DISJOINT PATTERN ENUMERATION FOR CUSTOM INSTRUCTIONS IDENTIFICATION

Pan Yu   *Tulika Mitra*

Department of Computer Science
National University of Singapore
{panyu,tulika}@comp.nus.edu.sg

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 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 paper, we describe an efficient algorithm to generate all feasible disjoint patterns starting with the set of feasible connected patterns. Compared to the state-of-the-art technique, our algorithm achieves orders of magnitude speedup while generating the identical set of candidate disjoint patterns.

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

1. INTRODUCTION

The transition from desktop to embedded computing has made it crucial to design high performance, low cost embedded 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 [3], NIOS from Altera [2] and Microblaze from Xilinx [21]) or ASIC (for example Lx [12] and Xtensa [13]). 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 [10, 12].

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 parallelization and chaining of operations. At the same time, custom instructions result in compact code size, reduced number of instruction fetch/decode and elimination of temporary registers. All these factors reduce 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* — enumeration of a set of candidate subgraphs from the program's DFG and (2) *custom instruction selection* — evaluation of the performance, power, area of each candidate and then selection of an optimal subset under various design constraints. In this paper, our focus is on the first problem. Interested readers can refer to [4, 10, 14, 18, 22] for various solutions to the second problem.

Enumerating all possible subgraphs of a given graph is intractable and computationally expensive. Previous approaches either put very limiting constraints on the number of operands [11, 19] or use heuristics [7, 10] to explore the design space quickly. However, it has been shown [6, 22] that such approaches can significantly restrict the performance potential of using custom instructions. There are only two works [18, 23] targeting exhaustive enumeration of feasible patterns[1]. In [18], the algorithm walks through the enumeration space represented by a binary decision tree, and prunes the space effectively based on constraint violation of the 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 [23] 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 [18] generates the set of feasible connected and disjoint patterns. In this paper, an efficient algorithm for the enumeration of disjoint patterns is introduced. The algorithm uses the set of feasible connected patterns as the base and can be integrated with any connected pattern enumeration algorithm.

[1] The method in [18] is an improved version of a previous one in [6] by the same authors

2. 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. [19] have developed a linear time algorithm to identify the maximal Multiple Inputs Single Output (MISO) patterns. J. Cong et al. [11] enumerate 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. As a result, most previous works [4, 7, 10] only generate a subset of the candidate patterns that meet input, output, and convexity constraints using various heuristics. However, they may miss opportunities to produce the globally optimal set of custom instructions. Other than ours, Pozzi's work [18] 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 the critical path. [4, 7, 10, 11, 19, 23] identify subgraphs with only one component, while [8], [18] and [14] combine disjoint components.

Overlap As the final set of selected custom instructions do not normally overlap in the DFG, [7, 19] do not consider overlapped candidate patterns (e.g., patterns $\{1, 2, 3\}$ and $\{2, 4\}$ in Fig. 1 overlap at node 2, so only one of them is enumerated). However, other works enumerate overlapped patterns to produce a better optima through pattern reuse, specially under tight area budget.

Explicitness Two recent works [5, 17] use ILP formulation to generate the single best performing pattern in each iteration of their algorithms. In this way, all the patterns are potentially enumerated in an 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.

We aim to enumerate all feasible patterns (connected, disjoint, and possibly overlapped) that meet the input, output and convexity constraints. This gives 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.

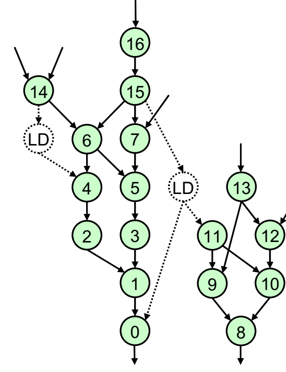

Fig. 1. An example dataflow graph. Invalid nodes corresponding to memory load operations (LD) are unshaded. Valid nodes are labeled in reverse topologically sorted order.

3. CUSTOM INSTRUCTION ENUMERATION PROBLEM

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

3.1. 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 $\mathrm{u} \in$ **predecessor(v)**. Note that $\mathrm{v} \in \mathrm{predecessor(v)}$.

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. 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.

3.2. 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 $\langle \mathrm{u}, \mathrm{v} \rangle$ 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 **IN(p)** and **OUT(p)**, respectively.

The following special patterns are of interest for custom instruction enumeration 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 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.

3.3. 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 $\mathrm{m} \in \mathrm{p}$ to another node $\mathrm{n} \in \mathrm{p}$ that contains a node $\mathrm{x} \notin \mathrm{p}$. 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 $\{4, 6, 14\}$ is non-convex (assuming memory load is an invalid operation).

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*.

3.4. Problem Definition

Given the DFG corresponding to a code fragment, the problem is to generate all feasible disjoint patterns corresponding to the DFG. We now transform the problem using the following theorem.

Theorem 1. *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. □

The above Theorem shows that a feasible disjoint pattern can be generated from one or more feasible connected patterns. Given the DFG and all the feasible connected patterns corresponding to a code fragment, the problem then is to enumerate all feasible disjoint MIMO patterns for the DFG.

4. DISJOINT MIMO PATTERN ENUMERATION

Disjoint pattern enumeration algorithm produces the set of all feasible disjoint MIMO patterns denoted as DPS. Each such 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

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

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.

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

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

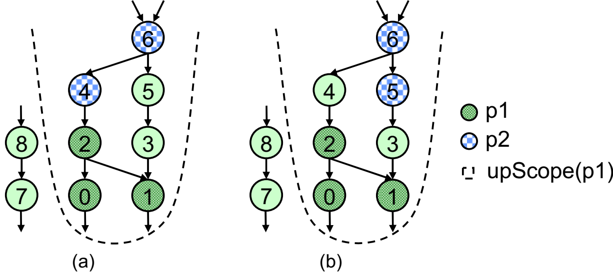

Fig. 2. Non-connectivity/Convexity check based on upward scope. (a) $\mathrm{p2}$ connects with $\mathrm{p1}$. (b) $\mathrm{p2}$ introduces non-convexity.

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 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. 2 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 1 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 (line 18–19). Upward scope 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). A comprehensive illustration of the algorithm is given in [24] with an example DFG.

Algorithm 1: Feasible disjoint pattern enumeration

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

end

4.1. Optimizations

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 patterns (as in line 15 of Algorithm 1 through bit-wise OR operation). Many other information related to node set, such as predecessors and upward scope of a pattern, are also represented as bit vectors.

Patterns in a pattern set are sorted according to their bit vector values. To perform efficient insertion that cannot be achieved either with sorted array or linked list, we maintain a set of patterns as a 2-3 Tree [1]. Every insertion of a pattern can be achieved within $\mathrm{O(\log_2(n))}$ time, where n is the total number of patterns present in the 2-3 tree. Sorted pattern set enables further pruning in Algorithm 1.

Further Pruning In Algorithm 1, when combining $\mathrm{p1}$ and $\mathrm{p2}$ fails $\mathrm{upScope}$ check (line 13–14), $\mathrm{p2}$ is skipped. Moreover, all the patterns in the set that are super graphs of $\mathrm{p2}$ can also be skipped. Unfortunately, these patterns are scattered throughout the pattern list. Due to the sorted pattern list, we can still efficiently skip the patterns that are super graphs of $\mathrm{p2}$ and the additional nodes have higher reverse topologically sorted order than the nodes in $\mathrm{p2}$. Similar reasoning applies to line 10–11 for $\mathrm{p1}$.

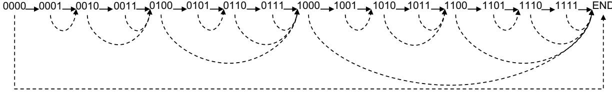

Fig. 3. Bypass pointers (dashed arrows) on a linked list of patterns.

Suppose node i occupies the ith bit from the left (i.e., node 0 is represented as the leftmost bit). Under such representation, the patterns that can be safely skipped with p are the ones with the same bit sequence up to p's rightmost "1". For example, if p 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 $\mathrm{upScope}$ check is failed. Fig. 3 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.

5. EXPERIMENTAL EVALUATION

We compare the efficiency of our algorithm against the current state-of-the-art algorithm [18] in this section. We first briefly describe the current state-of-the-art algorithm for exhaustive enumeration, as we use it as the baseline for comparison purposes.

5.1. SingleStep Algorithm

We call the algorithm in [18] **SingleStep** algorithm 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.

Benchmark	Domain	BB Size	% of Total Exec. Time
rijndael†	Encryption	894	61%
blowfish†	Encryption	334	46%
sha(unroll)†	Encryption	1468	54%
cjpeg†	Encoding	154	7%
MD5§	Encryption	943	67%

Table 1. Benchmark Characteristics. The size of basic block are given in terms of number of nodes (instructions).

5.2. Experiment Setup

Table 1 shows the characteristics of the benchmarks used in our experiments. Benchmarks marked with † are taken from MiBench [15], and § from the internet[2]. 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. 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 [9]. We have run all the experiments on a 3.0GHz Pentium 4 machine with 1GB memory. We have measured the time taken by the enumeration algorithms using the Pentium time-stamp cycle counter.

5.3. Efficiency Comparison on Pattern Enumeration

Our algorithm generates all the feasible connected patterns using algorithm presented in our previous work [23], and then enumerates all the feasible disjoint patterns.

Table 2 contains the results for all the benchmarks under different input/output constraints. Two algorithms produce the same sets of feasible patterns (connected and disjoint) for each benchmark (under "No. of Feasible Patterns" column). The fourth column is the number of patterns subjected to different constraint checks by SingleStep algorithm. The fifth and sixth columns are the number of patterns checked in connected pattern enumeration step and disjoint pattern enumeration step of MultiStep algorithm respectively[3]. In general, the search space of MultiStep algorithm is much smaller than that of SingleStep algorithm. Moreover, the search space of MultiStep algorithm for connected patterns is very small, thus the search for disjoint patterns dominates the enumeration. The last two columns presents the actual execution time of

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

[3]Reported numbers for connected pattern enumeration is different from [23] due to new pruning techniques introduced. Interested readers can refer to [24] for more information.

Benchmark	IN	OUT	Search Space *SingleStep*	Search Space *MultiStep* *Connected*	Search Space *MultiStep* *Disjoint*	No. of Feasible Patterns	Time *SingleStep* (sec)	Time *MultiStep* (sec)	Speedup $\frac{\texttt{SingleStep}}{\texttt{MultiStep}}$
Rijndael	3	1	412567	1926	0	437	0.446	0.012	**37**
	3	2	33014612	3450	116666	3612	36.99	0.021	**1761**
	3	3	434738397	3744	812455	3612	518.7	1.102	**471**
	4	1	424929	2425	0	675	0.754	0.015	**50**
	4	2	44573604	13125	169762	54203	54.85	0.486	**113**
	4	3	1280116614	63051	13599267	66785	1564	18.54	**84**
	5	1	437287	2885	0	714	0.475	0.018	**26**
	5	2	49440953	19989	176534	115434	56.75	0.722	**79**
	5	3	2095522364	72771	26956483	520993	2296	43.93	**52**
Blowfish	3	1	65226	823	0	177	0.063	0.003	**21**
	3	2	430665	1378	3354	522	0.547	0.009	**61**
	3	3	751917	1528	11634	522	2.297	0.018	**128**
	4	1	70145	1163	0	279	0.168	0.004	**42**
	4	2	645364	3923	4580	2577	0.769	0.018	**43**
	4	3	1671412	4683	44452	2937	5.534	0.062	**89**
	5	1	71550	1527	0	307	0.069	0.005	**14**
	5	2	746739	9582	4608	4728	1.662	0.027	**62**
	5	3	2876509	11916	73442	8428	7.498	0.126	**60**
Sha(unroll)	3	1	6391404	12029	0	1222	11.41	0.047	**243**
	3	2	94121024	17682	79072	6172	217.6	0.331	**657**
	3	3	365542922	20545	515750	9796	1328	1.135	**1170**
	4	1	7836042	35680	0	2343	13.93	0.121	**115**
	4	2	152320527	57246	116723	38728	331.5	0.704	**471**
	4	3	866118119	81255	3905462	78566	2616	6.359	**411**
	5	1	8995689	90456	0	3997	15.91	0.297	**54**
	5	2	215044666	146414	166911	82022	449.8	1.360	**331**
	5	3	7577280675	321797	7487850	280809	4312	15.44	**279**
Cjpeg	3	1	34715	717	0	166	0.020	0.001	**20**
	3	2	2571515	970	39945	911	1.507	0.037	**41**
	3	3	37250374	998	228304	960	22.53	0.192	**117**
	4	1	37343	1537	0	306	0.022	0.003	**7**
	4	2	4234944	2985	84718	13590	2.485	0.113	**22**
	4	3	122703827	3391	4771054	18180	73.35	4.662	**16**
	5	1	39406	3789	0	387	0.223	0.006	**37**
	5	2	5571468	9221	116771	37603	3.277	0.210	**16**
	5	3	271219380	14118	15162301	142348	161.4	17.68	**9**
MD5	3	1	996513	3142	0	606	2.632	0.019	**139**
	3	2	4489507	4399	75841	1255	17.58	0.155	**113**
	3	3	8210790	4525	118955	1328	37.92	0.247	**109**
	4	1	1124690	5584	0	1200	3.186	0.028	**114**
	4	2	7006628	7593	110519	43106	27.36	0.354	**77**
	4	3	13460076	8245	6703984	46028	60.60	9.745	**6**
	5	1	1194981	9156	0	1613	4.030	0.041	**98**
	5	2	9730310	11936	134698	79737	34.27	0.543	**63**
	5	3	21367000	15215	9921718	119155	90.94	15.38	**6**

Table 2. Comparison of enumeration algorithms

the two algorithms. MultiStep outperforms SingleStep on orders of 10X to 1000X as depicted in the last column.

6. 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.

Input/output and convexity constraints are the most general and minimal constraints on the dataflow subgraphs for CFU implementation. The specialty of particular CFU architectures, if any, can be applied on the complete set of of enumerated subgraphs to obtain the conforming ones; thus helping to reduce the number of candidates in the later custom instruction selection phase.

Exhaustive enumeration of candidate subgraphs favors pattern reuse through isomorphism. Even though isomorphism check is costly in general, finger print of a subgraph such as number of different type of operations can be used to quickly exclude non-identical pairs. However, optimal instruction selection on the complete set of subgraphs may not be practical. Consequently, heuristics to filter out most insignificant subgraphs are crucial. Note however, it is not the same as using heuristics for enumeration right from the beginning. Exhaustive enumeration provides a complete set of patterns for reuse and scheduling possibilities, which cannot be provided by current enumeration heuristics.

7. ACKNOWLEDGMENTS

This work was supported by NUS grant R252-000-292-112.

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

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

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

[9] D. Burger et al. Evaluating Future Microprocessors: The SimpleScalar Toolset. Technical Report CS-TR96-1308, Univ. of Wisconsin - Madison, 1996.

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

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

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

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

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

[15] M. R. Guthausch et al. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

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

[17] R. Leupers et al. A design flow for configurable embedded processors based on optimized instruction set extension synthesis. In *DATE*, 2006

[18] 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), 2006.

[19] L. Pozzi et al. Automatic topology-based identification of instruction-set extensions for embedded processor. Technical Report 01/377, EPFL, 2001.

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

[21] Xilinx Inc. Microblaze soft processor core.

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

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

[24] P. Yu and T. Mitra. Efficient Custom Instruction Identification with Exact Enumeration. Technical Report TRB5/07, National University of Singapore, 2007

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