction identification problem.

- **MISO**: A pattern p with only one output operand is called a MISO (Multiple Input Single Output) pattern. Clearly, a MISO pattern should be connected. MISO patterns are supported by all instruction set architectures (ISA).
- **Connected MIMO**: A connected pattern with multiple input operands and multiple output operands is called a connected MIMO (Multiple Input Multiple Output) pattern. MIMO patterns may not be supported by all ISAs.
- **Disjoint MIMO**: A disjoint pattern with multiple input operands and multiple output operands is called a disjoint MIMO pattern. A disjoint MIMO pattern consists of two or more MISO or MIMO patterns. Disjoint MIMO patterns are more useful for architectures with limited or no mechanisms to exploit instruction-level parallelism.

In addition, we define a special kind of pattern called cone. A **cone** is a rooted DAG in the dataflow graph s.t. either there is a path from the root node r to every other node in the cone (**downCone(r)**) or there is a path from every other node to the root node (**upCone(r)**). An upCone(r) is a MISO if r is the only output node of the cone. In Fig. 1, pattern $\{0, 1, 2, 3\}$ is an upCone(0), while pattern $\{6, 4, 5\}$ is a downCone(6). We also define maximal upCone (downCone) of a node r in a DFG G, **maxUpCone(r,G)** (**maxDownCone(r,G)**), as the upCone (downCone) in G rooted at r s.t. for any other upCone (downCone) q in G which is rooted at r, $\mathrm{maxUpCone(r, q)} \supset \mathrm{Q}$.

C. Feasibility of Patterns

Given a DFG, not all patterns are feasible as custom instructions. A pattern p is **convex** if there does not exist any path in the DFG from a node $\mathrm{x} \in \mathrm{p}$ to another node $\mathrm{n} \in \mathrm{p}$ that contains a node $\mathrm{l} \notin \mathrm{p}$. For example, $\{6, 14, 15\}$ is a convex pattern in Fig. 1. A pattern can be implemented as custom instruction if it is convex as non-convex patterns cannot be executed atomically. For example, in Fig. 1, pattern $\mathrm{p1}$ with nodes $\{4, 6, 14\}$ is non-convex (assuming memory load is an invalid operation). Similarly, pattern $\mathrm{p2}$ with nodes $\{5, 6, 15\}$ is also non-convex. However, note that the non-convexities of $\mathrm{p1}$ and $\mathrm{p2}$ arise due to different reasons. $\mathrm{p1}$ is non-convex because we cannot include the invalid node corresponding to memory load operation in the pattern, while $\mathrm{p2}$ is non-convex because we *choose not to include* node 7 in the pattern. We call the first case external non-convexity and the second one internal non-convexity. A non-convex pattern p is **external non-convex** if their exists a path from a node $\mathrm{m} \in \mathrm{p}$ to another node $\mathrm{n} \in \mathrm{p}$, which contains an invalid node $\mathrm{x} \notin \mathrm{p}$. Otherwise, the non-convex pattern is **internal non-convex**.

In addition, restrictions on instruction length and number of ports to the register file can put constraints on the maximum number of allowed input and output operands for a pattern. We call these **input constraint** and **output constraint** respectively. For example, if a custom instruction is allowed to have only one output operand, then the pattern $\{6, 14, 15\}$ in Fig. 1 is infeasible. In summary, *a pattern extracted from the DFG is feasible only if it is convex and satisfies the input and output constraints*.

D. Problem Definition

Given the DFG corresponding to a code fragment, the problem is to enumerate all feasible MISO, connected MIMO, and disjoint MIMO patterns for that code fragment. In the worst case, the number of feasible patterns of a DFG is exponential in terms of the number of nodes of the DFG. Therefore, the overall complexity of any *exact* enumeration algorithm is exponential. However, our experience suggests that, in practice, the number of feasible patterns in a DFG is far from exponential. Therefore, it is possible to design an efficient algorithm for exact enumeration of feasible patterns.

IV. Exhaustive Pattern Enumeration

We describe an efficient algorithm to exhaustively enumerate all feasible patterns in a DFG under input, output, and convexity constraints. We first briefly describe the current state-of-the-art algorithm for exhaustive enumeration.

SingleStep: Previous Algorithm: To the best of our knowledge, method in [21] is the only previous way that exhaustively enumerates all feasible patterns of a DFG. In this section, we briefly describe this algorithm as we use it as the baseline for comparison purposes. We will call this **SingleStep** algorithm in the rest of the paper as it enumerates all feasible MISO, connected MIMO, and disjoint MIMO patterns through a combined design space exploration. In contrast, we call our algorithm **MultiStep** algorithm as it generates MISO, connected MIMO, and disjoint MIMO patterns in three different stages.

The SingleStep algorithm first assigns labels $0 \ldots \mathrm{N} - 1$ to the valid operations (nodes) of the DFG in reverse topological order, where N is the number of valid operations in the DFG. It then searches an abstract binary tree containing $\mathrm{N} + 1$ levels and $2^{\mathrm{N}+1} - 1$ nodes to generate feasible patterns. The root node at level 0 represents the empty pattern. The two children of the root represent the presence and absence of operation 0, i.e., an empty pattern and a pattern containing operation 0, respectively. The nodes at level i $(0 < \mathrm{i} \leq \mathrm{N})$ represent all possible patterns with operations $0 \ldots \mathrm{i} - 1$. Basically, the search tree visits the operations in reverse topological order and explores the patterns corresponding to presence/absence of each operation. Clearly, the search space is exponential. However, the algorithm uses a clever strategy to prune the search space. If the pattern corresponding to a node s in the abstract search tree violates output and/or convexity constraint, then there is no need to explore the subtree of s. As the operations in the DFG are visited in reverse topological order, all the patterns corresponding to the nodes in the subtree of s are guaranteed to violate output and/or convexity constraint. Besides, certain cases of input violation caused by permanent inputs, which cannot be resolved in the deeper subtree, can also be used to prune the search space.

MultiStep: Our Algorithm: In contrast to the SingleStep algorithm, our MultiStep algorithm does not attempt to generate all feasible patterns in a single step. It breaks up the pattern generation process into three steps corresponding to cone, connected MIMO, and disjoint MIMO patterns. The first step generates upCones and downCones. Recall that a MISO pattern is a downCone with only one output node. Therefore, the first step implicitly generates all the MISO patterns. The second step combines two or more cones to generate connected MIMO patterns, and finally the third step combines two or more cones/MIMO patterns to generate disjoint MIMO patterns.

The MultiStep algorithm is based on the intuition that it is advantageous to separate out connected and disjoint MIMO pattern generation. The reason is the following. On one hand, connected MIMO pattern generation algorithm does not need to consider nodes that are far apart and have no chance of participating in a connected pattern together. Thus we can prune the design space considerably. On the other hand, lots of infeasible patterns are filtered out during connected pattern generation step and are not considered subsequently during disjoint pattern generation step. Thus the separation of concern speeds up the algorithm substantially.

MultiStep algorithm resembles the following two theorems.

Theorem 1: Any connected MIMO pattern p can be generated by combining convex upCones with at most $\mathrm{IN(p)}$ input operands or convex downCones with at most $\mathrm{OUT(p)}$ output operands.

Proof: Let $\mathrm{v_1, \ldots, v_N}$ be the nodes of P, where N is the number of nodes in p. Clearly, in the extreme case, $\mathrm{maxCone(v_1, p) \cup \ldots \cup maxCone(v_N, p) = p}$, where $\mathrm{maxCone(v_i, p)}$ can either be $\mathrm{maxUpCone(v_i, p)}$ or $\mathrm{maxDownCone(v_i, p)}$. By definition, $\mathrm{IN(maxUpCone(v_i, p)) \leq IN(p)}$ for any $\mathrm{1 \leq i \leq N}$. We prove by contradiction that $\mathrm{p_i = maxUpCone(v_i, p)}$ is convex. Let us assume that $\mathrm{p_i}$ is non-convex. Then, there exists at least a pair of nodes $\mathrm{m, n \in p_i}$ s.t. there exists a path from m to n that contains a node $\mathrm{y \notin p_i}$. As $\mathrm{p_i}$ is the maxUpCone of node $\mathrm{v_i}$ in p, if $\mathrm{y \notin p_i}$, then $\mathrm{y \notin p}$. Therefore, p is also non-convex, which is a contradiction. Similarly, we can prove the case for downCones. The case for $\mathrm{maxDownCone(v_i, p)}$ can be proved similarly. ■

Loosely speaking, it is possible to generate any feasible connected MIMO patterns by combining one or more cones. For example, the pattern {6, 7, 14, 15} in Fig. 1 can be generated by combining upCone(6) = {6, 14, 15} with downCone(15) = {7, 15}. The above theorem provides a key search space reduction by excluding combinations of any arbitrary cones. Specifically, to generate all the connected MIMO patterns, MultiStep algorithm only needs to generate all upCones that satisfy convexity/input constraints and all downCones that satisfy convexity/output constraints. This allows the algorithm to prune aggressively.

Theorem 2: Any connected component p of a feasible disjoint pattern dp must be a feasible connected pattern.

Proof: A connected component p of a disjoint MIMO pattern dp is a maximal connected subgraph in dp. An input of p must also be an input of dp. So $\mathrm{IN(p)} \leq \mathrm{IN(dp)}$. As dp satisfies input constraint, p must also satisfy the input constraint. The same reasoning holds for the output constraint.

We prove by contradiction that p is convex. Let us assume p is non-convex. Then there exists at least a pair of nodes $\mathrm{m}, \mathrm{n} \in \mathrm{p}$ s.t. there exists a path from m to n that contains a node $\mathrm{x} \notin \mathrm{p}$. There are two cases for x. (1) $\mathrm{x} \notin \mathrm{dp}$: In this case dp is also non-convex, which is a contradiction; (2) $\mathrm{x} \in \mathrm{dp}$: As p is a maximal connected subgraph, x is not connected to p. So there must be two nodes $\mathrm{y}, \mathrm{z} \notin \mathrm{p}$ and connected to p on a path $\langle \mathrm{m}, \mathrm{y}, \ldots, \mathrm{x}, \ldots, \mathrm{z}, \mathrm{n} \rangle$. We have $\mathrm{y}, \mathrm{z} \notin \mathrm{dp}$, otherwise they will belong to p too. So now we have two paths $\langle \mathrm{m}, \mathrm{y}, \ldots, \mathrm{x} \rangle$ and $\langle \mathrm{x}, \ldots, \mathrm{z}, \mathrm{n} \rangle$ that make dp non-convex, which is again a contradiction. So p must be convex. ■

Theorem 2 shows that a feasible disjoint pattern can be generated from one or more feasible connected patterns. The combination space among feasible patterns is much smaller than that of arbitrary patterns, resulting in more efficient enumeration.

The rest of this section describes our MultiStep algorithm in detail.

A. Generation of Cones

The first step generates all the convex upCones that satisfy input constraints and convex downCones that satisfy output constraints. Recall that a cone is a connected pattern and hence cannot contain nodes from different regions of a DFG. Therefore, we generate cones for each region individually. First, we traverse the nodes of each region in topological order and calculate the set of possible convex upCones that satisfy input constraints at each node. Similarly, we traverse the nodes of each region in reverse topological order to calculate the set of possible

Algorithm 1: Enumeration of upCones of region R

```
ConeGen
begin
1    for all nodes v of R in topological order do
2        $\mathrm{upConeSet(v)} := \{\{v\}\}$;
3        for all possible combination of immediate predecessors of v do
4            Let $v_1, \ldots, v_i$ be the selected immediate predecessors;
5            $\mathrm{tmpConeSet} :=$ CrossProduct$(\mathrm{upConeSet}(v_1), \ldots, \mathrm{upConeSet}(v_i), \{\{v\}\})$;
6            prune tmpConeSet for convexity and input violation;
7            $\mathrm{upConeSet(v)} := \mathrm{upConeSet(v)} \cup \mathrm{tmpConeSet}$;
end

CrossProduct $(\mathrm{set}_1, \ldots, \mathrm{set}_n)$
begin
8    $\mathrm{coneSet} := \phi$;
9    $\mathrm{set} := \mathrm{set}_1 \times \ldots \times \mathrm{set}_n$;
10   for each $s \in set$ do
11       Let $s = \langle s_1, \ldots, s_n \rangle$;
12       $\mathrm{coneSet} := \mathrm{coneSet} \cup \{s_1 \cup \ldots \cup s_n\}$;
13   return coneSet;
end
```

convex downCones at each node that satisfy output constraints.

Algorithm 1 details the generation of upward cones for a region R. We define $\mathrm{upConeSet(v)}$ as the set of upward cones for node v satisfying both the input operands and convexity constraints. Recall that each upward cone (pattern) in the set $\mathrm{upConeSet(v)}$, in turn, is again represented as a set of nodes. Given a node v, let $\mathrm{v}_1, \ldots, \mathrm{v}_\mathrm{k}$ be its immediate predecessors in the region. As we are traversing the nodes in topologically sorted order, the set of upward cones of $\mathrm{v_i}$ ($\mathrm{v_i} \in \mathrm{predecessors(v)}$) is known when v is visited. Therefore, we can compute all possible upward cones of v. For example, the set of upward cones of node 14 and 15 (in Fig. 1) are {{14}} and {{15},{15, 16}}, respectively. Therefore, the set of upward cones computed for node 6 is {{6}, {6,14}, {6,15}, {6,15,16}, {6,14,15}, {6,14,15,16}}.

This step may generate some upward cones (e.g., {5, 6, 15} at node 5 in Fig. 1) that do not satisfy convexity and/or input operands constraint. The algorithm eliminates such upward cones in line 6. Such elimination is safe, i.e., all upward cones satisfying input and/or convexity constraint can be produced, due to similar reasoning of Theorem 1. Note that the algorithm does

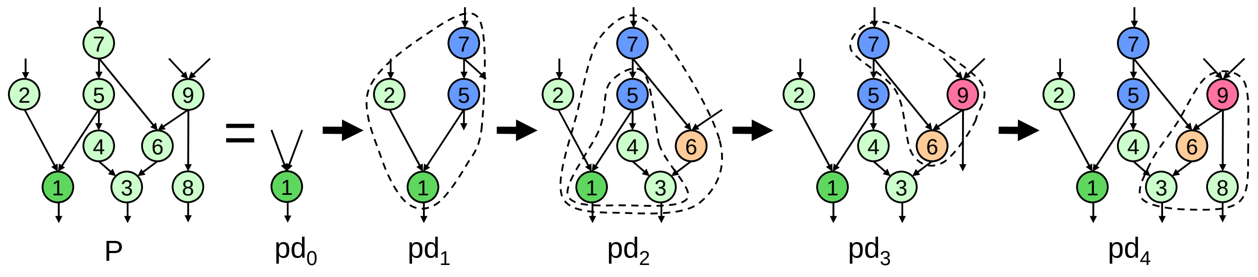

Fig. 2. Form a feasible connected MIMO pattern through partial decomposition. Decomposition cones are dashed on each step. Trivial decomposition cones, like 1 for every downward extension and 2 in pd_3, is omitted, which are eliminated in the algorithm.

not eliminate any upward cone that does not satisfy output constraint.

The generation of downward cones is similar to Algorithm 1. However, in this case, the traversal is in reverse topological order. Also the cones violating convexity and/or output constraints are eliminated.

B. Generation of Connected MIMO Patterns

1) Partial decomposition: In order to understand the mechanism of connected MIMO generation algorithm, let us first see how a feasible connected pattern can be decomposed and reproduced. Any feasible connected pattern p can be reproduced by concatenating a series of upward cones and downward cones. A **partial decomposition** is formed on each concatenation step, which is a connected subgraph of p. Starting from a sink node $\mathrm{v_s}$, which we treat as the initial partial decomposition $\mathrm{pd_0}$, we extend it upwards and downwards by adding upward cones and downward cones step by step until the partial decomposition becomes p. The process is as follows: Step 1, we extend $\mathrm{v_s}$ upwards, which is the initial extension node, by combining it with $\mathrm{maxUpCone(v_s, p)}$, such that $\mathrm{dp_1 = v_s \cup maxUpCone(v_s, p)}$; $\ldots$; Step $\mathrm{(n)}$, if the $\mathrm{(n-1)}$th step is upward, the nth step extends downwards through extension nodes set $\mathrm{ext = \{v | v \in OUT\_SET(dp_{n-1})\}}$, and produces $\mathrm{dp_n = dp_{n-1} \cup \Sigma_{v \in ext} maxDownCone(v, p)}$. If the $\mathrm{(n-1)}$th step is downward, the nth step extends upwards analogously on the reverse direction. The extension stops until the partial decomposition becomes the same as p. Fig. 2 shows an example graph, its decomposition cones and partial decompositions, starting with node

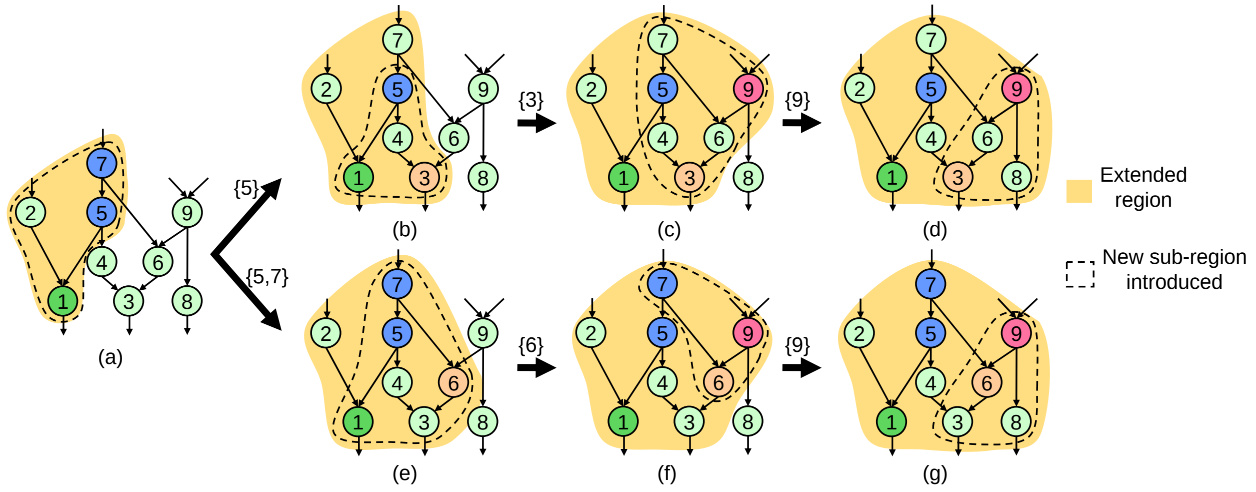

Fig. 3. Generating all feasible connected patterns involving node 1.

8.

We can get a few observations from the decomposition process. First, each constituent upward (or downward) cone satisfies input (or output) constraint and convexity constraint. Second, each partial decomposition after an upward (or downward) extension step satisfies input (or output) constraint and convexity constraint, and this suggests that intermediate patterns violating the constraints can be discarded. Third, a decomposition cone overlaps with the partial decomposition of the previous extension step at least on the corresponding extension nodes. Fourth, extension nodes that cannot introduce new nodes to the partial decompositions can be eliminated. For example, node 1 is a downward extension node in every downward extension step, however, it cannot extend to new nodes that the current partial decomposition has not reached, hence eliminated. The last three observations can help deduce pruning strategies in the connected MIMO generation algorithm.

Given a set of extension nodes, the set of feasible connected patterns containing all of them can be produced by combining cones rooted from them. So, as long as all the extension nodes are enumerated, the generation algorithm is complete. Instead of forming patterns individually, more productively, the set of patterns that can be extended through the same set of extension nodes can be processed together.

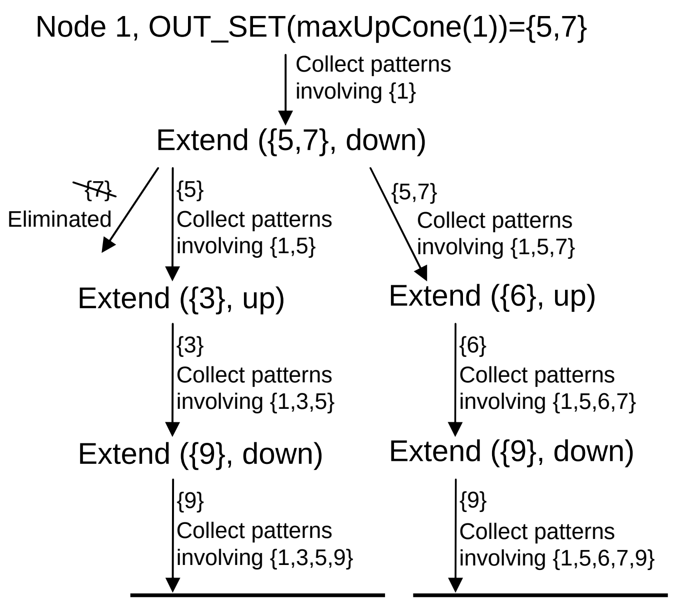
Node 1, OUT_SET(maxUpCone(1))={5,7}
Collect patterns involving {1}
Extend ({5,7}, down)
~~{7}~~
Eliminated
{5}
Collect patterns involving {1,5}
{5,7}
Collect patterns involving {1,5,7}
Extend ({3}, up)
Extend ({6}, up)
{3}
Collect patterns involving {1,3,5}
{6}
Collect patterns involving {1,5,6,7}
Extend ({9}, down)
Extend ({9}, down)
{9}
Collect patterns involving {1,3,5,9}
{9}
Collect patterns involving {1,5,6,7,9}

Fig. 4. A recursive process of collecting patterns for the example in Fig. 3.

We illustrate the key process of connected MIMO generation algorithm by walking through the generation of all feasible connected patterns involving node 1 in Fig. 3, assuming the graph in the previous example is a region itself and input and output constraints are not imposed. Firstly, we extend node 1 upwards, resulting in all the patterns in $\mathrm{upConeSet}(1)$. Instead of using the partial decomposition for a single pattern, we use the notion of extended region to identify extension nodes for a set of patterns. **Extended region** is a subgraph of region R that has been extended to. An upward (or downward) extension by node v will add $\mathrm{maxUpCone(v, R)}$ (or $\mathrm{maxDownCone(v, R)}$) to the existing extended region. The extended region after each extension step is shaded in the example. For now, the extended region is $\mathrm{maxUpCone(1, R)}$, and 2 downward extension nodes 5 and 7 are identified by taking its outputs (node 1 is a trivial downward extension node that is omitted). All possible downward (or upward) extensions to new nodes that have not been extended to are through the outputs (or inputs) of extended region. Two downward extension nodes 5 and 7 produces 3 combinations $\{5\}$, $\{7\}$ and $\{5,7\}$, indicating 3 extension possibilities each produces a different subset of patterns involving extension nodes 5 or 7. Extending through each combination would produce new extension nodes of its own, resulting in different extension paths, which implies a recursive process of further extension. For now, next step downward extension will split into three – extending through $\{5\}$, $\{7\}$ or $\{5,7\}$.

However, $\{7\}$ can be eliminated due to the same extension effects through $\{5,7\}$, because any further patterns involving node 1 and 7 must also contain node 5 for the sake of convexity. Such predecessor and successor relation between two extension nodes can help reduce the number of combinations greatly. Secondly, assume we take the $\{5\}$ extension path (Fig. 3 (b)), all the intermediate patterns from the previous extension step containing node 5 but not 7 are combined with downward cones rooted from it. Further upward extension node 3 is identified from the new extended region. Such extension goes on upwards and downwards interleavedly until no further extension nodes are identified. Fig. 4 illustrates the recursive extension process by function calls and patterns collected at each level. The top level obtains all the patterns involving node 1. Note that not all the patterns (including intermediate patterns) produced are feasible if inputs and outputs constraints are imposed, and they are deleted in the end.

2) Connected MIMO Generation Algorithm: We describe connected MIMO generation algorithm formally with Algorithm 2, and discuss the pruning techniques to eliminate redundant extension nodes combinations and subgraph combinations.

The algorithm traverses the nodes in a region R in reverse topological order (line 1). It maintains the following invariant: when the traversal of a node v is completed, all the feasible connected patterns involving v have been enumerated. Therefore, node v need not be considered further and can be masked, along with existing subgraphs/cones involving v.

For each starting node v as the initial extension node, which must be a sink node for the rest of region R, the algorithm resembles the process in the example of Fig. 3. It identifies new extension nodes (line 2, 21) and enumerates their combinations (line 9) upwards and downwards interleavedly, and split search recursively along each extension nodes combination with the `Extend` function. `Extend` takes in 4 arguments: (1) `direction` can take values `up` or `down`, and indicates whether the current extension step is upward or downward. (2) `MIMOSet` is a set of patterns passed from the previous level, as the base set of patterns to be combined with cones of extension nodes. (3) `oldExtStats` contains 3 sets of extension nodes, expressing the status of the extension: `newExt` is the set of extension nodes identified from the previous level, combinations of which will be enumerated at the current level; `extAll` and `extCombAll` are

Algorithm 2: Generation of feasible connected MIMO patterns of region R with redundancy reductions.

```
MIMOGen
begin
1   for all nodes $\mathrm{v}$ of $\mathrm{R}$ in reverse topological order do
2       $\mathrm{extStats.newExt} := $ ExtIdentify$(\mathrm{down}, \mathrm{extendedReg} :=$
            $\mathrm{maxUpCone(v, R), NODES(upConeSet(v)))};$
3       $\mathrm{extStats.extCombAll} := \mathrm{v};$
4       $\mathrm{extStats.extAll} := \mathrm{newExt};$
5       if $\mathrm{extension} \neq \phi$ then
6           $\mathrm{connectedMIMOSet(v)} := $ Extend$(\mathrm{down, upConeSet(v), extStats, extendedReg});$
7       remove $\mathrm{v}$ from $\mathrm{R}$;
end

Extend $(\mathrm{direction, MIMOSet, oldExtStats, oldExtendedReg})$
begin
8   $\mathrm{newMIMOSet} := \mathrm{MIMOSet};$
9   for all possible combination of $\mathrm{oldExtStats.newExt}$ do
10      Let $\mathrm{extComb} = \{\mathrm{v_1}, \ldots, \mathrm{v_i}\}$ be the current combination;
11      if ExtCombEli$(\mathrm{direction, extComb, oldExtStats.newExt})$ then continue;
12      $\mathrm{newExtStats.extCombAll} = \mathrm{oldExtStats.extCombAll} \cup \mathrm{extComb};$
13      $\mathrm{P} := \{\mathrm{p}|\mathrm{p} \in \mathrm{MIMOSet} \bigwedge \mathrm{p} \supseteq \mathrm{newExtStats.extCombAll} \bigwedge$
            $\mathrm{p} \cap (\mathrm{oldExtStats.extAll} - \mathrm{newExtStats.extCombAll}) = \phi\};$
14      if $\mathrm{direction} = \mathrm{down}$ then
15          $\mathrm{tmpMIMOSet} := $ CrossProduct$(\mathrm{downConeSet(v_1)}, \ldots, \mathrm{downConeSet(v_i)}, \mathrm{P});$
16          prune $\mathrm{tmpMIMOSet}$ for convexity and output violation;
17          $\mathrm{newExtendedReg} := \mathrm{oldExtendedReg} \cup (\bigcup_{\mathrm{v} \in \mathrm{extComb}} \mathrm{maxDownCone(v, R)});$
        else
18          $\mathrm{tmpMIMOSet} := $ CrossProduct$(\mathrm{upConeSet(v_1)}, \ldots, \mathrm{upConeSet(v_i)}, \mathrm{P}));$
19          prune $\mathrm{tmpMIMOSet}$ for convexity and input constraint violation;
20          $\mathrm{newExtendedReg} := \mathrm{oldExtendedReg} \cup (\bigcup_{\mathrm{v} \in \mathrm{extComb}} \mathrm{maxUpCone(v, R)});$
21      $\mathrm{newExtStats.newExt} := $ ExtIdentify$(\mathrm{!direction, newExtendedReg, NODES(tmpMIMOSet)});$
22      $\mathrm{newExtStats.extAll} := \mathrm{oldExtStats.extAll} \cup \mathrm{newExtStats.newExt};$
23      if $\mathrm{tmpExtension} \neq \phi$ then
24          $\mathrm{newMIMOSet} := \mathrm{newMIMOSet} \cup $ Extend$(\mathrm{!direction},$
                $\mathrm{tmpMIMOSet, newExtStats, newExtendedReg});$
        else
25          $\mathrm{newMIMOSet} := \mathrm{newMIMOSet} \cup \mathrm{tmpMIMOSet};$
26  prune $\mathrm{newMIMOSet}$ for input and output constraint violation;
27  return $\mathrm{newMIMOSet}$;
end
```

Algorithm 3: Auxiliary functions for the connected MIMO generation algorithm.

```
   ExtIdentify (direction, extendedReg, wetReg)
   begin
       /* Identify downward extension nodes */
 1     if direction = down then
 2         newExt := OUT_SET(extendedReg);
 3         for v ∈ newExt do
 4             if maxDownCone(v, R) ⊆ extendedReg then remove v from newExt;
 5             else if v ∉ wetReg then remove v from newExt;

       else
 6         // Upward case is analogous.
 7     return newExt;
   end
   bool ExtCombEli (direction, extComb, newExt)
   begin
 8     for any pair u and v ∈ newExt, where u ∈ predecessor(v) do
 9         if direction = down ⋀ u ∈ extComb ⋀ v ∉ extComb then return true;
10         else if direction = up ⋀ v ∈ extComb ⋀ u ∉ extComb then return true;
11     return false;
   end
```

the set of all the extension nodes identified and selected respectively up to the previous level, and are used to pick up a subset of patterns in `MIMOSet` to extend (line 13). (4) `oldExtendedReg` is the extended region from the previous step used to identify further extension nodes (line 2, 21).

Extension nodes are identified in the `ExtIdentify` function depicted in Algorithm 3. Downward extension nodes are identified as the output nodes of extended region. However, extension nodes that can not produce new patterns are eliminated in two ways. First, extension nodes introduce no new extended region are eliminated (line 4). Second, extension nodes falling outside the wet region are eliminated (line 5). **Wet region** is a subregion of R that contains nodes appearing in at least one partial decomposition in the nearest extension (as computed in line 21 of Algorithm 2), which supplies the base set of patterns to be extended. Recall that in the decomposition example, we have observed that a decomposition cone must be overlapping with the partial decomposition of the last extension step at least on the corresponding extension nodes. So, an extension node can be eliminated if none of the base patterns overlaps with it.

Function `ExtCombEli` tests a given extension node combination and bypasses it if it is redundant (line 11 in Algorithm 2). For two downward extension nodes u and v identified, if $\mathrm{u} \in \mathrm{predecessor(v)}$, the extension node combination with u but not v has the same effects with the combination containing both, thus can be bypassed safely (line 9). The reasoning for this is partial decompositions with node u must also contain node v, thus further extensions of two cases will be the same. Suppose previous upward extension node e is successor of both u and v (such e must exist obviously). So all the partial decompositions contain node e. As a result, if a partial decomposition contains u, it must also contain v to ensure the convexity to node e. Line 10 is the test along upward direction analogously.

C. Generation of Disjoint MIMO Patterns

Disjoint pattern enumeration algorithm produces the set of all feasible disjoint MIMO patterns denoted as DPS. According to Theorem 2, each disjoint pattern $\mathrm{dp} \in \mathrm{DPS}$ is composed of more than one connected patterns and satisfy the input, output and convexity constraints. We use the the set of all feasible connected MIMO patterns denoted as CPS as the base to produce all the disjoint patterns.

We observed that the number of output nodes of any feasible disjoint pattern is simply the summation of those of its constituent connected patterns. Based on this observation, we classify the patterns according to the the number of output nodes. We define $\mathrm{CPS_i}$ and $\mathrm{DPS_i}$ as set of all the feasible connected patterns and disjoint patterns with exactly i output nodes, respectively. Note that according to our definition $\mathrm{CPS_i} \cap \mathrm{DPS_i} = \emptyset$. Feasible disjoint patterns with n output nodes can be generated by combining feasible connected patterns with less than n output nodes. More formally, we have to consider all possible *partitions* of n (a partition of a positive integer n is a way of writing n as a sum of positive integers) except for the partition with single element n. For example, the partitions of integer 4 are 4, $3+1$, $2+2$, $2+1+1$, $1+1+1+1$. Therefore

$$\begin{aligned}\mathrm{DPS_4} \quad &= \quad (\mathrm{CPS_3} \times \mathrm{CPS_1}) \cup (\mathrm{CPS_2} \times \mathrm{CPS_2}) \cup (\mathrm{CPS_2} \times \mathrm{CPS_1} \times \mathrm{CPS_1}) \\ & \quad\quad \cup (\mathrm{CPS_1} \times \mathrm{CPS_1} \times \mathrm{CPS_1} \times \mathrm{CPS_1})\end{aligned}$$

where $\times$ and $\cup$ represent cross product and union operations, respectively. However, we can simplify the disjoint pattern generation process by replacing certain parts of the above equation with $\mathrm{DPS_i}$. Following we show the equations for disjoint patterns with up to 5 output nodes.

$$\begin{aligned}
\mathrm{DPS_1} &= \emptyset \\
\mathrm{DPS_2} &= \mathrm{CPS_1} \times \mathrm{CPS_1} \\
\mathrm{DPS_3} &= (\mathrm{CPS_2} \times \mathrm{CPS_1}) \cup (\mathrm{CPS_1} \times \mathrm{CPS_1} \times \mathrm{CPS_1}) \\
&= (\mathrm{CPS_2} \times \mathrm{CPS_1}) \cup (\mathrm{DPS_2} \times \mathrm{CPS_1}) \\
\mathrm{DPS_4} &= (\mathrm{CPS_3} \times \mathrm{CPS_1}) \cup (\mathrm{CPS_2} \times \mathrm{CPS_2}) \cup (\mathrm{CPS_2} \times \mathrm{CPS_1} \times \mathrm{CPS_1}) \\
&\quad \cup (\mathrm{CPS_1} \times \mathrm{CPS_1} \times \mathrm{CPS_1} \times \mathrm{CPS_1}) \\
&= (\mathrm{CPS_3} \times \mathrm{CPS_1}) \cup (\mathrm{CPS_2} \times \mathrm{CPS_2}) \\
&\quad \cup ((\mathrm{CPS_2} \times \mathrm{CPS_1}) \cup (\mathrm{CPS_1} \times \mathrm{CPS_1} \times \mathrm{CPS_1})) \times \mathrm{CPS_1} \\
&= (\mathrm{CPS_3} \times \mathrm{CPS_1}) \cup (\mathrm{CPS_2} \times \mathrm{CPS_2}) \cup (\mathrm{DPS_3} \times \mathrm{CPS_1}) \\
\mathrm{DPS_5} &= (\mathrm{CPS_4} \times \mathrm{CPS_1}) \cup (\mathrm{CPS_3} \times \mathrm{CPS_2}) \cup (\mathrm{DPS_4} \times \mathrm{CPS_1})
\end{aligned}$$

The above equations indicate that the disjoint patterns should be generated in increasing order of the number of output nodes (i.e., $\mathrm{DPS_2}$, $\mathrm{DPS_3}$, ...). Also each cross product operation is performed on two sets, i.e., each disjoint pattern is obtained by composing two previously generated patterns (connected or disjoint), thus simplifying the generation algorithm. Note that starting from $\mathrm{DPS_6}$, cross product operation on more than two sets need to be performed; for example $\mathrm{CPS_2} \times \mathrm{CPS_2} \times \mathrm{CPS_2}$ cannot be resolved. However, the term $\mathrm{CPS_2} \times \mathrm{CPS_2}$ appears during the generation of $\mathrm{DPS_4}$. By re-using these intermediate results, we can still ensure that the cross product is always performed with two sets.

Pruning: We observe that directly computing the right side of each equation may produce infeasible or redundant patterns. For example, if we combine two connected patterns that overlap with each other, the resulting pattern will either be connected or will have lesser number of output nodes than expected. Non-convex patterns may also be generated in this process. In order to avoid this, we must ensure that each feasible disjoint pattern is generated by combining two patterns $\mathrm{p1}$ and $\mathrm{p2}$ (disjoint or connected) that are (1) disjoint from each other and (2) there is no path from $\mathrm{p1}$ to $\mathrm{p2}$ or $\mathrm{p2}$ to $\mathrm{p1}$. The second condition ensures that combining the two patterns does not result in a non-convex disjoint pattern.

We define **upward scope** of a pattern p $(\mathrm{upScope(p)})$ for this purpose. It is the collection

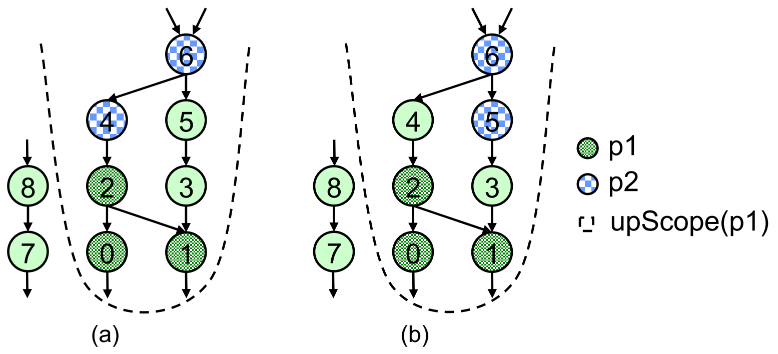

Fig. 5. Non-connectivity/Convexity check based on upward scope. (a) $\mathrm{p2}$ connects with $\mathrm{p1}$. (b) $\mathrm{p2}$ introduces non-convexity.

of all the predecessors of the nodes in pattern p. When combining two patterns $\mathrm{p1}$ and $\mathrm{p2}$, if $\mathrm{p1} \cap \mathrm{upScope(p2)} \neq \phi$ or $\mathrm{p2} \cap \mathrm{upScope(p1)} \neq \phi$, either non-connectivity and/or convexity condition will be violated; thus they need not to be combined. Fig. 5 shows these two cases. In disjoint pattern generation process, the upward scope for each pattern need to be computed and stored to perform this check.

To further prune the design space, we first number the nodes according to reverse topologically sorted order. Next we define $\mathrm{CPS_i^v}$ as the set of feasible connected patterns with i output nodes and v as the smallest numbered node. Similar definition applies to $\mathrm{DPS_i^v}$. Clearly,

$$\mathrm{DPS_i} = \bigcup_{\mathrm{v \in valid\ nodes}} \mathrm{DPS_i^v}$$

$$\mathrm{DPS} = \bigcup_{\mathrm{i=2}}^{\mathrm{MAXOUT}} \mathrm{DPS_i}$$

where MAXOUT is the output constraint.

Algorithm 4 details the disjoint pattern generation steps. It computes $\mathrm{DPS_i^v}$ for each valid node v in the innermost loop (line 17) according to the corresponding equation (line 8), aggregates them to form $\mathrm{DPS_i}$ (line 20) and finally DPS (line 21).

$\mathrm{DPS_i^v}$ is computed by combining pattern sets of node v with pattern sets of node u, where u is bigger than v in reverse topologically sorted order (line 6). Non-symmetrical terms, such as $\mathrm{CPS_1} \times \mathrm{CPS_2}$ should be combined twice with their place exchanged (line 18–19). Upward scope

Algorithm 4: Feasible disjoint pattern enumeration

```
DPSetGen
begin
1     $\mathrm{DPS} := \phi$;
2     for $\mathrm{i} = 2$ to $\mathrm{MAXOUT}$ do
3         $\mathrm{DPS_i} := \phi$;
4         for all valid nodes $\mathrm{v}$ of DFG in reverse topological order do
5             $\mathrm{DPS_i^v} := \phi$;
6             for all valid nodes $\mathrm{u}$ s.t. $\mathrm{order(u) > order(v)}$ do
7                 if $\mathrm{u} \in \mathrm{upScope(\{v\})}$ then continue with the next $\mathrm{u}$;
8                 for every term $\mathrm{T}$ on r.h.s. of the equation of $\mathrm{DPS_i}$ do
                      Let $\mathrm{T} = \mathrm{T1} \times \mathrm{T2}$;
9                     for all the patterns $\mathrm{p1}$ in $\mathrm{T1}$ with smallest node $\mathrm{v}$ do
10                        if $\mathrm{u} \in \mathrm{upScope(p1)}$ then
11                            continue with the next $\mathrm{p1}$;
12                        for all patterns $\mathrm{p2}$ in $\mathrm{T2}$ with smallest node $\mathrm{u}$ do
13                            if $\mathrm{p1} \cap \mathrm{upScope(p2)} \neq \phi$ or $\mathrm{p2} \cap \mathrm{upScope(p1)} \neq \phi$ then
14                                continue to the next $\mathrm{p2}$;
15                            $\mathrm{tmp} := \mathrm{p1} \cup \mathrm{p2}$;
16                            if InCheck $\mathrm{(tmp)}$ then
17                                $\mathrm{DPS_i^v} := \mathrm{DPS_i^v} \cup \{\mathrm{tmp}\}$;
18                    if $\mathrm{T1} \neq \mathrm{T2}$ then
19                        repeat lines 9 to 17 by exchanging the place of $\mathrm{T1}$ and $\mathrm{T2}$;
20                $\mathrm{DPS_i} := \mathrm{DPS_i} \cup \mathrm{DPS_i^v}$;
21        $\mathrm{DPS} := \mathrm{DPS} \cup \mathrm{DPS_i}$;
end
```

check helps reduce the design space at two places. First, node u can be entirely bypassed if it falls in $\mathrm{upScope(v)}$ (line 7); otherwise non-connectivity or convexity will be violated. Second, constituent pattern $\mathrm{p1}$ from pattern set of v can be bypassed if $\mathrm{upScope(p1)}$ overlaps with u (line 10). These two checks bypass a set of combinations at each time and greatly reduce the search space. A normal upward scope check between two constituent patterns is conducted before combining them (line 13). Lastly, the resultant pattern is added to $\mathrm{DPS_i^v}$ subject to input check (line 16–17).

D. Optimizations

In this section, we describe the data structures and some optimizations employed in the implementation of our pattern generation algorithm.

Data structures: We use fixed-length bit vectors to represent each pattern. The length of the bit vectors is equal to the number of nodes in the DFG. Given the bit vector of a pattern, each bit simply indicates the presence and absence of a node in that pattern. Bit vector representation provides a very natural and efficient means to combine two or more patterns (as in line 12 of Algorithm 1 through bit-wise OR operation). Many other information related to node set, such as max upward cone, predecessors and successors of a node, extended region, upward scope of a pattern, and etc., are also represented with bit vectors, and inter-operate with patterns using bit-wise operations.

Note that we need to remove duplicates while constructing a set of patterns. This step requires both efficient search as well as insertion that cannot be achieved either with sorted array or linked list. We maintain a set of patterns as a 2-3 Tree [1]. The patterns in a 2-3 tree are sorted by the value of their bit-vectors; every query or insertion of a pattern can be achieved within $\mathrm{O}(\log_2(\mathrm{n}))$ time, where n is the total number of patterns present in the 2-3 tree. A pattern is inserted in the 2-3 tree only if it is not present already.

Checking for Input/Output constraints: Given a pattern generated by combining patterns $\mathrm{p1}, \ldots, \mathrm{pn}$, $\mathrm{IN\_SET(p)} \subseteq \mathrm{IN\_SET(p1)} \cup \ldots \cup \mathrm{IN\_SET(pn)}$ (similarly for $\mathrm{OUT\_SET(p)}$). Therefore, in order to check for violation of input/output constraints in a pattern, we will need to look at the input/output nodes of the constituent patterns. For this purpose, we maintain the set of input/output nodes with each pattern.

Checking for convexity constraint: Convexity check of `DPSetGen` algorithm is done using upward scope because of the specialty of non-connectivity. Here we discuss convexity check in `ConeGen` and `MIMOGen` algorithms. In order to check for convexity of a individual pattern p, we consider all immediate successors from the nodes in $\mathrm{OUT\_SET(p)}$. If, for one such immediate successor $\mathrm{u} \notin \mathrm{p}$, $\mathrm{successors(u)} \cap \mathrm{p} \neq \phi$, then p fails the non-convexity constraint.

Recall that we also defined external non-convexity as the non-convexity caused by invalid nodes. Convexity check for individual patterns mentioned above is efficient for both internal and external non-convexity, however, external non-convexity can be identified even before a pattern is formed by observing the relations among its constituent patterns. Specifically, for any node v, there exists a **external conflicting set** ($\mathrm{ECS(v)}$, can be empty) s.t. any node within cannot coexist with v in a valid pattern, otherwise external non-convexity will occur. This is useful in the cross production of `ConeGen` (as in line 5 of Algorithm 1). As any resultant pattern contains v, constituent patterns from $\mathrm{upConeSet(v_1), \ldots, upConeSet(v_i)}$ involving any node in $\mathrm{ECS(v)}$ can be filtered out before the actual cross production, which reduce the number of combinations greatly. Similarly, in `MIMOGen`, because any resultant patterns contain all the selected extension nodes ($\mathrm{newExtStats.extCombAll}$), constituent patterns involving nodes in $\bigcup_{\mathrm{v_i}} \mathrm{ECS(v_i)}$ ($\mathrm{v_i} \in \mathrm{newExtStats.extCombAll}$) are filtered out before the cross productions (line 15 and 18).

Computing external conflicting sets involve a pre-processing step. Given a region R, we first identify special pairs of nodes, called **boundary pairs**. Two nodes u and v in R are called a boundary pair if there exists a path $\langle \mathrm{u}, \mathrm{x_1}, \ldots \mathrm{x_n}, \mathrm{v} \rangle$ in the DFG s.t. $\mathrm{x_1}, \ldots \mathrm{x_n}$ do not belong to R. For example in Fig. 1, $\langle 4, 14 \rangle$ and $\langle 0, 15 \rangle$ are boundary pairs. Clearly, if $\langle \mathrm{u}, \mathrm{v} \rangle$ is a boundary pair, then u and v cannot coexist in any convex pattern. Moreover for any node $\mathrm{x} \in \mathrm{maxUpCone(u, R)}$, it cannot coexist with any node $\mathrm{y} \in \mathrm{maxDownCone(v, R)}$ in a convex pattern and vice versa. $\mathrm{ECS(v)}$ is the collection of v's predecessors and successors that cannot coexists with v. Such predecessor set of v can be computed as the union of such predecessors of v's immediate predecessors and the ones that introduced by v if v forms a boundary pair. Hence the computation for all nodes in the region can be done through a single pass according to topological order. Analogously, such successor sets for all nodes can be obtained through a pass according to reverse topological order.

Refinement before cross productions: The number of temporary patterns generate from a cross production is the product of the number of patterns of participating pattern sets. **Refinement** filters away unnecessary patterns from constituent sets before the cross production, combining which certainly produce infeasible or redundant results, decreases the value of each factor of the multiplication and thus reduces the combination space greatly. Refinement can be used before cross production throughout the algorithm, according to different refine conditions. Line 13 in Algorithm 2 is an example of refinement explicitly written in the algorithm. Also, as discussed before, refinement can be applied according to external conflict sets. On the implementation part, in order to traverse all the patterns (which are stored as leaf nodes of a 2-3 tree) of a pattern set quickly, they are also linked as a linked list. Refinement is done by bypassing unnecessary patterns on the linked list before the cross production. The refined linked list should be restored before the set is used again, because other cross production may require different refinements.

Refinement according to selected extension nodes are used in Algorithm 2. Because every base pattern in P involves all the extension nodes in $\mathrm{oldExtStats.extCombAll}$ (note that $\mathrm{oldExtStats.extCombAll} \subseteq \mathrm{newExtStats.extCombAll}$), the resultant patterns of cross production must also involve these nodes. In a downward extension, a downward cone from $\mathrm{downConeSet}(\mathrm{v_i})$ without a selected extension node that is a successor of $\mathrm{v_i}$ has the same effect with a downward cone with the extension node, so the former one can be filtered away. For example in Fig. 3 (b), only downward cones containing node 1 are needed in $\mathrm{downConeSet}(5)$.

On demand downward cone set generation: The generation of downward cone sets of each node can be pushed to the time when they are needed in `MIMOGen`. The full set of $\mathrm{downConeSet}(\mathrm{v})$ is not necessary if v never becomes a downward extension node, or when it does, some nodes in $\mathrm{maxDownCone}(\mathrm{v})$ have already been masked. For instance, for the region in Fig. 3, $\mathrm{downConeSet}(2)$ is not needed because it will never become a downward extension node. Another example is suppose we visit node 3 instead of 1 first, node 7 is not a downward extension node for node 3. Node 7 will only become a downward extension node when `MIMOGen` visits other nodes (e.g., 1 or 8) after node 3 is done and masked. At that time, we only generate $\mathrm{downConeSet}(7)$ without the presence of node 3.

More pruning in `DpSetGen`: In `DPSetGen`, when combining $\mathrm{p1}$ and $\mathrm{p2}$ fails $\mathrm{upScope}$ check (line 13–14), $\mathrm{p2}$ is skipped. Besides, some patterns which are super graphs of $\mathrm{p2}$ that contain all the nodes in $\mathrm{p2}$ and other nodes reverse topologically ordered greater than all those in $\mathrm{p2}$ will also fail $\mathrm{upScope}$ check if combining with $\mathrm{p1}$, thus they can be skipped altogether

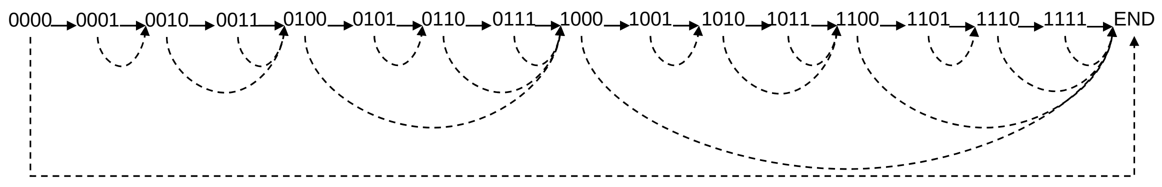

Fig. 6. Bypass pointers (dashed arrows) on a linked list of patterns.

with p2 [2]. Similar reasoning applies to line 10–11 for p1. Recall that a set of patterns are sorted on a 2-3 tree. We also link the patterns as a sorted linked list. The order is according to the value of the bit vector representing each pattern. We suppose node i occupies the ith bit (i.e., node 0 is represented as the leftmost bit). Under such representation, the patterns can be safely skipped with p2 are those whose bits to the left side of p2's rightmost "1" are the same with p2. For example, if p2 is 0101000, at most 8 patterns can be bypassed whose values range from 0101000 to 0101111. So we can safely jump to the first pattern with bit vector value larger than 0101111 (this pattern may not be 0110000 because patterns in the set may not be continuous). In order to make use of this, we add a **bypass pointer** to each pattern, pointing to the next pattern that can be skipped to if upScope check is failed. Fig. 6 illustrates a list of patterns with their bypass pointers. To compute the bypass pointers, we traverse the linked list once sequentially while maintaining a stack. We define **bypass value** as the largest value that can be skipped for each pattern (e.g., for 0101000, the bypass value is 0101111). When we are at pattern p, we pop out all the patterns on the top of the stack whose bypass value is less than p's bit vector value and set their bypass pointers to p, and then we push p onto the stack. At the end of the list, we set the bypass pointers of remaining patterns on the stack to the END of the linked list.

Miscellaneous: In `coneGen`, for two immediate predecessors of node v – v1 and v2, if v1 is predecessor of v2, v1 can be eliminated from all immediate predecessor combinations. Similar elimination holds when generating downward cone sets along the reverse direction. This elimination is very helpful to reduce combinations in some benchmark programs.

[2]In fact, all the patterns in the set which are super graphs of p2 are sure to fail upScope check. However, they are scattered discretely in the pattern list and unable to be skipped efficiently.

Benchmark	**Domain**	**BB Size**	**Size of Regions**	**% of Total Exec. Time**
rijndael†	Encryption	894	{562,68,4,4,4,4,1}	61%
blowfish†	Encryption	334	{133,120,2}	46%
sha(unroll)†	Encryption	1468	{1367,1}	54%
cjpeg†	Encoding	154	{92,40,1,1,1}	7%
MD5§	Encryption	943	{667,1×56}	67%

TABLE I

BENCHMARK CHARACTERISTICS. THE SIZE OF BASIC BLOCK AND REGION ARE GIVEN IN TERMS OF NUMBER OF NODES (INSTRUCTIONS).

V. EXPERIMENTAL EVALUATION

We compare the efficiency of our MultiStep algorithm against SingleStep algorithm in this section. Since designers may have different concerns on connected patterns and disjoint patterns, both cases will be compared.

A. *Experiment Setup*

Table I shows the characteristics of the benchmarks used in our experiments. Benchmarks marked with † are taken from MiBench [16], and § from the internet[3]. These benchmarks fall into encryption and multimedia encoding domains, which are typically computation oriented and involve very large DFGs. We choose one frequently executed basic block from each benchmark for the DFG. The regions for the DFGs are also shown in Table I. For example, the DFG in `rijndael` consists of seven regions with 562, 68, 4, 4, 4, 4, 1 nodes, respectively. Note that a large portion of time is spent in executing the chosen basic block for each benchmark, and this justifies the effort in selecting patterns from there large basic blocks.

The benchmarks are compiled and evaluated under SimpleScalar tool set using SimpleScalar ported gcc-2.7.2.3 with -O3 optimization [10]. We have run all the experiments on a 3.0GHz

[3]http://sourceforge.net/projects/libmd5-rfc by L. Peter Deutsch

Pentium 4 machine with 1GB memory. We have measured the time taken by the enumeration algorithms using the Pentium time-stamp cycle counter.

B. Comparison on Connected Pattern Enumeration

The first two steps of our MultiStep algorithm generate all the feasible connected patterns. Note that the original SingleStep algorithm enumerates both connected and disjoint patterns, therefore works on the entire DFG as opposed to individual regions in a DFG. To enumerate connected patterns, we invoke SingleStep algorithm for each region separately for comparison purpose. Also, for each generated pattern, we do an additional check to see if it is connected. We perform a depth first search of the pattern subgraph starting with the most recently added node. If the depth first search reaches all the nodes, then the pattern is connected. Experimental results indicate that the overhead for this additional check is minimal.

Table II contains the results for all the benchmarks under different input/output constraints. Two algorithms produce the same sets of feasible connected patterns for each benchmark (under "No. of Feasible Connected Patterns" column). Compared to the size of the regions, the number of feasible connected patterns is quite small. Therefore, it is possible to apply an optimal selection method, such as ILP formulation [19], [24], in the later stage for an optimal set of custom instructions.

The "Search Space" columns are the number of patterns subjected to different constraint checks by the two algorithms. In general, as SingleStep algorithm produces connected patterns by extending existing ones with neighbors, it is far more effective in pruning infeasible patterns. The last two columns presents the actual execution time of the two algorithms. Our MultiStep algorithm takes at most seconds to get connected feasible patterns in all cases, while SingleStep algorithm sometimes require thousands of seconds (e.g., 5-input 3-output cases of `Rijndael` and `Sha`). Compared to the previously reported results of our algorithm [25], efficiency of the current one is greatly improved due to various new pruning techniques and optimizations discussed.

Benchmark	**IN**	**OUT**	**Search Space** *SingleStep*	**Search Space** *MultiStep*	**No. of Feasible Connected Patterns**	**Time** *SingleStep* **(sec)**	**Time** *MultiStep* **(sec)**
Rijndael	3	1	322218	1926	437	0.339	0.012
	3	2	25988184	3450	619	27.10	0.021
	3	3	372627758	3744	619	427.2	0.030
	4	1	330585	2425	675	0.361	0.015
	4	2	33908883	13125	1177	35.35	0.041
	4	3	1031215148	63051	1495	1121	0.143
	5	1	338948	2885	714	0.357	0.018
	5	2	37153534	19989	1680	37.89	0.053
	5	3	1597049641	72771	2910	1702	0.202
Blowfish	3	1	32080	823	177	0.024	0.003
	3	2	189252	1378	252	0.149	0.004
	3	3	344635	1528	252	0.359	0.006
	4	1	34419	1163	279	0.026	0.004
	4	2	275745	3923	554	0.204	0.008
	4	3	743840	4683	704	0.670	0.016
	5	1	35120	1527	307	0.026	0.005
	5	2	314981	9582	894	0.230	0.014
	5	3	1205486	11916	1594	1.000	0.016
Sha(unroll)	3	1	6390037	12029	1222	11.32	0.047
	3	2	91239564	17682	2270	211.2	0.105
	3	3	355703427	20545	2987	1282	0.147
	4	1	7834675	35680	2343	13.83	0.121
	4	2	147686544	57246	5019	320.6	0.281
	4	3	824924965	81255	7931	2508	0.525
	5	1	8994322	90456	3997	15.91	0.297
	5	2	208654630	146414	8717	437.1	0.642
	5	3	1486041112	321797	16122	4086	1.935
Cjpeg	3	1	19782	717	166	0.015	0.001
	3	2	891973	970	249	0.541	0.003
	3	3	7223032	998	249	4.624	0.003
	4	1	21242	1537	306	0.016	0.003
	4	2	1476434	2985	511	0.890	0.008
	4	3	26641228	3391	633	16.68	0.011
	5	1	22321	3789	387	0.017	0.006
	5	2	1938275	9221	834	1.168	0.020
	5	3	61492729	14118	1191	38.08	0.039
MD5	3	1	795706	3142	606	0.874	0.019
	3	2	3349367	4399	948	4.217	0.031
	3	3	5761443	4525	979	8.258	0.034
	4	1	957428	5584	1200	1.040	0.028
	4	2	4133343	7593	2132	5.200	0.045
	4	3	8038476	8245	2360	11.38	0.054
	5	1	1015344	9156	1613	1.120	0.041
	5	2	5367195	11936	3472	6.625	0.062
	5	3	11380619	15215	4124	15.90	0.090

TABLE II

COMPARISON OF ENUMERATION ALGORITHMS – CONNECTED PATTERNS

C. Comparison on All Feasible Pattern Enumeration

All three steps of MultiStep algorithm generate all the feasible patterns, including disjoint ones. Meanwhile, for the SingleStep algorithm, the overhead of ensuring pattern connectivity in previous experiments no longer exist. However, its search space increases because it works on the entire DFG instead of single regions.

The results are shown in Table III. The "Additional Combination MultiStep" is the total number of pattern pairs subject to various checks in the third step of MultiStep algorithm. When output constraint is 1, no additional combination is required because the third step is not performed. The reason is each valid node has at least 1 output, thus a 1-output pattern must be a connected one. The "No. of Feasible Patterns" column is obtained by summing up the total number of connected patterns and disjoint patterns. As can be seen, the number of all the feasible patterns is far greater than that of connected ones in most cases. For example, `Rijndael` explodes 179 times and `Sha` 120 times in 5-input, 3-output cases, respectively. The large number of feasible patterns renders optimal custom instruction selection methods seemingly infeasible, and one should probably seek resort from high quality heuristics. As to the execution time, MultiStep outperforms SingleStep on orders of 10X to 1000X, due to reasons discussed in Section IV.

VI. Conclusion

In this paper, we have introduced an efficient algorithm to enumerate all feasible candidate patterns under various architectural constraints. Compared with a previously proposed approach targeting the same problem, the running time of our algorithm achieves orders of magnitude speedup. This gives us the opportunity to explore large DFGs. We believe that it is important to explore large DFGs as compilers for ILP processors now routinely employ if-conversion, loop unrolling and region formation to work on bigger DFGs. The efficiency of our algorithm makes it possible to be integrated into state-of-art ASIP tool chains to perform design space exploration even in the early stage of the design.

Benchmark	**IN**	**OUT**	**Search Space** *SingleStep*	**Additional Combination** *MultiStep*	**No. of Feasible Patterns**	**Time** *SingleStep* **(sec)**	**Time** *MultiStep* **(sec)**
	3	1	412567	0	437	0.446	0.012
	3	2	33014612	116666	3612	36.99	0.021
	3	3	434738397	812455	3612	518.7	1.102
	4	1	424929	0	675	0.754	0.015
Rijndael	4	2	44573604	169762	54203	54.85	0.486
	4	3	1280116614	13599267	66785	1564	18.54
	5	1	437287	0	714	0.475	0.018
	5	2	49440953	176534	115434	56.75	0.722
	5	3	2095522364	26956483	520993	2296	43.93
	3	1	65226	0	177	0.063	0.003
	3	2	430665	3354	522	0.547	0.009
	3	3	751917	11634	522	2.297	0.018
	4	1	70145	0	279	0.168	0.004
Blowfish	4	2	645364	4580	2577	0.769	0.018
	4	3	1671412	44452	2937	5.534	0.062
	5	1	71550	0	307	0.069	0.005
	5	2	746739	4608	4728	1.662	0.027
	5	3	2876509	73442	8428	7.498	0.126
	3	1	6391404	0	1222	11.41	0.047
	3	2	94121024	79072	6172	217.6	0.331
	3	3	365542922	515750	9796	1328	1.135
	4	1	7836042	0	2343	13.93	0.121
Sha(unroll)	4	2	152320527	116723	38728	331.5	0.704
	4	3	866118119	3905462	78566	2616	6.359
	5	1	8995689	0	3997	15.91	0.297
	5	2	215044666	166911	82022	449.8	1.360
	5	3	7577280675	7487850	280809	4312	15.44
	3	1	34715	0	166	0.020	0.001
	3	2	2571515	39945	911	1.507	0.037
	3	3	37250374	228304	960	22.53	0.192
	4	1	37343	0	306	0.022	0.003
Cjpeg	4	2	4234944	84718	13590	2.485	0.113
	4	3	122703827	4771054	18180	73.35	4.662
	5	1	39406	0	387	0.223	0.006
	5	2	5571468	116771	37603	3.277	0.210
	5	3	271219380	15162301	142348	161.4	17.68
	3	1	996513	0	606	2.632	0.019
	3	2	4489507	75841	1255	17.58	0.155
	3	3	8210790	118955	1328	37.92	0.247
	4	1	1124690	0	1200	3.186	0.028
MD5	4	2	7006628	110519	43106	27.36	0.354
	4	3	13460076	6703984	46028	60.60	9.745
	5	1	1194981	0	1613	4.030	0.041
	5	2	9730310	134698	79737	34.27	0.543
	5	3	21367000	9921718	119155	90.94	15.38

TABLE III

COMPARISON OF ENUMERATION ALGORITHMS – DISJOINT PATTERNS

REFERENCES

[1] A. Aho, J. Hopcroft, and J.D.Ullman. *Data structures and Algorithms*. Addison-Wesley, 1987.

[2] Altera. Nios embedded processor system development. `http://www.altera.com/products/ip/processors/nios`.

[3] M. Arnold and H. Corporaal. Designing domain-specific processors. In *CODES*, 2001.

[4] J. M. Arnold. S5: The architecture and development flow of a software configurable processor. In *FPT*, 2005.

[5] K. Atasu, D. Günhan, and and Özturan, Can. An integer linear programming approach for identifying instruction-set extensions. In *CODES+ISSS*, 2005

[6] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *DAC*, 2003.

[7] M. Baleani et al. Hw/Sw partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *CODES*, May 2002.

[8] E. Borin and et al. Fast instruction set customization. In *ESTImedia*, 2004

[9] P. Brisk, A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *CASES*, October 2002.

[10] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Toolset. Technical Report CS-TR96-1308, Univ. of Wisconsin - Madison, 1996. Available from `http://www.simplescalar.com`.

[11] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customization. In *MICRO36*, 2003.

[12] J. Cong, Y. Fan, G. Han, and Z. Zhang. Application-specific instruction generation for configurable processor architectures. In *FPGA*, 2004.

[13] P. Faraboschi et al. Lx: a technology platform for customizable VLIW embedded processing. In *ISCA*, 2000.

[14] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2), 2000.

[15] C. Galuzzi and et al. Automatic selection of application-specific instruction-set extensions In *CODES+ISSS*, 2006

[16] M. R. Guthausch et al. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001. Benchmark available from `http://www.eecs.umich.edu/mibench/`.

[17] Xilinx Inc. Microblaze soft processor core.

[18] R. Kastner et al. Instruction generation for hybrid reconfigurable systems. *ACM Transaction on Design Automation of Electronic Systems*, 7(2), 2002.

[19] J. Lee, K. Choi, and N. Dutt. Efficient instruction encoding for automatic instruction set design of configurable ASIPs. In *ICCAD*, 2002.

[20] R. Leupers and K. Karuri and S. Kraemer and M. Pandey. A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In *DATE*, 2006

[21] L. Pozzi, K. Atasu, and P. Ienne. Exact and Approximate Algorithms for the Extension of Embedded Processor Instruction Sets. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 25(7): 1209-29, July 2006.

[22] L. Pozzi, M. Vuletic, and P. Ienne. Automatic topology-based identification of instruction-set extensions for embedded processor. Technical Report 01/377, NSwiss Federal Institute of Technology Lausanne (EPFL), 2001.

[23] S. Talla. *Adaptive Explicitly Parallel Instruction Computing*. PhD thesis, New York University, 2000.

[24] P. Yu and T. Mitra. Characterizing embedded applications for instruction-set extensible processors. In *Proceedings of DAC*, 2004.

[25] P. Yu and T. Mitra Scalable custom instructions identification for instruction-set extensible processors. In *CASES*, 2004

[26] Z. A. Ye. et al. Chimaera: A high-performance architecture with a tightly-coupled reconfigurable functional unit. In *ISCA*, 2000.