RZ 3488 (# 99303) 04/21/03
Electrical Engineering 7 pages

Research Report

Organizing Pattern Libraries for ASIP Design

Gero Dittmann

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
ged@zurich.ibm.com

Organizing Pattern Libraries for ASIP Design

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

Abstract

In this paper we propose a new method to arrange a library of application-graph patterns. Such libraries are employed 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 $O(n \cdot p)$ with n the total number of operation nodes of all patterns in the library and p the size of the pattern sought.

Our new method employs identity operations to organize a library in such a way that a directed search strategy with only $O(d)$ and $d \leq p$ is possible. Furthermore, the organization reveals synergies between patterns for the ASIP design process and for 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, 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 which 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.

In current approaches, the pattern libraries are unordered collections of patterns—provided the library organization has been described at all. 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 with a number of heuristics already built in to keep library size low.

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, the library organization reveals opportunities to substitute one pattern by another. This may 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).

The remainder of this paper is structured as follows: In Section 2 we refer to related work on instruction-set generation. In Section 3 we introduce our concept of an identity graph and its use for a novel library organization. Section 4 presents a search algorithm, and Section 5 an insertion algorithm for identity-graph-based libraries. Bounds of the library size are derived analytically in Section 6 and compared with conventional libraries. We conclude the paper with Section 7 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 which share at least one operand. Applications are not represented by the compiler output directly but 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. From this list, patterns are manually selected, grouped, and implemented.

The library-construction algorithm 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. We conclude that 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 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 schedule 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 is not described in the paper.

Identity operations, which we use to find relations between patterns, have been exploited for basic algebraic transformations in compilers [5] and in high-level synthesis [6]. In [7] and [8], identity operations are *inserted* into sequences of operation nodes in order to increase the number of identical patterns. We go the opposite way by using identity elements to *eliminate* nodes from patterns. As a side effect, however, the library we construct in this way reveals the same opportunities to substitute one pattern by another. Moreover, our approach is not constrained to small sequential patterns.

3 Organizing Libraries as Identity Graphs

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 op_id, i.e. the algebraic identity element for that operator, to the other input such that

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

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
$\cdot$	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*, respectively.

A complex pattern can be transformed into a simpler pattern by applying the identity operands of its operation nodes to the appropriate inputs, thus 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 leafs or right-cyclops nodes to be removable, i.e., their right input must be accessible from outside the pattern.

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 operatations includes all operation types that occur in the pattern. The primitive operations finally all converge to a *move* operation.

If all patterns generated this way are entered into the pattern library then the sequence of applying the identity operands can be used to sort the patterns in the library. 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 graph (ID graph)*. Figure 1 shows an example ID-graph of a pattern from an application that parses headers of network packets [9].

The library ID-graph 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 that the possibly slower execution and the cost of applying the identity operands can be afforded. This cost may be, for instance, additional *move* instructions.

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 $C(pattern)$ associated with the patterns that usually increases with pattern complexity, capturing for instance operand encoding effort, die area, or latency. This cost function balances the derived tendency towards more complex patterns for implementation.

4 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.

The following is the pseudo code of a recursive algorithm that implements the proposed search strategy. It traverses the pattern sought depth-first and right-branch-first. It returns either the position of the pattern in the library or NULL.

```
patternInLibrary = find (patternRoot,
                         libLevel0)

libNode find (patNode, libNode) {
 libNode =
   libNode.next[patNode.operandNumber,
                patNode.operator]
 if patNode.rightOperand != NULL then
   libNode = find (patNode.rightChild,
                   libNode)
 if patNode.leftOperand != NULL then
   libNode = find (patNode.leftChild,
                   libNode)
 return libNode
}
```

Figure 2 shows the graph for the pattern in Figure 1 with only the reverse edges that are required for the search algorithm. We call this an ID-based search graph.

In order to search this graph for, e.g., the pattern in the middle consisting of a right shift followed by a subtraction, 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-`

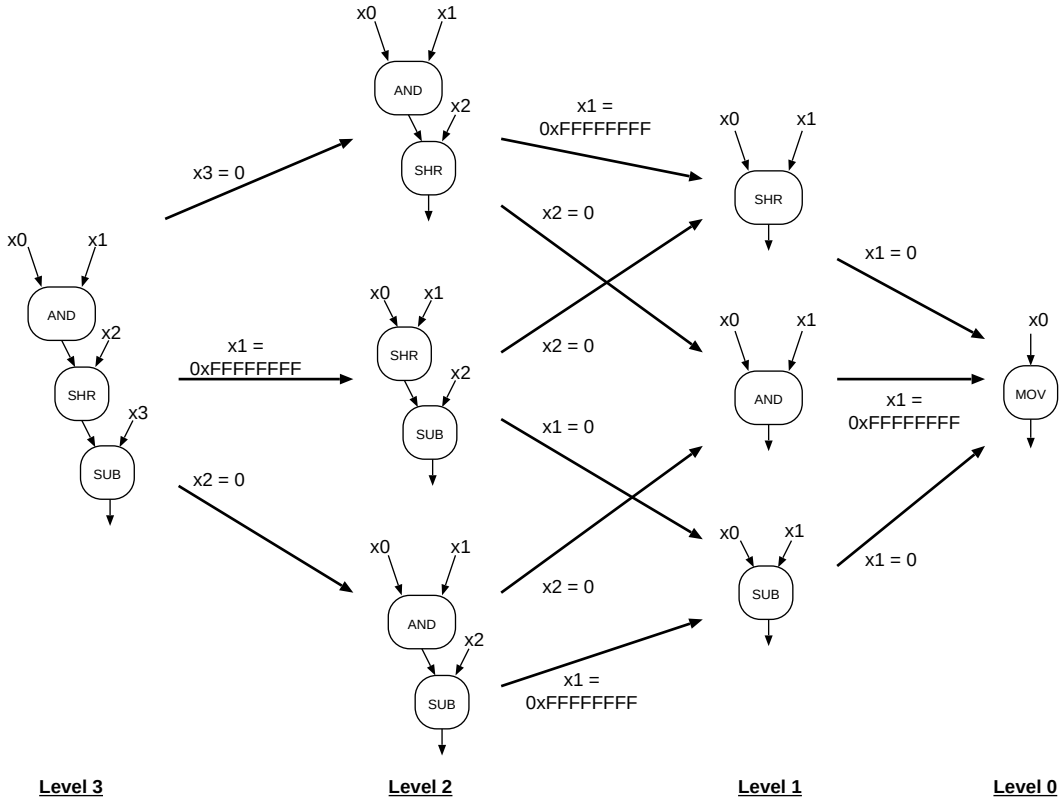

Figure 1: ID-Graph of a Pattern

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 a *SHR* operator and hereby sets `libNode` to the library entry we have been seeking. Both, the left and 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. 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 then 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 $d \leq p$.

5 Inserting Patterns into a Library

To insert a pattern into a pattern library that is organized as an ID-based search graph, we construct the ID graph of the new pattern and derive the corresponding search graph. Then, for each path in this search graph, we try to find the corresponding path in the library. If a path does not exist in its entirety in the library then the part existing has to be connected to the remainder of the path in the search graph of the new pattern. Those parts of the

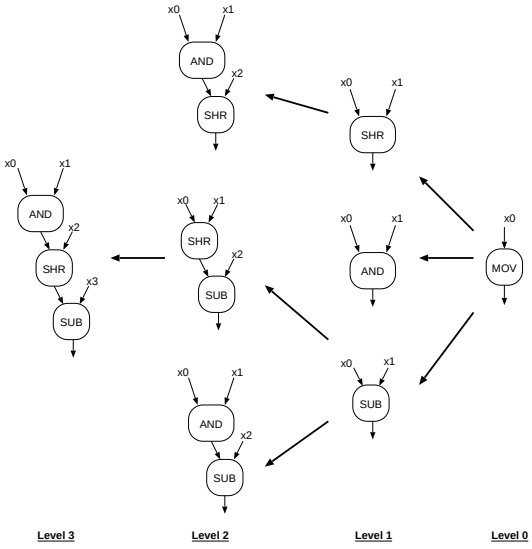

Figure 2: ID-Based Search Graph

new search graph that already existed in the library are deleted. A pseudo code of this algorithm is as follows:

```
newRoot = makeSearchGraph(newPattern)
insert (newRoot, libraryRoot)

insert (sourceNode, destNode) {
  for each operand {
    for each operator {
      if sourceNode.next[operand,operator]
         != NULL then {
        if destNode.next[operand,operator]
           == NULL then
          destNode.next[operand,operator]
          = sourceNode.next
            [operand,operator]
        else {
          insert (sourceNode.next
            [operand,operator],
            destNode.next
            [operand,operator])
          delete (sourceNode)
        }
      }
    }
  }
}
```

This *insert* function considers the possibilities of growing the patterns from one level of the search graph to the next. It compares these possibilities in the new search graph with the library and links those to the library that are missing. The number of possibilities, which is also the number of *next*-pointers stored with a pattern, corresponds to the number of possible continuations of a search for larger patterns: the number of external left operands on the top left fringe of the pattern, plus one right operand, times the number of primitive operators in the library.

$$\#_{\mathrm{NextPointers}} = (Operands_{\mathrm{topleft}} + 1)Primitives$$

Only the top right operand of a pattern has to be considered because the other right operands have been handled at lower levels of the search graph. The same is true for left operands that are not in the top left fringe. In the worst case, all left operands in a pattern are on the top left fringe. Then the number of next-pointers for this pattern with p nodes is $(p+1) \cdot Primitives$. Note that pointers that are NULL still have to be stored because they are a termination condition of the recursion in the search and insert algorithms.

We can provide a coarse estimate of the computational complexity of the *insert* function when we assume the derived worst-case number of possibilities of $(p+1) \cdot Primitives$ for each pattern in the search graph. The number of primitive operations is constant. Then the algorithm is $O(s \cdot p)$, with p the number of operation nodes in the inserted pattern and s the number of patterns in its search graph.

6 Library Size

6.1 Analytical Bounds

The patterns with the largest ID graphs are those for which each operation node can be eliminated by its ID operand. This is the case for patterns that consist only of right-cyclops nodes with right identity operands, such as the pattern in Figure 1. These patterns form a sequence of nodes, each of which obtains its left operand from a previous node—except for the first node—and that provides its result as the left operand to the following node—except for the root node. All right operands are pattern-external inputs through which the ID operands can be applied to eliminate any node at any level of the ID graph. For the worst-case library size, we consider patterns of this kind in the following.

The derivation of patterns from a parent pattern can be formulated as a combinatorial problem at each ID-graph level: On level k, how many different ways are

there to choose k nodes out of the n nodes in the parent pattern? Hence, a pattern of n *different* right-cyclops nodes generates a graph of

$$\sum_{k=0}^{n} \binom{n}{k}$$

patterns through identity-operand transformations. This includes the parent pattern itself, the primitive operations, and the final *move* node. For the first pattern to be inserted into an empty library, this is also the number of new patterns for the library.

When a parent pattern that is being inserted has offspring patterns that are already present in the library then the number of new patterns that the parent introduces into the library is accordingly lower. Each ID subgraph that the library and the ID graph of the new parent pattern have in common comprises the merge pattern where both graphs meet, and its complete cone of ID transformations down to the final *move* operation. For a merge pattern that comprises m right-cyclops nodes out of the n nodes of the parent, the number of additional patterns that are introduced to the library by the parent pattern is only

$$\sum_{k=0}^{n} \binom{n}{k} - \sum_{i=0}^{m} \binom{m}{i}.$$

If the library and the parent have more than one merge pattern then the ID graphs of the merge patterns may overlap. In this case, the overlapping region must be subtracted only once from the contribution of the parent to the library. For a parent that has two merge patterns with the library, comprising m_1 and m_2 nodes, respectively, and the ID graphs of the two merge patterns merging at a pattern of m_3 nodes, the contribution of the parent to the library is computed by

$$\sum_{k=0}^{n} \binom{n}{k} - \left(\sum_{i=0}^{m_1} \binom{m_1}{i} + \sum_{j=0}^{m_2} \binom{m_2}{j} - \sum_{g=0}^{m_3} \binom{m_3}{g} \right).$$

The patterns with the smallest ID graphs are those for which only one node can be removed at each transformation level. This is true for patterns that consist only of left-cyclops nodes of non-commutative operators. In these patterns, the only removable node is the leaf node because it is the only node with a pattern-external left operand. The ID graph of such a parent has only one pattern at each level, namely, the pattern at one level higher without the leaf node. If the parent comprises n operation nodes its ID graph will consists of n patterns and the *move* node.

Patterns that comprise multiple instances of the same operation will result in smaller ID graphs as redundant child patterns will occur only once in the ID graph. Each duplicated operation node results in one primitive node fewer on level 1 of the ID graph. Furthermore, duplicated operations will probably also result in redundant patterns on higher ID-graph levels. Each of these redundancies reduces the number of patterns in the ID graph.

How many patterns a library will ultimately incorporate strongly depends on the properties of the parent patterns that have been inserted and therefore cannot be derived analytically.

6.2 Comparison with Unordered Libraries

An unordered pattern library only comprises parent patterns and their sub-graphs. Many offspring patterns in ID graphs are also sub-graphs of the parent pattern. They would have been added to the library by a conventional library-construction algorithm as well. But there are other patterns in an ID graph that are not sub-graphs of the parent and that, compared with conventional libraries,therefore constitute an overhead.

For each transformation step from a parent pattern to primitive patterns, a simpler pattern is a sub-graph of its parent if the operation node eliminated was a leaf or a root node, i.e., the eliminated node had only pattern-external inputs or its only output was a pattern-external output. Eliminating other nodes, i.e. cyclops nodes, always leads to connecting previously unconnected nodes. This new connection cannot occur in sub-graphs of the parent pattern. Hence, compared with an unordered pattern library, patterns that incorporate such a connection represent the overhead of an ID graph.

The number of patterns in the ID graph that are sub-graphs of the parent is equal to the sum of leaf nodes and removable root nodes of the parent pattern and of all its generated sub-graphs. If a leaf that is being eliminated from a pattern feeds into a cyclops node then the child pattern generated has the same number of leaves. If the leaf feeds into an internal node then the child has one leaf fewer than its parent.

The patterns with the highest number of children that are not sub-graphs of the parent, i.e. the patterns with the largest ID-graph overhead compared with unordered pattern libraries, are again patterns of only right-cyclops

nodes and a single leaf node, such as the pattern in Figure 1. Each ID-graph level from the parent pattern to level 1 has one sub-graph more than the previous level—starting with one in the highest level. Subtracting the total number of sub-graphs from the total number of patterns in the ID graph of a pattern, minus the final move operation, which will not be represented in a practical library, results in the following formula for the worst-case ID-graph overhead for a parent pattern of n nodes:

$$\sum_{k=0}^{n} \binom{n}{k} - 1 - \sum_{i=1}^{n} i = \sum_{k=1}^{n} \binom{n}{k} - k$$

In practice, inserted patterns will have a significant number of internal nodes that cannot be eliminated. Moreover, they will comprise redundant sub-graphs that are inserted into the library only once. Consequently, the overhead of such patterns is significantly lower than in the worst case given here.

7 Conclusions and Future Work

In this paper we have presented a novel method to organize libraries of DFG patterns as ID graphs. Compared with conventional unordered libraries, ID graphs enable more efficient searches with a computational complexity of $O(d)$ instead of $O(n \cdot p)$ with $d \leq p$. Because this eliminates the dependency between computational complexity and library size, large pattern libraries can be handled. Given the memory sizes of today's workstations, the need for heuristics that exclude less promising patterns is significantly reduced, and exact methods become possible. Furthermore, ID graphs reveal opportunities to substitute patterns by others. This can be exploited for instruction-set generation and code generation.

We have presented our method for operation nodes with only one output, i.e., for tree-shaped patterns. The next step in developing the method will be to extend it for any kind of directed acyclic graph (DAG). We plan to achieve this by introducing a *visited*-flag for pattern nodes to keep the search algorithm from searching the same node twice while traversing a pattern.

Furthermore, we will 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.

Finally, we will explore the possibility to substitute the pattern-finding algorithms that feed a pattern library by inserting the complete DFG of each basic block into an ID-graph 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 ID operands.

References

[1] Gero Dittmann and Andreas Herkersdorf. Multilayer 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] Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, San Francisco, 1997.

[6] 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.

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

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

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