Organizing Libraries of DFG Patterns

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

Abstract

We propose to arrange a library of tree patterns into a hierarchy by means of identity operations. Compared with current unstructured approaches, our new method reduces the computational complexity of searching a pattern from $O(n \cdot p)$ to only $O(d)$, $d \leq p$. Furthermore, the organization reveals synergies between patterns for ASIP instruction-set synthesis, data-path sharing, and code generation.

1 Introduction

A crucial step in the design of Application-Specific Instruction-set Processors (ASIPs) is the instruction-set generation. Methods for automating this process extract patterns from the data-flow graphs (DFGs) of applications 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.

In current approaches, the pattern libraries are unordered collections of patterns. A search algorithm on such a library has a computational complexity of $O(n \cdot p)$, with n the total number of operation nodes of all patterns in the library and p the size of the pattern sought.

In this paper, we introduce a novel organization for pattern libraries that enables a search algorithm with only $O(d)$ time. Here, d is the size of the sought pattern up to the maximum pattern size in the library, therefore $d \leq p$. Furthermore, the new library organization reveals opportunities to substitute one pattern by another. This can be exploited for more efficient instruction selection and code generation. The method is presented for tree-shaped patterns but can be extended to directed acyclic graphs (DAGs).

2 The identity graph of a pattern

Most primitive operations in the instruction sets of general-purpose processors can be used to map one input operand a to itself by applying an identity operand op_{id} to the other input, i.e. the algebraic identity element for the operation:

$$a \circ op_{id} = a.$$

This turns the primitive operation $\circ$ into an identity operation. For example, the identity operand for an addition is 0 and for a multiplication it is 1.

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 values can be applied to operands that are accessible from the outside.

By applying identity operands to one node at a time, a pattern of n nodes, of which m are removable in this way, 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 outer nodes of a pattern at all stages 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.

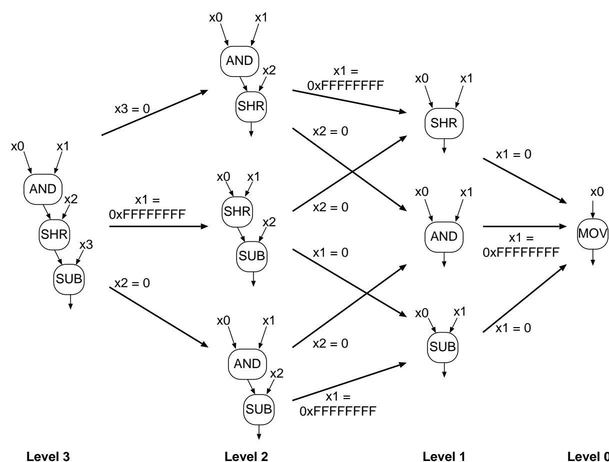

Figure 1. ID Graph of a Pattern

If all patterns generated in this way are entered into the pattern library then the sequence of applying the identity

1530-1591/04 $20.00 (c) 2004 IEEE

operands can be used to sort the patterns in the library. We represent this sorting as a graph in which the graph nodes are the patterns and the directed graph edges represent 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 graph (ID graph)*. Figure 1 shows the ID graph of an example pattern.

Merging the ID graphs of all patterns results in a library ID graph. This graph in turn reveals which simpler patterns can be covered by a complex instruction during code generation, again by applying the appropriate identity operands to its inputs. Thus, these simpler patterns need not be implemented as individual instructions if the complex pattern is chosen for implementation—provided that the possibly slower execution and the cost of applying the identity operands can be afforded. This cost may, for instance, be additional *move* instructions. If the cost is lower than the benefit, the ID graph reveals opportunities to substitute patterns by more complex ones during instruction-set synthesis and code generation, leading to fewer special instructions that provide the same benefit. In a similar fashion, an ID graph could be employed in logic synthesis to find opportunities for data-path sharing.

3 Searching an ordered library

The access to the pattern library can be accelerated significantly by exploiting the order of the patterns. 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 library ID graph against their direction. In this way, we arrive at the complete pattern, provided it exists in the library.

Figure 2 shows the *ID-based search graph* for the pattern in Figure 1 with only the reverse edges that are required for the search algorithm. In order to search this graph for, e.g., the pattern on the right, which consists of a right shift (SHR) followed by a subtraction (SUB), the search algorithm starts with the pattern root—in this case the subtraction. In the library it follows the pointer to the subtraction operation on level 1. 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 operation.

Consequently, the search function is called recursively and follows the pointer in the library that attaches a right shift to the left operand of the subtraction. This time, both the left and the right operand of the pattern node are NULL. Therefore, there are no further pattern nodes to be discovered, and the library entry sought has been found.

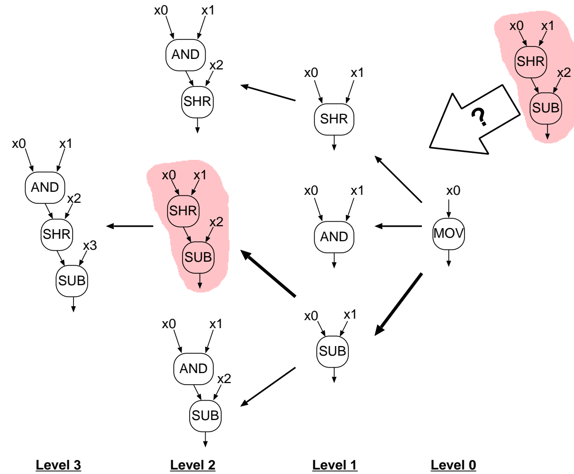

Figure 2. ID-Based Search Graph

The search function is called at most once for each node in the pattern sought. The pointers to the next library nodes are stored in an array with linear access time. Therefore, this search is $O(p)$, with p the number of operation nodes in the pattern sought. If this pattern has more operation nodes 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. It follows that $d \leq p$.

4 Results

ID graphs reveal opportunities to substitute patterns by others. This can be exploited for instruction-set generation, resulting in a leaner instruction set with the same speedup. Moreover, ID graphs can be used during code generation to increase the number of opportunities for the use of specialized instructions, resulting in faster code. Another application could be data-path sharing in logic synthesis.

Our experiments with DSP benchmarks show that a search-graph library can be constructed in roughly the same time as a simple linked-list library. The resulting library comprises eight times as many patterns. However, the search of a pattern in this large library is on average more than eight times as fast. For more details on the properties and use of ID graphs, the reader is referred to [1].

References

[1] Gero Dittmann. Organizing pattern libraries for ASIP design. Technical Report RZ3488, IBM Research, `www.zurich.ibm.com/~ged/`, April 2003.