Integrated Approach to Optimized Code Generation for Heterogeneous-Register Architectures with Multiple Data-Memory Banks

Stefan Fröhlich and Bernhard Wess

Abstract— **This paper focuses on heterogeneous-register architectures with multiple data-memory banks. An evolutionary hybrid is introduced that combines evolutionary optimization strategies with tree techniques and list scheduling. It minimizes the execution time of the final code by jointly optimizing the schedule, selected instructions, allocated registers and data memory banks. The core of the proposed technique is a linear-time algorithm translating expression trees into optimal straight-line code segments. Typically, the proposed technique executes an order of magnitude faster than pure genetic implementations and achieves better results than with successively applied greedy techniques for the individual code generation steps. The proposed technique is well-suited to applications with stringent timing constraints.**

I. Introduction

Mapping digital signal processing algorithms to optimal programs for digital signal processors (DSPs) represents a complex combinatorial optimization problem. Even very simple DSP applications result in an extremely large solution space. Code generation consists of several highly-interdependent optimization problems which typically belong to class NP including operation scheduling, instruction selection, data register allocation, data memory bank allocation, code compaction, data memory layout generation, and address register allocation.

Typical DSPs have irregular data paths with different functional units and dedicated registers. Commonly, data memory is partitioned into multiple banks to increase memory bandwidth. Memory accesses may occur in parallel if the referenced variables belong to different memory banks and the registers involved are allocated according to a strict set of rules. Traditional compiler techniques can hardly cope with the irregularities of DSP architectures [1]. As a consequence, available DSP compilers produce code of unacceptable quality for most real-time applications. Often, the only alternative is to write programs manually in assembly code which is both time-consuming and error-prone.

A large number of signal processing algorithms (e.g. linear time-invariant systems) can be specified by directed acyclic graphs (DAGs). Likewise, basic blocks in high-level language programs (e.g. C) can be represented by DAGs. In this paper, we introduce a new technique that translates DAGs into optimized programs for heterogeneous memory-register architectures with multiple data memory banks. Our technique minimizes the execution time of the compacted code by jointly optimizing the schedule, selected instructions, and allocated registers and memory banks. The core of our approach is a linear-time algorithm that translates expression trees into code segments satisfying a set of boundary conditions for the tree interface variables. A significant advantage of our code generation approach is the fact that interdependencies of the code generation phases are taken into account.

INTHFT, Vienna University of Technology, Gusshausstrasse 25/389, A-1040 Vienna, Austria. Phone: +43 1 58801 38901, Fax: +43 1 58801 38999

The paper is organized as follows. In section II, there is a short discussion of related work. An overview of the proposed technique is given in section III. Section III-A explains the evolutionary part of the algorithm, sections III-B to III-D cover the deterministic optimization procedures, divided into tree evaluation, loop optimization, and code compaction. In section IV, experimental results are discussed.

II. Related Work

Unfortunately, optimal code generation for DAGs is NP hard even for very simple target architectures [2]. In contrast, expression trees can be translated into optimal straight-line programs with linear complexity [3], [4], [5]. Thus, it seems reasonable to split DAGs into expression trees and to translate them separately. In a second step, all trees are translated separately by an optimal tree algorithm. Several techniques have been proposed that aim at reducing the overall instruction costs for the data transfers between the trees [6], [7], [8]. These techniques have in common that the value of each tree is always transferred to memory and there is at most one opportunity for another tree to use this value without reloading from memory. In general, however, further optimization allows more data transfers to be saved.

A severe drawback of code generation in separate steps is the fact that all interdependencies of the code generation phases are neglected. All local optimized code segments are generated independently and therefore no joint optimization of the tree code and the data transfers between these segments takes place. Both register allocation and memory bank allocation strongly affect the compaction result and consequently should not be done in separate steps.

For the specific case of regular data paths, tree pattern matching has been applied to generate code for data flow graphs [9]. Recently, two papers on optimized DAG translation have been published [10], [11]. However, the procedures of both papers only produce straight-line code. Thus, code compaction has to be performed in a separate step. Since code compaction strongly depends on register and memory bank allocation, we believe that the interactions of these tasks should be taken into account as proposed in this paper. In [12], an approach is presented that integrates register and memory bank allocation. This algorithm is based on labeling a constraint graph that represents the restrictions on register and memory allocation. The algorithm starts by compacting a given symbolic straight-line code. Constraint graph labeling is performed in a separate step and therefore register and memory bank allocation is decoupled from compaction. However, decoupling in general adversely affects code quality.

III. Proposed Technique

The proposed technique starts with splitting the DAG into maximum-sized expression trees. We ensure that an optimal program exists for each tree which computes the respective tree without storing intermediate results in memory. The data transfers between the trees occur via so-called *tree interface variables.* These variables may reside either in registers or memory. The objective of our technique is to minimize the overall costs for the compacted program by optimizing variable lifetimes, register allocation, and memory bank allocation. We propose a genetic hybrid that combines evolutionary optimization strategies with optimum tree techniques and greedy heuristics such as list scheduling. The result is an efficient technique that executes an order of magnitude faster than a pure genetic implementation and achieves better results than with successively applied greedy techniques for the individual code generation steps.

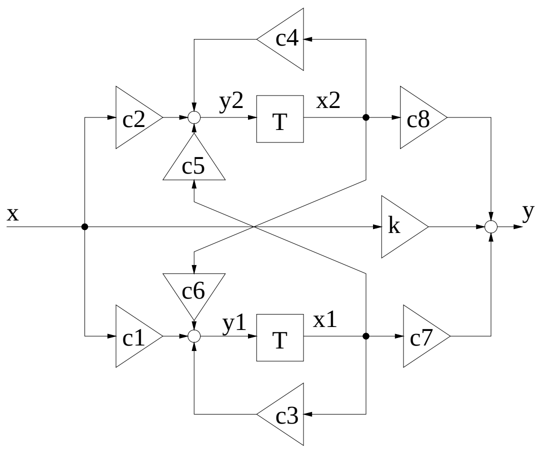

Fig. 1. DFG of a 2nd order state space filter

The examples in the discussion of our genetic hybrid refer to a 2nd order state space filter to be implemented on Motorola's DSP56k. The DFG is shown in Fig. 1 and the set of expressions is listed in Fig. 2 corresponding to five expression trees $T_1, T_2, \dots T_5$.

$$\begin{aligned}
&T_1\colon y = x_1 * c_7 + x_2 * c_8 + x * k\\
&T_2\colon y_1 = x_1 * c_3 + x_2 * c_6 + x * c_1\\
&T_3\colon y_2 = x_1 * c_5 + x_2 * c_4 + x * c_2\\
&T_4\colon x_1 = y_1\\
&T_5\colon x_2 = y_2
\end{aligned}$$

Fig. 2. Expressions for a 2nd order state space filter

A. Genetic Optimization

A.1 Chromosome Encoding

Each individual consists of a set of trees (in a dedicated order) and a set of tree interface variables. The tree schedule together with the attributes of the variables form the chromosome of an individual. The tree schedule can be expressed as a permutation of n elements where n denotes the number of trees. Information about access type (load, store, ...) and register or memory bank are assigned to each variable and stored in attributes. As shown in Fig. 3, this information may be displayed in a matrix constituting the chromosome of an individual. The lifetime periods of each variable can be derived from this matrix.

	T_1	T_2	T_3	T_4	T_5
y	*cs*				
y_1		*cs*		*lr*	
y_2			*ca*		*ur*
x_1	*la*	*ur*	*lr*	*cs*	
x_2	*lr*	*lr*	*lr*		*cs*
x	*la*	*ua*	*ur*		

Fig. 3. Attribute matrix

Each matrix element is encoded by three letters:
- $\{l|u|c\}$ defines whether the variable is *l*oaded from memory, *u*sed from a register, or *c*omputed.
- $\{a|r|s\}$ defines whether the variable remains *a*ctive in a register, is *r*eleased or *s*tored to memory.
- An optional last letter, not shown in Fig. 3, identifies the register or memory bank.

A.2 Selection

The selection mechanism picks random samples of the old population to create the next generation. The selection process is controlled by the fitness value of the individuals, those with a higher fitness value are more likely to be selected. Scaling is applied to the fitness values as the differences between the raw fitness values are too small to establish an efficient selection mechanism. The range of fitness values is mapped to the interval from 1 to p. p is called *selection pressure* and determines the factor between the scaled fitness of the best and of the worst individual.

A.3 Crossover

To create a new individual, two individuals are picked at random from the old population. Their chromosomes are mixed up by the crossover operator. This operator ensures that parts of the genetic information of the new individual are from one ancestor and the rest from the other one. By doing this, a new chromosome is created which may result in a better solution (or, as likely, in a worse one, but in this case the next selection process would weed out such an individual).

Separate crossover operators are implemented for modifying tree schedules, access types and register/memory bank assignments. After crossover, it is not guaranteed that the chromosome corresponds to a valid individual, so a repair algorithm is required to ensure that only valid individuals are generated.

A.4 Mutation

After an individual has been created and before it is added to the new population, a mutation operator is applied with certain probability. This operator randomly modifies one of the individual's attributes.

- *Attribute mutation*

The attribute mutation operator randomly chooses an element of the attribute matrix and modifies the attribute. The operator takes care that the changes are consistent and adjusts, if necessary, neighboring attributes. An example for attribute mutation is shown in Fig. 4. The attribute of x_2 in T_1 is changed from *lr* to *la*. For consistency, the attribute of x_2 in T_2 needs to be adjusted. As a result, the life period of x_2 is extended.

	T_1	T_2	T_3	T_4	T_5
y	cs				
y_1		cs		lr	
y_2			ca		ur
x_1	la	ur	lr	cs	
x_2	**la**	**ur**	lr		cs
x	la	ua	ur		

Fig. 4. Attribute mutation

• *Memory bank mutation*
The memory bank for a randomly chosen variable is changed. This kind of mutation can never be inconsistent.
• *Tree schedule mutation*
The evaluation order of two consecutive trees may be changed if no data dependencies are violated. This can be easily checked in the attribute matrix. An example for tree schedule mutation is given in Fig. 5.

	T_1	$\mathbf{T_3}$	$\mathbf{T_2}$	T_4	T_5
y	cs				
y_1			cs	lr	
y_2		ca			ur
x_1	la	**ur**	**lr**	cs	
x_2	lr	lr	lr		cs
x	la	**ua**	**ur**		

Fig. 5. Tree schedule mutation

A.5 Replacement Scheme

The child generation consists only of new items resulting from the successive application of selection, crossover, and mutation operators.

Optionally, the best individuals may be automatically selected into the next generation passing by the random selection mechanism. This concept is called elitism and it guarantees that good solutions cannot be lost due to the following random decisions. On the other hand, there is a higher risk of saturating at a local minimum too early when using elitism. However, no performance gain has been observed by applying elitism.

B. Tree Optimization

The core of the proposed procedure is an algorithm that takes the interdependencies of instruction selection, register allocation, and scheduling into account when translating expression trees into optimal contiguous sequential code for heterogeneous memory-register architectures. In contrast to existing code generation algorithms, our algorithm produces optimal code for expression trees that have leaves with predefined register assignments [11]. All code generation constraints for the individual trees are determined by the columns of the attribute matrix. Specifically, attributes are assigned to all tree interface nodes (root and leaves). Attributes determine whether registers for tree interface variables are released or remain active. Active registers restrict register resources and consequently constrain tree code generation.

Optimum contiguous code for an expression tree is generated by minimizing a cost vector for each node of the tree. This can be done in linear time by a single bottom-up traversal of the tree. For each cost vector entry, the optimum instruction together with the order of the operands and their cost vector indices are stored.Once the cost vector has been computed for each tree node, an optimum straight-line code can be emitted by traversing the tree a second time.

C. Loop Optimization

If assembly code is executed within a time-critical loop, variables should be kept in registers over the loop boundaries resulting in further savings of instruction cycles. This is done by modifying the attribute matrix accordingly. Without loop optimization, each register content is discarded the last time it is used and all variables are reloaded in the next iteration. By enabling loop optimization, registers maintain their contents after the last tree which may then be used in the next cycle.

Loop optimization requires modifications in the register assignment procedure. The same register must be provided at the beginning of the cycle as well as at the end if a value should be preserved. This raises boundary conditions which cannot always be satisfied. Individuals, for which no valid solution can be found, are assigned a fitness value of 0, so they do not have any chance to survive.

D. Code Compaction

Code compaction is an important step within DSP assembly code optimization, as the vast majority of DSPs support multiple operations to be executed simultaneously. As an example, Motorola's DSP56k family allows to execute up to two data transfers in parallel to data-path operations if certain constraints for register and memory bank assignment are met. This is typical for a broad class of DSPs.

There are different interdependent phases in the code generation process which strongly affect code quality:
• Memory bank selection:
There are instructions which may only be parallelized if the operands are stored in different memory banks. For typical DSPs there are two memory banks and load operations may be executed simultaneously if they do not access the same bank.
• Register bank selection:
Often, instructions may only be parallelized if certain register banks are used which gives some boundary conditions to the selection of register banks. This is done during the straight line code generation as well as in the genetic algorithm but always before the code compaction. Thus the boundary conditions are evaluated after their assignment and a fitness value is fed back to the genetic algorithm.
• Tree scheduling:
A change in the tree schedule changes data dependencies. As an example, it may be allowed to load an operand into the source register earlier if there is no other access to this register in between. Exploiting this fact permits out-of-order execution of operations which may result in more compact code.

As all optimization steps are under direct control of the genetic algorithm, the compaction step is fully integrated into the rest of the optimization. Thus, phase coupling problems are mostly avoided.

In [13] several different methods suited for code compaction are compared. Most of them are very general but do not execute in linear time which is required to restrict the run-times. The list scheduling algorithm, which is a heuristic derived from a branch and bound algorithm, executes in linear time while still achieving good results. The fact that most current DSPs have only limited parallelism makes list scheduling a good choice for code compaction. However, the algorithm has to be modified slightly to fit to the data dependencies mentioned above.

To apply list scheduling, a dependency graph has to be built. This is done by the following algorithm:
1. Find all *strong dependencies*, i.e. for each operation find the instructions calculating the source operands.
2. Find all *weak dependencies*, i.e. look for instructions reading from registers which are overwritten by the current instruction.
3. Calculate the weight factor for all instructions. The weight is calculated by counting the number of instructions which are dependent on the current instruction.

Now the dependency graph is used to generate optimized parallel assembly instructions. This is done by applying the following steps:
1. Find and mark all instructions which do not have any strong dependencies. Weak dependencies are allowed at this point of time.
2. Sort the marked instructions by descending weight. This means that the command with the most dependent instructions is listed first.
3. Use the instruction with maximum weight as the basis for the current parallel command.
4. Check the remaining instructions and parallelize if weak dependencies exist only with instructions already selected for the current parallel command. Of course, parallel execution of these instructions must be supported by the target architecture.
5. If all conditions are met, add the instruction to the current parallel command. Continue with the previous step for the next marked instruction.
6. When no more instruction can be parallelized, the current multiple command is finished and added to the final code.
7. Start over at step one and repeat until all sequential instruction are put into parallel commands.

IV. Experimental Results

Several different DAGs have been processed to obtain a representative set of experimental results. As an example for highly optimized code, the program output for a 2nd order state space filter is shown in Fig. 6 (without generating code for the address generation units; this is not part of the paper but can easily be done by applying established algorithms like in [14]). The calculation was done with a population of 500 individuals, a selection pressure $p = 50$, and over 1500 generations. The result is optimal for the given problem, as there are 9 distinctive arithmetic operations which cannot be parallelized any further and one move instruction which cannot be saved either.

Fig. 7 illustrates the convergence of the algorithm for 100 test runs. The center line corresponds to the best solution averaged over all test runs, the outer lines show the best and the worst result achieved. Experimental results show that the average code size drops exponentially at the beginning of the optimization process, starts saturating after about 80 iterations, and reaches saturation at about 300 iterations. The average result after 1500 iterations is 10.03 with an optimum solution of 10 assembly instructions. That is, only 3 out of 100 runs did not reach the optimum.

By choosing other parameter sets for the genetic algorithm, the performance can be tuned. Generally, a larger population size will always increase the quality of the result

```
Setup code:
 1:   move Y:x1,Y0
 2:   move Y:x2,A

Loop body:
 1:    move A,X0        Y:c8,Y1
 2:    mpy X0,Y1,B      X:c7,X0     Y:k,Y1
 3:    mac Y0,X0,B      X:x,X0
 4:    mac X0,Y1,B      X:c6,X0     A,Y1
 5:    mpy X0,Y1,B      X:c3,X0     B,Y:y
 6:    mac Y0,X0,B      X:x,X0      Y:c1,Y1
 7:    mac X0,Y1,B      A,X0        Y:c4,Y1
 8:    mpy X0,Y1,A      Y:c5,X0
 9:    mac Y0,X0,A      X:x,X0      Y:c2,Y0
 10:   mac X0,Y0,A      B,Y0
```

Fig. 6. Compacted code for a 2nd order state space filter

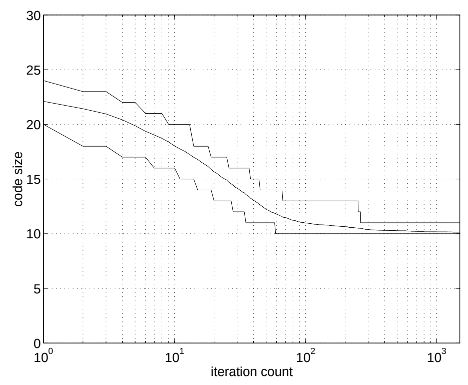

Fig. 7. Convergence of the evolutionary process

at the cost of a longer run-time. For an unbiased comparison, one has to use the total number of individuals calculated throughout the entire simulation which is done in Fig. 8.

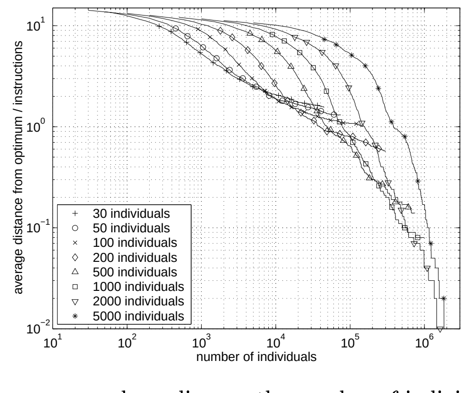

Fig. 8. Convergence depending on the number of individuals calculated for different population sizes

It can be seen that for each number of individuals calculated, which is equivalent to a certain amount of run-time, there is exactly one population size producing the best result. In other words, the population size can be chosen with respect to the desired quality of the solution. Note that there is a point of saturation when practically all runs reach the optimum solution. Further enlargement of the population will not lead to better performance anymore. This saturation point is highly problem dependent and may not be reached at all for large DFGs.

When varying the selection pressure, code quality shows a local optimum which, however, is not very sharp. Therefore the choice of the selection pressure is uncritical as long as the order of magnitude is correct.

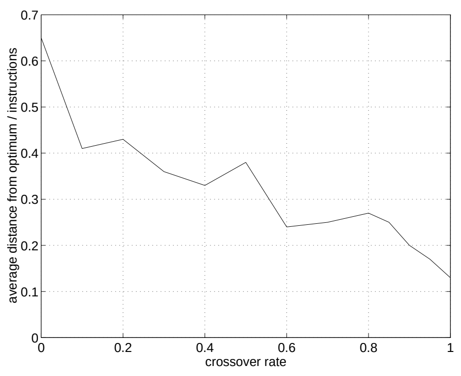

Fig. 9. Code quality depending on the crossover rate

In Fig. 9 the dependency of the compiler performance on the crossover rate is shown. Higher crossover rates lead to a better performance. As the algorithm allows several identical individuals within a population, a crossover rate of 100% does not necessarily mean that the next generation contains only new individuals (it is likely that some individuals are crossed with identical ones). Code quality improves with higher crossover rates, but also the convergence of the algorithm is significantly faster, if crossover is enabled.

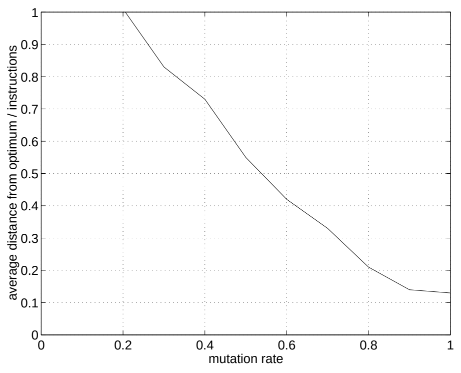

Fig. 10. Code quality depending on the mutation rate

Fig. 10 shows the dependency of the overall result on the mutation rate. It can be seen that higher mutation rates lead to a better performance with a saturation at about $p_m = 0.8$. This is significantly more than usual mutation rates for genetic algorithms. The reason for this behavior is the complex construction of the chromosome since different tasks like tree scheduling, access types and register bank selection have to be mixed. Without a high mutation rate, too many attributes would be lost forever after the selection procedure.

A summary of typical results is shown in Fig. 11. Each group of bars corresponds to a typical DSP problem, mostly different kinds of 2nd order filters. Within each group, the optimum solution and the output of the official Motorola C-compiler are compared to the experimental results of the proposed technique. As the C-compiler is not able to exploit the benefits of two separate memory banks, different test runs have been done.

It can be seen that our technique is always at or close to the optimum solution when using all optimization techniques. With restriction to a single memory bank and without

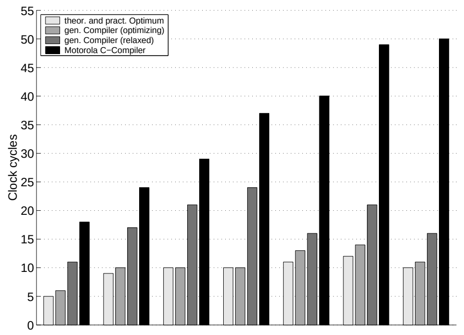

Fig. 11. Comparison of results for different applications

loop optimization, results are still significantly better than those of the C-compiler. Depending on the problem and on the degree of optimization, an improvement of a factor 2 to 5 may be achieved compared to the C-compiler.

V. Conclusions

The proposed technique is well-suited to time-critical applications. Greedy techniques, as applied in DSP-C-compilers, produce code that is unacceptable if there are stringent timing constraints. The procedure discussed in this paper generates high-quality code that can typically only be obtained by cumbersome hand-coding.

References

[1] V. Zivojnovic, J. M. Velarde, C. Schläger, and H. Meyr, "DSPstone: a DSP-oriented benchmarking methodology," in *Proc. 5th Int. Conf. on Signal Processing Applications & Technology*, Dallas, October 1994, vol. 1, pp. 715–720.

[2] A. V. Aho, S. C. Johnson, and J. D. Ullman, "Code generation for expressions with common subexpressions," *Journal of the ACM*, vol. 24, no. 1, pp. 146–160, January 1977.

[3] A. V. Aho and S. C. Johnson, "Optimal code generation for expression trees," *Journal of the ACM*, vol. 23, no. 3, pp. 488–501, July 1976.

[4] B. Wess, "Code generation based on trellis diagrams," in *Code Generation for Embedded Processors*, P. Marwedel and G. Goossens, Eds., chapter 11, pp. 188–202. Kluwer Academic Publishers, 1995.

[5] G. Araujo and S. Malik, "Optimal code generation for embedded memory non-homogeneous register architectures," in *Proc. 7th Int. Symp. on System Synthesis*, Cannes, France, September 1995, pp. 36–41.

[6] B. Wess and W. Kreuzer, "Optimized DSP assembly code generation starting from homogeneous atomic data flow graphs," in *Proc. 38th Midwest Symp. on Circuits and Systems*, Rio de Janeiro, August 1995, vol. 2, pp. 1268–1271.

[7] S. Liao, S. Devadas, K. Keutzer, and S. Tjiang, "Instruction selection using binate covering for code size optimization," in *Proc. Int. Conf. on Computer-Aided Design*, 1995, pp. 393–399.

[8] G. Araujo and S. Malik, "Using register-transfer paths in code generation for heterogeneous memory-register architectures," in *Proc. 33rd ACM/IEEE Design Automation Conf.*, Las Vegas, June 1996.

[9] M. A. Ertl, "Optimal code selection in dags," in *Proc. ACM Int. Symp. on Principles of Programming Languages*, 1999.

[10] R. Leupers, "Register allocation for common subexpressions in DSP data paths," in *Proc. Asia Pacific Design Automation Conference*, Yokohama, January 2000.

[11] B. Wess, "Simulated evolutionary code generation for heterogeneous memory-register DSP-architectures," in *Proc. European Signal Processing Conference*, Tampere, September 2000, vol. 4.

[12] A. Sudarsanam and S. Malik, "Memory bank and register allocation in software synthesis for ASIPs," in *Proc. IEEE Int. Conf. on Computer-Aided Design*, San Jose, November 1995.

[13] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett, "Local microcode compaction techniques," *ACM Computing Surveys*, vol. 12, no. 3, pp. 261–294, September 1980.

[14] B. Wess, "Minimization of data address computation overhead in DSP programs," *Kluwer Design Automation for Embedded Systems*, vol. 4, pp. 167–185, March 1999.