Scalable Custom Instructions Identification for Instruction-Set Extensible Processors

Pan Yu
panyu@comp.nus.edu.sg

Tulika Mitra
tulika@comp.nus.edu.sg

School of Computing
National University of Singapore
Republic of Singapore 117543

ABSTRACT

Extensible processors allow addition of application-specific custom instructions to the core instruction set architecture. However, it is computationally expensive to automatically select the *optimal* set of custom instructions. Therefore, heuristic techniques are often employed to quickly search the design space. In this paper, we present an efficient algorithm for *exact* enumeration of all possible candidate instructions given the dataflow graph (DFG) corresponding to a code fragment. Even though this is similar to the "subgraph enumeration" problem (which is exponential), we find that most subgraphs are not feasible candidates for various reasons. In fact, the number of candidates is quite small compared to the size of the DFG. Compared to previous approaches, our technique achieves orders of magnitude speedup in enumerating these candidate custom instructions for very large DFGs.

Categories and Subject Descriptors: C.1.3 [Other Architecture Styles]

General Terms: Algorithm, Performance, Design.

Keywords: ASIPs, customizable processors, 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 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 [2, 11, 12, 15, 18] that consist of an existing processor core extended with application-specific *custom instructions*. These custom instructions execute on reconfigurable *custom functional units (CFU)*. Application-specific custom instructions help simple embedded processors to 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 [9, 11].

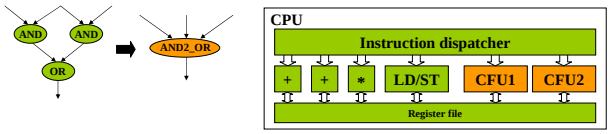

Figure 1: An example of custom instruction and architecture.

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. Figure 1 shows an example of a dataflow subgraph, its corresponding custom instruction, and the datapath of an extensible processor that includes the CFUs. Optimized hardwired CFUs help to improve performance through parallelism and chaining of operations. At the same time, custom instructions result in compact code size as well as less number of instruction fetches and decodes. Also the elimination of temporary registers within a custom instruction reduces register pressure. 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 focus on the first problem. Interested readers can refer to [3, 4, 9, 14, 17] 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

CASES'04, September 22–25, 2004, Washington, DC, USA.

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 [4] – see Section 3.1 for details).

Previous approaches either put very limiting constraints on the number of operands [10, 16] or use heuristics [6, 9] to explore the design space quickly. However, it has been shown [4, 17] that these approaches can significantly restrict the performance potential of using custom instructions. To the best of our knowledge, [4] is the only work that exhaustively enumerates all feasible patterns. The algorithm uses an effective pruning strategy for various constraints. However, in the worst case, it will look at 2^N patterns where N is the number of nodes in the DFG. Therefore, the exhaustive enumeration algorithm in [4] has scalability issues if the DFG is very large and/or the micro-architectural constraints are not very restrictive. Note that there exits a very efficient algorithm [5] to enumerate all connected induced subgraphs of a graph. However, the presence of microarchitectural constraints necessitates the design of a customized algorithm that can prune the design space containing infeasible subgraphs efficiently. It is not clear how [5] can be extended to do so.

The contributions of our work are the following.

- We present an efficient algorithm to enumerate all feasible patterns of a DFG under number of input/output operands and convexity constraints. The algorithm exploits the structure of the DFG and prunes the design space substantially based on the constraints. Experimental results indicate that our algorithm achieves orders of magnitude speedup over previous exact enumeration approach. Our algorithm is quite scalable so that it can be applied on large DFGs with relaxed micro-architectural constraints.

- Our experimental results confirm that the number of feasible patterns is indeed very small compared to the size of the DFG. Therefore, the set of patterns generated by our algorithm can be easily subjected to accurate design space exploration for an optimal subset.

The rest of the paper is organized as follows. We discuss related work in instruction-set extensible processors in Section 2. We define the problem formally and describe our algorithm in Section 3. Experimental results are presented in Section 4 and concluding remarks appear in Section 5. Finally, the proof of completeness of our algorithm appears in Appendix.

2. RELATED WORK

Identifying optimal set of custom instructions has received a lot of attention recently. As mentioned before, the process consists of two steps: enumeration of patterns and selection of an optimal subset of these patterns. The first step constructs a DFG for each basic block in the program and enumerates the patterns from that DFG. The previous work in pattern enumeration can be classified according to the restrictions imposed on the feasibility of patterns 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. [16] has developed a greedy algorithm that can identify the maximal Multiple Inputs Single Output (MISO) patterns. The complexity of the algorithm is linear in the number of nodes in the DFG. J. Cong et al. [10] 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. However, their algorithm does not enumerate all possible patterns and is restricted to very small number of input and output operands. Clark et al. [9] 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. [6] 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 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. Atasu's work [4] is the only known approach that exhaustively enumerates all possible patterns. It searches a full binary tree and decides at each step whether or not to include a particular instruction in a pattern. The potentially exponential search space is pruned based on violation of convexity and input or output constraints. Even though the algorithm is quite fast for small DFGs, it is not very scalable as DFG size or number of input/output operands 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. Also, including multiple disjoint components in a single custom instruction may generate very large patterns that have very little chance of reuse in a program or across programs. [3, 6, 9, 10, 16] identify subgraphs with only one component, while [4] and [7] combine disjoint components together. In this paper, we only identify connected custom instructions.

Overlap. As the final set of selected custom instructions do not normally overlap in the DFG, [6, 16] do not consider overlapped candidate patterns (e.g., patterns {1, 2, 3} and {2, 4} in Figure 2 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, especially under tight area budget.

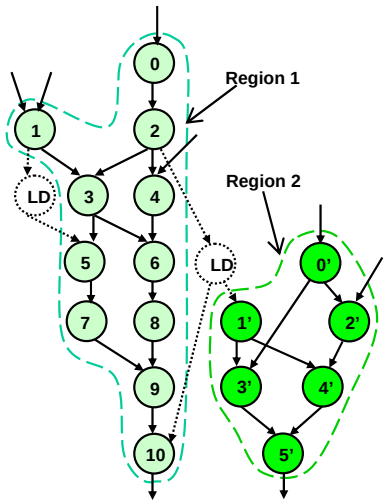

Figure 2: An example dataflow graph and its regions

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 [9], 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. [4] proposes an optimal method to select N patterns. Both ILP-based [14, 17] and heuristic-based methods [9, 17] 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 at enumerating all possible connected patterns (may be overlapped) that meet the input, output and convexity constraints. This gives the selection process an opportunity to find the globally optimal solution. As mentioned before, Atasu et al. [4] is the only work that has a similar goal. However, the scalability problem of [4] restricts its applicability to DFGs of large basic blocks. In contrast, our approach is scalable both in terms of DFG size as well as number of input/output operands.

3. TECHNIQUE

In this section, we describe our algorithm to generate all feasible patterns of a dataflow graph (DFG) under microarchitectural constraints. First, we formally define the problem and describe the previous approach for exhaustive enumeration. Next, our algorithm is presented followed by a discussion of optimizations employed in the implementation.

3.1 Problem Formulation

Given a program, custom instructions are identified using the dataflow graphs corresponding to the basic blocks. A **DataFlow Graph (DFG)** 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. G(V,E) is always a directed acyclic graph (DAG). 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, we partition the nodes of the DFG into valid and invalid nodes. A node in the DFG, whose corresponding operation can be included as part of custom instruction, is a **valid node**; otherwise, it is an **invalid node**.

Note that our algorithm is not restricted to generating patterns only within a basic block. In fact, our previous work [17] explores patterns across basic block boundaries and shows significant improvement in performance. In that case, we construct DFGs corresponding to a sequence of basic blocks ignoring the control flow instructions. Once the patterns are selected, it is left to the compiler to exploit patterns crossing basic block boundaries through formation of superblocks, hyperblocks and/or loop unrolling.

The DFG is partitioned into multiple regions. Given a DFG G(V,E), we define a **region** R(V',E') as a maximal subgraph of G s.t. (1) V' contains only valid nodes, (2) there exists an undirected path between any pair of nodes in V', and (3) there does not exist any edge between a node in V' and a valid node in (V - V'). Invalid nodes do not belong to any region. We apply custom instruction identification algorithm individually on each region. Figure 2 shows a DFG divided into two regions by a memory load operation (assuming memory load is an invalid operation).

Given a DFG, a **pattern** is a subgraph that belongs to a region in the DFG. A pattern is a possible candidate for custom instructions. A region is a trivial pattern. For convenience, we represent a pattern simply by its set of nodes P. The subgraph corresponding to a pattern can be re-constructed by simply taking the induced subgraph of P (i.e., all the edges in the DFG that connect any two nodes in P). In addition, each pattern has incoming edges and outgoing edges corresponding to the input and output operands of the pattern, respectively. The set of nodes in P connected to input operands of P are called **input nodes**, defined as IN(P). Similarly, the set of nodes in P connected to output operands of P are called **output nodes**, defined as OUT(P).

A pattern P is **convex** if there exists no path in the DFG from a node m $\in$ P to another node n $\in$ P, which contains a node p $\notin$ P. For example, $\{1, 2, 3\}$ is a convex pattern in Figure 2. Feasible patterns should be convex as non-convex patterns cannot be executed atomically. For example, in Figure 2, the pattern P1 with nodes $\{1, 3, 5\}$ is non-convex. Similarly, the pattern P2 with nodes $\{2, 3, 6\}$ is also non-convex. However, note that the non-convexity of P1 and P2 arises due to different reasons. P1 is non-convex because we *cannot include* the invalid node corresponding to the memory load operation in the pattern. On the other hand, P2 is non-convex because we *choose not to include* node 4 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 m $\in$ P to another node n $\in$ P, which contains an invalid node p $\notin$ 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 max-

imum number of allowed input and output operands for a pattern. For example, if a custom instruction is allowed to have only one output operand, then the pattern $\{1, 2, 3\}$ is infeasible.

Problem Definition. Given the DFG corresponding to a code fragment, our problem is to enumerate all feasible patterns (i.e., patterns that satisfy convexity and input, output constraints) for that code fragment. However, the goal is to avoid duplication of work, that is, minimize the number of times a pattern is enumerated. Ideally, we would like to enumerate a pattern exactly once. Note that *this problem is similar but not exactly the same as the subgraph enumeration problem within each region.* There are two important differences.

- The internal and external non-convexity as well as input and output constraints disallow many subgraphs as candidate custom instructions. Therefore, it is possible to develop efficient algorithms that can enumerate the patterns by exploiting this condition.
- As the dataflow graph is a DAG, its structure can be exploited to avoid enumerating a pattern multiple times.

3.2 Previous Algorithm

To the best of our knowledge, [4] is the only previous work that exhaustively enumerates all feasible patterns of a DFG. We will refer to this algorithm as **exhaustive** in the rest of the paper. In this section, we briefly describe the exhaustive algorithm as we use it as the baseline algorithm for comparison purposes.

The exhaustive algorithm first assigns labels $0 \ldots N-1$ to the instructions (nodes) of the DFG in reverse topologically sorted order, where N is the number of instructions in the DFG. (Note that in Figure 2, the instructions are labeled differently, i.e., in topologically sorted order.) It then searches an abstract binary tree containing N+1 levels and $2^{N+1} - 1$ 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 instruction 0, i.e., an empty pattern and a pattern containing instruction 0, respectively. The nodes at level i ($0 < i \leq N$) represent all possible patterns with instructions $0 \ldots i-1$. Basically, the search tree visits the instructions in reverse topological order and explores the patterns corresponding to presence/absence of each instruction. Clearly, the search space is exponential. However, the algorithm uses a clever strategy to prune the design space. If the pattern corresponding to a node S in the abstract search tree violates either the output constraint or the convexity constraint, then there is no need to explore the subtree of S. As the instructions in the DFG are visited in reverse topologically sorted order, all the patterns corresponding to the nodes in the subtree of S are guaranteed to violate output or convexity constraint.

Note that the original exhaustive algorithm as described in [4] enumerates both connected and disconnected patterns. Therefore it works on the entire DFG as opposed to individual regions in a DFG. For comparison purposes, we invoke the exhaustive algorithm for each region separately. 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.

3.3 Our Algorithm

Our algorithm generates all possible connected feasible patterns of a DFG. It first partitions the DFG into regions and finds patterns within each region. We will refer to our algorithm as **union** algorithm for reasons that will become clear later. Note that, in the worst case, number of feasible patterns of a DFG is exponential in terms of the number of nodes of the DFG. Therefore, the overall complexity of union algorithm is also exponential. However, experimental results indicate that, in practice, union algorithm achieves orders of magnitude speedup over exhaustive algorithm while producing the same set of patterns.

To present an overview of the algorithm, we first need to define the notion of cones. An **upward cone** of node v is a pattern P containing v such that for any other node $u \in P$, there exists a path from u to v in P. Informally, an upward cone of v is a pattern with v as the only sink node. Similarly, a **downward cone** of node v is a pattern with v as the only source node. In Figure 2, pattern $\{0, 1, 2, 3\}$ is an upward cone at node 3, while pattern $\{2, 3, 4\}$ is a downward cone at node 2.

The algorithm consists of two phases. In the first phase, we traverse the nodes of each region in topologically sorted order and calculate the set of possible convex upward cones at each node. Note that the upward cones for different nodes in a region may overlap (i.e., two cones may share nodes) but they are always unique (i.e., a cone is present only once). Similarly, we traverse the nodes of each region in reverse topologically sorted order to calculate the set of possible convex downward cones at each node. However, there exist feasible patterns that cannot be represented as upward or downward cone of a node. For example, the feasible pattern $\{1, 2, 3, 4\}$ cannot be represented as upward or downward cone of any of its constituent nodes. We enumerate these patterns in the second phase of the algorithm as union of two or more upward and downward cones. For example, the pattern $\{1, 2, 3, 4\}$ can be obtained from the union of upward cone $\{1, 2, 3\}$ of node 3 and downward cone $\{2, 4\}$ of node 2. The next two subsections present in detail the first and second phase of the algorithm, respectively.

3.3.1 First Phase: Enumeration of Upward and Downward Cones

Algorithm 1 details the generation of upward cones for a region R. As mentioned before, the algorithm traverses the nodes of a region in topologically sorted order (nodes 0 to 10 in Figure 2). We define UC_SET(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 UC_SET(v), in turn, is again represented as a set of nodes. Given a node v, let $v_1, \ldots, v_k$ be its predecessors in the region. As we are traversing the nodes in topologically sorted order, the set of upward cones of v_i ($1 \leq i \leq k$) 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 1 and 2 (in Figure 2) are $\{\{1\}\}$ and $\{\{2\}, \{0, 2\}\}$, respectively. Therefore, the set of upward cones computed for node 3 is { {3}, {1,3}, {2,3}, {0,2,3}, {1,2,3}, {0,1,2,3} }.

```
1  for all nodes v of R in topologically sorted order do
2      UC_SET(v) := {v};
3      Let v_1, ..., v_k be the predecessors of v;
4      for i = 1 to k do
5          for j= 1 to ( k choose i ) combinations do
6              Let v_{x_1}, ..., v_{x_i} be the predecessors selected in
               the j^th combination;
7              tmp := UC_SET(v_{x_1}) × ... × UC_SET(v_{x_i});
8              for each t ∈ tmp do
9                  Let t = {uc_1, ..., uc_i};
10                 uc := uc_1 ∪ ... ∪ uc_i ∪ {v};
11                 if CONVEX(uc) ∧ IN_CHECK(uc) then
12                     UC_SET(v) := UC_SET(v) ∪ {uc};
               end
           end
       end
   end
```

Algorithm 1: Enumeration of upward cones of region R.

This step may generate some upward cones (e.g., $\{2, 3, 6\}$ at node 6 in Figure 2) that do not satisfy convexity and/or input operands constraint. The algorithm eliminates such upward cones in line 11. We prove in the Appendix that such elimination is safe, i.e., Algorithm 1 still produces all upward cones satisfying input and convexity constraints. Note that the algorithm does not eliminate any upward cone that does not satisfy output constraint.

Moreover, an upward cone uc that violates input and/or convexity constraint can still be used to compose a feasible pattern p. As we generate all possible patterns as union of one or more cones, we have to make sure that the pattern p can still be generated even if we eliminate the upward cone uc. The proof is given in the appendix.

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.

3.3.2 Second Phase: Enumeration of All Feasible Patterns

As mentioned before, the second phase computes all the feasible patterns as union of one or more upward and downward cones. It traverses the nodes in a region R in reverse topological order. It maintains the following invariant: *when the traversal of a node v is completed, all the feasible patterns involving v have been enumerated.* Therefore, node v need not be considered further and can be deleted along with the constituent cones involving v. Algorithm 2 details the steps.

We first define a few terms. The **maximal upward cone** of a node v in a DAG R, MAX_UC(v, R), is the maximal upward cone of v s.t. for any upward cone Q of v in R, MAX_UC(v, R) $\supseteq$ Q. For example, $\{0, 1, 2, 3\}$ is the maximal upward cone of node 3. The upward cones of node v, UC_SET(v), can *only* be extended along the output nodes of the maximal upward cone of v, that is, OUT(MAX_UC(v, R)) (lines 4–6 in Algorithm 2). The reasoning for this is that any output node of an upward cone in UC_SET(v) must also be the output node of MAX_UC(v,R). Suppose u (u$\neq$v) is an output node of an upward cone Q in UC_SET(v) as generated by Algorithm 1. As Q is convex, all the paths from u to v must be included in Q. Clearly, u must also have at least one edge to a node x outside Q (otherwise u cannot be

```
1  begin
2      for all nodes v of R in reverse topological order do
3          PATTERNS(v) := UC_SET(v);
4          ext := OUT(MAX_UC(v, R));
5          if ext ≠ φ then
6              PATTERNS(v) := UNION(PATTERNS(v),
               ext, down);
7          remove v from R;
       end
   end

   UNION(core, ext, direction)
   begin
8      new_core := core;
9      Let ext = {v_1, ..., v_k};
10     for i = 1 to k do
11         for j = 1 to ( k choose i ) combinations of ext do
12             Let V = {v_{x_1}, ..., v_{x_i}} be the j^th combination
               of extension points;
13             P := { p | p∈core ⋀ p⊇V ⋀ p∩(ext-V)=φ};
14             if direction = down then
15                 tmp:=DC_SET(v_{x_1})× ... ×DC_SET(v_{x_i})×P;
               else
16                 tmp:=UC_SET(v_{x_1})× ... ×UC_SET(v_{x_i})×P;
               end
17             tmp_core := φ;
18             for each t ∈ tmp do
19                 Let t = {pat_1, ..., pat_{i+1}};
20                 pat := pat_1 ∪ ... ∪ pat_{i+1};
21                 if (direction = down) ⋀ CONVEX(pat) ⋀
                   OUT_CHECK(pat) then
22                     tmp_core := tmp_core ∪ {pat};
23                 if (direction = up) ⋀ CONVEX(pat) ⋀
                   IN_CHECK(pat) then
24                     tmp_core := tmp_core ∪ {pat};
               end
25             if direction = down then
26                 tmp_ext := ⋃_{v_{x_i}∈V} IN(MAX_UC(v_{x_i}, R));
               else
27                 tmp_ext:=⋃_{v_{x_i}∈V} OUT(MAX_DC(v_{x_i}, R));
               end
28             tmp_ext := REMOVE_EXT(tmp_ext ∩
               {vertices present in new_core});
29             if tmp_ext ≠ φ then
30                 new_core := new_core ∪ UNION(tmp_core,
                   tmp_ext, !direction);
               else
31                 new_core := new_core ∪ tmp_core;
               end
           end
       end
32     new_core := { p | p ∈ new_core ⋀ IN_CHECK(p) ⋀
       OUT_CHECK(p) };
33     return new_core;
   end
```

Algorithm 2: Generation of feasible patterns of region R.

an output node of Q). Now there cannot exist any path from x to v (otherwise Q will be non-convex). As MAX_UC(v, R) only contains nodes that have paths to v, x is not included in MAX_UC(v, R). Therefore, u is also an output node of MAX_UC(v, R). The terminology and reasoning for downward cones are similar. Given a pattern, the nodes along which it is extended are called the extension points.

The patterns are enumerated using the UNION function. This is a recursive function that takes in three arguments: (1) `core`, a set of generated patterns, (2) `ext`, the nodes to be used as starting points for extension, and (3) `direction` of extension, which can take in values `up` and `down`. The algorithm combines patterns in the `core` with the patterns in the cones of the extension points. Given a combination of extension points $\{v_{x_1}, \ldots, v_{x_i}\}$ (lines 10–13), we only select the patterns P in the core that satisfy the following constraints: (1) the pattern contains all the extension points $\{v_{x_1}, \ldots, v_{x_i}\}$ and (2) the pattern does not contain any other extension points (as those patterns will be considered anyway in connection with other extension points combinations and will result in redundancy).

The cones for the extension points are chosen as upward or downward cones depending on the direction (lines 14–16). Next, some infeasible patterns are eliminated at lines 21–24. We prove that these eliminations are safe in the Appendix. Finally, the patterns generated might be extended further. This is achieved by computing the extension points of the newly generated patterns (lines 25–28) and calling UNION function again (lines 29–30). The recursion ends when either no new pattern is generated or there are no new extension points. The final set of generated patterns are subjected to input, and output check.

The function REMOVE_EXT eliminates extension points that cannot produce new patterns. The conditions under which an extension can be eliminated if `direction` is `down` are explained below. That is, we want to combine downward cones of the extension points. The explanation for the reverse direction is similar.

- If the direction is downwards, the extension points are chosen as output nodes of the current core. However, if none of the outgoing edges of an extension point leads to a node in the region that does not belong to the current core, then the extension point can be eliminated.
- Given two extension points u and v, if MAX_DC(u, R) $\subseteq$ MAX_DC(v, R), then u can be eliminated from further consideration.
- If an extension point has already been considered before, it can be eliminated.

Figure 3 (a) illustrates the algorithm with an example. Assume that we are currently visiting node 8 in the DFG. The maximal upward cone of 8 contains three output nodes: 1, 2, and 8. Among them, it is only possible to extend along nodes 1 and 2. These nodes result in three possible combinations: $\{1\}$, $\{2\}$, and $\{1, 2\}$. For the first combination, we take cross product of all cones in UC_SET(8) (containing node 1 but not 2) with DC_SET(1). This step does not introduce any new extension points. However, the second combination generates a new extension point 10 and UNION is called again. Figure 3 (b) illustrates the function call graph generated for node 8.

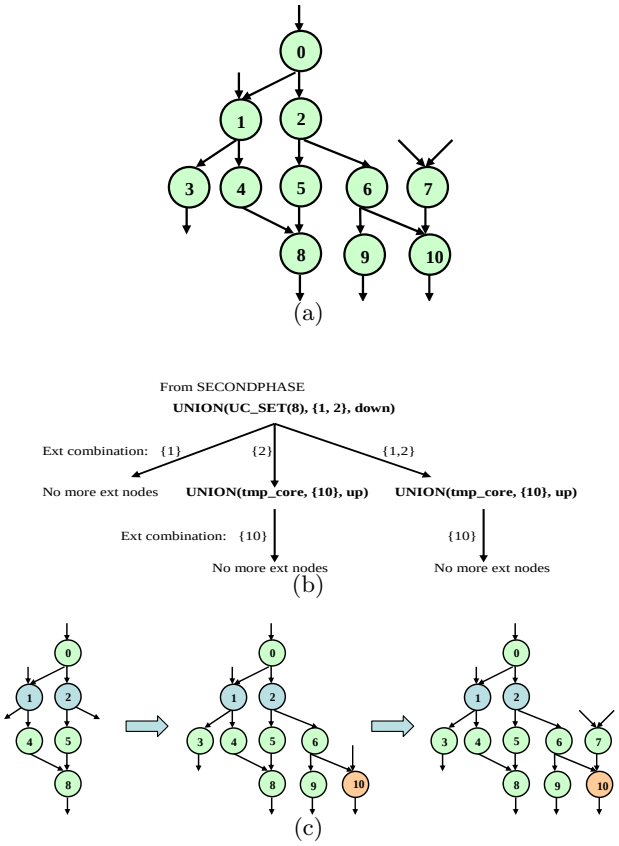

Figure 3: Illustration of pattern generation.

Figure 3 (c) illustrates the procedure with a concrete pattern $\{0, 1, 2, 4, 5, 8\}$ in the UC_SET(8) and the combination $\{1, 2\}$. We first extend the pattern by taking the downward cones at 1 and 2. The intermediate pattern produced is extended with the upward cones at node 10.

3.4 Optimizations

The basic idea of the algorithm is described in the previous section. In this section, we describe the data structures and some optimizations employed in the implementation.

Data structures. We use fixed-length bit vectors to represent each pattern. The length of the bit vectors for a particular region is equal to the number of nodes in that region. 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 20 of Algorithm 2 through bit-wise OR operation).

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 ordered by the value of their bit-vectors; every query or insertion of a pattern can be achieved within $O(\log_2(n))$ 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 convexity constraints. As discussed before, a non-convex pattern can have external or internal non-convexity. In order to check for internal non-convexity of a pattern P, we consider all immediate successors from the nodes in OUT(P). If, for one such immediate successor $u \notin P$, there exists a path to a node $v \in P$, then P fails the non-convexity constraint.

Most of the convexity violations in our problem arise due to violation of external non-convexity. We have developed an efficient method to check for this violation. The method involves 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 $\langle u, x_1, \ldots x_n, v\rangle$ in the DFG s.t. $x_1, \ldots x_n$ do not belong to R. For example in Figure 2, $\langle 1, 5\rangle$ and $\langle 2, 10\rangle$ are boundary pairs. Clearly, if $\langle u, v\rangle$ is a boundary pair, then u and v cannot coexist in any convex pattern. Moreover for any node $x \in \text{MAX\_UC}(u, R)$, it cannot coexist with any node $y \in \text{MAX\_DC}(v, R)$ in a convex pattern and vice versa. Therefore, given a node v in a region R, we compute the set of nodes in R that cannot coexist with v in a convex pattern due to external non-convexity. We call them v's **external conflicting set**. For example, in Figure 2, the external conflicting set of node 10 is $\{0, 1, 2\}$.

During the generation of patterns, we need to make sure that a pattern containing node v does not include any node from v's external conflicting set. If a pattern P is generated by combining patterns $P_1, \ldots, P_n$, then we check that the union of the external conflicting sets of $P_1, \ldots, P_n$ does not share any common node with P.

Checking for Input/Output constraints. Given a pattern P generated by combining patterns $P_1, \ldots, P_n$, $\text{IN(P)} \subseteq \text{IN}(P_1) \cup \ldots \cup \text{IN}(P_n)$ (similarly for OUT(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.

4. EXPERIMENTAL EVALUATION

In this section, we compare the performance of our union algorithm (see Section 3.3) against the exhaustive algorithm proposed in [4]. The algorithm in [4] allows multiple disjointed components within a pattern. We have constrained it to generate only connected patterns (see Section 3.2). To the best of our knowledge, the algorithm in [4] is the only known approach to enumerate all feasible patterns under input, output, and convexity constraints.

4.1 Experiment Setup

Table 1 shows the characteristics of the benchmarks used in our experiments. They are all from MiBench [13], a free, commercially representative embedded benchmark suite. In the benchmarks, `rijdael`, `blowfish`, and`sha` are encryption algorithms, `cjpeg` is an encoding algorithm and `bitcnts` is simply a computationally intensive application. We choose one frequently executed basic block from each benchmark. The regions for the basic block are also shown in Table 1. For example, the chosen basic block in `rijdael` consists of six regions with 562, 68, 4, 4, 4, 4 nodes, respectively. As can be seen from Table 1, the first four benchmarks have very large basic blocks, each containing large regions. For the first three benchmarks, the time spent in executing the chosen basic block alone is around 50% of the total execution time. This justifies the effort in selecting patterns from these large basic blocks. The basic block chosen for `sha(unroll)` contains the fully unrolled version (80 times) of the block transform function. The major computations in `cjpeg` and `bitcnts` are distributed among several basic blocks. We choose the ones with big regions as they may be the bottleneck of the pattern enumeration algorithm. We compare the efficiency of our algorithm (called **union**) against the algorithm in [4] (called **exhaustive**) for each of these basic blocks.

The benchmarks are compiled and evaluated under SimpleScalar tool set using SimpleScalar ported gcc-2.7.2.3 with -O3 optimization [8]. 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.

4.2 Experiment Results

Table 2 compares the two algorithms for all the benchmarks under different input/output constraints. The column `No. of Valid Patterns` shows the number of feasible patterns generated for each basic block. Two algorithms produce the same sets of feasible patterns for each benchmark. Compared to the size of the basic blocks, the number of feasible patterns is quite small. Therefore, it is possible for the later custom instruction selection procedure to work on the entire set of feasible patterns and produce an optimal subset as custom instructions.

The `Search Space` column shows the number of patterns subjected to different constraint checks by the two algorithms. In general, as union algorithm produces patterns by extending existing ones with neighbors, it is far more effective in pruning the infeasible patterns. The results indicate that our algorithm scales well with increasing DFG size and increasing number of input, output constraints. Note that for a given benchmark and output constraint, the search spaces of exhaustive algorithm are the same irrespective of the number of input. This is because the exhaustive algorithm does not prune the search space based on the number of input constraint. However, the number of input constraint is considered as one of our pruning strategy; so the efficiency of our algorithm is more evident especially under small value of input constraint.

The last but one column provides the speedup in execution time of our enumeration algorithm compared to the exhaustive approach. The speedup varies from modest to extremely high. The last column presents the actual execution time of our algorithm. For `rijdael`, union algorithm reduces the computation time from thousands of seconds to less than one second. For `blowfish`, even though it contains large regions, the computation within a region is clustered such that there exists non-convexity among nodes from different clusters. This feature is exploited by both the algorithms to prune efficiently. For `sha` and `cjpeg`, our algorithm is very efficient when the number of input is small. However, recall that our algorithm may produce the same pattern multiple times. The amount of redundancy increases with increasing pattern size and number of input constraints. Therefore, the speedup of our algorithm is less evident under larger value of input constraint. As `bitcnts` and `sha (loop)` contain small DFGs, both algorithms perform very efficiently.

Benchmark	Category	BB Size	Size of Regions	% of Total Execution Time
rijdael	Security	894	{562, 68, 4, 4, 4, 4}	61.19%
blowfish	Security	334	{133, 120, 2}	46.10%
sha(unroll)	Security	1468	{1367}	53.65%
cjpeg	Consumer	154	{92, 40}	6.97%
sha(loop)	Security	22	{11, 3, 2}	16.53%
bitcnts	Automotive	36	{27}	6.20%

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

Benchmark	No. of Input	No. of Output	Search Space *Exhaustive*	Search Space *Union*	No. of Valid Patterns	$\frac{ExecTime_{exhaust}}{ExecTime_{union}}$	Absolute Time *Union* (seconds)
Rijdael	3	1	336607	1924	438	36.2	0.014
	3	2	39691113	8325	620	1924.2	0.030
	3	3	2190185753	26472	620	46560.5	0.066
	4	1	336607	2424	676	29.7	0.017
	4	2	39691113	16267	1178	1267	0.045
	4	3	2190185753	66442	1496	19931.1	0.154
	5	1	336607	2884	716	25.2	0.020
	5	2	39691113	20879	1680	1025.3	0.056
	5	3	2190185753	73608	2904	15884.8	0.192
Blowfish	3	1	350120	823	177	4.1	0.0016
	3	2	339058	8938	252	11.1	0.0056
	3	3	1718557	9299	252	35.8	0.0084
	4	1	350120	1163	279	3.0	0.0022
	4	2	339058	9450	554	9.1	0.0066
	4	3	1718557	10714	704	26.0	0.0116
	5	1	350120	1527	307	2.4	0.0027
	5	2	339058	10098	894	8.0	0.0077
	5	3	1718557	12864	1594	18.3	0.0164
Sha(unroll)	3	1	12983317	12256	1222	500.9	0.06
	3	2	714743500	686033	2270	766.6	2.10
	3	3	16326817016	7220211	2987	1513.2	23.37
	4	1	12983317	36059	2343	206.1	0.14
	4	2	714743500	5054356	5019	138.3	11.68
	4	3	16326817016	983219339	7951	11.3	3122.47
	5	1	12983317	91072	3997	83.1	0.36
	5	2	714743500	8120906	8717	51.6	31.30
	5	3	16326817016	1973510852	16180	2.7	12917.43
Cjpeg	3	1	22659	995	159	5.3	0.001
	3	2	3369446	129114	233	5.2	0.133
	3	3	433196214	130978	233	502.5	0.197
	4	1	22659	2092	298	2.8	0.002
	4	2	3369446	137989	458	4.8	0.136
	4	3	433196214	145613	578	479.8	0.207
	5	1	22659	4347	379	1.3	0.003
	5	2	3369446	552169	726	2.2	3.135
	5	3	433196214	15292783	1003	10.3	9.640
Bitcnts	3	1	5463	137	23	17.3	1.3E-4
	3	2	36237	171	31	57.4	2.5E-4
	3	3	57021	172	31	89.6	2.5E-4
	4	1	5463	290	44	9.0	2.2E-4
	4	2	36237	419	80	34.0	4.2E-4
	4	3	57021	422	81	49.7	4.4E-4
	5	1	5463	637	65	4.3	4.7E-4
	5	2	36237	1184	169	16.2	9.1E-4
	5	3	57021	1191	172	24.3	9.4E-4
Sha(loop)	3	1	325	55	14	2.0	6.2E-5
	3	2	374	55	15	1.7	8.5E-5
	3	3	386	55	15	1.8	8.4E-5
	4	1	325	79	27	1.6	7.7E-5
	4	2	374	79	28	1.4	1.2E-4
	4	3	386	79	28	1.6	1.1E-4
	5	1	325	95	44	1.4	9.8E-5
	5	2	374	95	45	1.1	1.4E-4
	5	3	386	95	45	1.2	1.4E-4

Table 2: Comparison of Enumeration Algorithms

5. CONCLUSION

Enumerating all feasible candidate patterns under various architectural constraints is a key step is selecting the optimal set of custom instructions. In this paper, we have introduced an efficient algorithm to solve this problem and discussed its implementation issues. Compared with a recently proposed approach targeting the same problem, 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. We are currently integrating our algorithm in the compilation framework of a real extensible processor – Altera NIOS.

6. ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their helpful comments. This work was partially supported by NUS research grants R252-000-088-112, R252-000-171-112 and A*STAR Project 022/106/0043.

7. 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/nio-index.html`.

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

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

[5] D. Avis and K. Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65, 1996.

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

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

[8] 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`.

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

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

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

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

[13] 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/`.

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

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

[16] 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.

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

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

Appendix

LEMMA 1. *Given a region R, Algorithm 1 generates all the upward cones satisfying convexity and input operands constraints.*

Proof: Let the nodes of R in topologically sorted order be $v_0, \ldots v_{n-1}$. The proof is by induction on n. For the base case, node v_0 has only one upward cone $\{v_0\}$ which is generated in line 2 of Algorithm 1. We assume that the constraints on number of input and output operands are 2 and 1, respectively, so that cone $\{v_0\}$ satisfies all the constraints.

For the induction step, let us consider an upward cone C at node v_x $(1 \leq x \leq n-1)$ satisfying convexity and input operands constraints. We prove that C can be generated from its predecessors' UC_SETs. Let $\{v_{x1}, \ldots, v_{xk}\}$ be the predecessor nodes of v_x and let $P_i = \text{MAX\_UC}(v_{xi}, C)$. Clearly,

$$C = \bigcup_{i=1}^{k} P_i \cup \{v\}$$

Therefore, we need to show that

$$P_i \in \text{UC\_SET}(v_{xi}) \quad \forall i = 1, \ldots, k$$

The above is true iff P_i is convex and satisfies input constraints. As $\text{IN}(P_i) \subseteq \text{IN}(C)$ and C satisfies input constraints, P_i also satisfies input constraints. We prove by contradiction that P_i is convex. Let us assume that P_i is non-convex. Then, there exists at least a pair of nodes $m, n \in P_i$ s.t. there exists a path from m to n that contains a node $y \notin P_i$. As P_i is the maximal upward cone of node v_{xi} in C, if $y \notin P_i$, then $y \notin C$. Therefore, C is also non-convex, which is a contradiction. □

LEMMA 2. *Given a region R, Algorithm 2 generates all the feasible patterns in R.*

Proof: We prove that given any feasible pattern P in R, P can be generated by Algorithm 2. Let $P = \{v_1, \ldots, v_n\}$. We also assume that the nodes of P have been visited in the order $v_1, \ldots, v_n$ in Algorithm 2 (reverse topological order). Clearly, P has been generated by v_1 as a node is removed from R after it has been visited.

The algorithm starts with the UC_SET(v_1). We will show that (P_0 = MAX_UC(v_1, P)) $\in$ UC_SET(v_1) and P can be generated by starting with the cone P_0. Now, UC_SET(v_1) contains P_0 iff P_0 is convex and satisfies input constraints. Note that IN(P_0) $\subseteq$ IN(P) as P_0 is a maximal upward cone in P. Therefore, if P satisfies input constraints, then P_0 should satisfy input constraints as well. We prove by contradiction that P_0 cannot contain non-convexity. Let us assume that P_0 is non-convex. Then, there exists at least a pair of nodes m, n $\in P_0$ s.t. there is path from m to n that contains a node y $\notin P_0$. As P_0 is a maximal upward cone in P, if y $\notin P_0$, then y $\notin$ P. Therefore, P is also non-convex, which is a contradiction.

Now, we prove that P can be generated from P_0 by recursively combining the current pattern with the maximal upward/downward cones of its extension points. Let the intermediate patterns generated be $P_0, \ldots, P_M$ s.t. $P_M = P$. The recursion depth M is bounded by the number of vertices n in P. Clearly, the proof is trivial if no patterns or cones were eliminated through convexity, input, and output checks. We prove that it is safe to eliminate patterns or cones by induction on M. We have already proved the base case M = 0.

For the induction step, let us assume (without loss of generalization) that $P_{i+1}, i = 1, \ldots, M-1$ is generated by combining P_i with the maximal downward cones of P_i's extension points. Let $\{e_1, \ldots, e_k\}$ be the set of extension points of P_i. The algorithm combines DC_SET(e_1), ..., DC_SET(e_n) with P_i. Using similar reasoning as in the previous paragraph, we can prove that MAX_DC(e_j, P) $\in$ DC_SET(e_j) for all $j = 1, \ldots, k$. Therefore, the algorithm can generate the pattern

$$P_{i+1} = P_i \bigcup \text{MAX\_DC}(e_1, P) \bigcup \ldots \bigcup \text{MAX\_DC}(e_k, P)$$

. We need to prove that P_{i+1} is convex and satisfies output constraints as it will be eliminated otherwise.

By definition, all the nodes in OUT(P_{i+1}) belong to OUT(P) as we have included the maximal downward cones of P_i's extension points (OUT(P_i)) in P_{i+1}. Therefore, as P satisfies output constraints P_{i+1} should satisfy output constraints. We prove by contradiction that P_{i+1} is convex. Let us assume that P_{i+1} is non-convex. Then, there exists at least a pair of nodes m, n $\in P_{i+1}$ s.t. there is path from m to n that contains a node y $\notin P_{i+1}$. By definition, the predecessor of y in the path is an extension point of P_{i+1} and its maximal downward cone in P is included in P_{i+1}. Therefore, if y $\notin P_{i+1}$, then y $\notin$ P and P is non-convex. $\square$