Pattern Libraries for Fast Searching and Data-Path Sharing

Gero Dittmann
IBM Research, Zurich Research Laboratory
Säumerstrasse 4 / Postfach
8803 Rüschlikon, Switzerland
ged@zurich.ibm.com

ABSTRACT

We propose a new method to organize a library of application-graph patterns. Such libraries are employed, e.g., in the design process for application-specific instruction-set processors (ASIPs) to find opportunities to specialize a processor instruction-set for an application domain. In current approaches, these libraries are unordered data collections. Therefore, to search a library for a specific pattern entails comparing the pattern with each entry in the library, which is $O(n \cdot p)$, where n is the total number of operation nodes of all patterns in the library and p the size of the pattern sought.

Our new method arranges a library as a tree based on the structure of the patterns. We present a directed search strategy on the tree with only $O(d)$ and $d \leq p$. Furthermore, we propose a second graph that employs identity operations to reveal synergies between patterns. The synergies can be exploited in ASIP instruction-set synthesis, data-path sharing, and code generation. Both library graphs can easily be superimposed to combine fast searching with synergy detection in a pattern library.

1. INTRODUCTION

A crucial step in the design of ASIPs is the instruction-set generation. Methods for automating this process, surveyed in [1], extract patterns from applications, usually in the form of data-flow graphs (DFGs), and insert them into a pattern library. Along with each pattern, statistical data is stored, such as the number of occurrences of a pattern in the applications. Based on this data, a subset of the patterns in the library is then selected for implementation as specialized instructions.

For each pattern that is found in the applications a search in the library is performed to check whether the pattern is already present, and the pattern is then either added to the library or only the statistics are updated. The pattern libraries described in current approaches are unordered collections of patterns. A search algorithm on such a library has to compare all operation nodes of the pattern in question with all operation nodes of all patterns in the library, in the worst case. Hence, the computational complexity of this search is $O(n \cdot p)$, with n the total number of operation nodes of all patterns in the library and p the size of the pattern sought [2].

Because a search is conducted for each pattern in the applications and because the pattern libraries tend to be large, the computational complexity of the search algorithm has a significant impact on the total running time of the instruction-set generation. In [2], for instance, memory requirements of more than 200 MB are given for single benchmark applications, resulting in a running time of more than 24 hours—even with a number of heuristics already built in to keep library size low. The memory size of today's workstations alleviates the problem of memory consumption, but the latency of pattern searches is still critical.

In this paper, we introduce a novel organization for pattern libraries that enables a search algorithm with only $O(d)$, where d is the size of the pattern sought, up to the maximum pattern size in the library ($d \leq p$). Furthermore, we propose a method that employs identity operations to reveal synergies between patterns. The synergies can be exploited in ASIP instruction-set synthesis, data-path sharing, and code generation. Both methods can easily be combined for fast searching and synergy detection in a pattern library.

The remainder of this paper is structured as follows: In Section 2 we review some related work. In Section 3 we introduce our concept of a pattern-search graph and its fast search and insert algorithms. Section 4 presents our identity-operand graph for pattern substitution. Experimental results concerning library size and search time are given in Section 5. In Section 6 we conclude the paper and indicate some directions for future work.

2. RELATED WORK

An early approach to instruction-set generation for ASIPs can be found in [3]. *Parallel* operations in DFGs are scheduled into time steps, and operations in the same time step form an instruction. A simulated annealing algorithm is then used to modify the original operation schedule to find better instruction sets. Moreover, different operand encodings are tried out in order to meet a given instruction-size constraint. A data structure for the collection of instruction candidates is not described.

In [2], existing processors are extended for an application domain by implementing patterns as special instructions that consist of *sequential and parallel* operations sharing at least one operand. Applications are not represented by the compiler output directly but rather by execution traces, which enables the detection of patterns across control-flow boundaries and a better estimate of their frequency of occurrence.

The pattern-matching algorithm that works on these traces develops its pattern library on the fly: It starts with a library of basic operations and then iteratively adds all possible combinations of each operation node with its neighbors, i.e., combinations with other nodes that share at least one operand with it in the application graph. Patterns from this library are then selected to cover the application graph such that each operation is covered by exactly one pattern. This selection is called a *cover* of the application graph.

A variation of dynamic programming is employed to minimize the implementation cost of the cover. The patterns in the library are sorted by the number of times they occur in the application graphs and by the number of times they were selected for a cover. At the end of the process, patterns from the list are manually selected, grouped, and implemented.

The library-construction algorithm that is employed in this methodology tries to find pattern matches by iterating over all operation nodes of all patterns in the library and comparing them with all nodes in a subject pattern. Accordingly, the library is an unordered collection of patterns. The search algorithm given has a computational complexity of $O(n \cdot p)$, as explained in Section 1. In order to keep n low, heuristics are introduced to limit the library size by excluding patterns that do not seem beneficial.

A different method to cluster *parallel* operations in order to form new instructions is proposed in [4]. DFG nodes are scheduled as soon as possible and as late as possible to determine their mobility. From this information, a graph is derived in which two nodes are connected by an edge if they can be scheduled in the same control step. The edges are weighted with the number of times the nodes can be scheduled together. For instruction selection, a profiling function is employed to find the most frequently occurring operation pairs. This function must maintain a library of candidate pairs in order to collect profiling information, but it is not described in the paper.

The covering of code sections with appropriate elements in a pattern library for implementation has traditionally been performed by graph-based pattern-matching algorithms, e.g., for code generation in compilers [5, 6] and for technology mapping in logic synthesis [7, 8]. These algorithms assume a fixed set of patterns, and they perform preprocessing on the entire set to speed up the actual matching. Methods for instruction-set generation, in contrast, construct their libraries on the fly, iteratively filling the library with new patterns. For such dynamic libraries the preprocessing of the entire pattern set would have to be repeated for each new pattern. This problem renders the algorithms very inefficient for our purposes.

Recently, a fundamentally different approach to the matching problem has been proposed in using combinational equivalence checking on binary decision diagrams [9]. Because of its independence of the internal structure of patterns this method finds more matches than traditional graph-matching algorithms. A drawback of the method is its complexity. The authors report matching times of many hours for a library with ten instructions where a graph-matching algorithm needs less than one hour with a library of many thousands of patterns.

Identity operations, which we use to find relations between patterns, have been exploited for basic algebraic transformations in compilers [10] and in high-level synthesis [11]. In [12] and [13], identity operations are *inserted* into sequences of operation nodes in order to increase the number of identical patterns. In contrast, we propose to use identity elements to *eliminate* nodes from patterns. As a side effect, the library we construct in this way reveals the same opportunities to substitute one pattern by another—and more, because our approach is not constrained to small sequential patterns.

3. THE PATTERN-SEARCH GRAPH

Arranging patterns in a linked list results in a completely arbitrary order. There is no relation in the order of the patterns that could be exploited for directed and therefore faster searches. Patterns do, however, have a structure that lends itself to ordering. In particular, a pattern can be a subgraph of another pattern, or two patterns can have common subgraphs. In the following, we present a method to order patterns according to their structure and the operation type of their nodes. First, we introduce the method for tree-shaped patterns and then generalize it in Section 3.3 to patterns that are directed acyclic graphs (DAGs). The resulting new structure provides significantly faster searches than conventional library structures.

We arrange the patterns as a tree. Each node in this tree represents a pattern. The root of the tree has edges to all patterns that consist of a single basic operation, forming a first level of patterns. The basic operations can be extended to form two-node patterns by connecting the output of another operation to one of the operand inputs of the basic operation. Each of the first-level patterns has edges to all two-node patterns with the corresponding first-level pattern as their root node. The two-node patterns are the second level of the library tree. This process continues for more complex patterns, adding more levels to the library graph. We call this tree graph a *pattern search graph (PSG)* as it facilitates directed and fast searches as we will show in Section 3.1. Figure 1 shows an example PSG of a pattern from an application that parses headers of network packets [14].

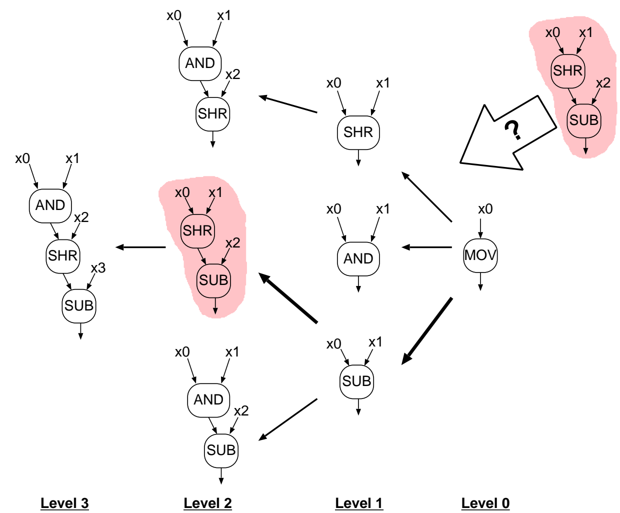


Figure 1: Example of a pattern search graph.

Compared with the linked list, the structure of a PSG is two-dimensional rather than one-dimensional. This entails the following redundancy. Take a pattern consisting of three operation nodes, two of which feed their result as operands to the root node. In this case, we could have two PSG paths to the pattern: First adding the left operand to the root operation and then the right, or adding the right operand first and then the left. In order to have only one path from the library root to each pattern, we posit that appropriate operations are connected first to right operands and then to left operands, assuming basic operation nodes with two operands. The search algorithm presented in the next section requires only the remaining PSG paths with precedence for the right operand.

This PSG organization entails that all patterns on the path to a complex pattern must be present in the library. Hence, when inserting a pattern into the library, we sometimes must also insert additional patterns that are on the PSG path to the new pattern but that are not yet present in the library. Our performance measurements in Section 5 show, however, that the overhead introduced by this effect is negligible.

3.1 Searching Patterns in a PSG

The access to the pattern library can be accelerated significantly by exploiting the order of the patterns in a PSG. When searching for a particular pattern in the library, we start with one of the primitive operation nodes it comprises, namely, the root node. We then add operation nodes in the pattern in reverse topological order by following the edges in the search graph. In this way, we arrive at the complete pattern, provided it exists in the library.

The recursive algorithm in Figure 2 implements the proposed search strategy. It traverses the pattern sought depth-first and right-branch-first, corresponding to the rule for avoiding redundant paths we posited earlier. The procedure returns either the position of the pattern in the library or NULL.

```
activeOpnd = 0;      /* global variable */
patInLibrary = find( patRoot, libRoot );


LibNode find( patNode, libNode ) {
 LibNode nextLibNode = libNode.next[activeOpnd][patNode.operator];

 activeOpnd = 0;      /* right operand is always 0 */
 if (patNode has rightOpnd and nextLibNode exists)
  nextLibNode = find( patNode.rightOpnd, nextLibNode );

 activeOpnd++;
 if (patNode has leftOpnd and nextLibNode exists)
  nextLibNode = find( patNode.leftOpnd, nextLibNode );

 return nextLibNode;
}
```

Figure 2: Pseudo code: Searching a tree pattern in a PSG.

Consider the example PSG in Figure 1. In order to search this graph for, e.g., the pattern in the upper right-hand corner, the algorithm starts with the pattern root—in this case the subtraction. In the first line of the *find*-function, `libNode` is set to the library entry corresponding to the pattern root by following the `next`-pointer indexed by the only operand of the library root and by a *SUB* operator. Then the right operand of the pattern root—labeled *x2*—is examined, which is NULL because it is an external pattern input. Therefore, it is skipped and the left operand is checked, which is not NULL because it is connected to the output of the shift operator. Consequently, the *find*-function is called recursively with the shift operation as the next `patNode` and the subtraction as the `libNode`.

This time, the first line of the *find*-function follows the `next`-pointer indexed by the second operand of the subtraction and by an *SHR* (shift-right) operator, and hereby sets `libNode` to the library entry we have been seeking. Both the left and the right operand of `patNode` are NULL, and therefore the library entry sought is returned, after unwinding the recursive calls, to be assigned to `patternInLibrary`.

The *find*-function is called at most once for each node in the pattern sought. The pointers to the next library nodes are stored in the array *libNode.next* with linear access time. Therefore, this search is $O(p)$, with p the number of operation nodes in the pattern sought. If the pattern sought is larger than the largest pattern in the library, the search stops even earlier. Hence, the worst-case computational complexity of a search is $O(d)$, with d the size of the pattern sought, up to the maximum number of operation nodes in any pattern in the library—which is equal to the maximum depth of the library search-graph. Note that $d \leq p$. Our experimental results presented in Section 5 show that a search in a PSG is indeed orders of magnitude faster than a search in a linked list.

The pattern-matching algorithms found in the literature, including those based on hash tables, have, in the best case, a similar computational complexity as our PSG search [5]. In addition, however, they require significant preprocessing of the library with at least $O(n)$, where n is the total number of operation nodes of all patterns in the library. Such a preprocessing step renders these algorithms inefficient for frequent updates to the library as required in ASIP design. Our PSG, in contrast, does not require any preprocessing.

3.2 Inserting Patterns into a PSG

We insert a pattern into a PSG by searching it and complementing the path if it ends before finding the pattern. The algorithm is given in Figure 3. The difference to the search algorithm in Figure 2 is in creating a new `LibNode` if `nextLibNode` does not yet exist.

```
activeOpnd = 0;      /* global variable */
patInLibrary = insert( patRoot, libRoot );


LibNode insert( patNode, libNode ) {
 LibNode nextLibNode = libNode.next[activeOpnd][patNode.operator];
 if (nextLibNode does not exist) {
  liveOpnds = libNode.liveOpnds − activeOpnd + 1;
  nextLibNode = new LibNode( liveOpnds );
  libNode.next[activeOpnd][patNode.operator] = nextLibNode;
 }

 activeOpnd = 0;      /* right operand is always 0 */
 if (patNode has rightOpnd)
  nextLibNode = insert( patNode.rightOpnd, nextLibNode );

 activeOpnd++;
 if (patNode has leftOpnd)
  nextLibNode = insert( patNode.leftOpnd, nextLibNode );

 return nextLibNode;
}
```

Figure 3: Pseudo code: Inserting a tree pattern into a PSG.

The parameter passed to the new `LibNode` is the the number of operands to which another operation node can be attached. This number is needed to dimension the `next`-array correctly. It is not simply the number of free operands because the operands that have been visited en route to the current `libNode` will not be used anymore. If there were an operation node to attach to any of them, this would already have happened at the first visit to the operand. Hence, these operands are "dead" and need neither an entry in the `next`-array, nor a number. The remaining numbered operands we call *live operands*. Compared with `libNode`, the number of live operands for the new `LibNode` is reduced by the number of the current active operand where the next pattern node is attached. The result is then increased by one because the new node occupies the current active operand but contributes two new operands—left and right. Hence, the number of live operands used in the pseudo code for the new `LibNode` computes to

$$\text{libNode.liveOpnds} - \text{activeOpnd} + 1.$$

The modification of the insert algorithm compared with the search algorithm does not affect the computational complexity. It is also $O(d)$, with d the size of the pattern sought, up to the maximum number of operation nodes in any pattern in the library, and $d \leq p$.

3.3 Extension for DAG Patterns

In order to extend the PSG concept from trees to directed acyclic graphs, we need to allow for operator nodes whose result feeds more than one operand of other operator nodes. We achieve this by introducing a `revisit`-array with each PSG entry, similar to the `next`-array, through which operator nodes are revisited that have been encountered before via a different operand. The `revisit`-array is indexed by the operand number to which the next node is attached and by the unique number of the operator node that is being revisited via this operand. Therefore, the operator nodes must be numbered, which we do in the same order in which the search and insert algorithms traverse them. In this way we can detect that we revisit a pattern node because it will have a lower number k than the current node $j > k$, whereas otherwise the number of the next node is higher: $k = j + 1$. Whenever we encounter a revisited node in a pattern, we take the associated hop in the library graph through the `revisit`-array instead of the `next`-array. This is the only modification of the tree algorithms necessary. It enables a PSG to handle DAGs without transforming them to trees first as required by earlier methods [7].

We formulate the computational complexity of our DAG algorithms as a function of the number of edges in the pattern rather than the number of operator nodes because it is the edges that the algorithms traverse and in a DAG there can be more than one edge per node. This, however, also results in a linear complexity: $O(e)$, with e the number of edges in the pattern sought, up to the maximum number of edges in any pattern in the library—which is again equal to the maximum depth of the library search-graph—plus one because the first step from the library root to the first library entry does not correspond to an edge in the pattern. This complexity is equal to that of the tree-pattern algorithms if we also formulate it as a function of the number of edges in the searched pattern.

4. THE IDENTITY-OPERAND GRAPH

As we have seen, the pattern-search graph exploits relations between patterns to speed up searches in a library. The method we introduce in this section is based on similar relations and allows a more complex pattern to substitute a class of simpler patterns. We achieve this using identity operands to disable operation nodes in a pattern in order to mimic any of a group of simpler patterns.

Our method can be employed in ASIP design to handle different application patterns by only one special instruction, resulting in a leaner instruction set. Used in code generation, the method increases the number of opportunities to deploy complex instructions. Furthermore, it provides a systematic approach to finding opportunities for data-path sharing in high-level synthesis.

4.1 Identity Operands in DFG Patterns

Most primitive operations that are found in the instruction sets of general-purpose processors can be used to map one input operand a to itself by applying an identity operand $\mathrm{op_{id}}$, i.e. the algebraic identity element for that operator, to the other input such that

$$a \circ \mathrm{op_{id}} = a, \quad \mathit{or}$$
$$\mathrm{op_{id}} \circ a = a,$$

turning the primitive operation $\circ$ into an identity operation. Examples of identity operands are given in Table 1.

Table 1: Identity operands.

primitive operation	left operand	right operand
$+$	0	0
$-$	n/a	0
$*$	1	1
$/$	n/a	1
$<<, >>$	n/a	0
AND	all 1's	all 1's
OR, XOR	0	0

An operand for an operation node in a DFG pattern is either generated by another node in the same pattern or is an external input to the pattern. Depending on their operands, we distinguish three types of nodes:

- A *leaf node* has two operands that are external inputs to the pattern.
- An *internal node* has two operands that are both generated by other nodes in the same pattern.
- A *cyclops node* has only one operand that is an external input to the pattern and the other operand is generated within the pattern. Depending on whether the external input is the right or left operand, we call the node a *right cyclops* or a *left cyclops*.

A complex pattern can be transformed into a simpler pattern by applying the identity operands of its operation nodes to the appropriate inputs, thereby effectively eliminating nodes from the pattern. Particular operands can be applied directly to leaf nodes and to cyclops nodes. The non-commutative operations in Table 1 have no left identity operand. Nodes of these operation types must be leaves or right-cyclops nodes to be removable, i.e., their right input must be accessible from outside the pattern.[1]

By applying identity operands to one node at a time, a pattern of n nodes, of which m are removable can be transformed into m patterns of $n-1$ nodes. By recursively repeating this on each of the simpler patterns, the complex pattern can eventually be reduced to primitive operations. If all leaf nodes and all cyclops nodes at any stage of the recursion are removable then the set of primitive operations includes all operation types that occur in the pattern. The primitive operations finally all converge to a *move* operation.

The sequence of applying the identity operands can be used to sort the patterns. We represent this sorting as a graph with the graph nodes being the patterns and the directed graph edges representing the application of an identity operand to one particular operation node in the pattern. The edges are directed from the more complex pattern to the derived smaller one. We call this type of graph an *identity-operand graph (IOG)* and say that a complex pattern *dominates* the simpler patterns in its IOG. Figure 4 shows the IOG for the level-3 pattern in the PSG in Figure 1. Evidently, both graphs contain the same patterns and only differ in the number and the orientation of the edges. This suggests that both graphs can be efficiently constructed simultaneously and be held in a unified data structure: a combined PSG/IOG.

The library IOG shows which simpler patterns can be covered by a complex instruction during code generation, again by applying the appropriate identity operands to its input. Therefore, these simpler patterns need not be implemented as individual instructions if the complex pattern is chosen for implementation—provided the possibly slower execution and the cost of applying the identity

[1]Load and store operations have no identity operand at all, but because of their long latency they are less relevant for instruction-set generation than arithmetic operations.

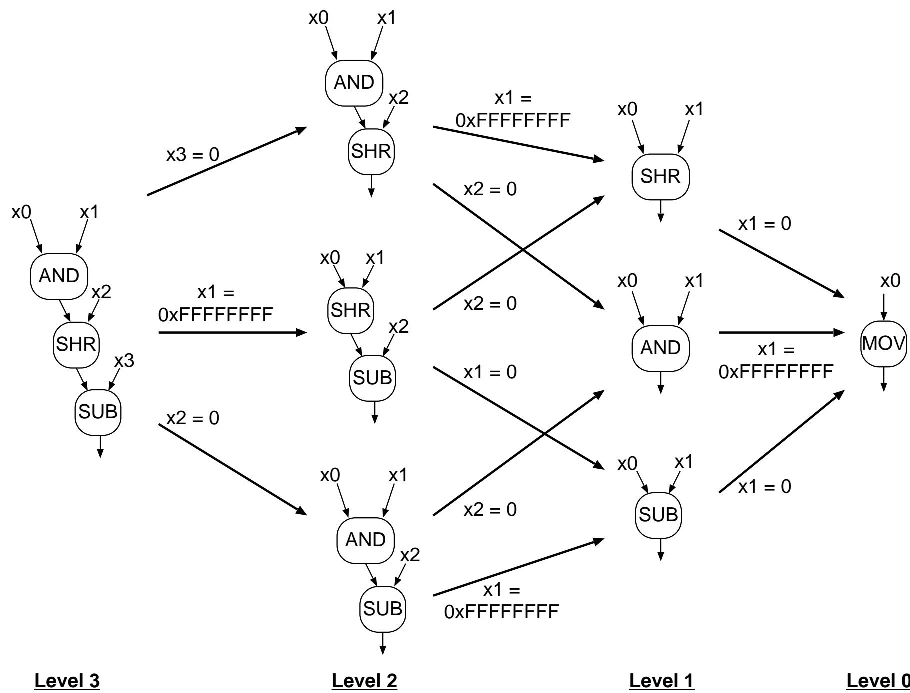


Figure 4: IOG of a Tree Pattern.

operands can be afforded. This cost may be, for instance, additional *move* instructions. If the cost is lower than the benefit, then the IOG reveals opportunities to substitute patterns by more complex ones during instruction-set synthesis and code generation, leading to a reduced number of special instructions that provide the same benefit.

To this end, the power of a complex pattern to cover all derived simpler patterns seems to suggest that only the most complex patterns should be chosen for implementation. But in ASIP design methodologies, there is an implementation cost function $C(\text{pattern})$ associated with the patterns, which usually increases with pattern complexity and captures, for instance, operand-encoding effort, die area, or latency. This cost function counterbalances the tendency toward more complex patterns for implementation.

In practice, nodes could also simply be deleted from the pattern for construction of the library graph, regardless of an appropriate identity operand. This approach would, however, eliminate the IOG property that a pattern can substitute all other patterns in its IOG by applying identity operands accordingly, a property that is needed for more efficient instruction selection.

4.2 Inserting Patterns into an IOG

In the following we describe an algorithm that combines the construction of the IOG of a pattern with its insertion into a PSG library. The resulting combined pattern-search/identity-operand graph enables fast searches and shows synergies between patterns at the same time. The search algorithm requires only the PSG edges, and therefore the code in Figure 2 does not need to be modified. In contrast, the insertion algorithm has to create also the IOG edges.

The algorithm traverses the pattern to be inserted in the same way as the algorithms in Section 3, discovering right branches first. While doing so the algorithm maintains a list of patterns it has inserted, called *fragments*, that are consecutively combined with discovered pattern nodes to form all possible combinations of pattern nodes that occur in the IOG of the pattern. Each of these combinations is inserted into the PSG library and the IOG edges are created provided that the corresponding node is removable.

Discovered pattern nodes must be combined with fragments in such a way that only patterns are generated that can be modeled by disabling operation nodes from the original pattern to be inserted. In particular, a first node can only be operand to a second node on the same side—left or right—on which the pattern branch that contains the second node is attached to the first node.

Consider, for instance, the pattern in Figure 5. The nodes are labeled in the order in which they are discovered. Node A is discovered first and inserted into the library as well as into the list of fragments. Node B is discovered next. Now, the only possible combination with node A in the fragment list is to make B the right operand of A. There is no set of nodes in the original pattern to be disabled such that node B would be the left operand of A.

To ensure that discovered nodes are combined with fragments in possible ways only, each fragment has an operand marked as *active operand*. In the combination process, a discovered pattern node is always attached to the current active operand of the fragment. Once the pattern branch attached to the active operand of a fragment has been discovered completely, the *active* mark is moved to the next possible operand of the fragment before nodes in the branch attached to this new active operand are being discovered. If there is no possible operand left to become active then all combinations with the fragment have been generated and the fragment is deleted from the list of fragments.

Attaching a discovered node to the active operand of a fragment, however, can still result in impossible combinations. This is the case if the pattern branch that is currently discovered is a left branch of a pattern node that is not contained in the fragment, but the fragment does contain a node from the right branch. The node from the right branch then blocks the only operand at which a node from the left branch could be attached to create a possible combination.

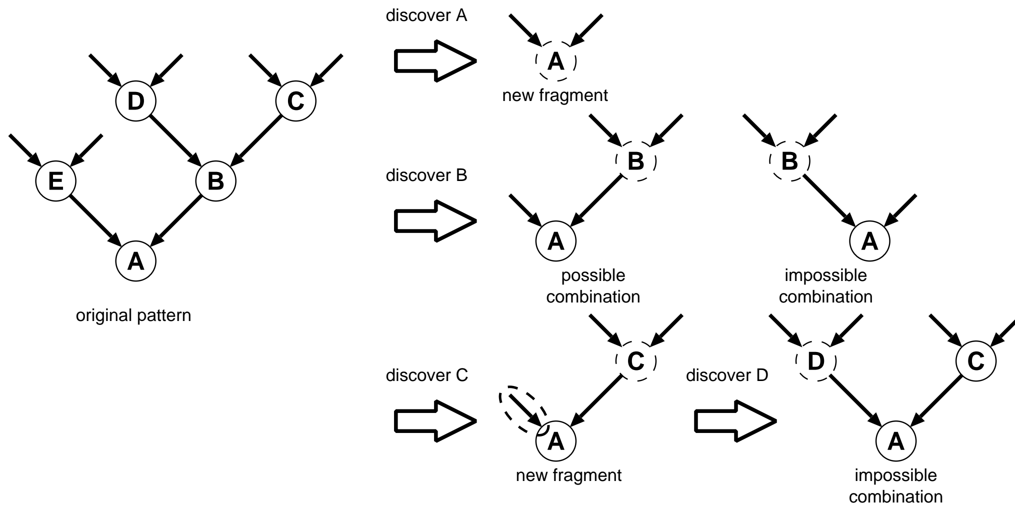


Figure 5: Fragment combination.

To illustrate this, consider again the pattern in Figure 5. When node C is discovered, it is combined with node A from the fragment list as its right operand. As in the original pattern no operand of C is fed by an operation node, the next possible operand of the new fragment is the left operand of A, which is therefore marked as the active operand, indicated by the dashed oval. If we now discover pattern node D and attach it to the active operand of the fragment we just created, as shown as the last step in the figure, then we arrive at an impossible combination again.

To prevent this type of impossible combination, we temporarily disable fragments if the equivalent of their active operand in the original pattern is not on the path to the pattern node to be discovered next. In the example, the left operand of A is not on the path to node D. Therefore, a fragment that has the left operand of A marked active has to be disabled at this point. A disabled fragment is enabled as soon as pattern nodes in its branch are being discovered. The example fragment is enabled just before node E is discovered.

With these mechanisms in place, our fragment algorithm generates all patterns of the IOG to be constructed.

4.3 IOGs of DAG Patterns

Some steps in a DAG search in a PSG do not add another operation node to a pattern on the search path. Instead they attach another operand input to the output of a node that is already there. These steps cannot be reversed by applying identity operands. Therefore, the IOG for a DAG pattern does not necessarily contain the patterns on the search path in the PSG for the same pattern.

Figure 6 shows an example of the IOG of a DAG pattern. Unlike the tree-pattern IOG in Figure 4, the DAG-pattern IOG does not comprise the generic basic operations on level 1, but the operations are more constrained, e.g., two inputs are equal, resulting in a squaring operation, or one input being a constant, resulting in an increment operation. Furthermore, the IOG does not converge to a move operation at level 0. The IOG of a DAG, however, still is useful to find substitution opportunities as in the tree case. For this purpose it can be constructed separately from the search graph.

5. EXPERIMENTAL RESULTS

We have implemented a pattern generator in the MachSUIF compiler-framework [15]. To grow the scope for the pattern generator

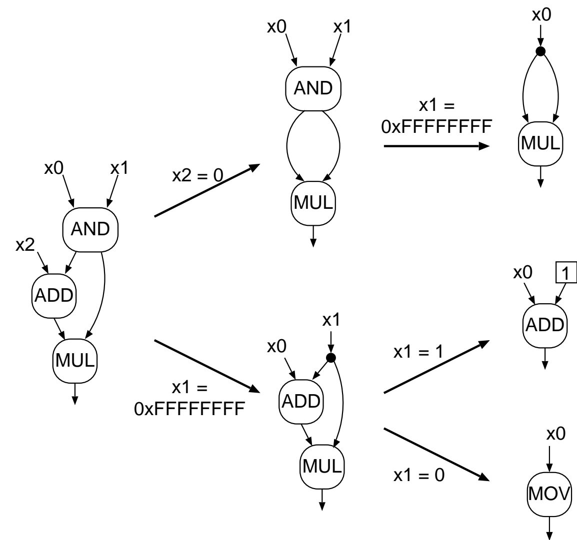


Figure 6: IOG of a DAG pattern.

beyond basic-block boundaries, the generator works on a static single assignment (SSA) representation which extends the DFGs to operand definitions in other basic blocks. We include operations in other DFGs in a pattern if these operations are reached directly, i.e. not through a ϕ-function, because these operations can easily be moved across control-flow boundaries.

We fed the patterns to one of four types of pattern libraries:

- A PSG for DAG patterns, constructing only the search path to each pattern.
- A combined PSG/IOG for tree patterns, constructing the entire IOG for each pattern.
- An unordered linked-list library for trees.
- An unordered linked-list library for DAGs.

We tested the libraries on a subset of the MediaBench benchmark suite [16]. We measured the search times while the library was being constructed, i.e., from the first pattern inserted to the last. Furthermore, we measured the number of entries in each library after the last pattern had been inserted. The pattern generator passed between 60 and 41,233 different patterns per run to the library.

First we compare PSG libraries of DAGs with a linked-list implementation. Our experiments show that the number of patterns in the PSG libraries is not significantly larger than in the linked lists. This is intuitive because we only supplement patterns on the search path to each pattern rather than constructing the entire IOG for each pattern inserted.

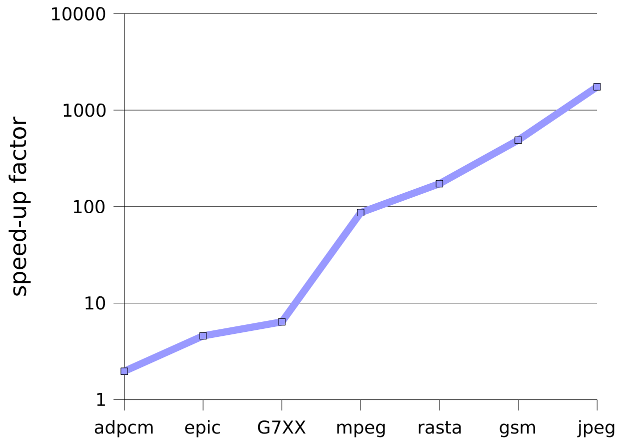
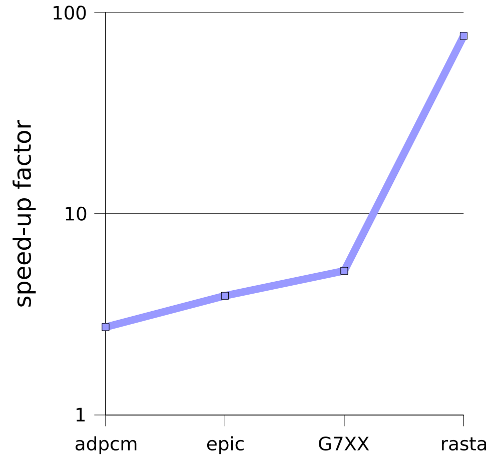


Figure 7: Search speed-up for DAG patterns in a PSG.

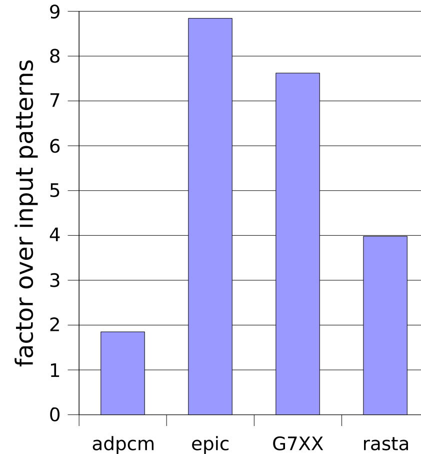
Figure 7 compares the search times for DAG patterns, giving the speed-up of PSG libraries over linked lists. The benchmarks are ordered by the number of different patterns that the pattern generator passed to the library. The more patterns in the library, the longer the linked list and the longer the worst-case search time in the list. In contrast, the search time on a PSG is independent of the library size. Hence, the larger the library, the more significant the advantage in search speed of the PSG. In the JPEG library with 42,864 DAG patterns, we measured a PSG speed-up factor of 1743 compared with the linked list.

In a second set of experiments, we constructed the entire IOG for each pattern inserted. As shown in Section 4.3, a PSG/IOG library can only be constructed for tree-shaped patterns. Because of the increase in size due to constructing an IOG for each pattern, a PSG/IOG organization is viable for medium-sized libraries only. The largest library in our experiments was the rasta benchmark with 11,371 patterns in the PSG/IOG. Figure 8(a) shows the resulting speed-up for searches for tree patterns, comparing a PSG/IOG with a linked list. Figure 8(b) shows the increase in size of the PSG/IOG over the linked list for the same workload.

It can be seen that the overhead due to constructing the IOG for each entered pattern increases the size of the library to up to ninefold compared with a linked list, which holds only the entered patterns, i.e., the minimal set. However, the hierarchical organization of the PSG/IOG still reduces the search times as dramatically as the pure PSG—in spite of the larger size. Note that only the IOG enables the use of complex patterns to substitute simpler ones. The linked list does not provide this advantage. If the substitution is not required, the pure PSG provides the search speed without the overhead.



(a) Search speed.



(b) Library size with IOGs.

Figure 8: Performance of a combined PSG/IOG library of tree patterns.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a novel method to organize pattern libraries for ASIP design: PSGs. Compared with conventional unordered libraries, PSGs enable more efficient searches with a computational complexity of $O(d)$ instead of $O(n \cdot p)$ with $d \leq p$. Our experiments show that our libraries can be searched orders of magnitude faster than state-of-the-art libraries. We measuered speed-up factors of up to 1743.

Because this eliminates the dependency between computational complexity and library size, large pattern libraries can be handled. Current methods for library organization rely on heuristics that exclude less promising patterns to keep the library size low. Given the memory sizes of today's workstations, our approach eliminates the need for such heuristics. Therefore, exact methods are now possible that do not incur the risk of missing beneficial patterns that heuristics might eliminate.

Furthermore, we introduced IOGs which reveal opportunities to substitute patterns by others. This can be exploited for instruction-set generation, resulting in a leaner instruction set with the same speed-up. Moreover, IOGs can be used during code generation to increase the number of opportunities for the use of specialized instructions, resulting in faster code. We also see an application of IOGs in synthesis tools to find opportunities for data-path sharing.

Currently, we study how to find patterns deterministically that perform the same function but have different shapes because of commutative operations. We try to achieve this by ordering operators and operands, which results in a canonicalization of the patterns. Another interesting extension would be to handle patterns with multiple roots by joining the roots at a dummy node. The dummy node would become the single root and the pattern could be inserted into a PSG library.

Finally, we will explore the possibility of substituting the pattern-finding algorithms that feed a pattern library by inserting the complete DFG of each basic block into an PSG/IOG library. We expect to be able to generate all sub-graphs of a DFG by allowing a small set of nodes to be removed from a pattern that are not removable with identity operands.

7. REFERENCES

[1] Gero Dittmann and Andreas Herkersdorf. Multi-layer intermediate representation for ASIP design and critical-path optimization. Technical Report RZ 3484, IBM Research, February 2003.

[2] Marnix Arnold. *Instruction Set Extension for Embedded Processors*. PhD thesis, Delft University of Technology, Delft, The Netherlands, March 2001.

[3] Ing-Jer Huang and Alvin M. Despain. Generating instruction sets and microarchitectures from applications. In *Proceedings of ICCAD-94*, pages 391–396, November 1994.

[4] Philip Brisk, Adam Kaplan, Ryan Kastner, and Majid Sarrafzadeh. Instruction generation and regularity extraction for reconfigurable processors. In *Proceedings of CASES 2002*, pages 262–269, October 2002.

[5] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM (JACM)*, 29(1):68–95, 1982.

[6] Alfred V. Aho and Mahadevan Ganapathi. Efficient tree pattern matching: an aid to code generation (extended abstract). In *Proceedings of the 12th SIGACT-SIGPLAN*, pages 334–340. ACM Press, 1985.

[7] Kurt Keutzer. DAGON: technology binding and local optimization by DAG matching. In *Proceedings of the 24th DAC*, pages 341–347. ACM Press, June 1987.

[8] Min Zhao and Sachin S. Sapatnekar. A new structural pattern matching algorithm for technology mapping. In *Proceedings of the 38th Conference on Design Automation*, pages 371–376. ACM Press, 2001.

[9] Newton Cheung, Sri Parameswaran, Jörg Henkel, and Jeremy Chan. MINCE: Matching instructions using combinational equivalence for extensible processor. In *Proceedings of DATE'04*, Paris, France, February 2004. ACM Press.

[10] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, San Francisco, 1997.

[11] Birger Landwehr and Peter Marwedel. A new optimization technique for improving resource exploitation and critical path minimization. In *Proceedings of ISSS'97*, pages 65–72, 1997.

[12] Miodrag Potkonjak and Sujit Dey. Optimizing resource utilization and testability using hot potato techniques. In *Proceedings of DAC'94*, pages 201–205, 1994.

[13] Dirk Herrmann and Rolf Ernst. Improved interconnect sharing by identity operation insertion. In *Proceedings of ICCAD-1999*, pages 489–493, 1999.

[14] Gero Dittmann. Programmable finite state machines for high-speed communication components. Master's thesis, Darmstadt University of Technology, `http://www.zurich.ibm.com/~ged/`, 2000.

[15] Michael D. Smith's Research Group on Compilation and Computer Architecture. `http://www.eecs.harvard.edu/~hube/software/`.

[16] C. Lee, M. Potkonjak, and W.H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proc. of the 30th Annual Intl. Symp. on Microarchitecture*, pages 330–335, Dec. 1997.