# Polynomial-Time Subgraph Enumeration for Automated Instruction Set Extension

Paolo Bonzini and Laura Pozzi
Faculty of Informatics
University of Lugano (USI)
Switzerland
Email: paolo.bonzini@lu.unisi.ch, laura.pozzi@unisi.ch

University of Lugano
Faculty of Informatics
Technical Report No. 2006/07
December 2006

## Abstract

*This paper proposes a novel algorithm that, given a data-flow graph and an input/output constraint, enumerates all convex subgraphs under the given constraint in polynomial time with respect to the size of the graph. These subgraphs have been shown to represent efficient Instruction Set Extensions for customizable processors. The search space for this problem is inherently polynomial but, to our knowledge, this is the first paper to prove this and to present a practical algorithm for this problem with polynomial complexity. Our algorithm is based on properties of convex subgraphs that link them to the concept of multiple-vertex dominators. We discuss several pruning techniques that, without sacrificing the optimality of the algorithm, make it practical for data-flow graphs of a thousands nodes or more.*

## 1. Introduction

A common practice in designing system-on-chip processors is to define a basic customizable processor and to extend it with units specialized for particular applications. Since typical embedded systems already include several Application-Specific Integrated Circuits (ASICs), it is conceivable to develop differently customized versions of the processor for each product. Such processor extensions can increase performance in specific domains, without the cost of advanced RISC processors and the complexity of entirely customized instruction sets. Many manufacturers are proposing customizable processors, such as Tensilica Xtensa and ARC ARCtangent. Instead of ASICs, other manufacturers use reconfigurable fabrics as accelerators. On some Xilinx systems, for example, an FPGA and a hard-wired PowerPC processor coexist on the same die and share the same bus.

Independently of the technology adopted, it is important to be able to *automatically generate the best performing instruction set extensions* for an application. An exact solution to this problem is not feasible if we want to generate more than one instruction [15]. On the other hand, several algorithms exist to find the single best performing instruction set extension in a graph. Exact algorithms for this problem may be based on Integer Linear Programming [3] or pruned exploration of the subgraphs [15, 4].

An important subproblem in optimal ISE identification is enumeration of the valid subgraphs. While algorithms like [15, 4] focus on an exponential search space (each node of a basic block can be either *in or out* of a subgraph), we focus attention on the subgraphs input and output nodes. This way, as it can be seen in detail later, we identify a problem space that is no longer exponential in the size of the graph.

Based on this observation, we propose a new practical algorithm to enumerate all subgraphs under a given microarchitectural constraint, whose complexity is polynomial in the size of the graph. The algorithm is based on the relationship between convex subgraphs and multiple-vertex dominators. Even though the complexity is still rather high—up to $O\left(n^7\right)$ for commonly used input/output constraints—several pruning techniques allow it to process basic blocks of around a thousand nodes in about a minute.

The next section discusses related work in the domain of instruction-set selection and overviews previous uses of

multiple-vertex dominators. Section 3 provides a theoretical framework for the problem, and section 4 discusses the relationship between subgraph enumeration and multiple-vertex dominators. Section 5 introduces the algorithm, together with some techniques to prune the search space. Finally, section 6 presents our experimental setup and discusses the results.

## 2. Related work

Past literature on custom instruction identification can be roughly divided in two groups. Some authors aimed at optimal instruction identification, developing algorithms with exponential worst-case complexity. Other works reduced the complexity, at the expense of optimality or generality.

Atasu et al. developed a widely known optimal algorithm [4], which Pozzi et al. further refined with additional pruning techniques [15]. Both algorithms are based on subgraph enumeration: they exhaustively explore the search space algorithm, pruning it through constraint propagation. Neither poses limits on the latency or connectedness of the instructions. The latter algorithm is reasonably efficient even for large basic blocks, but its performance quickly deteriorates if the custom instructions can have multiple outputs. An Integer Linear Programming formulation of the same problem was presented in [3]: in this case, the enumeration of subgraphs is implicit in the formulation's constraints, and the worst-case complexity is still exponential.

Other authors simplify the problem formulation and then find exact solutions for it. For example, Choi et al. [9] limit the latency (depth) of subgraphs, and Yu and Mitra [17] only focus on identifying connected custom instructions. Our algorithm can be adapted to run faster under this kind of limitation.

Other approaches include greedy techniques such as those in Baleani et al. [5], Clark et al. [11] and Biswas et al. [6]. While fast, these algorithms have limited effectiveness [10]. More recent work seems to focus on optimal algorithms, possibly using subgraph enumeration as a building block.

The algorithm we present is based on multiple-vertex dominators. Also known as generalized dominators, they were introduced by Gupta in [13]. Single-vertex dominators are a widely studied problem [1], but they are rare in circuit graphs or data-flow graphs: multiple-vertex dominators are more common and can be useful to explore or simplify such graphs. Gupta presents an algorithm to enumerate all multiple-vertex dominators, but only with exponential worst-case complexity. A smaller bound for this problem was never published, and it is an open problem to find an efficient way of representing the set of all possible dominators.

Dubrova et al. [12], however, do present an algorithm to enumerate $k$-vertex dominators in polynomial time $O\left(n^k\right)$. As we show in section 5, we can employ this algorithm successfully to find optimal custom instructions. The authors report that the algorithm is very slow for $k > 2$; however, we developed pruning techniques for subgraph enumeration, that make this algorithm practical also for somewhat higher values of $k$, and for graphs with a thousand nodes or more.

The algorithm in [12] looks for single-vertex dominators in many different reduced graphs. For this, efficient algorithms exit. Aho and Ullman's work [1] was successively improved Purdom and Moore [16], and by Lengauer and Tarjan. [14] presents two very effective algorithms with complexity $O\left(e \log e\right)$ and $O\left(e\, \alpha(e,n)\right)$, where $e$ is the number of edges, $n$ is the number of nodes, and $\alpha$ is the slowly-growing functional inverse of the Ackermann function. The asymptotic complexity was further reduced to linear by Alstrup et al. [2]; these improvements however do not imply reduced run-times, because the algorithms are complex and $\alpha$ can be considered constant in practice.

## 3. Problem statement

The data flow of each basic block is represented by a graph $G(V,E)$. The graph $G$ may have an arbitrary number of root vertices $I_{ext}$, that is vertices that have no predecessors. These vertices represent input variables of the basic block. The graph may also have a set of vertices $O_{ext}$ that is a superset of those vertices that have no successors. Figure 1(a) shows an example data-flow graph with 3 roots (vertices A, B, C) and 2 $O_{ext}$ vertices (X and Y).

A *cut* is defined on a rooted, direct, acyclic graph. $G$ is transformed into a rooted graph, by augmenting it with a single vertex that is a predecessor of every vertex in $I_{ext}$. We also create an additional vertex (the *sink*) and connect $O_{ext}$ to the sink. This way, the reverse graph of $G$ also a rooted graph, which is useful when computing postdominators.

The definition of cut, and in particular of convex cut, are as follows.

**Definition 1 (Cut):** A cut $S$ is a subgraph of a graph $G$. We call *inputs of* $S$ the set $\mathrm{I}(S)$ of predecessor vertices of those edges which enter the cut $S$ from the rest of the graph $G$, that is $\mathrm{I}(S) = \bigcup_{v \in S} \mathrm{pred}(v) \backslash S$. Similarly, we call *outputs of* $S$ the set $\mathrm{O}(S)$ of vertices which are part of $S$, but have at least one successor $v \notin S$.

**Definition 2 (Convex cut):** A cut $S$ is convex if there is no path from a vertex $u \in S$ to another vertex $v \in S$ which contains a vertex $w \notin S$.

The shaded areas in figure 1(b)(c)(d) are all examples of a convex cut. Nodes with a double border are outputs[1] and grey nodes are inputs.

[1] The terms *vertex* and *node* will be used interchangeably.

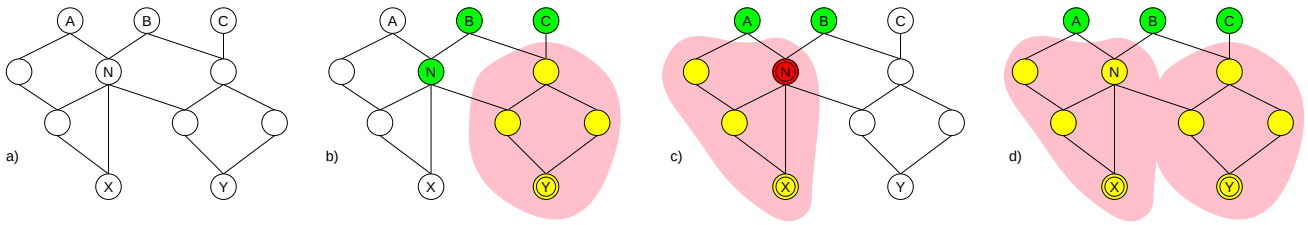


**Figure 1. a) A data-flow graph (all edges are towards the bottom of the figure); b) a valid 1-output convex cut. The three green nodes N, B, C are inputs to the output node Y. They are also a multiple-vertex dominator of the output. c) an invalid 1-output convex cut. The two input nodes are a multiple-vertex dominator of the output node X, but there is an additional output (the red node). d) a valid 2-output convex cut. The three input nodes A, B, C dominate Y. The other output X is only dominated by A and B.**

The microarchitecture may pose several additional constraints on the cuts that can be considered valid. First of all, the values $N_{\text{in}}$ and $N_{\text{out}}$ indicate the maximum number of read and write ports in the register file, respectively, which a custom instruction can use. Secondly, some nodes of $G$ may be *forbidden*, that is, they may not be included in a cut[2].

Some forbidden nodes will be marked as such by the user, and represent operations that are not allowed in a special instruction—for example, loads and stores if the custom functional unit cannot have any memory port. In addition to these nodes, the algorithm will consider other nodes to be forbidden. $I_{ext}$ nodes are implicitly forbidden, because their values are calculated outside the basic block. Likewise, the newly introduced source and sink will be considered forbidden because they do not map to an actual computation in the program. We will denote forbidden nodes with $F$.

User-specified forbidden nodes may have no predecessor: without losing generality, all the nodes $v \in F$ can be connected to the same "artificial" source as the external inputs $I_{ext}$.

Thus, given a graph $G$, the posed problem is to find all convex cuts $S \subseteq G$ under the constraints that $|\mathrm{I}(S)| \leq N_{\text{in}}$, $|\mathrm{O}(S)| \leq N_{\text{out}}$, and $S \cap F = \emptyset$.

In the remainder of this paper, we add another condition for the validity of the convex cut. For each input $w \in \mathrm{I}(S)$, there is a vertex $v \in S$, such that at least one path from the root of $G$ to vertex $v$ contains $w$ but not any other input of $S$.

This condition excludes from consideration a few valid cuts, namely those where an input $w$ only has other inputs as predecessors (in this case, the inputs to $w$ must be predecessors of $w$ for the cut to be convex). These cuts violate the condition we just added because all the paths from the root of $G$ to $w$ contain other inputs. Given the cut in figure 1(d), we would have such a cut if the node marked as N was a fourth input. All paths from the root to N would pass through the inputs A and B.

Note that *all* the predecessors of $w$ need to be inputs, or there would be a path from the root of $G$ to $v$ that only contains the input $w$. Then, all the inputs but $w$ will be inputs to the valid cut $S \cup \{w\}$. This cut—like the one in figure 1(d)—will be found by our algorithm, and can be used to find the cuts that were lost by the addition of the technical condition.

## 4. Properties of convex cuts

We can define more characteristics and prove some properties of convex cuts.

**Definition 3 (Inputs to a vertex):** In a convex cut $S$, the *inputs to* a vertex $v$ are defined to be the set of vertices $I_v(S) \subseteq \mathrm{I}(S)$ such that:

1. every path from the root of $G$ to $v$ contains at least one vertex $i \in I_v(S)$;
2. and, for every vertex $i \in I_v(S)$, at least one path from the root to $v$ contains $i$.

The following related definition will also be useful:

**Definition 4 (Connected convex cut):** A convex cut $S$ is connected if it has a single output, or if for any two outputs $o_1$ and $o_2$ there exists a vertex $i \in \mathrm{I}(S)$ which is an input to both $o_1$ and $o_2$.

For example, A and B are the inputs to X in figure 1(d).

There is an important link between the *generalized dominators* of an output $o$, and the inputs to $o$. Generalized dominators are defined as follows on a rooted graph $G$.

**Definition 5 (Generalized dominator):** A set of vertices $V$ in a rooted graph $G$ *dominates* a vertex $v$ iff it meets the following two conditions:

[2]Note that forbidden nodes may still be chosen as inputs to a cut.

1. all paths from the root of $G$ to vertex $v$ contain at least one vertex $w \in V$;

2. for each vertex $w \in V$, there is at least one path from the root of $G$ to vertex $v$, which contains $w$ but not any other vertex in $V$.

We can then prove the following theorem and provide the link between inputs and generalized dominators:

*Theorem 1*: If $S \subseteq G$ identifies a convex cut, then for every output $o$ of $S$ the set of vertices $I_o(S)$ that are inputs to $o$ is a generalized dominator of $o$ in $G$.

*Proof.* From definition 3, every path from the root to $o$ contains at least one of the vertices in $I_o(S)$. Condition 1 is then verified in the definition of generalized dominators.

With the more restrictive definition of convex cut that we presented above, they also satisfy condition 2. Because of that restriction, for each input $i \in I_o(S)$, there is a vertex $v \in S$, such that at least one path from the root of $G$ to vertex $v$ contains $i$ but not any other input of $S$.

In particular, we can find one such path for any $v$ that is a successor of $i$ and not an input. If we pick a vertex that is also contained in a path between $i$ and $o$ (there will always be such a path, because $i$ is an input to $o$), then the path can be extended to stop at $o$ instead of $v$. This proves that condition 2 is also verified. □

For example, given the convex cut in figure 1(b), this theorem proves that the set of inputs to the cut is a generalized dominator of the output. The theorem provides a useful starting point to derive the nodes in a convex cut from its input and output vertices, and to prove that inputs and outputs uniquely identify the convex cut. However, this proof needs another definition:

**Definition 6 (Vertices between $V$ and $w$):** $B(V, w)$, the set of vertices between a set $V$ and a vertex $w$, is the set of vertices contained by at least one path between a vertex $v \in V$ and $w$ (if there is such a path). The starting vertex of the path is *not* includedin $B(V, w)$, while the final vertex $w$ is.

For example, in figure 1(b), the shaded area represents the vertices between its inputs (nodes B, C and N), and the output node Y. Note that $B(V, w)$ can be computed easily; the complexity is linear in the size of the set $B$ itself, and hence $O(n)$.

*Theorem 2*: Any convex cut is uniquely identified by its sets of input and output vertices, respectively $\mathrm{I}(S)$ and $\mathrm{O}(S)$. In other words, two convex cuts of the same graph are equal iff they have the same inputs and outputs.

*Proof.* *If two convex cuts $S$ and $T$ are equal, they have the same sets of inputs and outputs*. This is true because the inputs and outputs of a convex cut are functions of the vertices in the cut.

*If two convex cuts have the same sets of inputs and outputs, they are equal*. To prove this part, we consider the cut $S' = \bigcup_{v \in \mathrm{O}(S)} B(\mathrm{I}(S), v) \backslash \mathrm{I}(S)$, and prove that $S = S'$.

If a vertex $v$ was in $S'$ but not in $S$, then there would be a path from an input to an output going through $v \notin S$, violating the definition of convexity. This because $S'$ is defined to include all the vertices, along every possible path from an input to an output.

If a vertex $v$ was in $S$ but not in $S'$, this means that no path from the inputs to the outputs contains $v$. However, every path from the root to $v$ must contain at least one input in $I_v(S)$. Then, there is always a path from this input to $v$—that is, $v \in B(\mathrm{I}(S), v)$. If $v$ was an output, $B(\mathrm{I}(S), v)$ would be included in $S'$, hence $v \in S'$ and we have a contradiction. If $v$ was not an output, there must be a path from $v$ to an output $o$, otherwise at least one successor of $v$ would not belong in $S$ and $v$ would be an output. Therefore $v \in B(\mathrm{I}(S), o)$ and we also have a contradiction. □

We proved that a convex cut is uniquely identified by the sets of inputs and outputs. However, the reverse is not true: given two sets of vertices $I$ and $O$, they identify a convex cut only under rather strict conditions. The following weaker theorem, nevertheless, forms the basis, and at the same time the correctness proof, for our algorithm.

*Theorem 3*: Given two sets of vertices $I$ and $O$, if for every vertex $o_j \in O$, there is a set of vertices $I_j \subseteq I$ such that $I_j$ dominates $o$, then $S = \bigcup_{o_j \in O} B(I_j, o_j) \backslash I_j$ is a convex cut with $\mathrm{I}(S) \subseteq I$.

*Proof.* The convexity of $S$ derives directly from the definition of $B(I_j, o_j)$: all the vertices on a path between an input and an output are included in $S$, therefore no such path can cross a vertex that is not part of $S$.

If an $i \in \mathrm{I}(S)$ was not in any $I_j$, we would have a path from the root to every $o_j$ that passed through a vertex not in $I_j$. This contradicts the hypothesis that $I_j$ dominates $o_j$. □

This theorem provides a way to compute the vertices in a cut in $O(n)$ time (since $B(I_j, o_j)$ can also be computed in linear time). It also guarantees that the cut will not have any more inputs. On the other hand, it does not guarantee that the cut will have precisely the requested inputs and outputs—see figure 1(c) for an example where an additional output appears between B and X. As we will see later, however, this can actually be exploited to speed up our algorithm.

## 5. Algorithm

Since the convex cuts in $G$ are uniquely identified by its input and output vertices, it is possible to enumerate them by coupling every possible set of outputs with all the possible sets of inputs. If, as in the posed problem, we put a constraint on the number of inputs and outputs, the number of valid convex cuts is clearly polynomial; more precisely, it is $O\left(n^{N_{\text{in}}+N_{\text{out}}}\right)$.

Section 5.1 describes a basic solution to the problem, which would be feasible only for small basic blocks. By switching to an incremental mode of operation, the choice of inputs and outputs can be pruned at each step: section 5.2 details the refinements in the algorithm, and section 5.3 presents the corresponding pruning techniques. Our choice of data structures is explained in section 5.4.

### 5.1. Basic solution

An implementation of the algorithm is presented in figure 2. To find all sets of vertices that can be inputs to an output vertex $o$, we try all the (possibly multiple-vertex) dominators of the output. We start by picking an output node, and explore all its dominators. After finding one, we may add another output node and recursively explore the new output's dominators.

In principle, any $n$-uple of vertices could be taken as an output. In practice, not all of them are useful: an output vertex $o_2$ is not admissible if another output vertex $o_1$ post-dominates it. Note that a vertex $v \in O_{ext}$ will not be post-dominated by any vertex but the artificial sink, because it is connected by an edge to the sink.

Note how, in agreement with theorem 3, we need an additional check that the cut really has no outputs besides $O$. If a cut fails this test, as in figure 1(c), it may still be extended with a new output and become valid: for example, node N itself may be added, or node Y may be added yielding the cut in figure 1(d).

To analyze the complexity of the algorithm we take into account each step. Setting up the algorithm includes the computation of single- and multiple-vertex dominators (with maximum cardinality $N_{\text{in}}$) of every node. Let $O\left(\tau(n)\right)$ be the complexity of computing single-vertex dominators on a graph with $n$ vertices. In theory, this can be as low as $O\left(n\right)$ but, as mentioned in section 2, practically used algorithms have a slightly higher complexity. Computing single-vertex dominators (and postdominators) requires a time $O\left(\tau(n)\right)$, while computing the multiple-vertex dominators requires a time $O\left(n^{N_{\text{in}}-1}\tau(n)\right)$.

Given the inputs and outputs of a cut, the nodes that are part of the cut can be enumerated in $O\left(n\right)$ time. Checking that the cut has no extraneous outputs has the same cost. Function POLY-ENUM has a complexity of $O\left(n^{N_{\text{in}}+N_{\text{out}}+1}\right)$. As this dominates all the setup phases, it is also the complexity of subgraph enumeration for the algorithm in figure 2.

```
DO-ENUM(I, O, S, N_out)
    for each admissible output o do
        O' = O ∪ {o}
        for each dominator D of o with |D ∪ I| ≤ N_in do
            S' = S ∪ B(D, o)
            I' = I ∪ D
            if O(S') = O' ∧ S' ∩ F = ∅ then
                S' is a valid cut
            if N_out > 1 then
                DO-ENUM(I', O', S', N_out − 1)

POLY-ENUM()
    DO-ENUM(∅, ∅, ∅, N_out)
```

**Figure 2. A polynomial-time algorithm for subgraph enumeration**

### 5.2. Incremental operation

The pseudo-code in figure 2 is agnostic of the algorithm used to compute multiple-vertex dominators. Since we know of only one such algorithm with polynomial complexity, we can tailor our implementation to it in order to improve its speed.

The multiple-vertex dominator algorithm from [12] picks every possible *seed* set $\{v_1, \ldots, v_{n-1}\}$. Then, it removes the vertices in the seed set from the graph, together with any other node they dominate. Then, if a node $u$ dominates a node $w$ in this reduced graph, $\{v_1, \ldots, v_{n-1}, u\}$ is a multiple-vertex dominator of $w$ in the original graph.

The exploration of the seed sets (which covers the inner **for each** loop of figure 2) can be done recursively, just like for the output nodes. Every recursive call pushes a vertex on the seed set, calls the Lengauer–Tarjan algorithm [14] on the reduced graph, and then pops the vertex. There will be up to $N_{\text{in}} - 1$ recursive calls, giving the algorithm in figure 3.

Note how $S = \bigcup_{o \in O} B(D, o)$ is built incrementally. Pushing an input or an output adds nodes to $S$, such that on every recursive call $S$ can only grow. In particular, the newly added nodes for an input $v$ are $S_{new} = B(\{v\}, o) \backslash S$. For an output $o$, they are $S_{new} = B(I, o) \backslash S$. As a further optimization, our implementation maintains a single copy of $S$. We keep track of when each node was added to $S$, and remove them before leaving the invocation that added them.

Unlike the naïve algorithm, this algorithm does everything in a single pass without any setup phase. The complexity is the same as for the previously presented one, that is $O\left(n^{N_{\text{in}}+N_{\text{out}}+1}\right)$. In the **for each** loops, the cost of invoking PICK-INPUTS dominates the linear-time work to update $S$; when CHECK-CUT is called, instead, updating $S$ is

```
CHECK-CUT(I, O, S, N_in, N_out)
    if O(S) = O ∧ S ∩ F = ∅ then
        S is a valid cut
    if N_out > 0 then
        PICK-OUTPUT(I, O, S, N_in, N_out)

PICK-INPUTS(I, o, O, S, N_in, N_out)
    ▷ the next line invokes Dubrova et al.'s algorithm
    for each node w such that I ∪ {w} dominates O do
        I' = I ∪ {w}
        S' = S ∪ B({w}, o)
        CHECK-CUT(I', O, S', N_in − 1, N_out)
    if N_in > 1 then
        ▷ add a node to the seed set
        for each ancestor i of o do
            I' = I ∪ {i}
            S' = S ∪ B({i}, o)
            PICK-INPUTS(I', o, O, S', N_in − 1, N_out)

PICK-OUTPUT(I, O, S, N_in, N_out)
    for each admissible output o do
        O' = O ∪ {o}
        S' = S ∪ B(I, o)
        if I dominates o then
            CHECK-CUT(I, O', S', N_in, N_out − 1)
        elseif N_in > 0 then
            PICK-INPUTS(I, o, O', S', N_in, N_out − 1)

POLY-ENUM-INCR()
    PICK-OUTPUT(∅, ∅, ∅, N_in, N_out)
```

**Figure 3. Building $S$ incrementally**

covered by the additional $+1$ in the exponent.

### 5.3. Pruning techniques

The improvements described in the previous section only affect a constant factor in the algorithm's complexity. Therefore, starting from the refined version of the algorithm, we implemented several techniques to improve the algorithm's performance. These do not reduce the asymptotic complexity either. However, they do reduce the $n$ in the complexity. The decrease can be quite dramatic, so that the algorithm is practical even for graphs with 1,000 or more nodes.

**Output–output pruning.** To prune the set of outputs, we can rely on a "weak" point of theorem 3. The theorem does not guarantee that no internal nodes are outputs; we can only prove that all the chosen outputs are part of the cut. Still, this can be exploited to greatly reduce the search space.

We call an output *internal* if it is not chosen explicitly, but just "happens to be" part of the cut. First, we save a candidate whenever $|\mathrm{O}(S)| \leq N_{\mathrm{out}}$, even if it has internal outputs. Then, given a chosen output node $o$, and as long as the total number of outputs is less than $N_{\mathrm{out}}$, the algorithm will accept any node from $o$'s ancestors as a valid output: for example, the cut in figure 1(c) will now be accepted as a 2-output subgraph. Then, a node $o$ will not need to be examined in POLY-ENUM-INCR if it is the ancestor of another selected output.

Optimization of internal outputs will be subject to refinements in the rest of this section.

**Connectedness.** The algorithm can be set up to only search for connected cuts. To do so, any output after the first must be reachable from at least one input.

When allowing internal outputs, these may be removed by adding another chosen output. The search must then proceed even if a cut has more than the maximum number of allowed outputs, the search must proceed even though the cut is not yet valid. However, in this case we will accept only a connected output—one that is reachable from at least one already selected input. Otherwise, there would be no path from any input to the new output, and hence no path from the excess internal outputs to the new output. No successor of the internal outputs could be included, and all the internal outputs would remain, leaving the cut invalid.

For example, figure 1(c) depicts a cut with an internal output (the dark node). Adding an unconnected node, such as node M in the same figure, will not allow the algorithm to remove the internal output. On the other hand, adding a connected node will remove the output, as is the case with node N. Adding node N results in the two-output cut in figure 1(d).

**Pruning while building $S$.** Some cuts can be determined to be invalid just by looking at $S$. This is the primary added value of the incremental algorithm described in section 5.2. Note that with this optimization, the algorithm will not find all the dominators anymore. This however does not affect its correctness, because none of the skipped dominators would have been a valid cut.

Examples of invalid cuts are cuts that include a forbidden node, and cuts where more than $N_{\mathrm{out}}$ nodes have edges to forbidden nodes. Internal outputs can be discovered as we update $S$; if $N_{\mathrm{out}}$ has reached zero, cuts that have internal outputs are rejected.

**Output–input pruning.** Given an output $o$, the algorithm in figure 3 restricts the seed set to include only ancestors of $o$. We can actually do better by anticipating some validity checks for $S$, and discarding inputs that would fail such checks.

A possible source of optimization comes from forbidden nodes, which are effectively partitioning the search space. When a node $v$ is picked as an output, some of its ancestors

are not possible inputs. Given a pair of nodes, one input and one output, *if a path between the two includes a "forbidden" node, that pair will never appear in a valid cut's set of inputs and outputs*. Then, if a forbidden node $w$ is an ancestor of $v$, $w$'s ancestors will not be valid inputs to $v$.

These optimizations do not improve the speed when $F$ is empty or very small, but this case is quite rare. In particular, large basic blocks usually include many memory loads and/or stores.

Also, given an input $v$ and an output $w$, forbidden nodes provide a way to compute a lower bound to the inputs needed by a cut that includes them. The previous paragraph excluded forbidden nodes from any path between $v$ and $w$. However, if a node on the path has a forbidden predecessor, that predecessor will will become an input. In a precalculation step, we can find all pairs $(v, w)$ of nodes, and count the number of forbidden predecessors on paths between $v$ and $w$. If these nodes are $N_{\text{in}}$ or more, $v$ will not be a valid input for $w$.

The set of valid inputs can be limited further, depending on the targeted acceleration device. Configurable Compute Accelerators [10], for example, may pose a limit on the depth of the accelerated expression. In this case, it would be possible to exclude nodes that are "too far" from the chosen output.

**Input–input pruning.** To find multiple-vertex dominators of size $n$, we start from seed sets of $n-1$ nodes, $\{v_1, \ldots, v_{n-1}, w\}$. Definition 5 implies that at least one path from an input to an output should not pass through any other input. This condition can be used to quickly dismiss invalid seed sets $\{v_1, \ldots, v_{n-1}\}$ prior to the execution of the Lengauer-Tarjan algorithm. We also test a sufficient condition, pruning the sets for which an input $w$ postdominates another input $v$. In this case, all paths from $v$ to an output would pass through the other input $w$ before reaching the output.

**Dominator–input pruning.** Finally, once we find a dominator, we can use it to prune the rest of the search. To do so, we take inspiration from Gupta's original algorithm for multiple-vertex dominators. An *immediate* multiple-vertex dominator is a subset of each node's predecessors. Then, the algorithm recursively replaces a node with its immediate multiple-vertex dominator to derive all the dominators.

Then, once we find a valid multiple-vertex dominator $\{v_1, \ldots, v_{n-1}, w\}$, the nodes $v_i$ can be replaced only by their ancestors—or if they are forbidden nodes, they cannot be removed at all. This test is hard to implement correctly without sacrificing speed, so we implement a simplified version of this test that only affects the last node in the seed, $v_{n-1}$. If this node is forbidden, we do not process it further and proceed with the next possible $v_{n-2}$. If it is not, we choose its replacement among its ancestors.

### 5.4. Data structures

The data-flow graph is represented with both an adjacency matrix, and adjacency lists for predecessors and successors. In addition, we precompute the presence of paths between two nodes, and whether any of these paths touches a forbidden node (see section 5.3). This information is also kept both as a matrix, and as a list. Other precomputed information include the dominator and postdominator trees. Ancestor queries (either on dominators or on postdominators) can be performed in constant time.

$S$ is maintained as a singly linked list of nodes; nodes are added at the tail. $B(I, o)$ can be added to $S$ in $O(n)$ time using a work-list. The old tail is saved whenever $S$ is updated, so that a previous value of $S$ can be restored in constant time.

To compute dominators, we implemented the $O(n \log n)$ variant of the Lengauer–Tarjan algorithm, which employs path compression but no tree balancing. The speed of the algorithm is crucial, because at least 70% of the time is spent in it. Careful engineering of the algorithm sped it up by a factor of 3. Even ensuring that the algorithm's data for a node is exactly as big as a cache line—quite a low-level optimization!—had an impact of 10% or more in the execution time.

We initially used a recursive implementation for the algorithm's *eval* function, since GCC's profile-directed inlining eliminated the recursive calls almost completely. However, path compression connects all the nodes to the same ancestor, and compiler optimization could not deduce this. Switching to an iterative implementation cut the number of memory accesses by a third.

The algorithm maintains several arrays of nodes. Carefully choosing whether to store the node themselves, or rather their *dfnum*[3], can improve the speed widely.

Finally, for some nodes the algorithm does not determine an immediate dominator, but only a node with lower *dfnum* and with the same dominator. In the original algorithm, a final pass on the graph (running from the highest to the lowest *dfnum*) computes the immediate dominators for all nodes. Instead, we resolve immediate dominators lazily using union-find. This does not change the asymptotic complexity of the algorithm, which is still $O(n \log n)$. However, if $k$ is the number of dominator queries performed, it changes their overall complexity from $O(n + k)$ to $O(k \log n)$. This is effective because we only perform a limited number of dominator queries.

[3]The *dfnum* is a number given to each node in a pre-order, depth-first visit of the graph.

## 6. Results

In order to evaluate our algorithm's performance, we collected the data-flow graphs of 250 basic blocks from MiBench. The sizes of the blocks range from 10 to 1196 nodes. We also used four synthetic data-flow graphs, tree-shaped as in figure 4. Their depth varies from 4 to 7 levels. We found them empirically to exhibit worst-case performance for algorithms like [4] and [15]; in particular, for [4] the complexity can be proved to be exponential, $O\left(1.6^n\right)$, on this kind of graph.

The graph in figure 5 compares, for each basic block, the speed of our algorithm versus the implementation in [15]. Subgraphs are enumerated with a constraint of four inputs and two outputs. For each basic block processed, a point is plotted at the intersection between our algorithm's execution time (on the X axis) and that of [15] (on the Y axis). Therefore, data points above the diagonal line represent executions where our algorithm is faster, and data points below the straight line represent those cases where our algorithm is slower. Different symbols group the data points in four clusters; three of these correspond to different sizes and the fourth is reserved to the synthetic DFGs, shaped as in figure 4.

The algorithm's performance is in general better than that of [15]. Some exceptions are expected, because both algorithms employ pruning techniques whose effectiveness can vary widely for different data-flow graph topologies. [15] also seems to achieve a similar polynomial time bound; however, while already stated in [15], this has not been proved formally so far.

In fact, the main contribution of this paper is the proof that the enumeration problem actually has polynomial complexity in $n$. State-of-the-art algorithms explored a binary search space, where each node could either be part of the subgraph or not. We look at the search space from a different perspective: each convex subgraph can have at most $N_{\text{in}}$ inputs and $N_{\text{out}}$ outputs nodes, and these nodes univocally identify the subgraph that lays between them.

Finally, we show here only the performance of the enumeration algorithm presented, and we are not concerned with the speedup enabled by Instruction Set Extensions designed using this technique: the effectiveness of using this technique for identification of custom instruction has been widely validated by past papers, including [7, 4, 10].

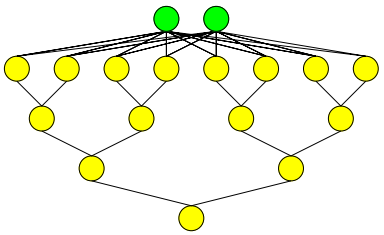

**Figure 4. A tree-shaped data-flow graph, the worst case for algorithms such as [15].**

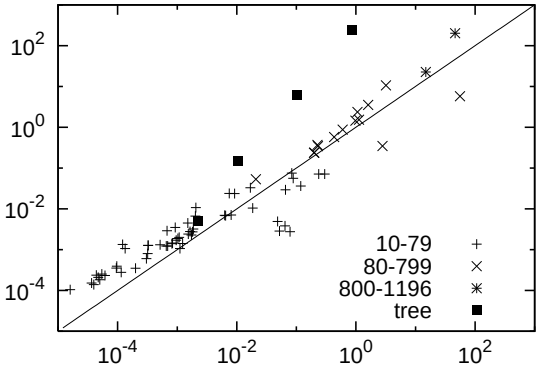


**Figure 5. Run time comparison with [15]. The X axis represents times for our algorithm, the Y axis represents pruned exhaustive search. Data points above the line represent executions where our algorithm is faster.**

## 7. Conclusion

We presented a novel algorithm for full enumeration of the subgraphs of a given data-flow graph. The algorithm supports arbitrary input/output constraints, is not restricted to connected subgraphs and, to our knowledge, is the first one to be presented with polynomial time complexity.

Our approach focuses on choosing the input and output nodes for each cut, instead of exploring an exponential search space where each node may be part of a cut or not. This high-level view allowed us to develop several very effective pruning techniques based on the graph topology—for example, on the dominator and postdominator relationships.

This algorithm was successfully used in in our compiler toolchain [8]; full subgraph enumeration allows detection of high-performance custom instruction sets, yielding speedups up to 6x. In addition to having a known polynomial bound to the complexity, the performance of the algorithm is in parallel, and usually better, than the algorithm in [15]. This represents a further contribution to the field of automatic identification of Instruction Set Extensions for embedded processors.

## References


[1] A. V. Aho and J. D. Ullman. *Theory of Parsing, Translation and Compiling*. Prentice Hall Professional Technical Reference. Prentice Hall, Englewood Cliffs, N.J., 1973.

[2] S. Alstrup, D. Harel, P. W. Lauridsen, and M. Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, Dec. 1999.

[3] K. Atasu, G. Dündar, and C. Özturan. An integer linear programming approach for identifying instruction-set extensions. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 172–77, Jersey City, N.J., Sept. 2005.

[4] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th Design Automation Conference*, pages 256–61, Anaheim, Calif., June 2003.

[5] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, and A. Sangiovanni-Vincentelli. HW/SW partitioning and code generation of embedded control applications on a reconfigurable architecture platform. In *Proceedings of the 10th International Workshop on Hardware/Software Codesign*, pages 151–56, Estes Park, Colo., May 2002.

[6] P. Biswas, S. Banerjee, N. Dutt, L. Pozzi, and P. Ienne. ISEGEN: Generation of high-quality instruction set extensions by iterative improvement. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1246–51, Munich, Mar. 2005.

[7] P. Biswas, V. Choudhary, K. Atasu, L. Pozzi, P. Ienne, and N. Dutt. Introduction of local memory elements in instruction set extensions. In *Proceedings of the 41st Design Automation Conference*, pages 729–34, San Diego, Calif., June 2004.

[8] P. Bonzini and L. Pozzi. Code transformation strategies for extensible embedded processors. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, Seoul, South Korea, Oct. 2006.

[9] H. Choi, J.-S. Kim, C.-W. Yoon, I.-C. Park, S. H. Hwang, and C.-M. Kyung. Synthesis of application specific instructions for embedded DSP software. *IEEE Transactions on Computers*, C-48(6):603–14, June 1999.

[10] N. Clark, A. Hormati, and S. Mahlke. Scalable subgraph mapping for acyclic computation accelerators. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, Seoul, South Korea, Oct. 2006.

[11] N. Clark, H. Zhong, and S. Mahlke. Processor acceleration through automated instruction set customisation. In *MICRO 36: Proceedings of the 36th Annual International Symposium on Microarchitecture*, San Diego, Calif., Dec. 2003.

[12] E. Dubrova, M. Teslenko, and A. Martinelli. On relation between non-disjoint decomposition and multiple-vertex dominators. In *Proceedings of the 2004 IEEE International Symposium on Circuits and Systems*, pages 493–496, May 2004.

[13] R. Gupta. Generalized dominators and post-dominators. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language*, pages 246–257, New York, NY, USA, Nov. 1992. ACM Press.

[14] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, July 1979.

[15] L. Pozzi, K. Atasu, and P. Ienne. Optimal and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-25(7):1209–29, July 2006.

[16] P. W. Purdom Jr. and E. F. Moore. Immediate predominators in a directed graph [h] (algorithm 430). *Communications of the ACM*, 15(8):777–778, Aug. 1972.

[17] P. Yu and T. Mitra. Scalable custom instructions identification for instruction set extensible processors. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 69–78, Washington, D.C., Sept. 2004.