Designing Domain-Specific Processors

Marnix Arnold
Delft University of Technology
Department of Electrical Engineering
marnix@cardit.et.tudelft.nl

Henk Corporaal
IMEC Leuven
heco@imec.be

ABSTRACT

We present a semi-automated method for the detection and exploitation of application domain specific instruction set extensions for embedded (VLIW) processors. It consists of three steps: the first step detects frequently occurring operation patterns, in the second step, the patterns are grouped and implemented in a number of Special Function Units (SFUs) and the third step incorporates the custom operations into the code generation process.

Experiments show that the SFUs generated and exploited with our methodology can result in architectures that perform up to 30% better than architectures of the same cost without SFUs.

Keywords

Instruction Set Synthesis, Design Space Exploration

1. INTRODUCTION

In recent years, the emphasis in the microprocessor market has increasingly been shifting from general-purpose CPUs for personal computers and workstations, to processors that have to perform only a limited number of tasks, meant to be embedded in various electronic systems. Examples of these embedded processors can be found all around us today: in mobile phones, cars, cameras, toys, etc. It is expected that the market for these embedded devices will only grow as the mobile voice and data networks continue to expand.

The research presented in this paper is aimed at tuning (embedded) processors towards their intended application domain. Figure 1 shows a number of different hardware/software systems, each with its own advantages and disadvantages. The systems on the left side of the figure tend to be more application-specific (efficient but inflexible), to the right they become more general-purpose (very flexible but also expensive). By designing an *Application Domain Specific Processor* (ADSP), we want to combine the efficiency of hardware with the flexibility of software.

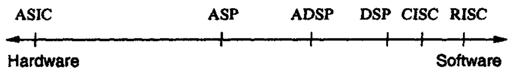

Figure 1: Hardware/software systems.

A popular[1] processor tuning approach, and also the one we will take, is to extend the operation repertoire of the processor to include application domain specific custom operations. Because we believe time-to-market to be a critical factor in the successful customization of embedded processors, we strive to automate this process as much as possible.

This paper addresses the problem of automatically finding the proper instruction set extensions for the intended application domain, as well as the problem of how to make use of these extensions. We will demonstrate a technique for the automatic detection of instruction set extensions on the basis of a set of benchmarks representative of the intended domain. We will also present a code generation technique that allows instruction-level parallel (VLIW) processors to make use of the extensions we found. Using this technique, we will perform a design space exploration in order to find a suitable processor configuration for the application domain.

Section 2 gives an overview of related work from the fields of instruction set synthesis, pattern matching and code generation with complex operations. In section 3, we describe our strategy for the automatic detection of opportunities for application (domain) specific instruction set extensions. A code generation strategy that combines instruction selection with scheduling is described in section 4. Experiments with the pattern construction method and the resulting Special Function Units are presented in section 5. In section 6, we present design space exploration results for searches with and without SFUs. A summary and conclusions are given in section 7.

2. RELATED WORK

The work presented in this article relates to the fields of pattern matching, instruction set synthesis and instruction selection/code generation. Matching and covering algorithms are well-known in the fields of code generation and logic synthesis. Pattern matchers used in logic synthesis ([2], [3]) and compilers ([4], [5]) generally require the pattern graphs to be trees (single-output, acyclic, non-reconvergent graphs).

The study described in [6] involves the search for chainable operation sequences, in order to find instruction set extensions for application-specific instruction set processors

(ASIPs). An earlier version of some of the work presented in this paper can be found in [7]. In [8], the performance potential of data dependence collapsing for sequences of operations was studied in the context of a superscalar architecture.

The work presented in [9] derives the best instruction set on a fixed data path for a set of benchmarks representative for an application domain. An integrated instruction set synthesis and code generation tool was described in [10], which also focuses on a fixed, single-issue, pipelined processor.

In [11], instruction selection with complex operations is treated as a binate covering problem. Exact solutions can be found (for small basic blocks). Liem e.a. [12] use matching and covering techniques to identify recurring instances of hybrid operation and control flow patterns, in order to perform instruction selection for DSP and ASIP architectures. Another approach [13], aimed at code generation for DSPs, operates by partially postponing instruction selection until the compaction phase, in order to exploit pipelined, complex operations.

The methodology we present in this paper distinguishes itself from the related work in several ways. The matching algorithm we will use [14] is different from existing methods ([2], [3], [4], [5], [11]) in that it allows both the subject and pattern graphs to be DAGs with any number of outputs. This allows us to match operation patterns of any shape as long as they are acyclic. The algorithm constructs its own pattern library, rather than requiring a predefined one [12], by combining existing patterns as they are encountered in the subject graph. It is able to construct operation patterns of any (acyclic) shape, rather than just sequences [6]. Our instruction selection and scheduling algorithm is aimed at exploiting custom operations for instruction-level parallel architectures, whereas current efforts ([11], [12], [13]) focus primarily on single issue DSP architectures.

3. FINDING INSTRUCTION SET EXTENSIONS

We will describe an automated technique to identify frequently occurring operation patterns in a set of applications. By implementing these patterns in hardware, as special operations, we hope to be able to increase the efficiency with which the embedded processor can execute these applications. The general approach we will take is as follows:

1. Generate execution traces for a set of benchmarks representative of the application domain we are designing the embedded processor for.
2. In these traces, identify and isolate frequently occurring patterns of operations.
3. Evaluate the most frequently occurring operation patterns in terms of how useful it would be to implement them in hardware (as custom operations).

The execution traces are generated using the simulator from the Move compiler suite [15], which simulates the execution of (sequential) code as it is generated by the compiler front-end. The operations that the simulator executes are used to generate a (dynamic) execution trace in the form of a data-dependence graph to expose a large amount of available instruction level parallelism. This way, we can also detect patterns and their frequency counts across control flow boundaries.

For the second step, the automatic detection of recurring patterns of operations, a new matching algorithm was developed. This was necessary in order to overcome several limitations posed by existing matching algorithms, most notably the restriction that pattern graphs are only allowed to have one output. This restriction would make it impossible to detect opportunities for multi-output custom operations. In addition, the algorithm has been extended to allow on-the-fly construction of the pattern library, adding operation patterns as they are found in the trace. Conventional methods such as the one used in [12] require the designer to supply a predefined library of operation patterns, which means that the designer must know beforehand which patterns to expect.

An in-depth description of our pattern detection and construction algorithm is beyond the scope of this paper. Instead, we will illustrate the basic concepts by means of two examples in sections 3.1 and 3.2. A detailed description of the algorithm can be found in [14].

The third step involves gaining an understanding of how often a custom operation would be used during code generation. This is described briefly in section 3.3.

3.1 The Matching Algorithm

Given a subject graph G_{sub}, consisting of the operations and operands in an execution trace, we must find all matches of pattern graph G_{pat} in G_{sub}. Our matching algorithm operates by finding *partial matches* between individual subject and pattern operations, and then merging these until complete pattern matches are found. Figure 2 illustrates the concept of partial matches. Operation I matches operation n_1 and a partial match m_1 is created. This match consists of a vector of references to subject graph nodes, where the position in the vector of each reference indicates which pattern graph node it corresponds to. The rest of the entries in the match vector remain empty. In a similar fashion, partial match m_2 is constructed for operation II and n_2. Combining partial matches m_1 and m_2 around a shared *pivot* operand yields a complete match m_3 for pattern G_{pat}.

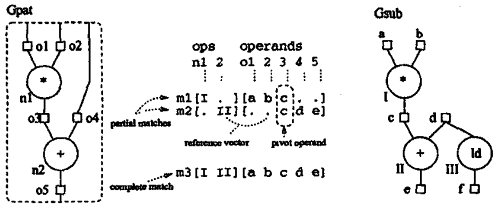

Figure 2: Partial matches merged into a complete match.

3.2 Pattern Library Construction

We start pattern library construction with a subject graph G_{sub} and a collection of single-operation patterns $PatLib = \{G_{pat_i}\}$, as shown in figure 3. Each time we finish detecting matches on an operation node, we start looking for opportunities to create new patterns.

After we finish matching on operations I and II of G_{sub},

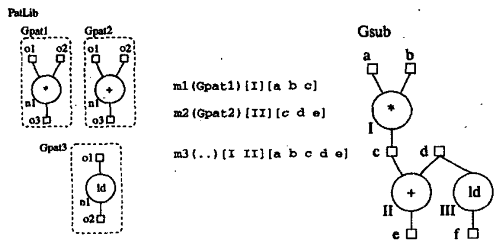

Figure 3: Pattern construction by combining matches.

we see that two matches m_1 and m_2 both contain a reference to operand node *c*. Note that now the matches refer to different patterns, which was not the case in the example of figure 2. If we combine the matches m_1 and m_2 into a new match m_3, we have a recipe for constructing a new pattern! By copying all the nodes (operations and operands) referenced in m_3 into a new pattern and adding it to the library, we will be able to log all future occurrences of this pattern.

3.3 Trace Covering

It is misleading to use the number of detected matches for each pattern as a measure of its usefulness as an instruction set extension. It has been observed [14] that some classes of patterns result in an enormous number of overlapping matches, from which only a few can be chosen for a cover. For this reason, we want to make a selection of matches, such that all subject operations are covered by one match. Selecting such a set of matches is called covering. In this paper we attempt to minimize the number of matches chosen for the cover, since that may give an indication which patterns could be useful in minimizing the code size when we are generating code.

The algorithm we use to select a cover is a variation on dynamic programming[16], an approach often used in compilers. It operates by recursively walking each path in the subject graph from its outputs upwards, until it meets either an already covered node, or a subject graph input. On each operand node it comes across, it determines the lowest implementation cost in much the same way that dynamic programming handles the minimum-cost covering problem on trees. Since we have to deal with non-tree graphs, however, some modifications have been made.

The calculation of the implementation cost of a match happens relative to the current subject operand node. In figure 2, the cost of a single operation match on operation *III*, relative to *f*, would be the cost of the match itself, plus the implementation cost at the input *d*. However, a pattern supporting a post-increment load (*II* and *III*) would be much more expensive since it is both larger and has a larger fan-in. Instead, we only consider the part of the match that is *relevant* to *f*. The implementation cost of the *II-III* match itself is divided across its outputs, meaning that for *f* only half the match cost is considered. Similarly, we divide the cost at its inputs by the number of uses of that input, so that for *d*, only half the implementation cost is considered.

4. CODE GENERATION WITH INSTRUCTION SET EXTENSIONS

Now that we have a method for finding a set of candidates for instruction set extensions, we also need a way to take advantage of these custom operations in the code that is generated for our target applications. The context in which we will generate code is an instruction-level parallel list scheduler, based on the Move compiler tools [15]. This scheduler is operation-based (as opposed to cycle-based): operations are scheduled in sequence, each time the *ready* (i.e, without unscheduled predecessors) operation with the highest priority is chosen for scheduling. As one of the priority measures, the *slack* of an operation is taken, that is, the difference between the last (ALAP) and first (ASAP) cycle in which it can be scheduled, taking only data dependency constraints into account (ignoring resource constraints). The lower the slack, the higher the operation's scheduling priority.

There are several strategies we can use to introduce custom operations into the scheduling process. The one we will use, described in this section, is based on the trace covering method introduced in section 3.3 and injects custom operations into the code before it is scheduled. Other strategies (described in [14]) introduce custom operations into the code *during* scheduling. However, research has shown that within the basic block scheduling scope that we will use, the method described in the following paragraphs yields the best results.

The most straightforward way to include the use of custom operations in the code generation process is to modify the data dependence graph (DDG) before presenting it to the scheduler. The DDG is in some ways similar to the execution traces we performed trace covering on. It contains operations (nodes) that use the result values of other operations as operands and are thus dependent (connected by data dependence edges). If we represent our custom operations in terms of patterns of *basic*[1] operations, we can use the matching algorithm from section 3.1 to detect all instantiation possibilities of our *custom* operations in the DDG. Each operation node will be contained in at least one match (the *default* match, with the basic operation itself). Any other matches containing the operation node provide an implementation of the operation node as part of a custom operation. We can then use the trace covering algorithm to perform instruction selection, i.e, to make a selection of matches, covering the DDG, such that each operation node is contained in exactly one match. The matches chosen for the cover are subsequently *implemented* by replacing the basic operations contained in each complex pattern match with the corresponding custom operation. Figure 4 illustrates this: in 4(a), all matches $m_{1...5}$ on the DDG fragment are shown, in (b) a selection of matches (cover) is made, containing only m_2 and m_4, and in (c) the custom operations corresponding to the matches from the cover are implemented in the graph.

Our code generation strategy attempts to find the optimal cover within each basic block that minimizes the number of selected matches. Fewer matches translates to fewer operations for the schedule, and it is expected that the increased scheduling freedom leads to better (i.e, shorter) schedules.

[1]I.e, only operations from the set that was used by the compiler front-end to generate the unscheduled, sequential code.

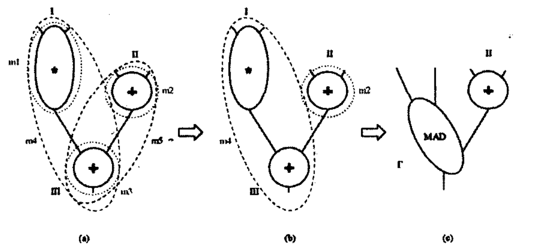

Figure 4: Matching, covering and match implementation on a DDG fragment.

5. EXPERIMENTS

In the previous sections, we presented a method for finding frequently occurring patterns of operations. We have used this method to find a set of custom operations for a set of benchmark applications from the digital signal processing domain. The approach taken has two steps:

1. We determine a set of frequently occurring operation patterns from the execution traces of all of the benchmarks.
2. Using the library of detected patterns and their matching and covering contributions as a guide, we construct Special Function Units (SFUs) and a testbed architecture.

An overview of the benchmark applications we will use throughout the experiments is given in section 5.1. The results for the pattern library construction experiment are presented and analyzed in section 5.2. The formation of SFUs is described in section 5.3.

5.1 Benchmark Applications

For our experiments, we use a number of common benchmarks from the digital signal processing domain. Frequently used for multimedia processing, these algorithms are good candidates for implementation on an embedded processor. It must be noted, however, that our method is not limited to finding processor optimizations for this set of benchmarks only. They are merely used to illustrate the generally applicable methodology for embedded processor customization described in this paper. An overview of the benchmarks, with their static and dynamic operation counts, is given in table 1.

Name	Description	#ops (stat)	#ops (dyn)
bspline	FIR Filter	31	6149
compress	Compression (dct 2d)	611	163513
dft	Discrete FFT	39	6666
edge	Edge detection	440	268717
expand	Decompression (idct 2d)	464	151083
fir	35 pt. Lowpass FIR	119	30459
flatten	Level histogram of image	148	33960
foewf	5th Order Elliptic Wave	43	13067
iir	IIR highpass filter	134	10794
intfft	Interpolate with FFT/IFFT	571	188421
pse	Sehwa's FIR filter	30	6917
smooth	Convolution w. 3x3 kernel	135	83365

Table 1: DSP benchmarks.

5.2 Pattern Construction Results

For the purposes of this paper, we have limited the number of operations per constructed pattern to three, mainly in order to reduce execution time and memory requirements, and because it is expected that larger patterns would be less generally applicable (as was shown in [14]).

The total number of (unique) new patterns found for all the benchmarks is 656. Figure 5 shows the contribution to the total number of matches vs. the contribution to the (trace)cover for each of the new patterns. All numbers are averages across all applications.

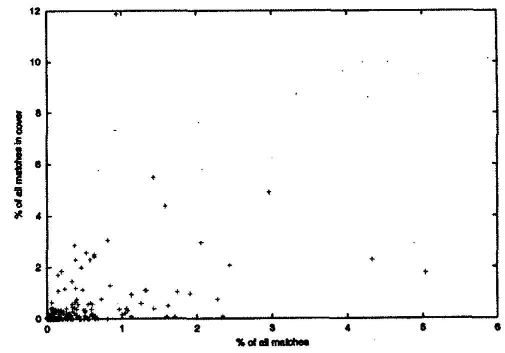

Figure 5: Match contribution vs. cover contribution for all patterns.

The first thing to observe about figure 5 is that most of the new patterns are not that often encountered or used in covers. A few patterns account for most of the matches and most of the matches chosen for covers. This indicates that a few, well-chosen custom operations can already yield large benefits.

We can now sort the library with the new patterns according to the sum of their matching contribution (position on the horizontal axis of figure 5) and their cover contribution (vertical axis). Looking only at the match contribution would be misleading, since not all matches are equally likely to be chosen for the cover. This explains why a pattern that was matched often (a high value on the horizontal axis of figure 5) is not necessarily chosen for the cover as frequently. Looking only at the cover contributions of patterns would be misleading as well: the choice of matches for the cover is made by an algorithm based on dynamic programming, and these tend to have a preference for chain-like patterns (i.e, sequential rather than parallel). By sorting by the sum of the matching and covering contributions, we get a mix of patterns with a high covering contribution (which are likely to have a good matching contribution, too) and patterns that have a high match count but were somehow not chosen by the covering algorithm.

5.3 Special Function Unit Construction

Since it is impossible to implement all patterns as custom operations, we have selected 40 patterns from the top-100 in order to create five Special Function Units (SFUs). No detailed hardware was designed, but the patterns were grouped into categories that are likely to map well onto the same hardware. Because this is difficult to do automatically it was done by hand.

The ADDSFU executes 12 patterns of up to three integer addition or compare operations. In this respect it is an extension of the 3-1 interlock collapsing ALU [17], which executes two chained integer operations as one. The most important difference, apart from the pattern size, is that our SFU also executes patterns with operations executing in parallel (with a shared operand).

The MEMSFU is a load/store unit that also performs address computations. It supports patterns of up to two integer additions and one memory operation.

The FPSFU supports patterns of up to three floating operations, at most one multiplication and up to two additions.

The MULSFU supports patterns of one integer multiplication followed by up to two integer additions.

The ASHSFU supports one integer addition or compare in sequence with, or parallel to a left shift operation.

In addition to the complex operations, all SFUs also support the individual, atomic operations that make up their respective patterns, as well as related atomic operations that do not appear in the patterns (e.g, although the floating-point subtract operation did not appear in any pattern, it is supported by the FPSFU). This is required in order to maintain full instruction set compatibility, or in the terms of section 4, to always allow implementation of any default match.

The latencies of the SFU are, as a first approximation, taken to be equal to the latencies of related basic FUs. Literature suggest this is a reasonable assumption in many cases. For instance, a chain of two integer additions can be collapsed into a single-cycle operation [17], an integer multiply-add takes the same time as an integer multiply in most DSP architectures and floating point multiply-add implementations (e.g, [18]) have been designed that have the same latency as a floating-point multiplication. In [19], the *SAM* or Sum-Addressed Memory technique is described. It performs base+offset (load address) calculations using the decoder of the RAM array, with very low latency, effectively combining the address calculation and the actual load in a single operation. The technique has been successfully implemented in the Ultrasparc III microprocessor.

In some other cases, however, the equal-latency assumption is bound to be overly optimistic. Unfortunately, the current Move software generation tools do not allow this latency to be specified on a per-operation basis: each FU has a single, fixed latency which applies to all the operations it supports. This may cause some distortion in the results presented in the next section, particularly where execution times are concerned.

6. DESIGN SPACE EXPLORATION

Now that a set of SFUs has been defined, we want to apply the automatic design space exploration method, described in chapter 7 of [15], to the problem of finding a suitable processor architecture for our application domain. In addition, the design space exploration process will let us analyze the added value of SFUs, relative to regular FUs. The initial, oversized architecture, which is used as the starting point of the design space exploration, contains both SFUs and regular FUs, and a number of transport buses. In the first phase of the design space exploration, all buses are connected to all (S)FUs.

The design space exploration process was performed twice, once with and once without making use of SFUs. Figure 6 shows the *Pareto curves*[16] for the abovementioned benchmark set. On the horizontal axis, the architectures' cost is shown (in μm^2), the vertical axis shows the total execution

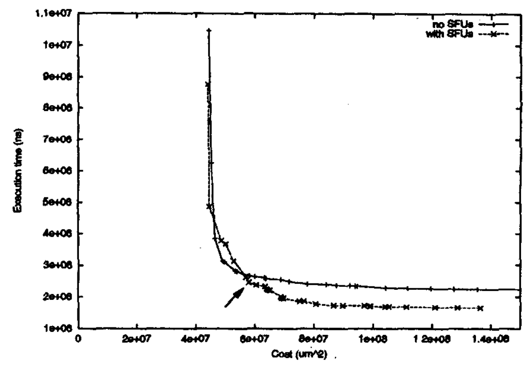

Figure 6: Design space exploration results.

time (in *ns*) of the benchmarks for the various architecture design points.

It can be seen in figure 6 that for higher architecture cost, the design points that include the SFUs are much more efficient than those with only the basic FUs (about 30% fewer cycles for the same cost). In the steepest part of the curve, the architectures consist of only the smallest set of (S)FUs necessary to execute the benchmarks. The only variation is in the number and type (width) of transport buses that are in the architectures. It can be seen here that the "with SFU" architectures have a slightly better performance vs. cost ratio that the "no SFU" architectures. This is due to the fact that the program storage requirements are lower for the code generated when making use of SFUs.

From the Pareto curve resulting from a design space exploration, the designer can choose an architecture configuration to perform *connectivity reduction* on. Connections between (S)FUs and transport buses are iteratively removed, which initially results in better results since the architecture's area and cycle time become smaller. This continues until the transport resources are limited to a degree where they constrain the scheduler too much and performance deteriorates.

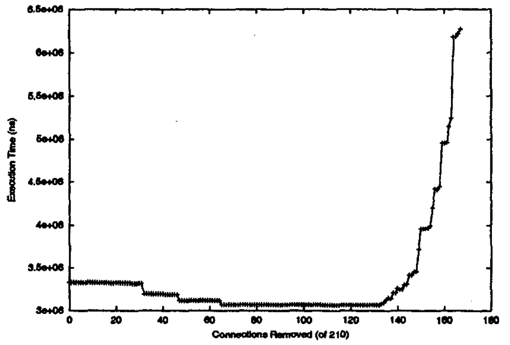

Figure 7: Connectivity reduction on a Pareto-point from figure 6.

Figure 7 shows the results for a connectivity reduction experiment performed on the architecture point marked with an arrow in figure 6. During the removal of the first 137 connections, the execution time (the product of the number of executed cycles and the cycle time) steadily decreases. This is due to the fact that the cycle time improves as the buses get shorter and have fewer connections (it changes from 27 to 25 ns during the removal of the first 137 connections). If more than 137 connections are removed, however, the remaining parallelism in the architecture is insufficient to generate efficient schedules. This means that the schedule length increases, which in turn causes an increase in the number of executed cycles. This increase is more rapid than the cycle time reduction, resulting in a higher execution time. It can also be seen in figure 7 that it is not possible to remove more than 167 connections. At this point, the minimal connectivity has been reached that is required to generate a working schedule.

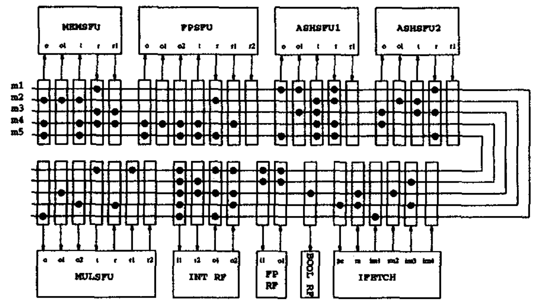

Figure 8: The architecture after removing 137 connections.

Figure 8 shows the architecture that results from the connectivity removal experiment, after 137 bus-socket connections have been removed. It has an area of 52 mm^2, a reduction of 6 mm^2. The most remarkable feature of the architecture is that there now are several SFU (result) sockets that are not connected to any buses. This does not mean that those units are incapacitated; it merely implies that there is no call for the more complex custom operations that use these result sockets. This in turn implies that an even cheaper architecture may be possible that will get the same performance as the one shown: by redesigning the SFUs without the unused, complex custom operations, a more efficient architecture can be designed. Note, however, that the experiments presented here all use basic block scheduling, it is very well possible that more advanced scheduling strategies with bigger scheduling scopes will be able to exploit the custom operations that have three outputs.

7. SUMMARY AND CONCLUSIONS

In this paper, we set out to automatically design application domain specific processors. We demonstrated our method for the automated detection and exploitation of instruction set extensions, which led to the specification of a number of Special Function Units (SFUs). These SFUs support a number of custom operations, which can be put to use by a special instruction selection and code generation strategy.

Design space exploration showed that using SFUs can have a beneficial effect on the cost vs. performance ratio of application domain specific processors. SFUs turn out to be an effective way to expand processors while maintaining flexibility and programmability.

8. REFERENCES

[1] Tom R. Halfhill. 1999 review: Embedded market breaks new ground. *Embedded Processor Watch*, 82, January 2000.

[2] K. Keutzer. Dagon: Technology binding and local optimization by dag matching. In *DAC, Proceedings of the Design Automation Conference*, pages 617–623, May 1987.

[3] R. Rudell. Logic synthesis for vlsi design. Technical Report UCB/ERL M89/49, University of California at Berkeley, April 1989.

[4] A.V. Aho, M. Ganapathi, and S.W.K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Programming Languages and Systems*, pages 491–516, October 1989.

[5] Christopher W. Fraser and Todd A. Proebsting. Custom instruction sets for code compression. *not published*, 1995.

[6] Frederick Onion, Alexandru Nicolau, and Nikil Dutt. Compiler Feedback in ASIP Design. Technical report, University of California, Irvine, September 1994.

[7] Marnix Arnold and Henk Corporaal. Automatic detection of recurring operation patterns. In *Seventh International Workshop on Hardware/Software Codesign*, pages 22–26, Rome, Italy, May 1999.

[8] Yiannakis Sazeides, Stamatis Vassiliadis, and James E. Smith. The performance potential of data dependence speculation & collapsing. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 238–247, Paris, France, December 1996.

[9] Bruce K. Holmer. A tool for processor instruction set design. In *EURO-DAC*, 1994.

[10] Ing-Jer Huang and Alvin M. Despain. Synthesis of instruction sets for pipelined microprocessors. In *DAC*, 1994.

[11] Stan Liao, Srinivas Devadas, Kurt Keutzer, and Steve Tjiang. Instruction selection using binate covering for code size optimization. In *International Conference on Computer-Aided Design*, pages 393–399, 1995.

[12] Clifford Liem, Trevor May, and Pierre Paulin. Instruction-set matching and selection for dsp and asip code generation. In *Proceedings of EDAC-ETC-EUROASIC*, pages 31–37, 1994.

[13] Rainer Leupers and Peter Marwedel. Instruction selection for embedded dsps with complex instructions. In *European Design Automation Conference (EURO-DAC)*, September 1996.

[14] Marnix Arnold. *Instruction Set Extension for Embedded Processors*. PhD thesis, Delft University of Technology, March 2001. ISBN 90-9014523-0.

[15] Jan Hoogerbrugge. *Code generation for Transport Triggered Architectures*. PhD thesis, Delft Univ. of Technology, February 1996. ISBN 90-9009002-9.

[16] Giovanni De Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill, 1994.

[17] S. Vassiliadis, J. Phillips, and B. Blaner. Interlock collapsing alus. *IEEE Transactions on Computers*, 42(7):825–839, July 1993.

[18] Troy N. Hicks, Richard E. Fry, and Paul E. Harvey. Power2 floating-point unit: Architecture and implementation. *http://www.austin.ibm.com/tech/fpu.html*, 1994.

[19] William L. Lynch, Gary Lauterbach, and Joseph I. Chamdani. Low load latency through sum-addressed memory (sam). In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 369–379, Barcelona, Spain, June 1998. ACM SIGARCH and IEEE Computer Society TCCA.